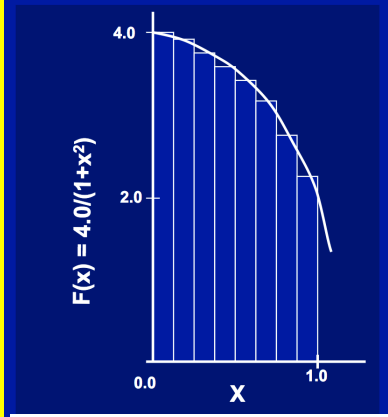
Exercise: Compute PI (The hello world of parallel programming)

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at

1. ssh yourName@frontera.tacc.utexas.edu
2. cd SimCenterBootcamp2020
3. git add .
4. git commit –m “yourname – updating my latest code”
5. git remote –v
6. git remote add upstream https://github.com/NHERI-SimCenter/SimCenterBootcamp2020.git
7. git fetch upstream/master
8. git merge

code/c/pi2.c

```
#include <stdio>
static int long numSteps = 100000;

int main() {
  double pi   = 0;
  double stepSize = 1.0/(double) numSteps;

  for (int i=0; i<numSteps; i++) {
      double x = (i+0.5)*stepSize;
       pi += 4./(1.+x*x);
  }

  pi *= stepSize;

  printf("PI = %16.14f\n",pi);
  return 0;
}
```

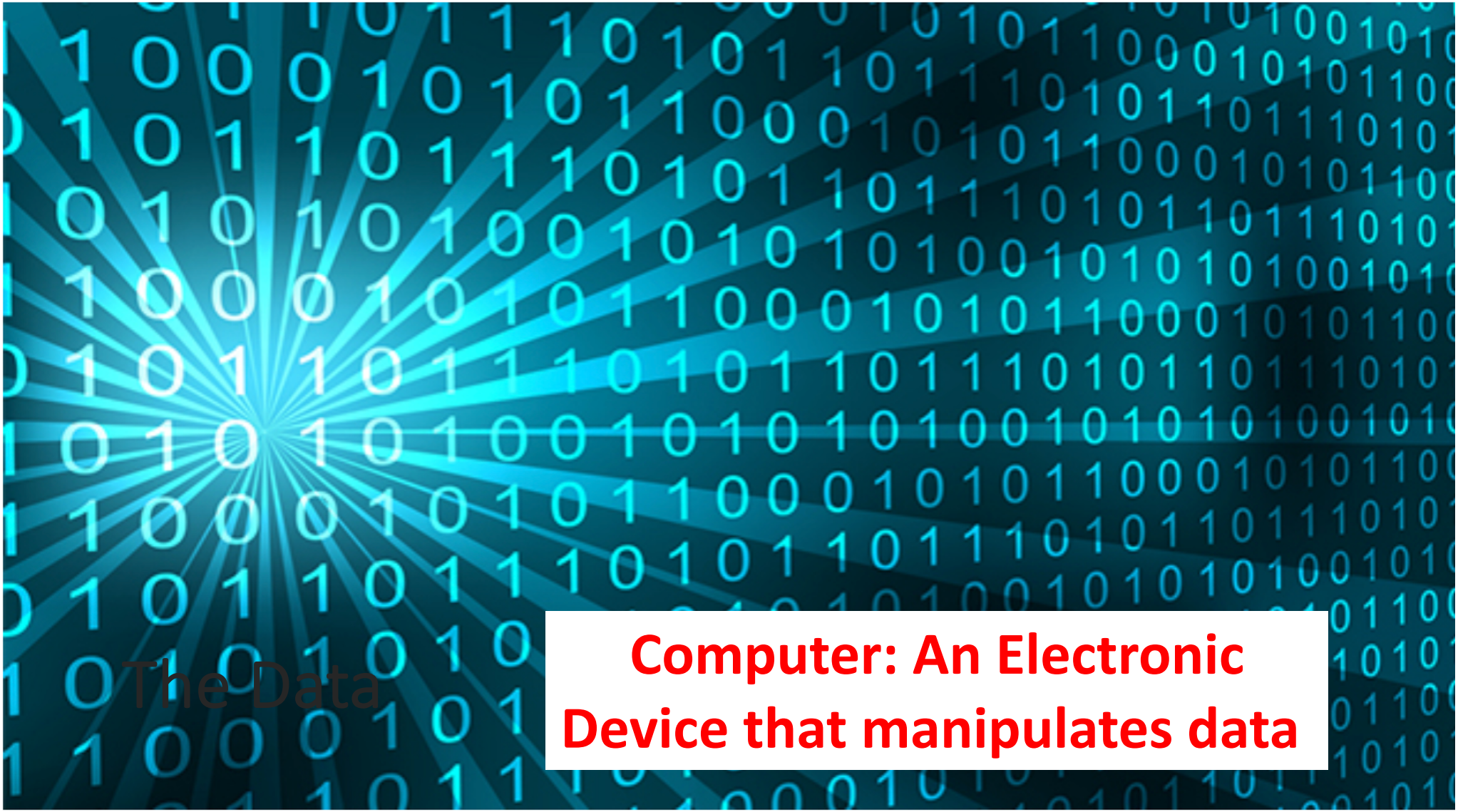
SimCenter NHERI
Center for Computational Modeling and Simulation

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

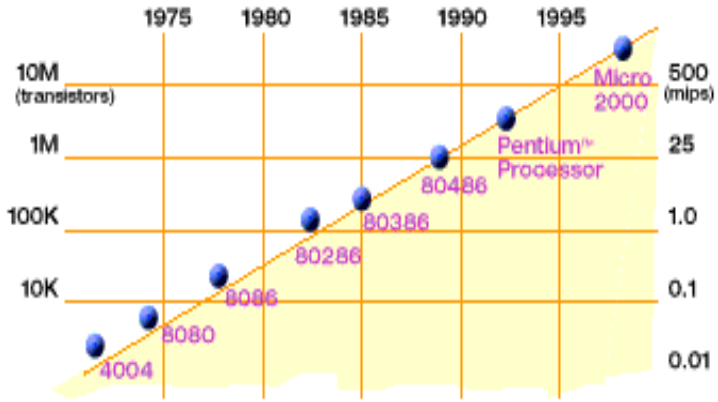
Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that 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] 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

