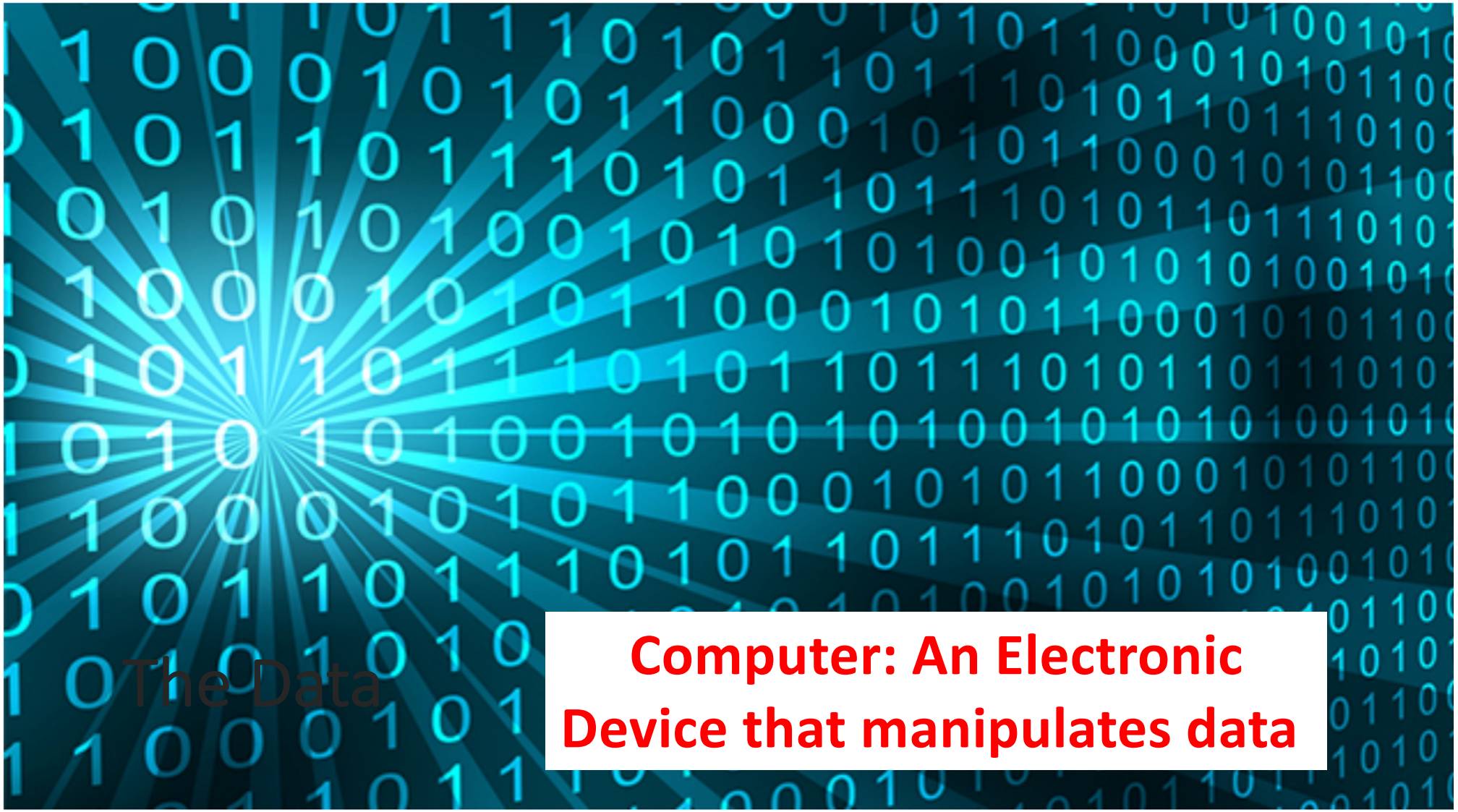
SimCenter NHERI
Center for Computational Modeling and Simulation

2020 Programming Bootcamp

Parallel Programming and MPI

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843



Computer: An Electronic Device that manipulates data

REMEMBER

```
git fetch upstream
git merge upstream/master
```

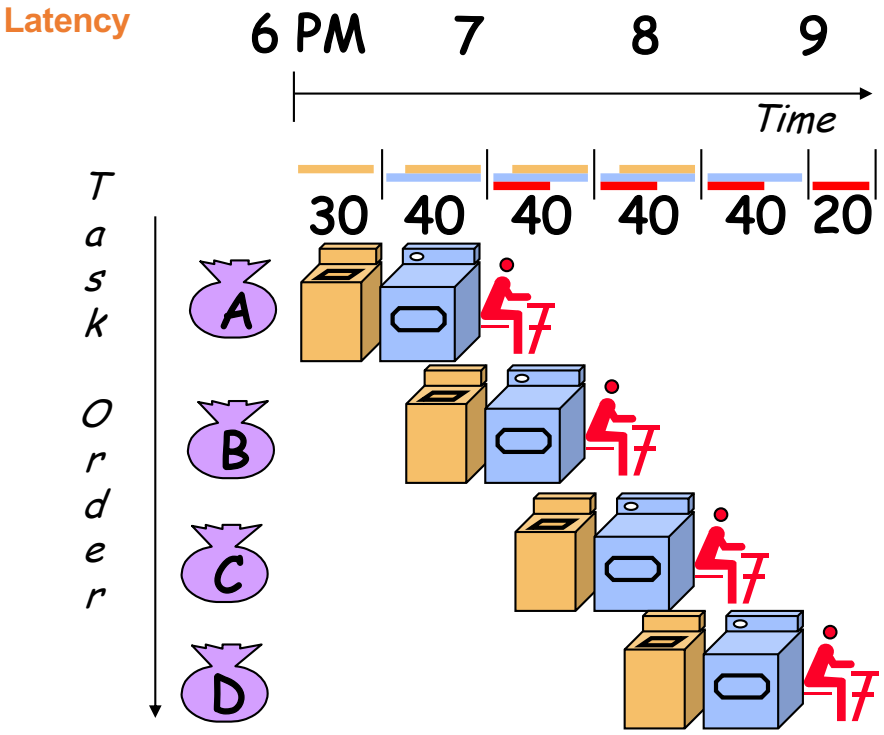
On Frontera today use idev before compiling & running mpi jobs

```
idev
```

