SimCenter NHERI
Center for Computational Modeling and Simulation

2020 Programming Bootcamp

Parallel Programming

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843

Running our small MPI Programs at TACC

```
idev
cmake  –build .
ibrun –n 4 ./pi
..
..
exit
```

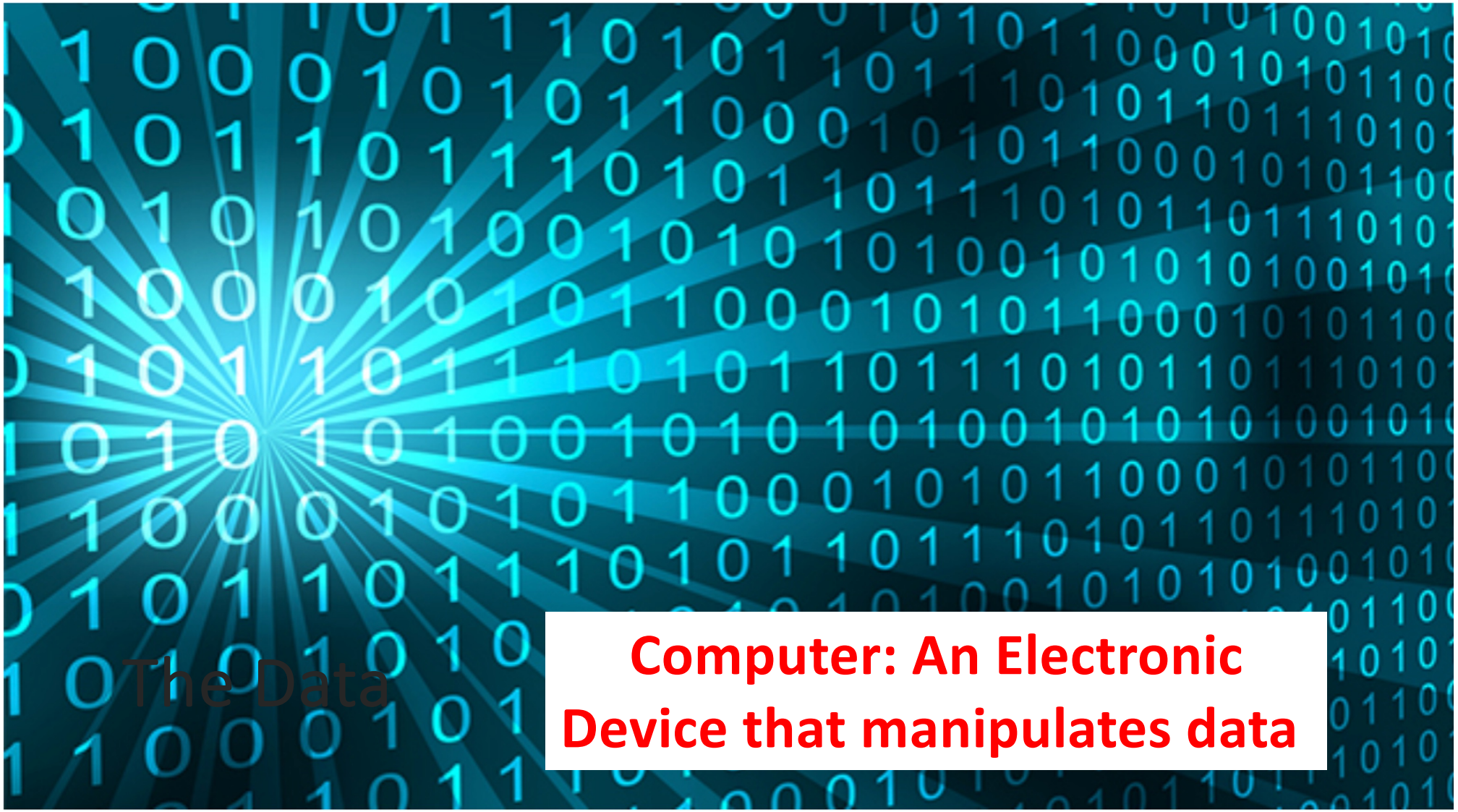
SimCenter NHERI
Center for Computational Modeling and Simulation

2020 Programming Bootcamp

Parallel Programming

Frank McKenna
University of California at Berkeley

NSF award: CMMI 1612843



Computer: An Electronic Device that manipulates data

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

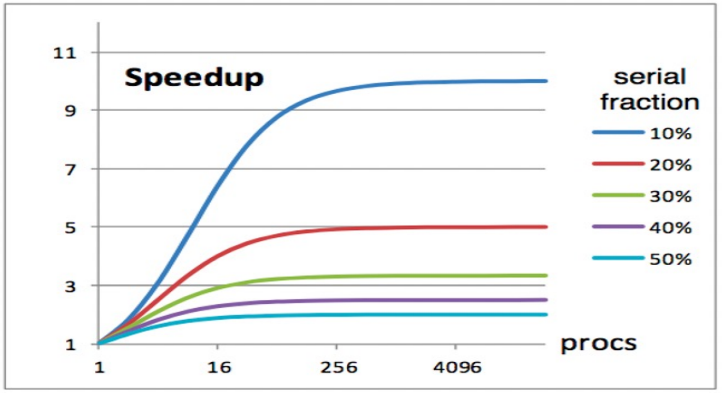
Can All Programs Be Made to Run Faster?

- 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

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

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

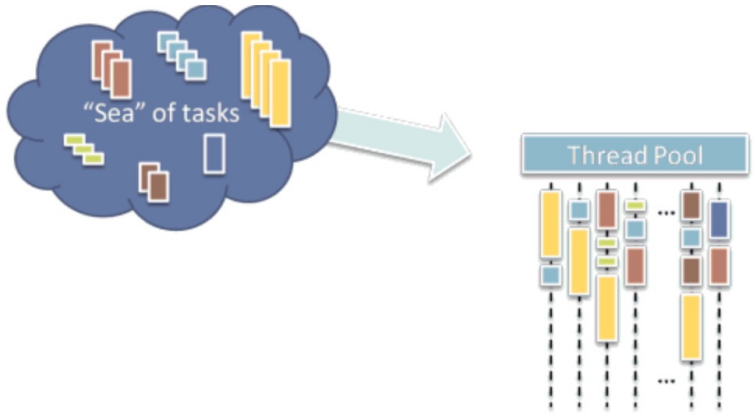
$$<= 1/s$$



- **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

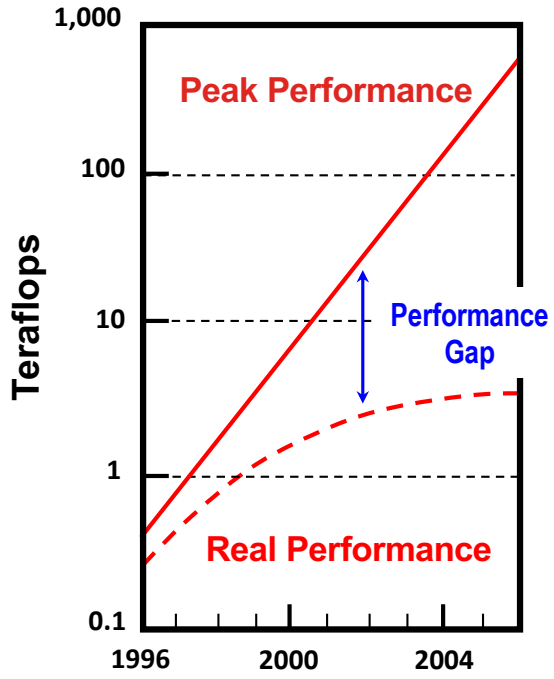
- In 1990's, peak performance increased 100x; in 2000's, it will increase 1000x