FRIDAY

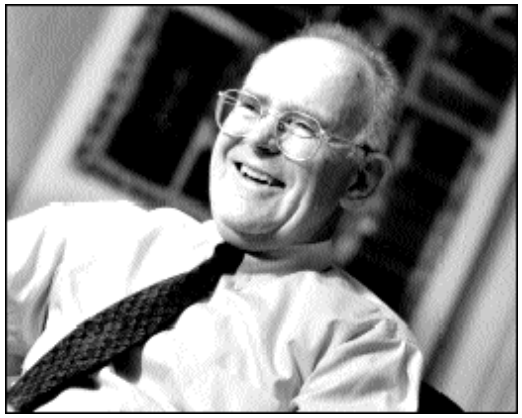
TODAY

Why is Parallel Programming Important



2X transistors/Chip Every 1.5 years
Called "Moore's Law"

Microprocessors have become smaller, denser, and more powerful.

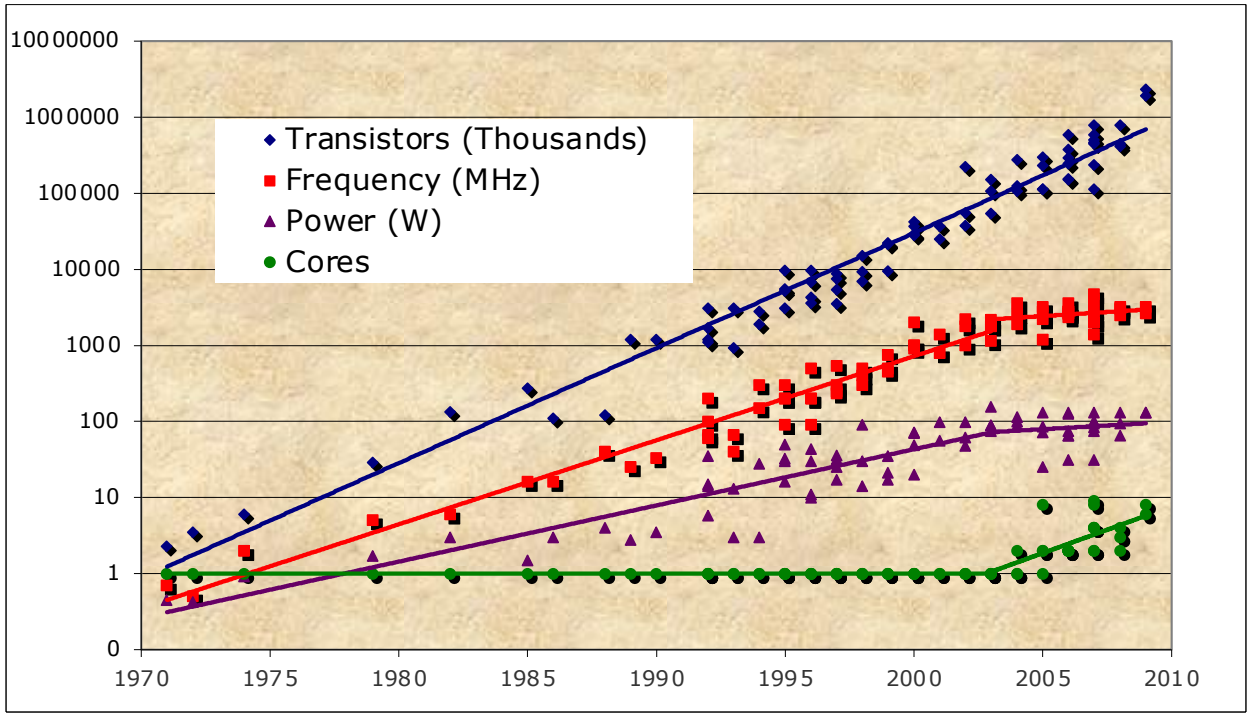


Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Slide source: Jack Dongarra

source: CS267, Jim Demmell

Revolution in Processors



- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
- Number of processor cores may double instead
- Power is under control, no longer growing

source: CS267, Jim Demmell

How many cores?

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

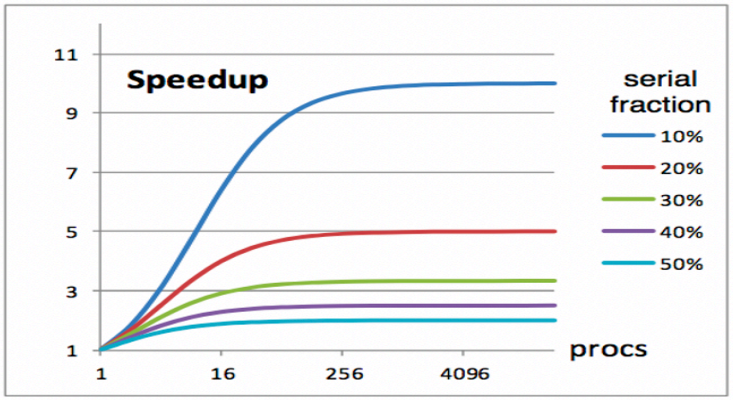
Can All Programs Be Made to Run Faster?

- 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

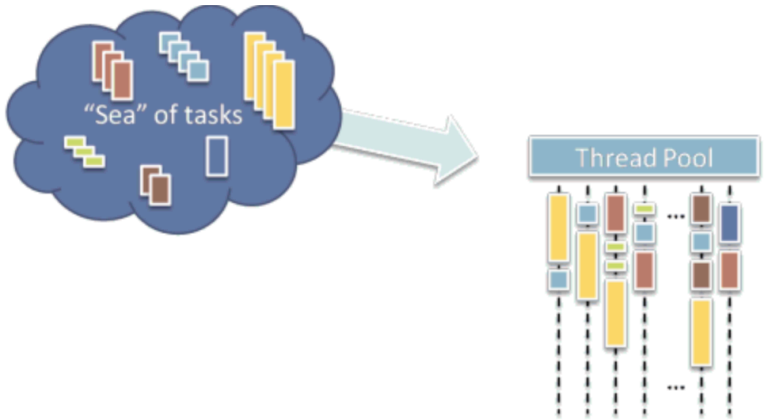
This Does not Take into Account Overhead of Parallelism

- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation
- Each of these can be in the range of milliseconds (=millions of flops) on some systems
- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work

source: CS267, Jim Demmell

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.

Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to
  - insufficient parallelism (during that phase)
  - unequal size tasks
- Examples of the latter
  - adapting to “interesting parts of a domain”
  - tree-structured computations
  - fundamentally unstructured problems
- Algorithm needs to balance load
  - Sometimes can determine work load, divide up evenly, before starting
    - “Static Load Balancing”
  - Sometimes work load changes dynamically, need to rebalance dynamically
    - “Dynamic Load Balancing,” eg work-stealing

Improving Real Performance

Peak Performance grows exponentially, a la Moore's Law

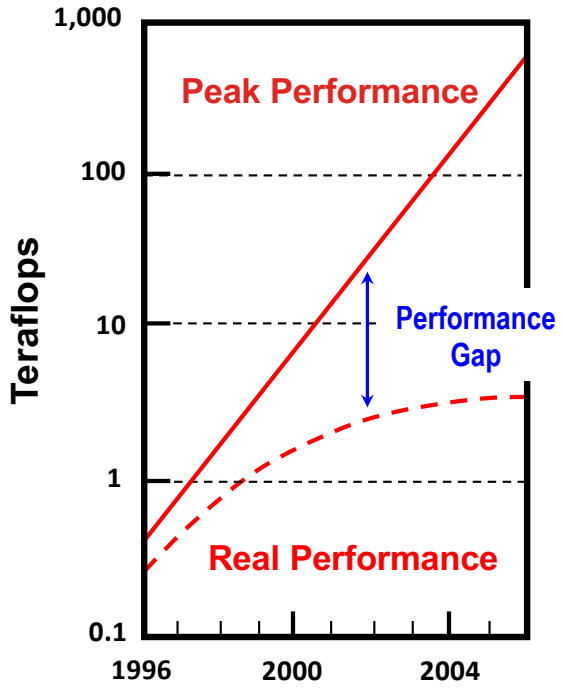
- In 1990's, peak performance increased 100x; in 2000's, it will increase 1000x

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

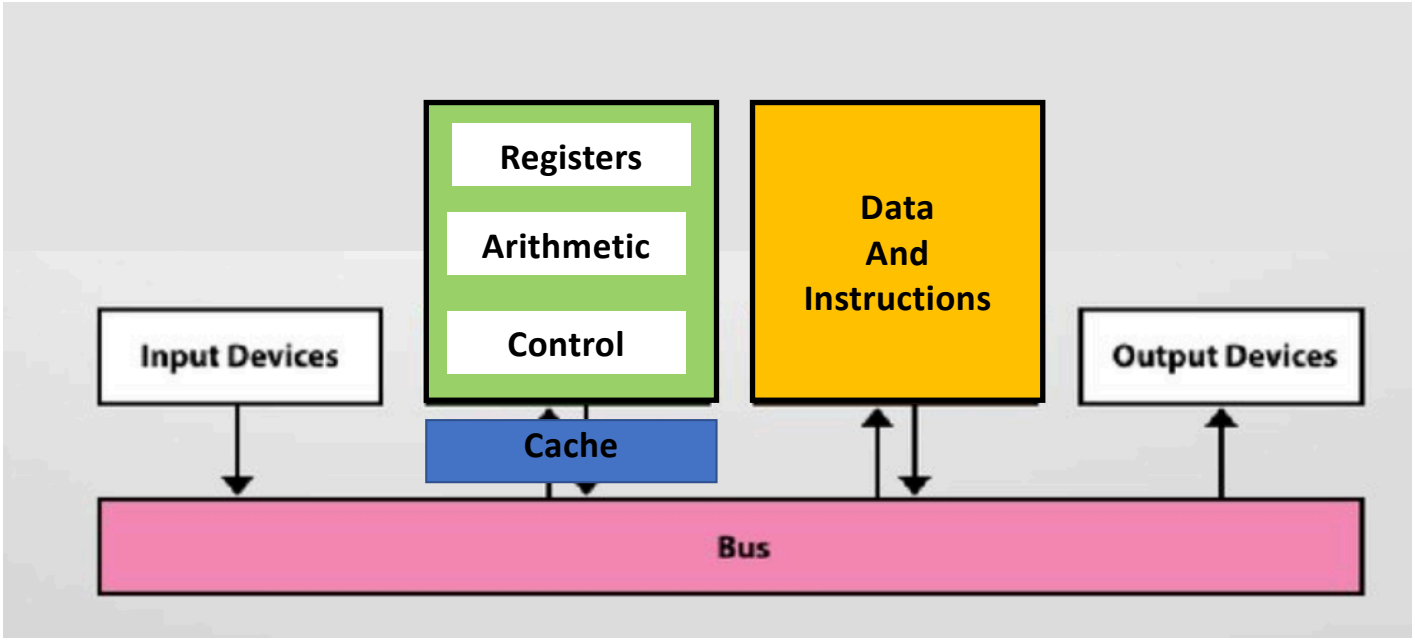
- was 40-50% on the vector supercomputers of 1990s
- now as little as 5-10% on parallel supercomputers of today