Pipelining

Dave Patterson's Laundry example: 4 people doing laundry

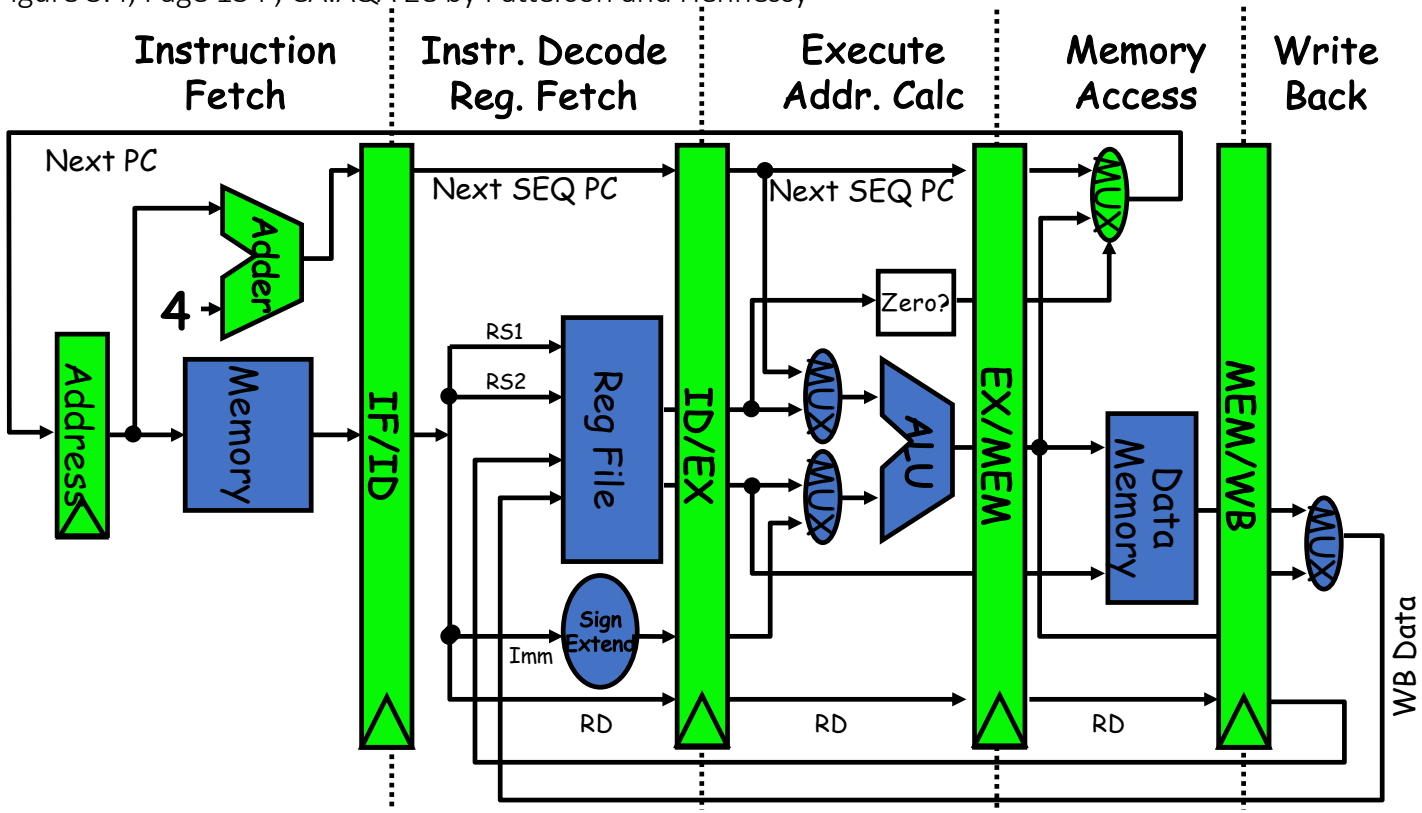
wash (30 min) + dry (40 min) + fold (20 min) = 90 min



- In this example:
  - Sequential execution takes 4 * 90min = 6 hours
  - Pipelined execution takes 30+4*40+20 = 3.5 hours
- Bandwidth = loads/hour
- BW = 4/6 l/h w/o pipelining
- BW = 4/3.5 l/h w pipelining
- BW <= 1.5 l/h w pipelining, more total loads
- Pipelining helps bandwidth but not latency (90 min)
- Bandwidth limited by slowest pipeline stage
- Potential speedup = Number pipe stages