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

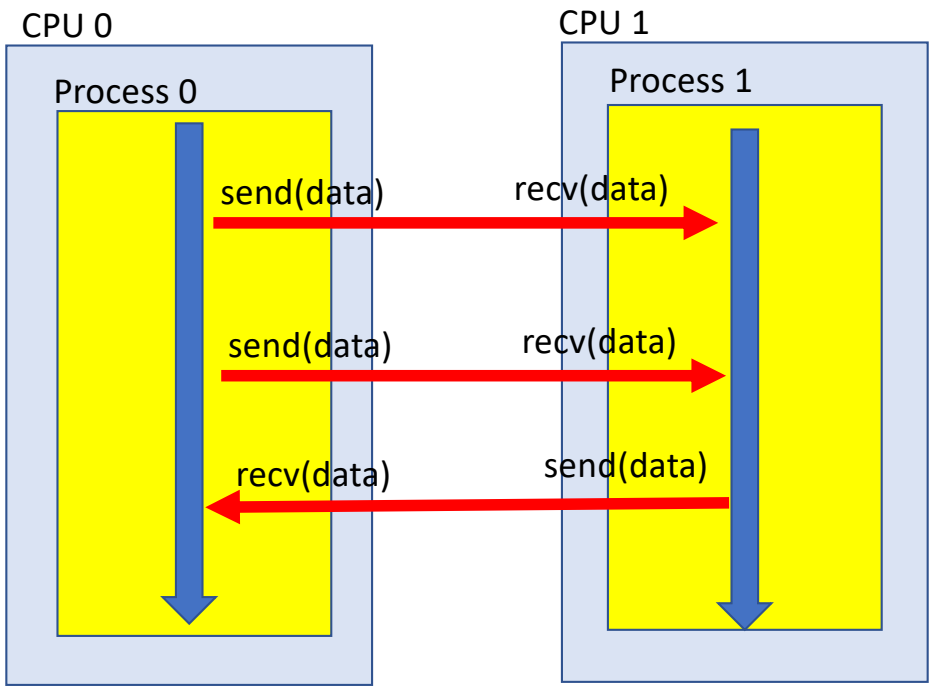


Writing Programs to Run on Parallel Machines

- 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.