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

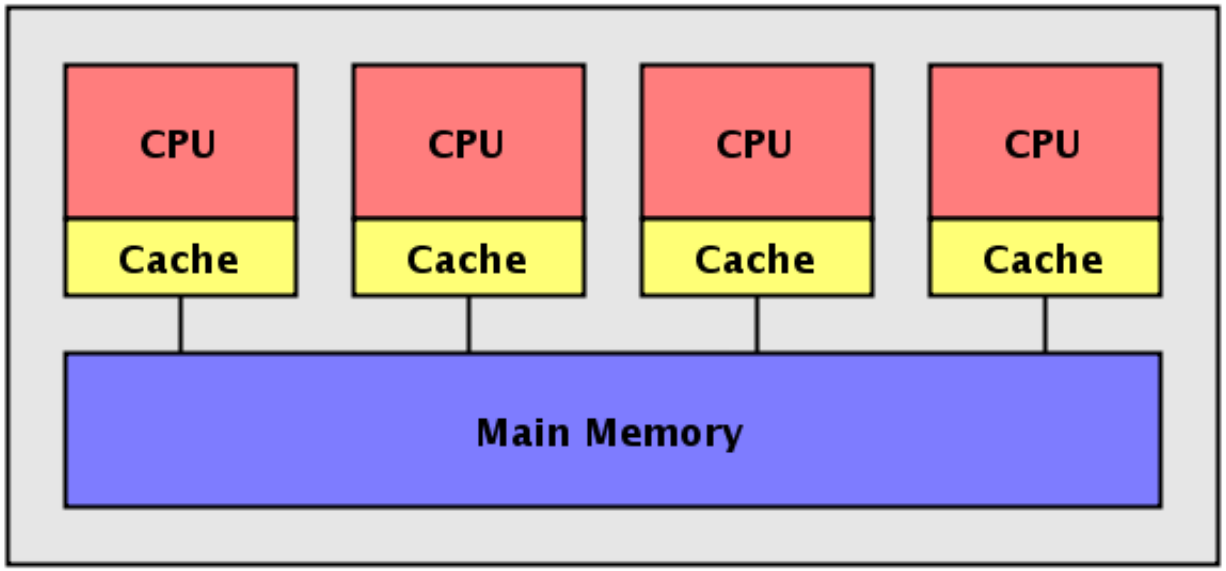


For Sequential Programming idealized view of Computer



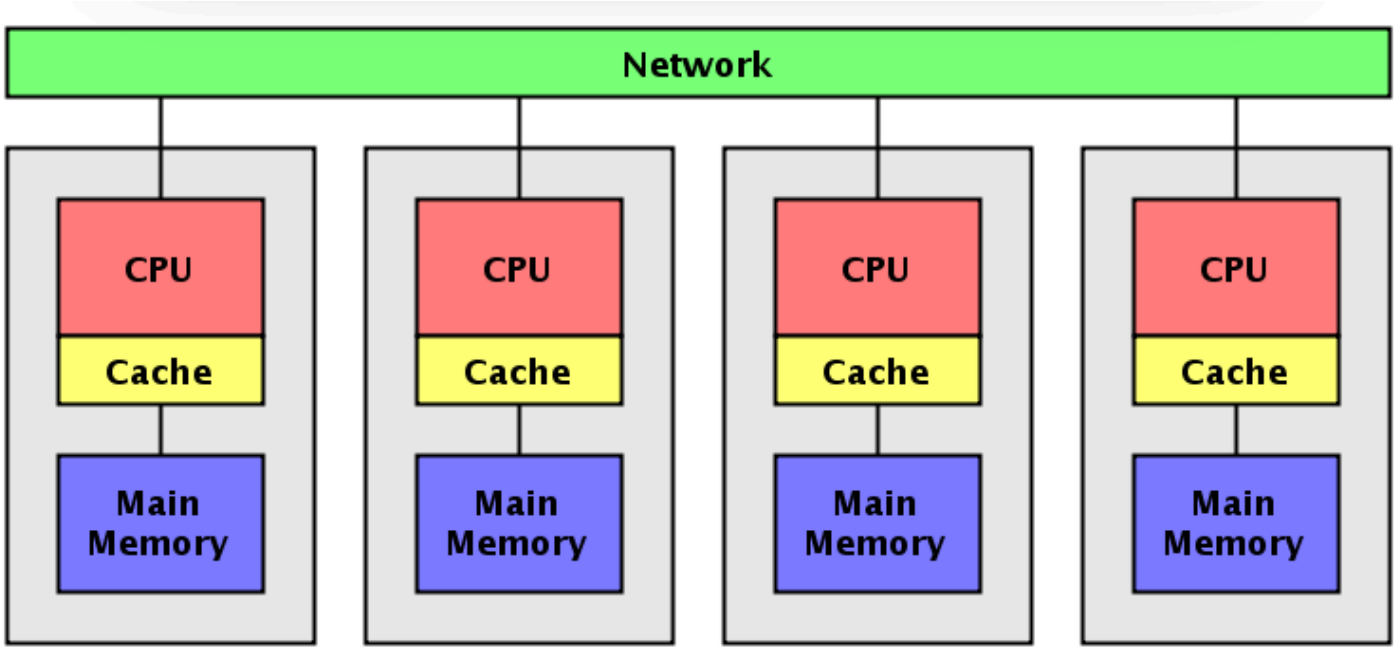
Simplified Model of Parallel Machine

#1: Shared Memory Model



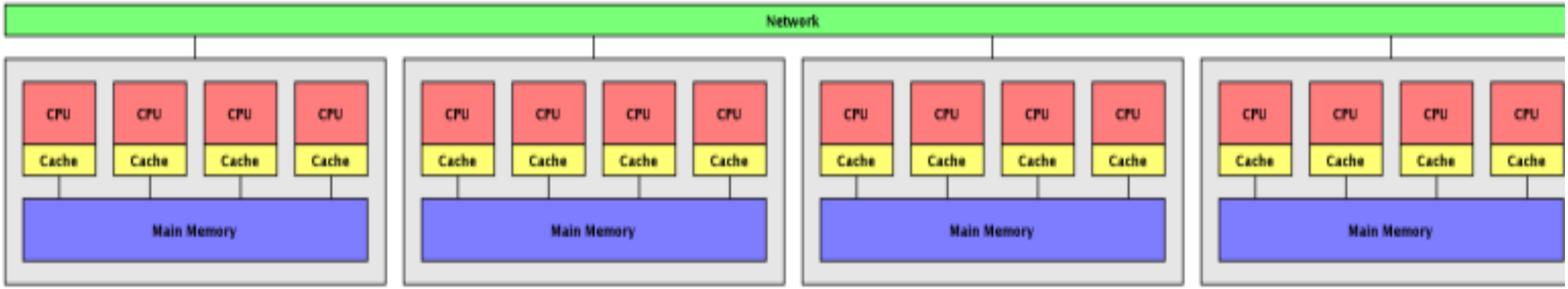
Simplified Model of Parallel Machine

#2: Distributed Memory Model

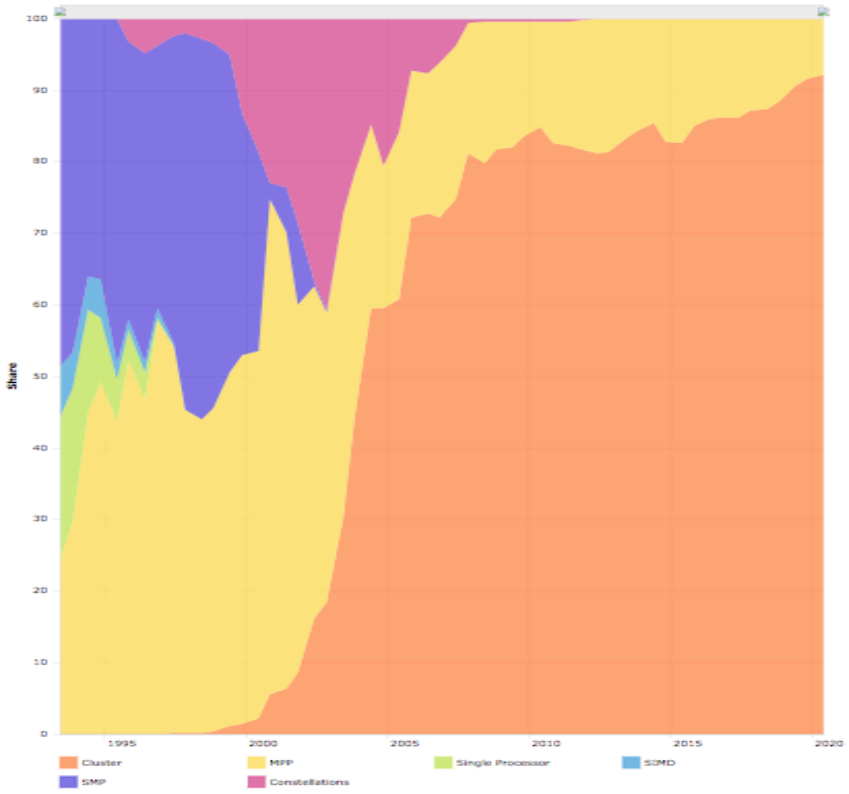