Example: 5 Steps of MIPS Datapath

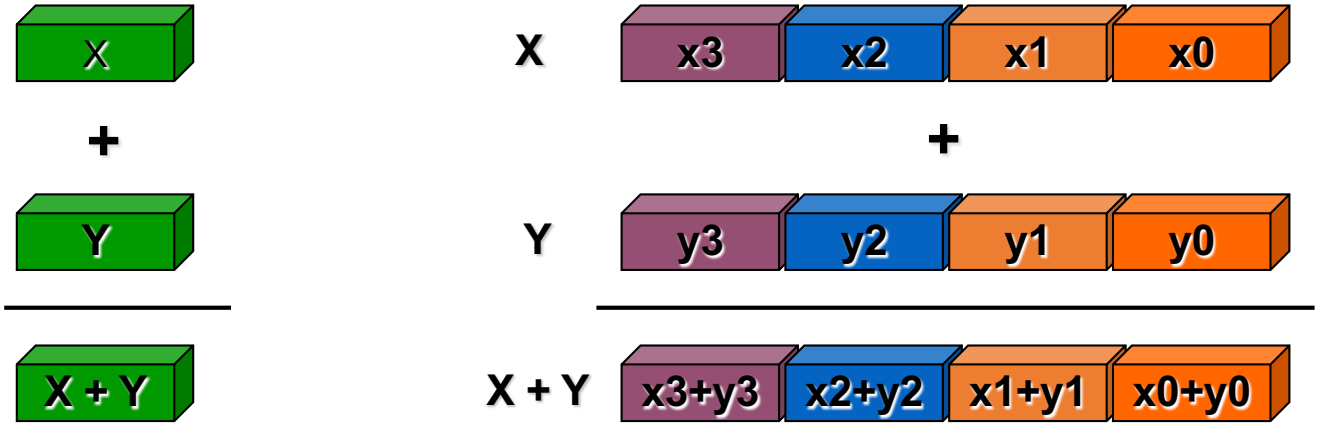
Figure 3.4, Page 134 , CA:AQA 2e by Patterson and Hennessy



- **Pipelining is also used within arithmetic units**
  - a fp multiply may have latency 10 cycles, but throughput of 1/cycle

SIMD: Single Instruction, Multiple Data

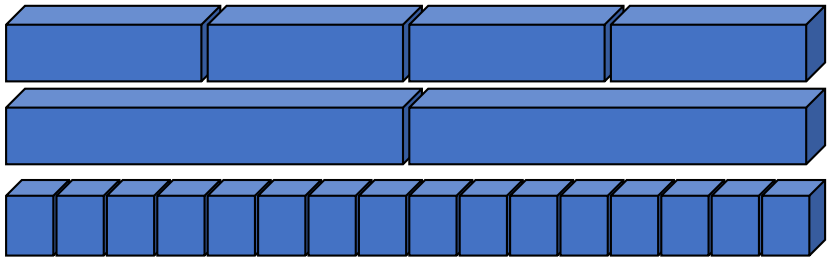
- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing
  - with SSE / SSE2
  - SSE = streaming SIMD extensions
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data around from one part of register to another
- Similar on GPUs, vector processors (but many more simultaneous operations)

What does this mean to you?

- In addition to SIMD extensions, the processor may have other special instructions
  - Fused Multiply-Add (FMA) instructions:

    x = y + c * z

    is so common some processor execute the multiply/add as a single instruction, at the same rate (bandwidth) as + or * alone
- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help
  - Choose a different compiler, optimization flags, etc.
  - Rearrange your code to make things more obvious
  - Using special functions ("intrinsics") or write in assembly ☹

Speed Comparison

There's plenty of room at the Top: What will drive computer performance after Moore's law?

Charles E. Leiserson, Neil C. Thompson*, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, Tao B. Schardl

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFL[illegible] See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

MONDAY

TODAY

Why is Parallel Programming Important

1 Intel i7 node, 6 cores + 16GB RAM

Apple A11, 6 cores + 64GB RAM

Frontera: 8008 “compute” nodes, Intel Xeon Cascade Lake with **56** cores per node

= **448,448** cores available, each node 192GB RAM, 480GB SSD local drive

 # 8, June 2020 TOP500 SUPERCOMPUTER SITES

DESIGNSAFE-CI

Consequence of Moore's Law Today

- Number of cores per chip can double every two years
- Clock speed will not increase (possibly decrease)
- Need to deal with systems with millions of concurrent threads
- Need to deal with inter-chip parallelism as well as intra-chip parallelism

What does it all mean for Programmers

"The Free Lunch is Over" Herb Sutter

source: CS267, Jim Demmell

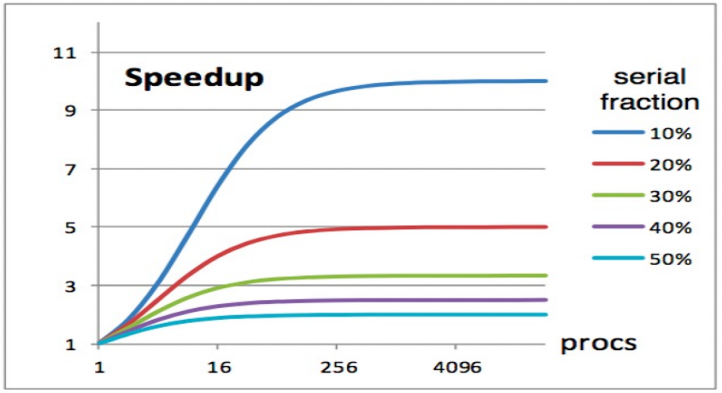
Can All Programs Be Made to Run Faster?