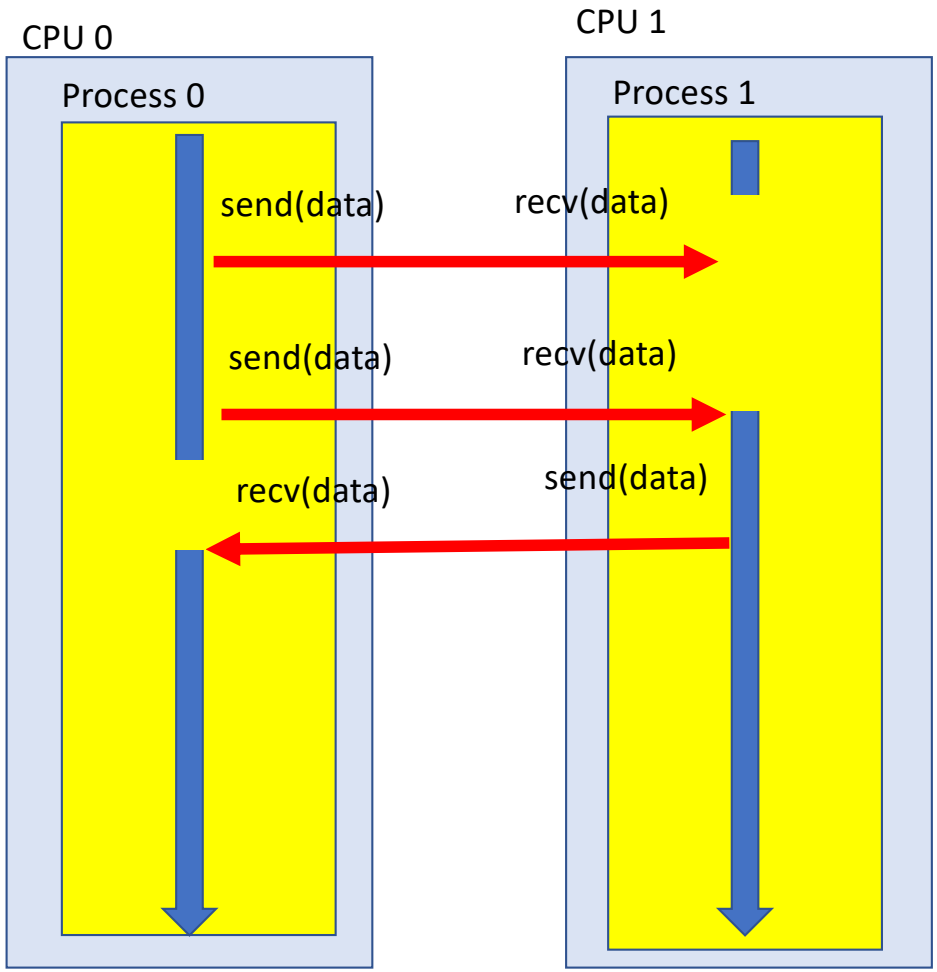
Message Passing Model

- 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

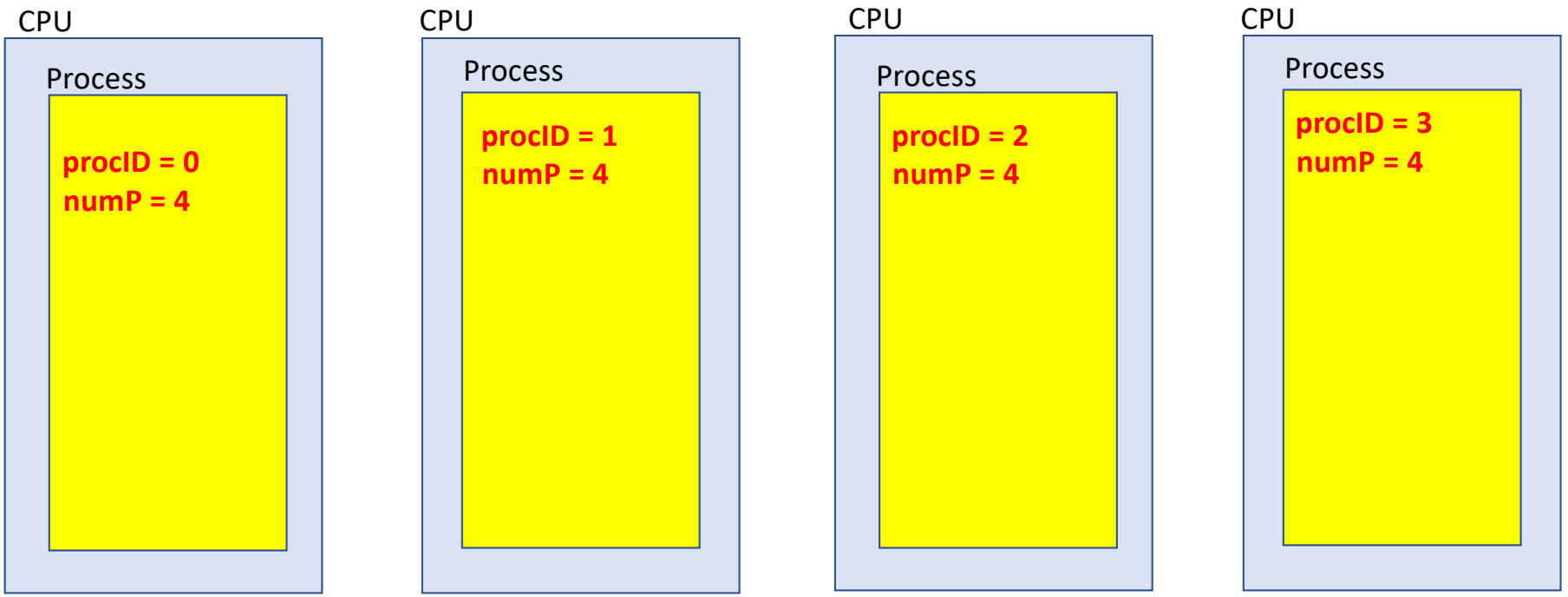
MPI_Init() must be first function called

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

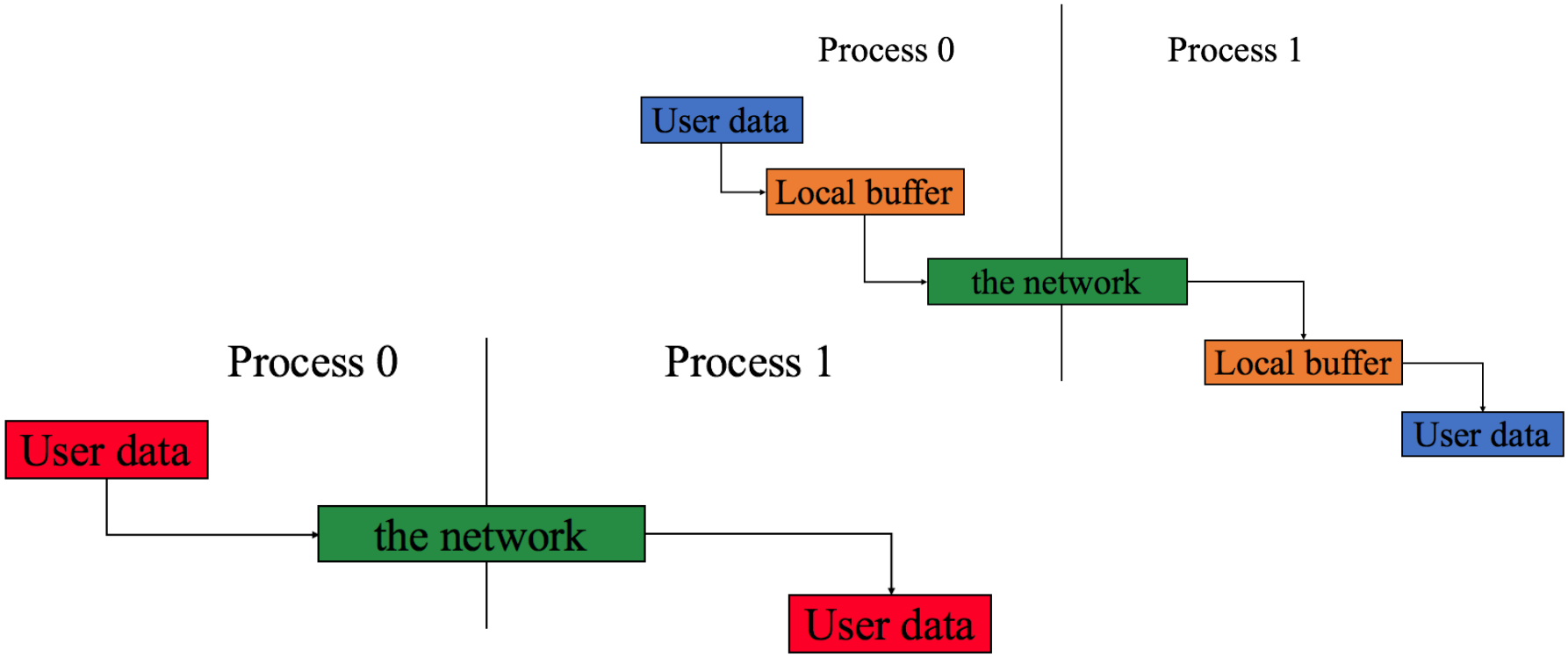
```
#include <mpi.h>
#include <stdio.h>
#define DATA_SIZE 1000
int main(int argc, char **argv) {
   int procID, numP;
   MPI_Status status;
   int buf[DATA_SIZE];

   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)`

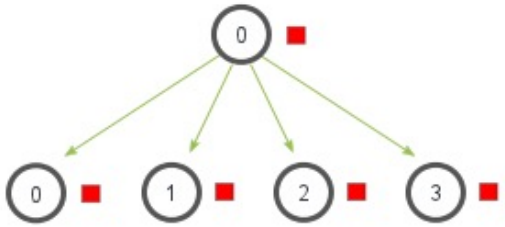
Could revise order
this requires some code rewrite:

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

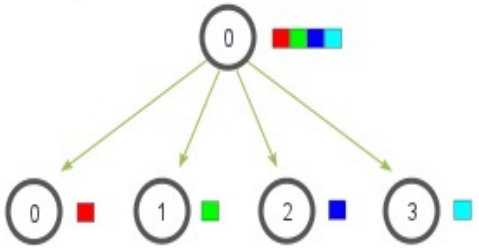
Alternatives use non-blocking sends.

Some Collective Functions

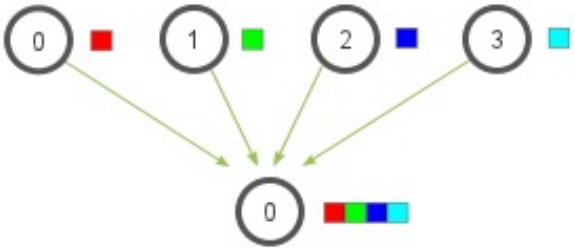
MPI_Bcast



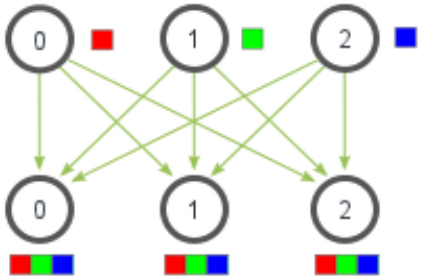
MPI_Scatter



MPI_Gather



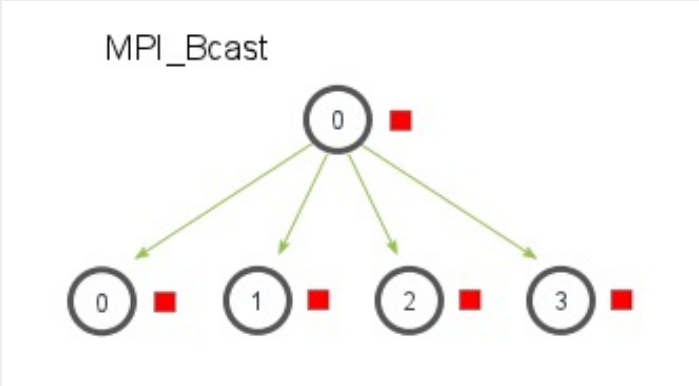
MPI_Allgather



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;
}
```