Simplified Model of Parallel Machine

#3: Hybrid Model



From Vector Supercomputers to Cluster Systems (many with GPU) - parallel computers on which we run change over time



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.

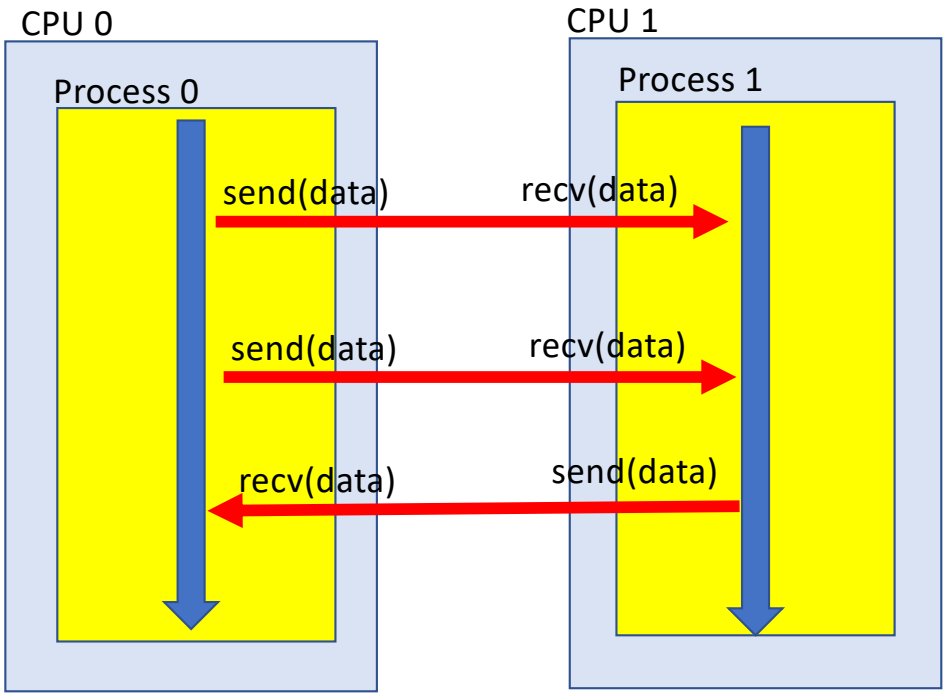
- Parallel Machines & Parallel Machine Models
- Parallel Programming
  - Message Passing With MPI
  - Shared Memory Programming with OpenMP