- Suppose only part of an application can run in parallel
- Amdahl's law
  - let s be the fraction of work done sequentially, so (1-s) is fraction parallelizable
  - P = number of processors

$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$

$$<= 1/(s + (1-s)/P)$$

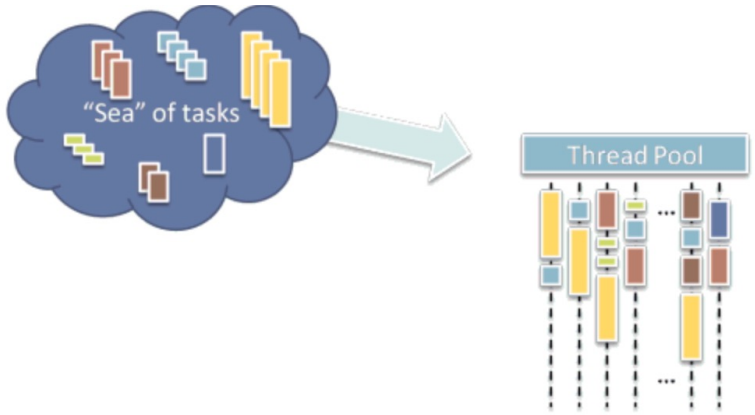
$$<= 1/s$$



- **Even if the parallel part speeds up perfectly performance is limited by the sequential part**
- Top500 list: currently fastest machine has P~7.3M; Frontera has 448,448

Considerations for Parallel Programming:

- Finding enough tasks that can run concurrently for parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning
- Where to put the task,



All of these things makes parallel programming harder than sequential programming.

Improving Real Performance

Peak Performance grows exponentially, a la Moore's Law

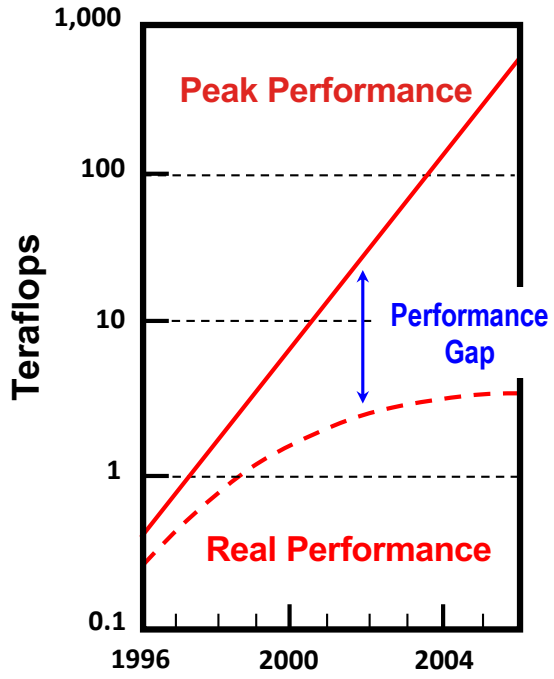
- In 1990's, peak performance increased 100x; in 2000's, it will increase 1000x

But efficiency (the performance relative to the hardware peak) has declined

- was 40-50% on the vector supercomputers of 1990s
- now as little as 5-10% on parallel supercomputers of today

Close the gap through ...

- Mathematical methods and algorithms that achieve high performance on a single processor and scale to thousands of processors
- More efficient programming models and tools for massively parallel supercomputers