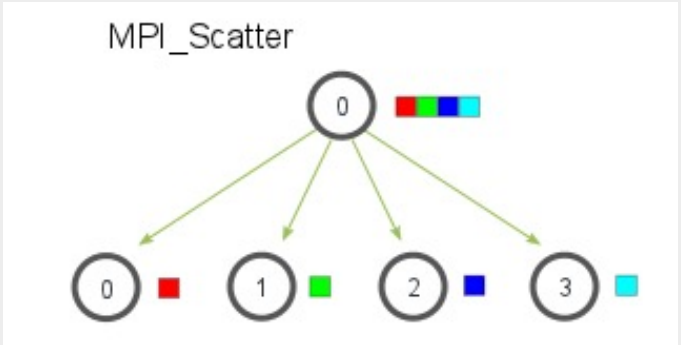


int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Scatter

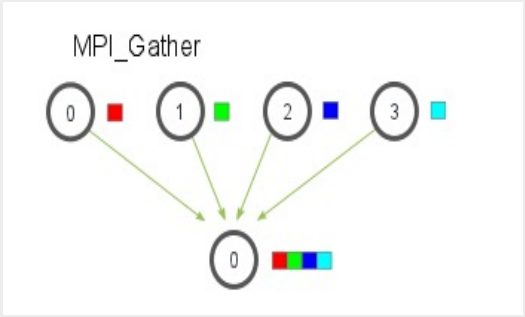
code/Parallel/mpi/scatter1.c

```
#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

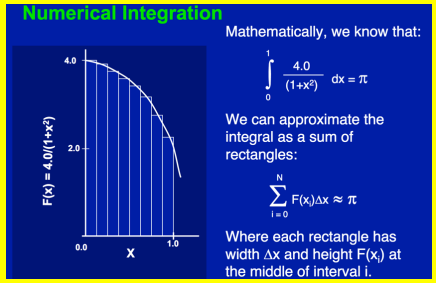



```
#include "mpi.h"
#include <stdio.h>
#define LUMP 5
int main(int argc, const charr **argv) {
     int procID, numP, ierr;
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numP);
     MPI_Comm_rank(MPI_COMM_WORLD, &procID);

      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



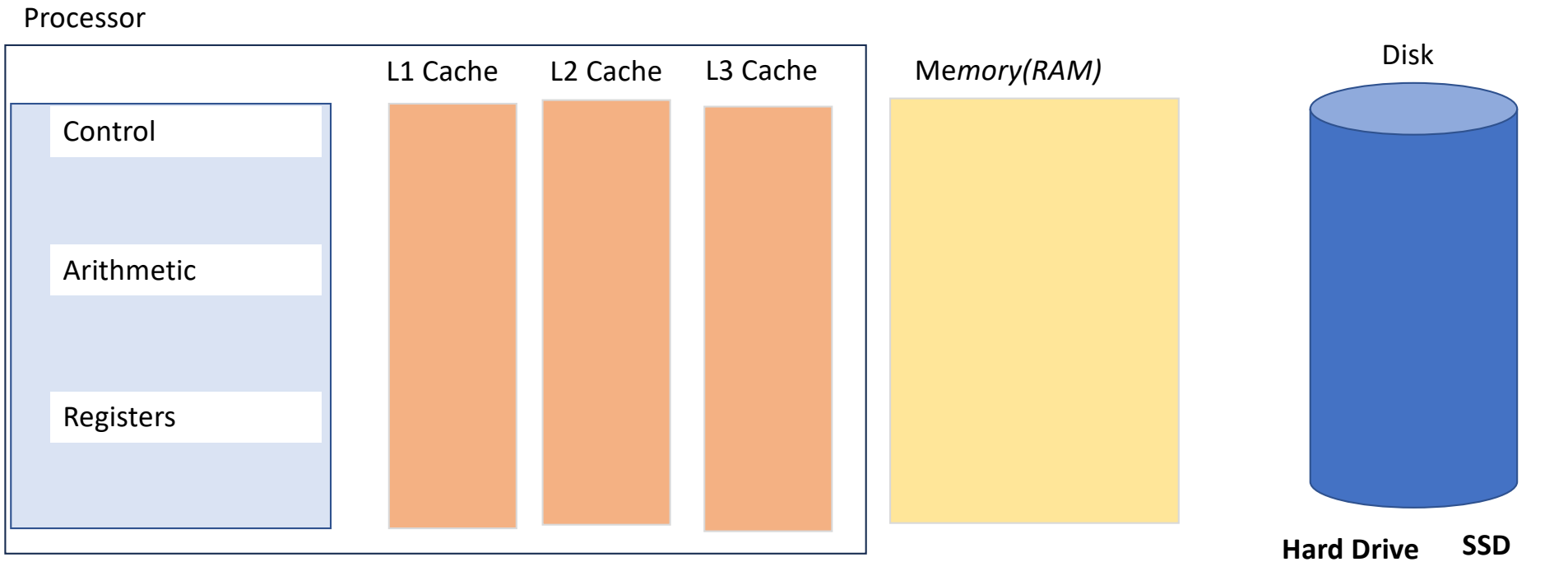
pi.c and gather1.c in assignments/C-Day4/ex1