Ignoring co-processors and GPUs

many slides source: CS267, Jim Demmell

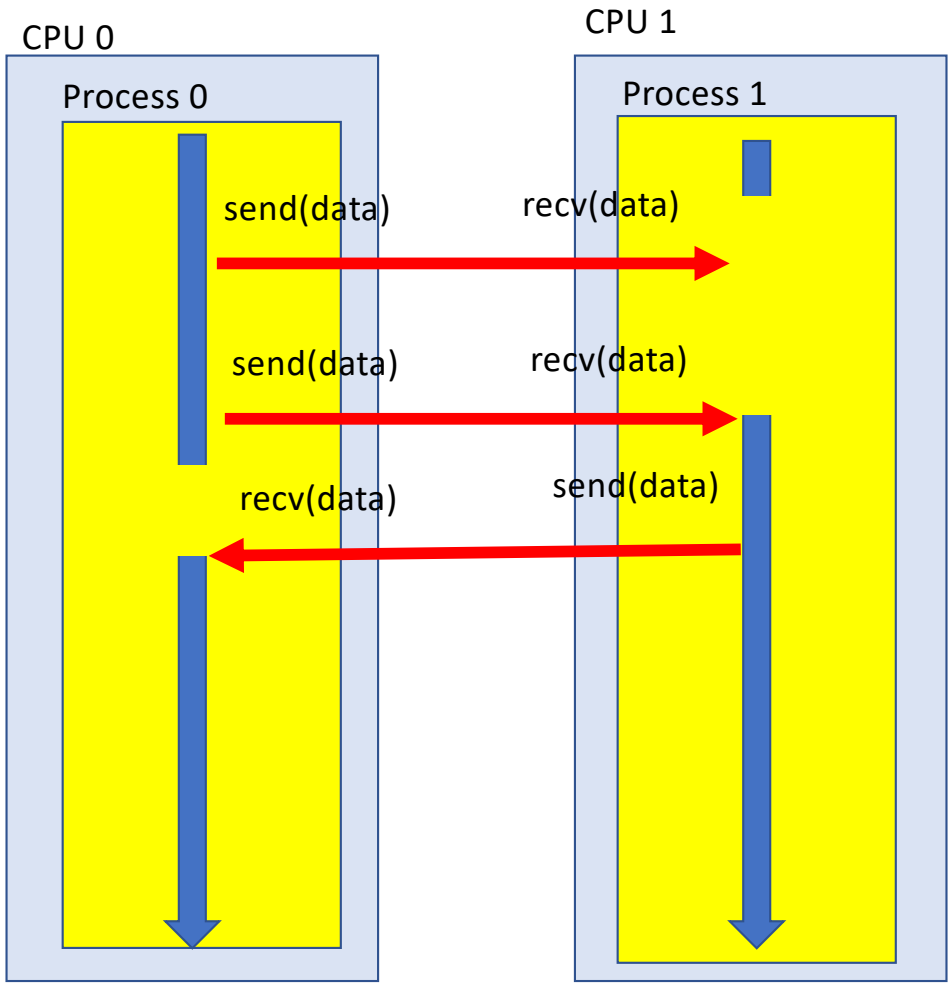
Message Passing Model

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

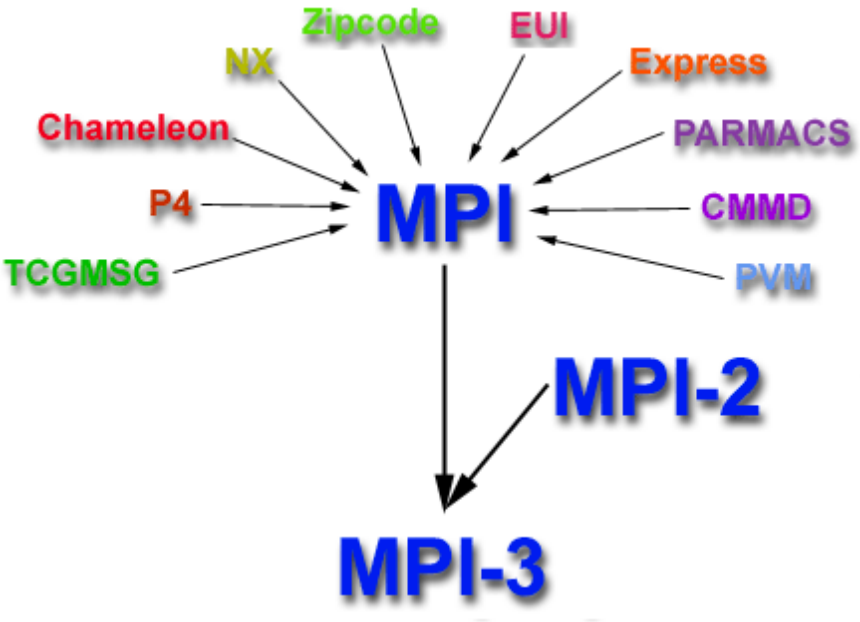


- Basically you write sequential applications with additional function calls to send and recv data.

Only Get Speedup if processes can be kept busy



Programming Libraries



- Coalesced around a single standard MPI
- Allows for portable code

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

Don’t freak out - Remember:

What I am about to show is just C code with some functions you have not yet seen.

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

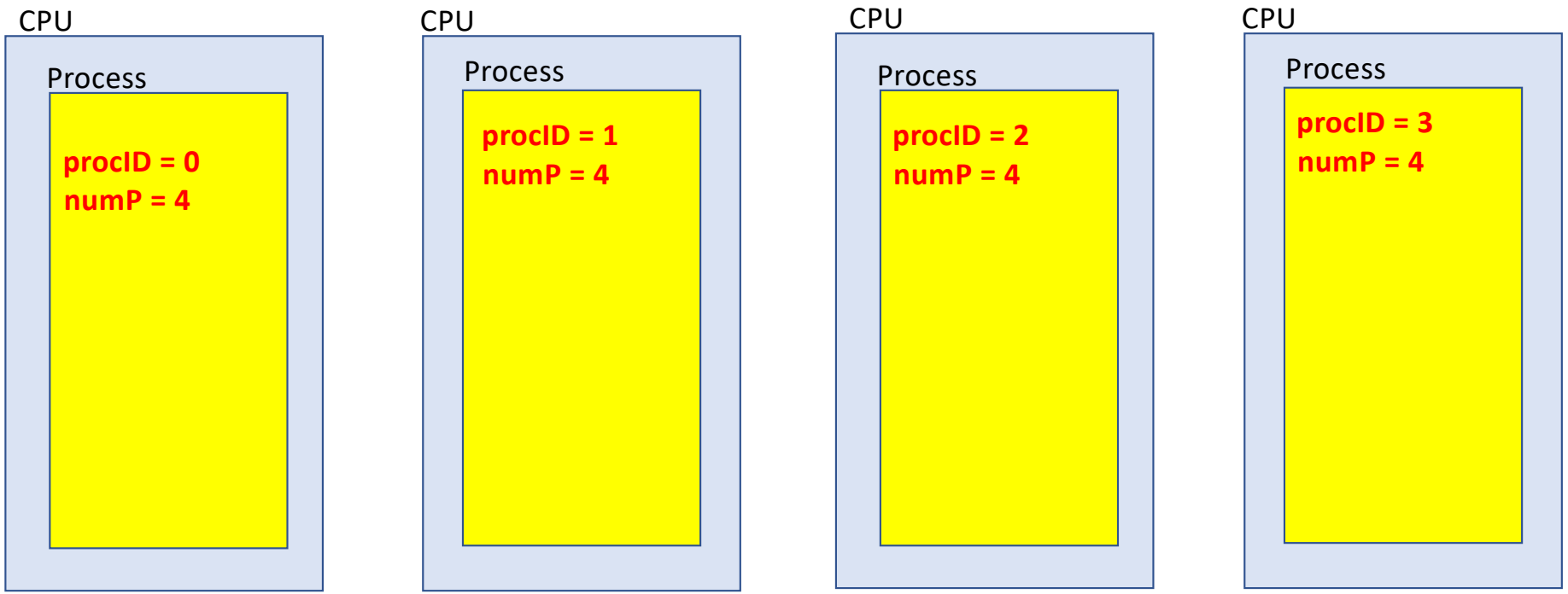
MPI_Init() must be first function called