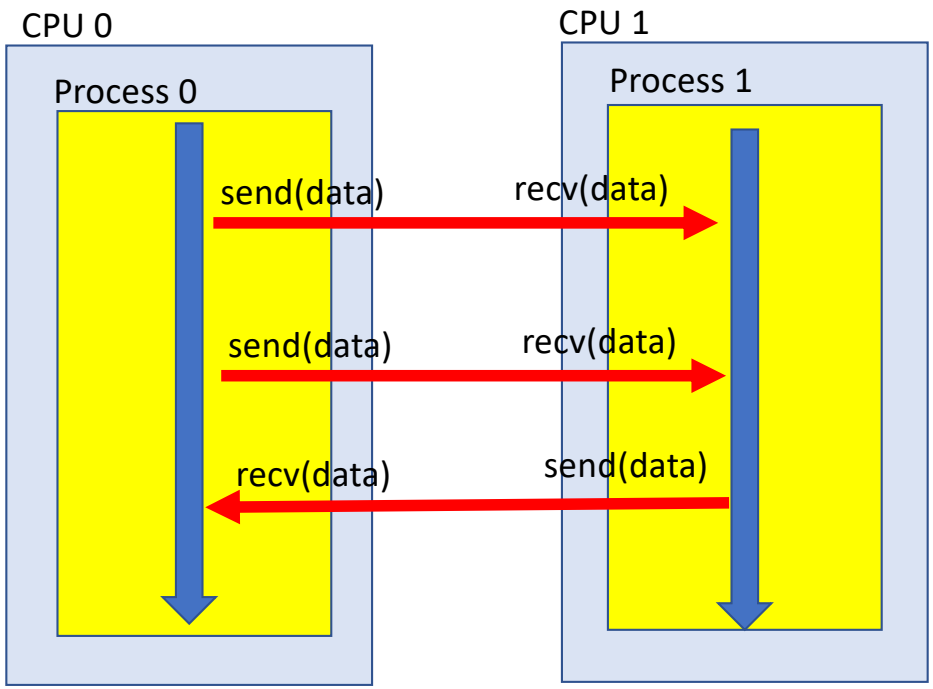


Writing Programs to Run on Parallel Machines

- C Programming Libraries Exist that provides the programmer an API for writing programs that will run in parallel.
- They provide a **Programming Model** that can be portable across architectures, e.g. most importantly the message passing model runs on a shared memory machine.
- We will look at 2 of these Programming Models and Libraries that support the model:
  - Message Passing Programming using MPI (message passing interface)
  - Thread Programming using OpenMP
- As with all libraries they can incur an overhead.

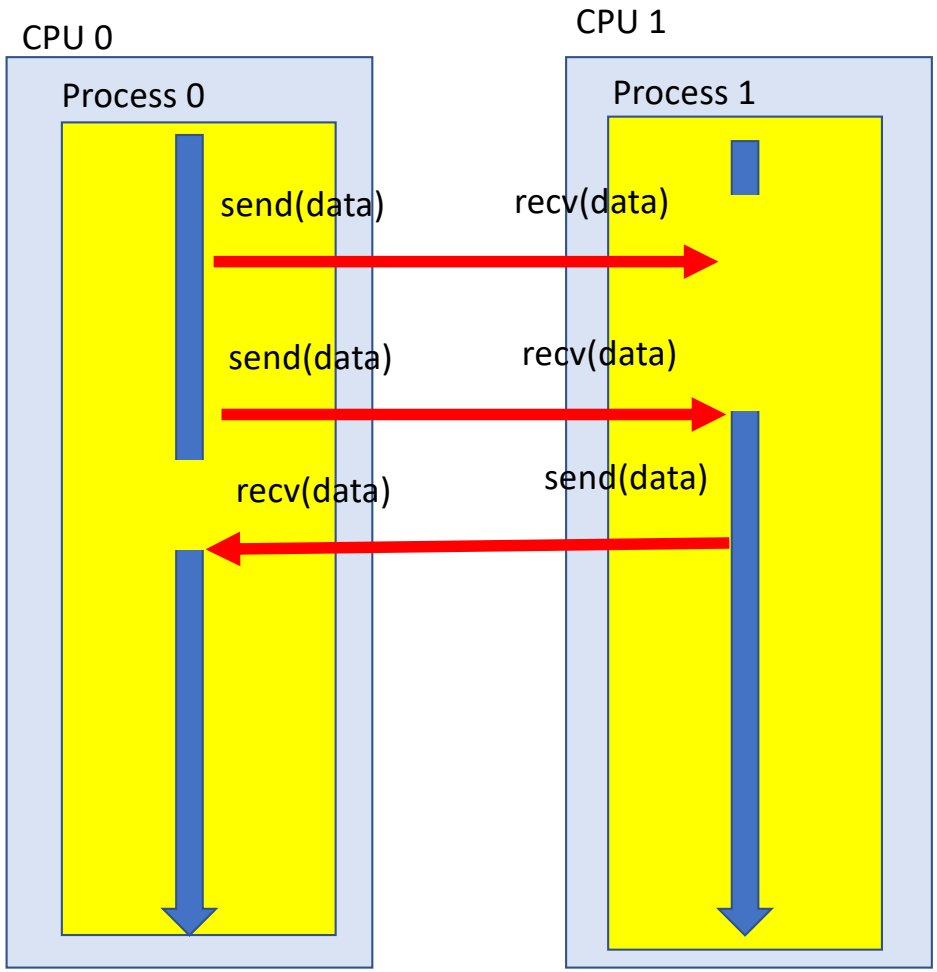
Message Passing Model

- Processes run independently in their own memory space and processes communicate with each other when data needs to be shared



- Basically you write sequential applications with additional function calls to send and recv data.

Only Get Speedup if processes can be kept busy



MPI

Provides a number of functions:

1. Enquiries
   - How many processes?
   - Which one am I?
   - Any messages Waiting?
2. Communication
   - Pair-wise point to point send and receive
   - Collective/Group: Broadcast, Scatter/Gather
   - Compute and Move: sum, product, max …
3. Synchronization
   - Barrier

Hello World - MPI

code/Parallel/mpi/hello1.c

```
#include <mpi.h>
#include <stdio.h>

int main( int argc, char **argv)
{
    int procID, numP;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numP );
    MPI_Comm_rank( MPI_COMM_WORLD, &procID );

    printf( "Hello World, I am %d of %d\n", procID, numP );
    MPI_Finalize();
    return 0;
}
```