Exercise 2: Compute Vector Norm using MPI

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

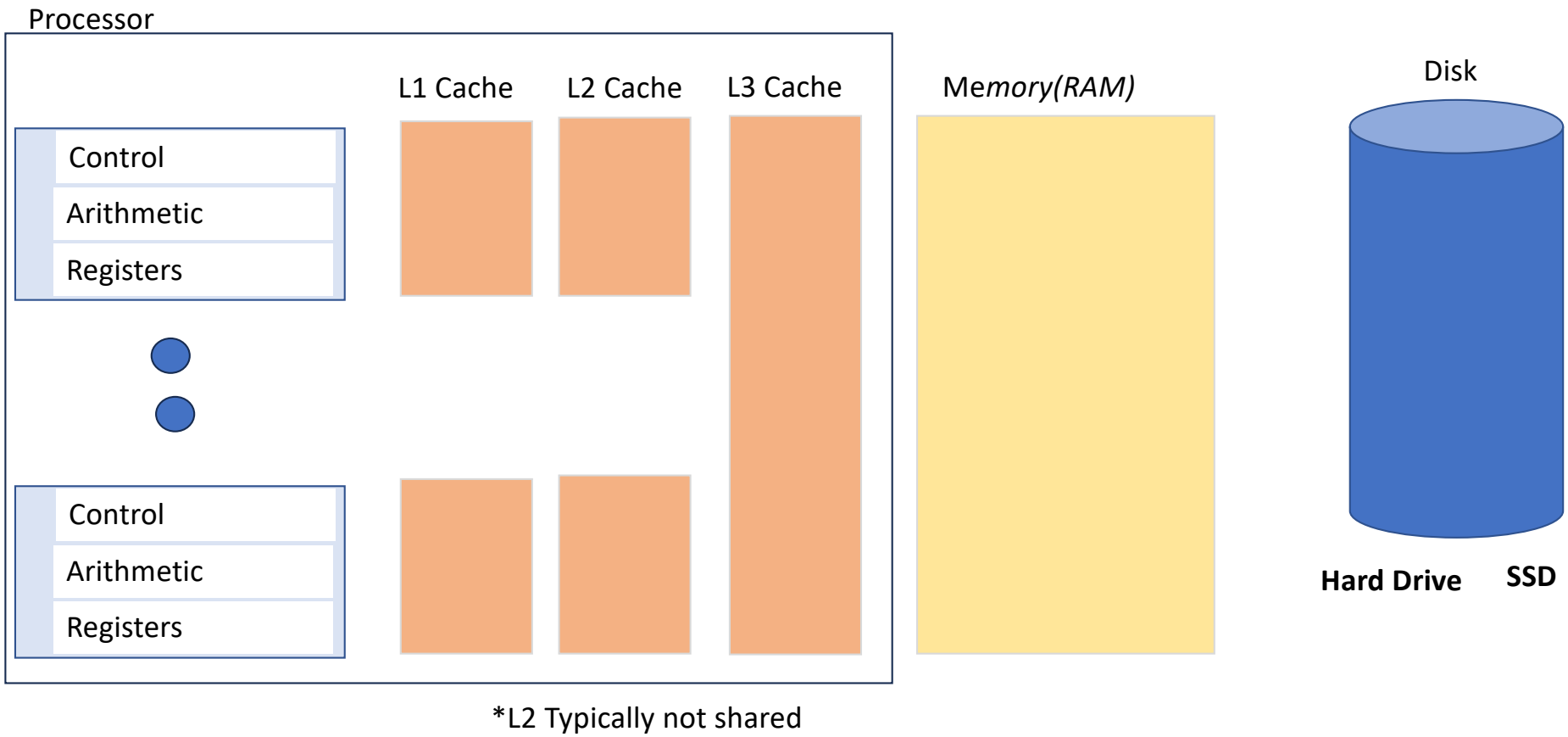
scatterArray.c in assignments/C-Day4/ex2

Idealized Processor Model

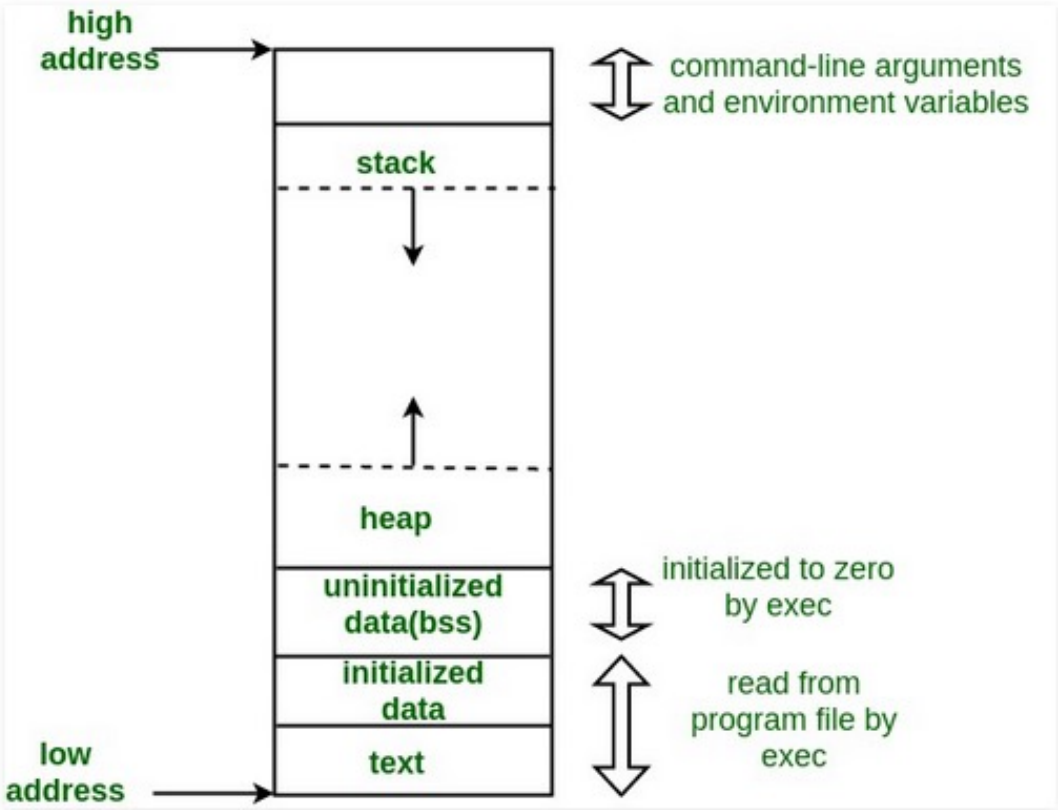


Size	1000 Bytes	64 KB	256 KB	2-4 MB	4-32 GB	4-16 TB	*.25-1TB*
Latency	0.3 ns	1 ns	3-10 ns	20-30 ns	50-100 ns	5-10e6 ns	25-50e3 ns
	Compiler	HW	HW	HW	Operating System	Operating System	