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)

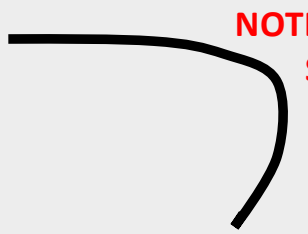
MPI_finalize() must be last function called


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

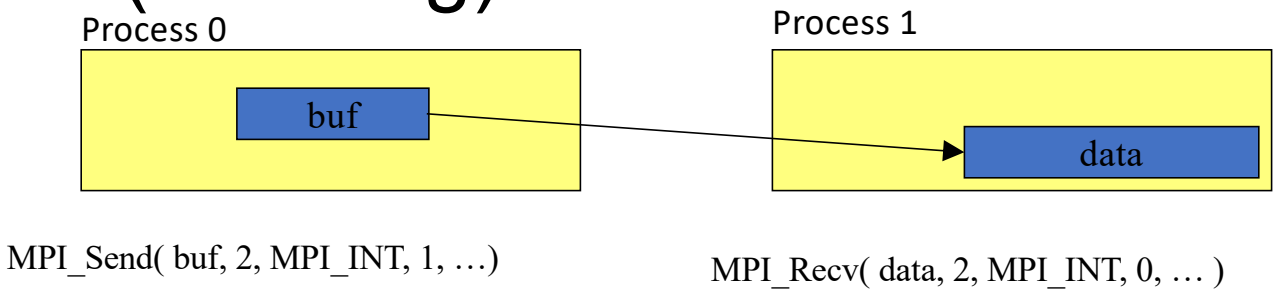
Send/Recv blocking

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;
}
```



MPI Basic (Blocking) Send

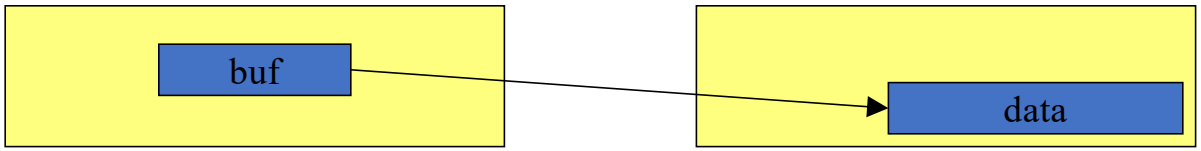


`MPI_SEND(start, count, datatype, dest, tag, comm)`

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**. The message has an identifier **tag.**
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI_CHAR	char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT **MPI_LONG_LONG**	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_COMPLEX **MPI_C_FLOAT_COMPLEX**	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_C_BOOL	_Bool
MPI_INT8_T **MPI_INT16_T** **MPI_INT32_T** **MPI_INT64_T**	int8_t int16_t int32_t int64_t
MPI_UINT8_T **MPI_UINT16_T** **MPI_UINT32_T** **MPI_UINT64_T**	uint8_t uint16_t uint32_t uint64_t
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

MPI Basic (Blocking) Receive



MPI_Send(buf, 2, MPI_INT, 1, …)

MPI_Recv(data, 2, MPI_INT, 0, …)

`MPI_RECV(start, count, datatype, source, tag, comm, status)`

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system, and the buffer can be used
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C:

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

Not Quite as Simple as ensuring PAIRS of send/recv

code/Parallel/mpi/send2.c

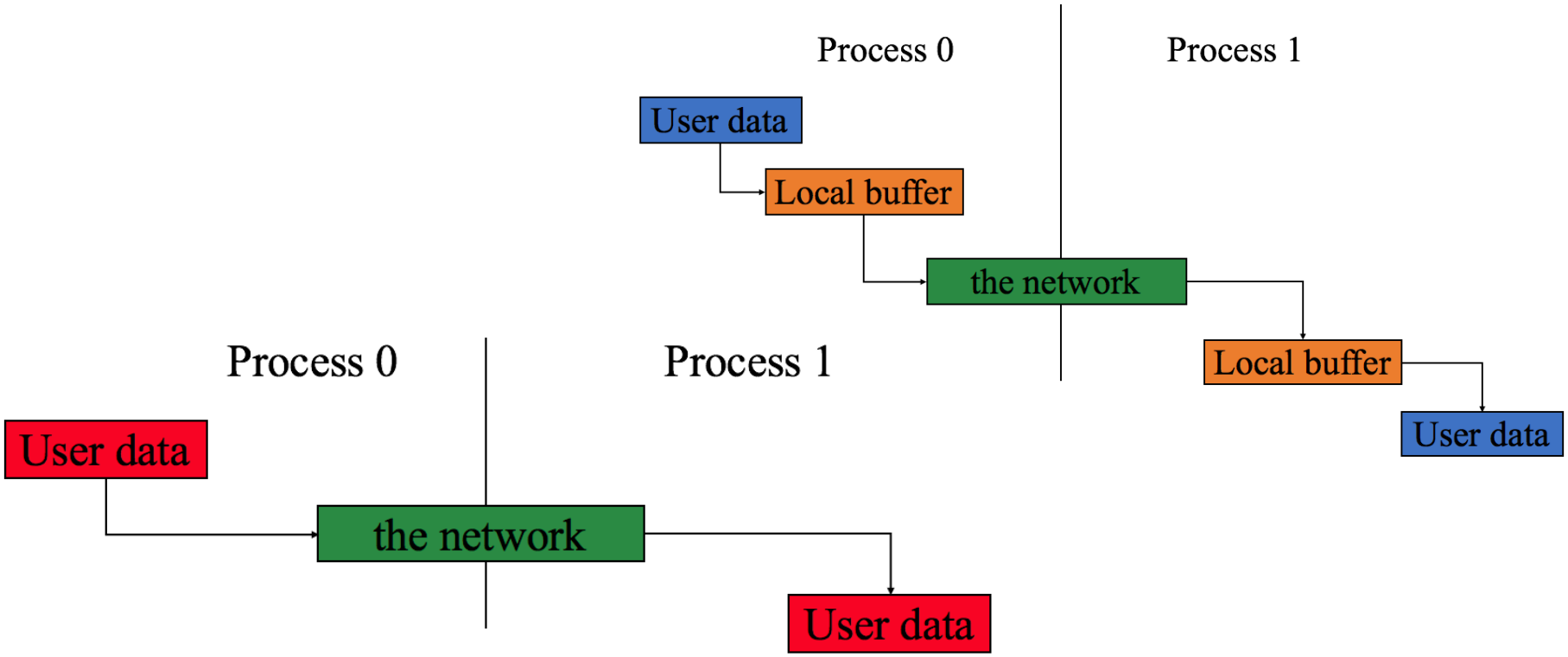
```
#include <mpi.h>
#include <stdio.h>
#define DATA_SIZE 1000
int main(int argc, char **argv) {
   int procID, numP;
   MPI_Status status;
   int buf[DATA_SIZE];