MPI functions (and MPI_COMM_WORLD) are defined in mpi.h

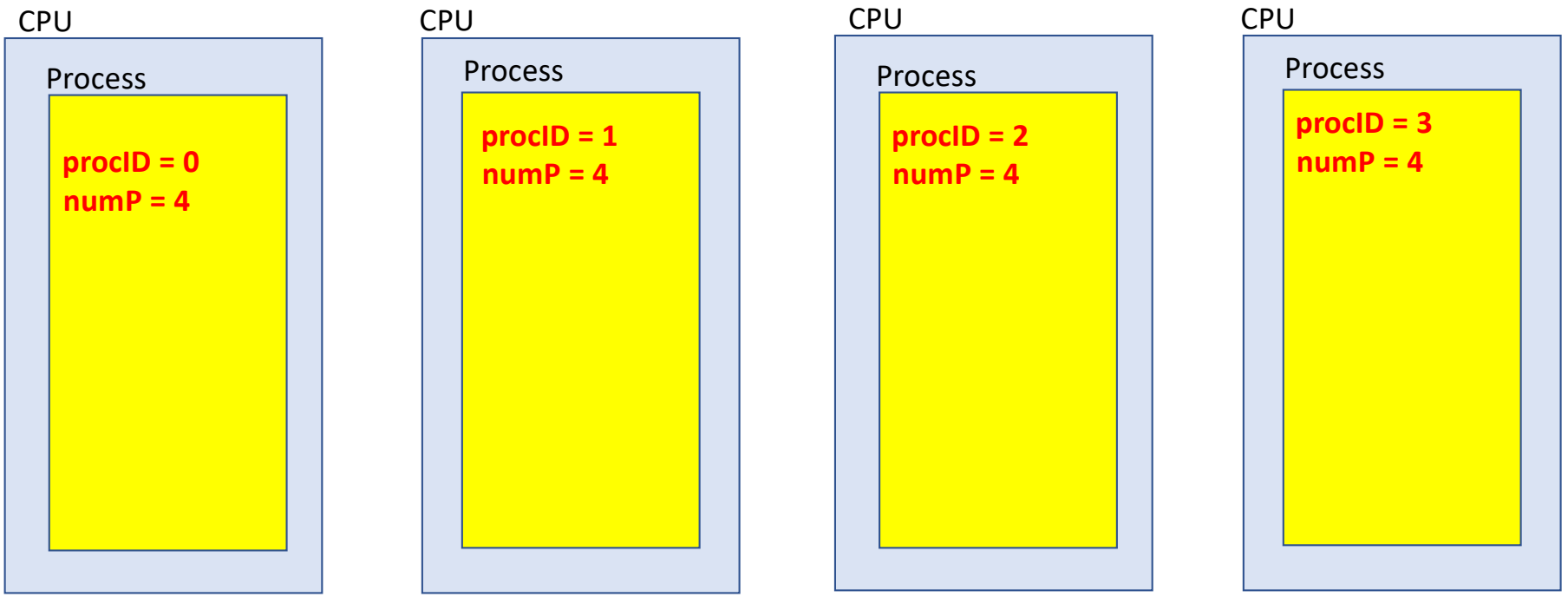
MPI_Init() must be first function called

MPI_COMM_WORLD is a default group contaning all processes

MPI_Comm_size returns # of processes in the group

MPI_Comm_rank returns processes unique ID the group, 0 through (numP-1)

MPI_finalize() must be last function called


CPU
Process
procID = 0
numP = 4
CPU
Process
procID = 1
numP = 4
CPU
Process
procID = 2
numP = 4
CPU
Process
procID = 3
numP = 4

Send/Recv blocking

code/Parallel/mpi/send1.c

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char **argv) {
  int procID;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &procID );

  if (procID == 0) { // process 0 sends
    int buf[2] = {12345, 67890};
    MPI_Send( &buf, 2, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (procID == 1) { // process 1 receives

    int data[2];
    MPI_Recv( &data, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    printf( "Received %d %d\n", data[0], data[1] );
  }

  MPI_Finalize();
  return 0;
}
```

NOTE the PAIR of Send/Recv

Not Quite as Simple as ensuring PAIRS of send/recv

code/Parallel/mpi/send2.c

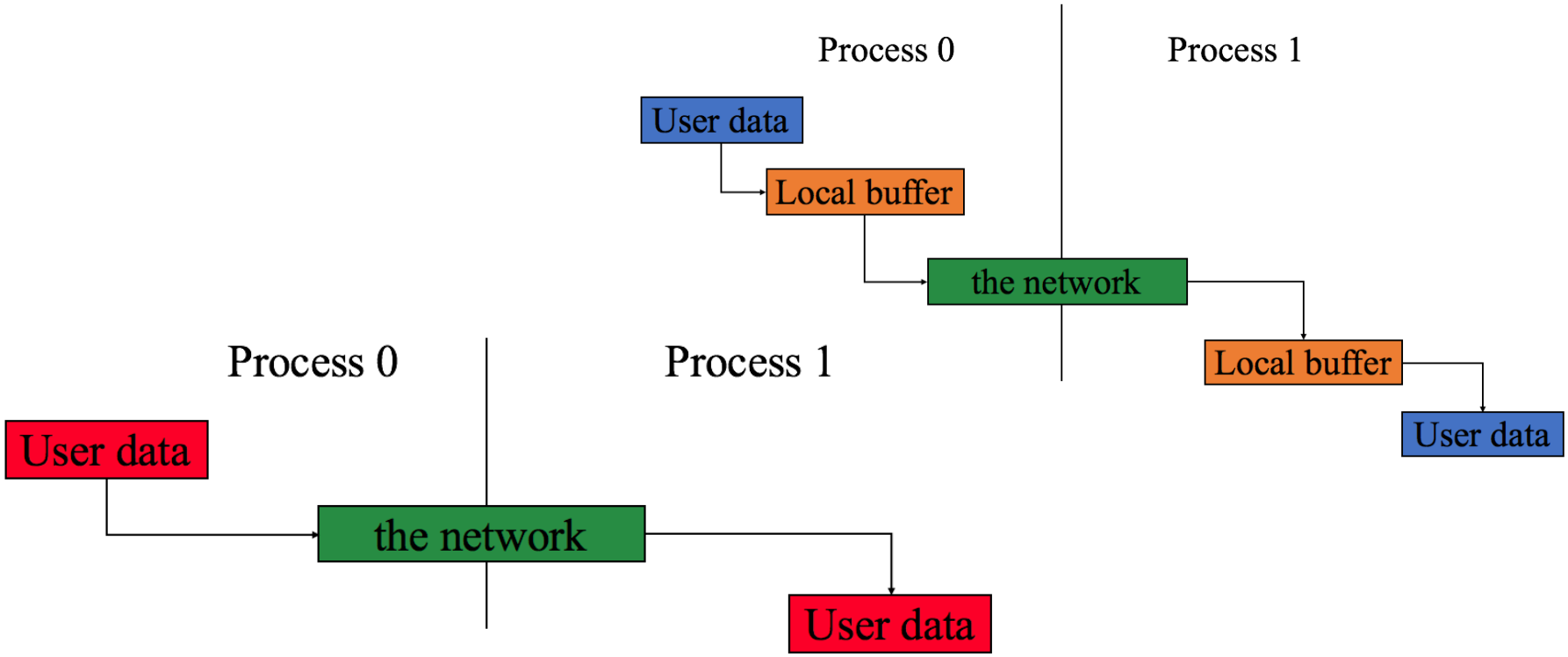
```
#include <mpi.h>
#include <stdio.h>
#define DATA_SIZE 1000
int main(int argc, char **argv) {
   int procID, numP;
   MPI_Status status;
   int buf[DATA_SIZE];