For Threaded Programming Let Us Revise

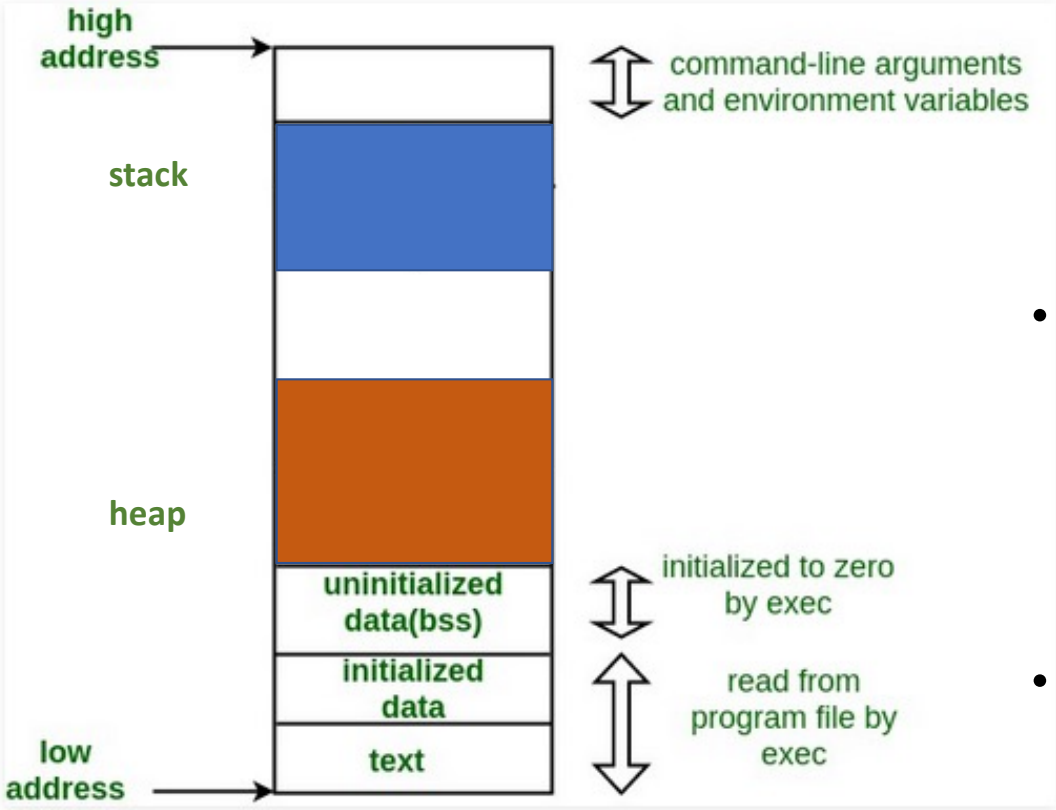


Process



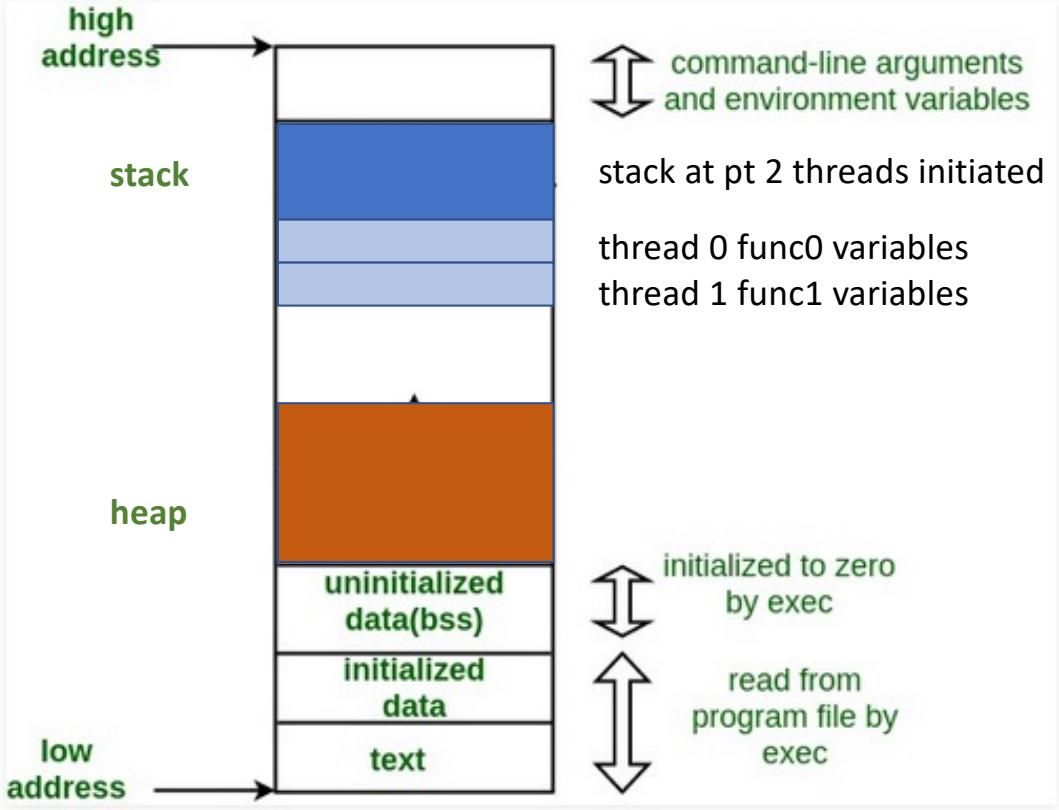
- An instance of a program execution.
- A process executes a program, you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- The execution context of a running program, i.e. resources associated with program, current state of memory, current instruction being executed, pc.

Process:



State of Memory at some point during program execution

- An instance of a program execution.
- A process executes a program, you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- The execution context of a running program, i.e. resources associated with program, current state of memory, current instruction being executed, pc.

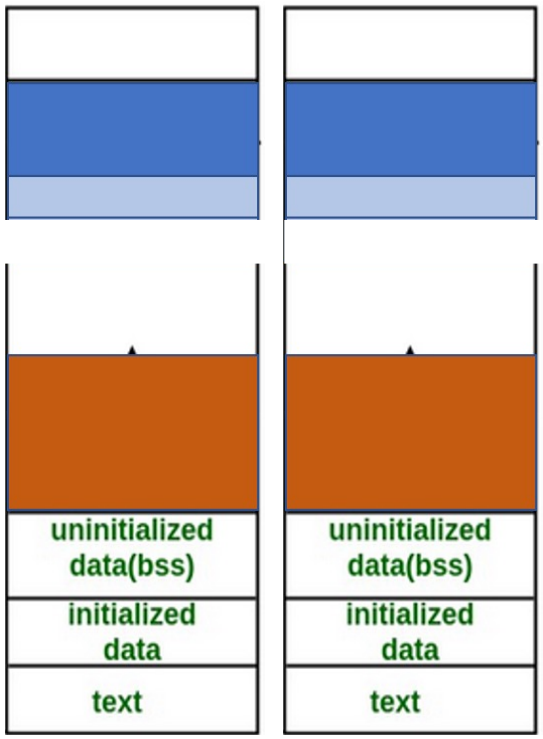


THREAD:

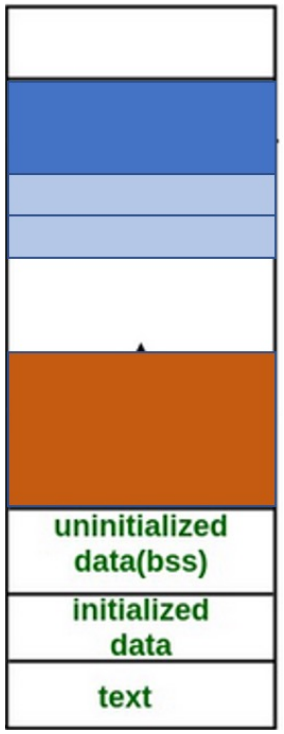
- A light weight process.
- Thread shares the Process state among multiple threads, has unique stack, shares contents of stack with other share heap.

State of Memory at point 2 threads are created (thread 0 in func0 with variables for func0, And thread1 in func1 with variables for func1. threads 0 and 1 share stack at point variables created, and share heap.

MPI v OpenMP

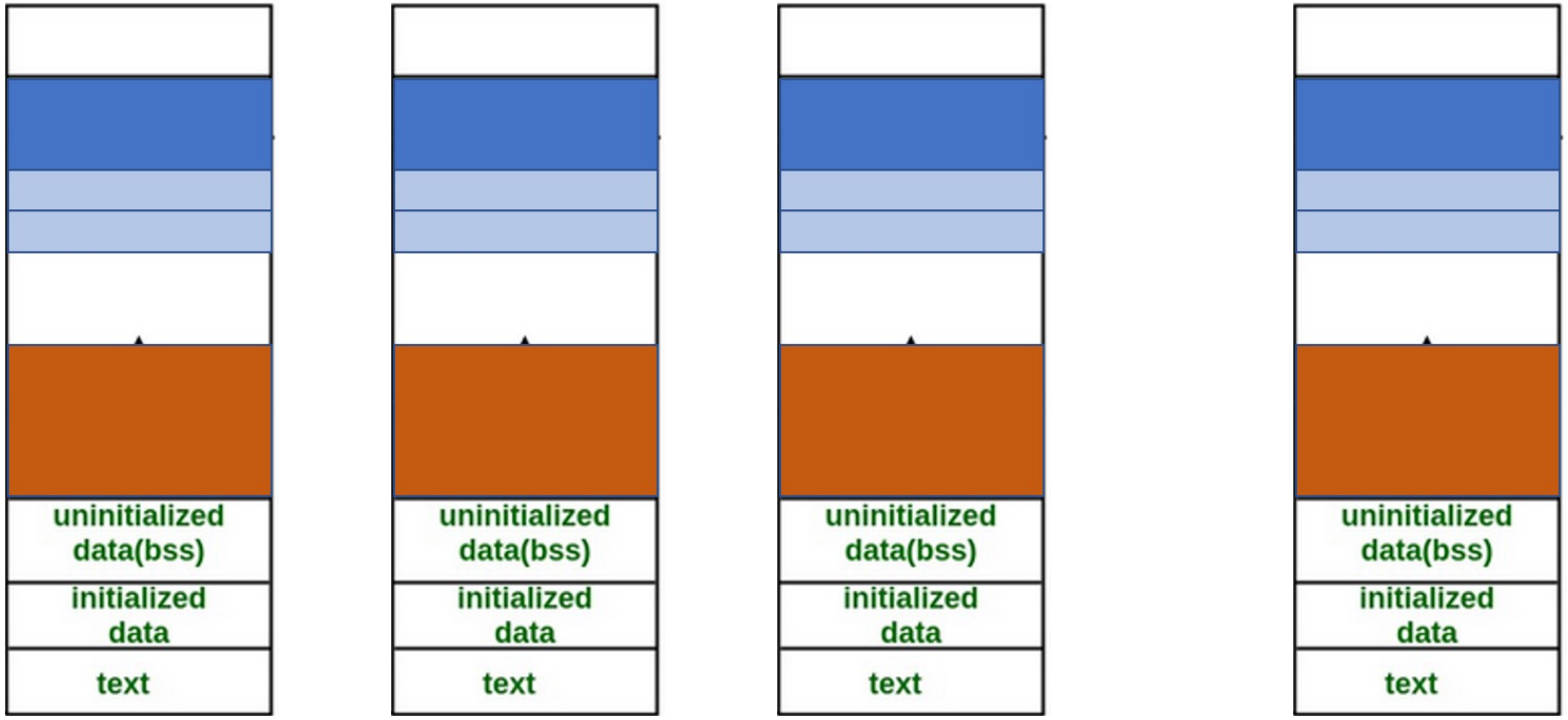


Multiple Processes



Multiple Threads
in a Single Processes

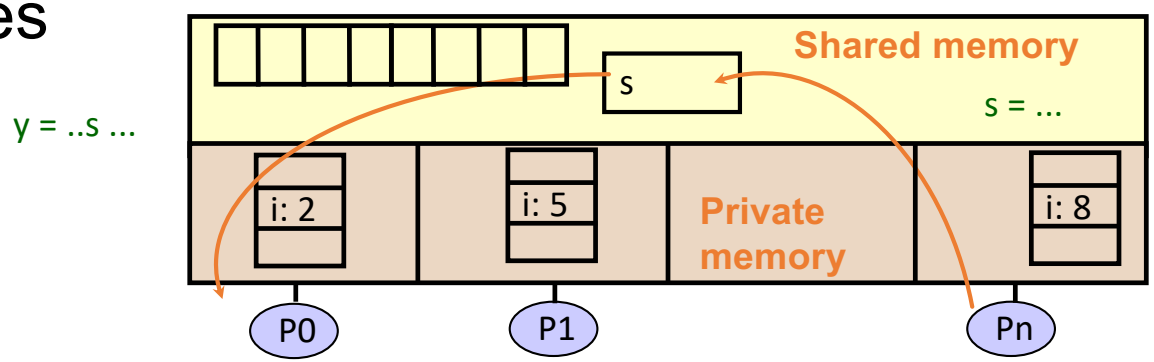
Hybrid – MPI + OpenMP



Multiple Processes each with Multiple Threads

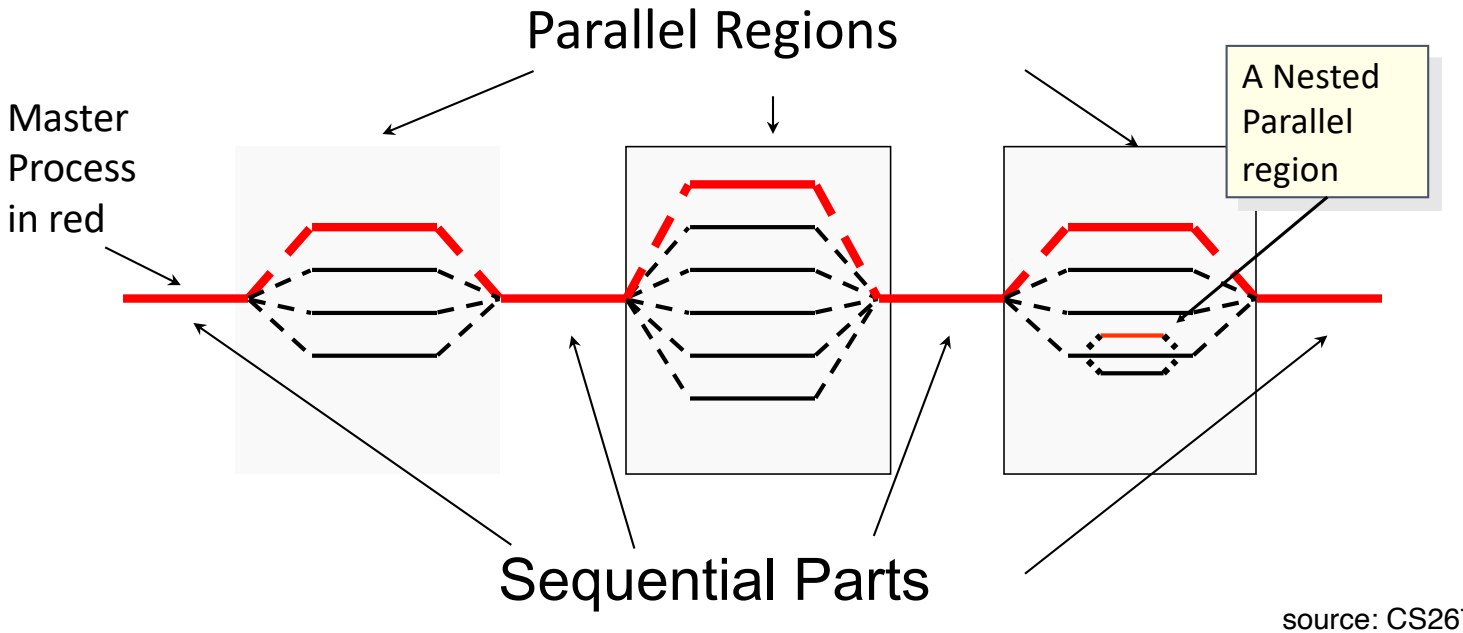
Threads

- Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables,** e.g., static variables, shared common blocks, or global heap.
- Threads communicate **implicitly by writing and reading shared variables.**
- Threads **coordinate by synchronizing** on shared variables



Programming for Threads

- Master Process spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



source: CS267, Jim Demmell

Runtime Library Options for Shared Memory

POSIX Threads (pthreads)

OpenMP

- OpenMP provides multi-threaded capabilities to C, C++ and Fortran Programs
- In a threaded environment, threads communicate by sharing data
- Unintended sharing of data causes **race conditions**
- **Race Condition**: program output is different every time you run the program, a consequence of the threads being scheduled differently
- OpenMP provides constructs to control what blocks of code are run in parallel and also constructs for providing access to shared data using synchronization
- Synchronization has overhead consequences, you have to minimize them to get good speedup.
- INTERFACE: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

- Mostly Set of Compiler directives (#pragma) applying to structured block

```
#pragma omp parallel
{

}
```

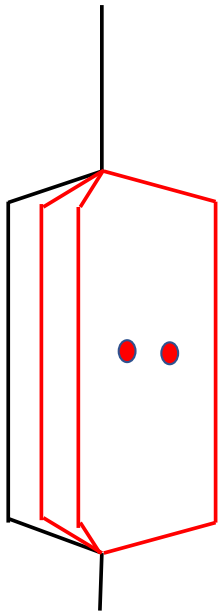
- Some runtime library calls

```
omp_num_threads(4);
```

- Being compiler directives, they are built into most compilers .
- Just have to activate it when compiling

```
gcc hello.c –fopenmp
icc hello.c  -qopenmp
```

Hello World



Code/Parallel/openmp/hello1.c

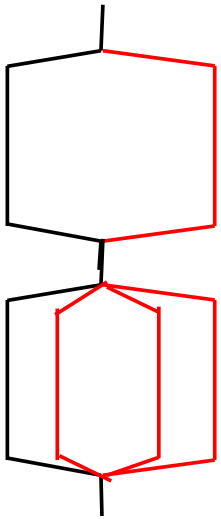
```
#include <omp.h>
#include <stdio.h>

int main( int argc, char *argv[])
{
  #pragma omp parallel
  {
     int id = omp_get_thread_num();
     int numP = omp_get_num_threads();
     printf("Hello World, I am  %d of %d\n",id,numP);
  }
  return 0;
}
```

```
openmp >gcc-7.2 hello1.c -fopenmp; ./a.out
Hello World, I am  0 of 4
Hello World, I am  3 of 4
Hello World, I am  1 of 4