   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &numP );
   MPI_Comm_rank( MPI_COMM_WORLD, &procID );

   If (procID == 0) {
      for (int i=0; i<DATA_SIZE; i++) buf[i]=1+i;
      MPI_Send(&buf, DATA_SIZE, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv(&buf, DATA_SIZE, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
      printf("%d Received %d %d\n", procID, buf[0], buf[DATA_SIZE-1]);
   } else if (procID == 1) {
      for (int i=0; i<DATA_SIZE; i++) buf[i]=DATA_SIZE-i;
      MPI_Send(&buf, DATA_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv(&buf, DATA_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
     printf("%d Received %d %d\n", procID, buf[0], buf[DATA_SIZE-1]);
   }
   MPI_Finalize();
   return 0;
}
```

```
mpi >mpicc send2.c;   ibrun  -n 2 ./a.out
Buffer Size: 1000
0 Received 1000 1
1 Received 1 1000
```

```
mpi >mpicc send2.c;   ibrun  -n 2 ./a.out
Buffer Size: 10000
^Cmpi >
mpi >
```

DEADLOCK .. PROGRAM HANGS .. WHY?

Why Deadlock? .. Where Does the Data Go



If large message & insufficient data, the send() must wait for buffer to clear through a recv()

source: CS267, Jim Demmell

Current Problem:

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

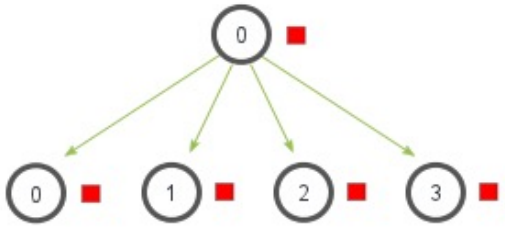
Could revise order
this requires some code rewrite:

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

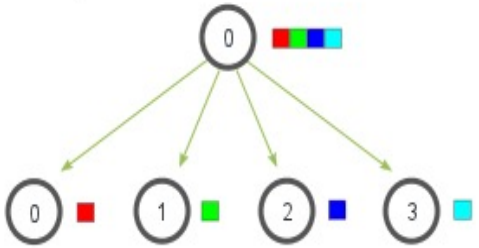
Alternatives use non-blocking sends.

Some Collective Functions

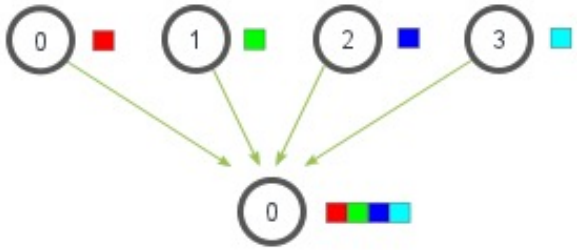
MPI_Bcast



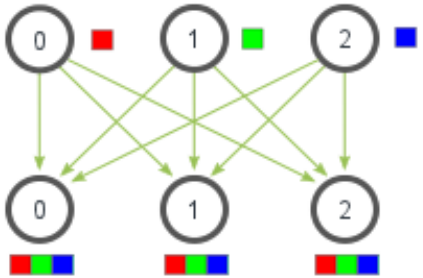
MPI_Scatter



MPI_Gather



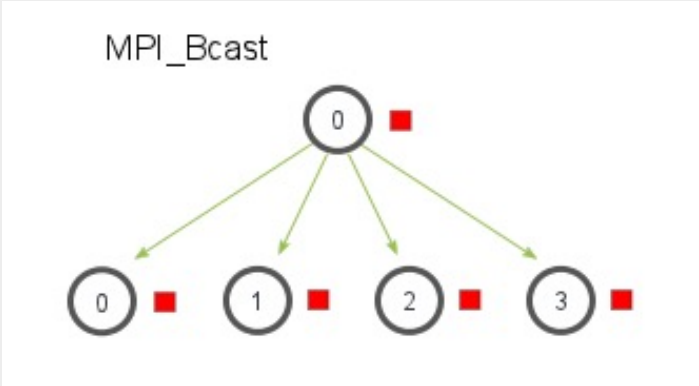
MPI_Allgather



Broadcast

code/Parallel/mpi/bcast1.c