Hello World, I am  2 of 4
openmp >
```

Each thread executes code within structured block

```
openmp >export env OMP_NUM_THREADS=2
openmp >./a.out
Hello World, I am  0 of 2
Hello World, I am  1 of 2
openmp >
```

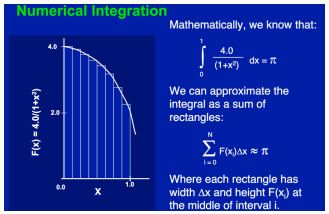
Different # threads in different blocks



Code/Parallel/openmp/hello4.c

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

int main(int argc, const char **argv) {

#pragma omp parallel num_threads(2)
  {
    int id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World, I am  %d of %d\n",id,numP);
  }
#pragma omp parallel num_threads(4)
  {
    int id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World Again, I am  %d of %d\n",id,numP);
  }
  return(0);
}
```

```
openmp >gcc-7.2 hello4.c -fopenmp; ./a.out
Hello World, I am  0 of 2
Hello World, I am  1 of 2
Hello World Again, I am  1 of 4
Hello World Again, I am  2 of 4
Hello World Again, I am  3 of 4
Hello World Again, I am  0 of 4
openmp >
```



icc –qopenmp pi.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 8
static long steps = 1000000000;
double step;
int main (int argc, const char *argv[])
  int i,j;
  double x;
  double pi, sum = 0.0;
  double start, delta;

  step = 1.0/(double) steps;

  // This is the beginning of a single PI computation
  // Compute parallel compute times for 1-MAX_THREADS
  for (j=1; j<= MAX_THREADS; j++) {

    printf(" running on %d threads: ", j);
    omp_set_num_threads(j);
      sum = 0.0;
    double start = omp_get_wtime();
#pragma omp parallel for reduction(+:sum) private(x)
    for (i=0; i < steps; i++) {
      x = (i+0.5)*step;
      sum += 4.0 / (1.0+x*x);
    }
   // Out of the parallel region, finialize computation
    pi = step * sum;
    delta = omp_get_wtime() - start;
    printf("PI = %.16g computed in %.4g seconds\n", pi, delta);
  }
```

```
login3.frontera(1025)$ ./a.out
 running on 1 threads: PI = 3.141592653589768 computed in 0.5752 se
 running on 2 threads: PI = 3.141592653589738 computed in 0.2843 se
 running on 3 threads: PI = 3.141592653589709 computed in 0.1908 se
 running on 4 threads: PI = 3.141592653589854 computed in 0.143 sec
 running on 5 threads: PI = 3.141592653589924 computed in 0.1143 se
 running on 6 threads: PI = 3.141592653589787 computed in 0.09513 s
 running on 7 threads: PI = 3.1415926535898 computed in 0.08192 sec
 running on 8 threads: PI = 3.141592653589883 computed in 0.0715 se
login3.frontera(1026)$
```