```
#include <mpi.h>
#include <stdio.h>

int main( int argc, char **argv) {
   int procID, buf[2];

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &procID );

   if (procID == 0) {
      buf[0] = 12345;
      buf[1] = 67890;
   }

   MPI_Bcast(&buf, 2, MPI_INT, 0, MPI_COMM_WORLD);

   printf("Process %d data %d %d\n", procID, buf[0], buf[1]);

   MPI_Finalize();
   return 0;
}
```

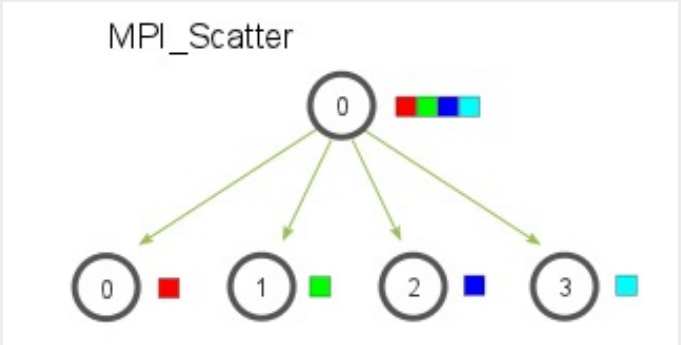


int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Scatter

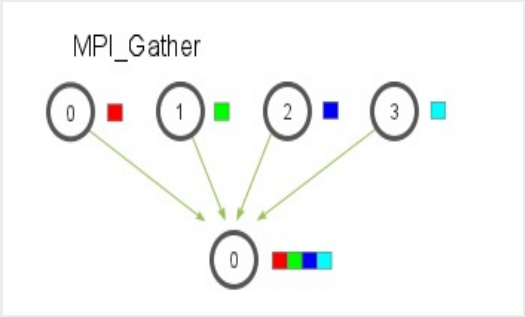
code/Parallel/mpi/scatter1.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define LUMP 5
int main(int argc, char **argv) {
  int numP, procID;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numP);
  MPI_Comm_rank(MPI_COMM_WORLD, &procID);

  int *globalData=NULL;
  int localData[LUMP];

  if (procID == 0) { // malloc and fill in with data
    globalData = malloc(LUMP * numP * sizeof(int) );
    for (int i=0; i<LUMP*numP; i++)
      globalData[i] = i;
  }
  MPI_Scatter(globalData, LUMP, MPI_INT, &localData, LUMP, MPI_INT, 0, MPI_COMM_WORLD);
  printf("Processor %d has first: %d last %d\n", procID, localData[0], localData[LUMP-1]);

  if (procID == 0)  free(globalData);

  MPI_Finalize();
}
```



int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

code/Parallel/mpi/gather1.c

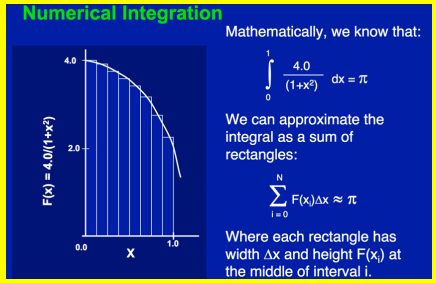



```
#include "mpi.h"
#include <stdio.h>
#define LUMP 5
int main(int argc, const charr **argv) {
     int procID, numP, ierr;
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numP);
     MPI_Comm_rank(MPI_COMM_WORLD, &procID);

      int *globalData=NULL;
      int localData[LUMP];
     if (procID == 0) { // malloc global data array only on P0
         globalData = malloc(LUMP * numP * sizeof(int) );
      }
     for (int i=0; i<LUMP; i++)
          localData[i] = procID*10+i;

     MPI_Gather(localData, LUMP, MPI_INT, globalData, LUMP, MPI_INT, 0, MPI_COMM_WORLD);

     if (procID == 0) {
         for (int i=0; i<numP*LUMP; i++)
               printf("%d ", globalData[i]);
         printf("\n");
     }
     if (procID == 0) free(globalData);
     MPI_Finalize();
```

MPI can be simple

- Claim: most MPI applications can be written with only 6 functions (although which 6 may differ)
- Using point-to-point:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECEIVE`
- Using collectives:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_BCAST/MPI_SCATTER`
  - `MPI_GATHER/MPI_ALLGATHER`
- You may use more for convenience or performance

Exercise 1: Parallelize Compute PI using MPI



pi.c and gather1.c in assignments/C-Day4/ex1

Exercise 2: Compute Vector Norm using MPI

$$||u||_2 = \sqrt{u_1^2 + u_2^2 + \cdots + u_n^2}$$

scatterArray.c in assignments/C-Day4/ex2