   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)`

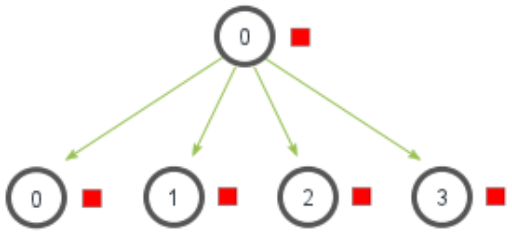
Could revise order
this requires some code rewrite:

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

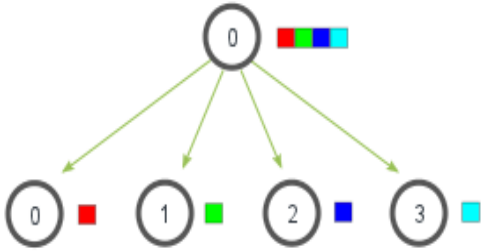
Alternatives use non-blocking sends.

Some Collective Functions

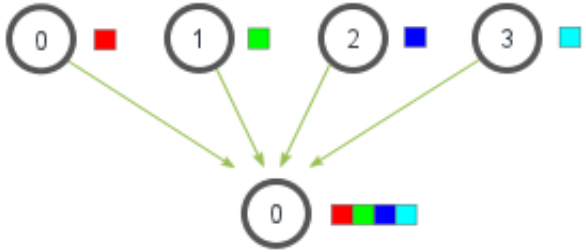
MPI_Bcast



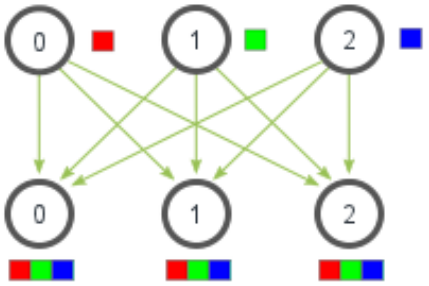
MPI_Scatter



MPI_Gather



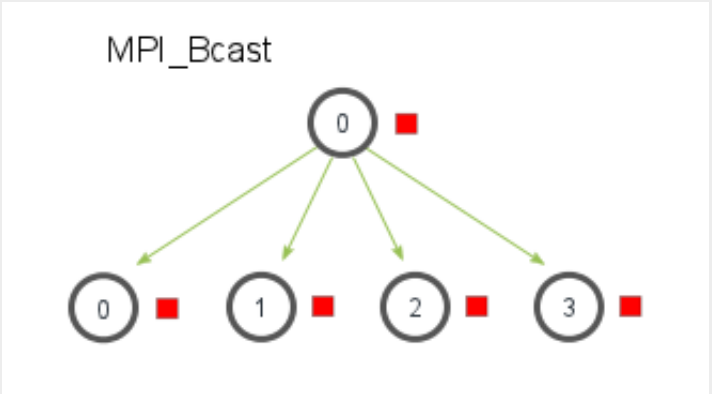
MPI_Allgather



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;
}
```

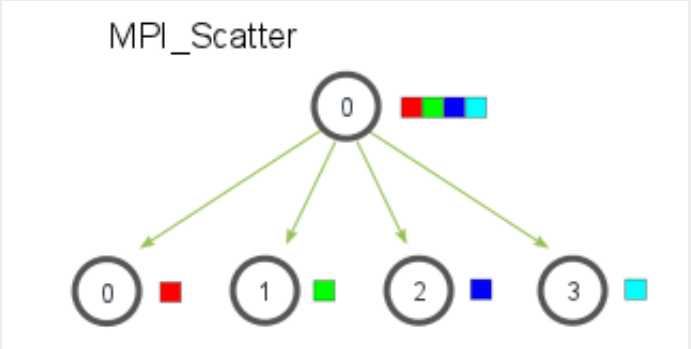


int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Scatter

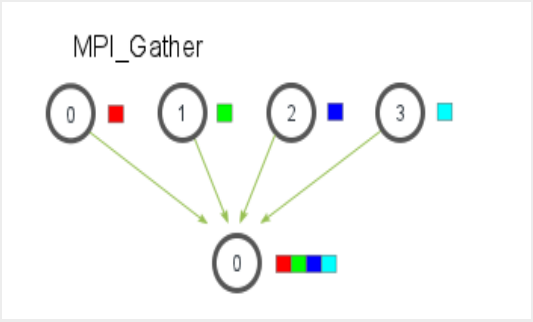
code/Parallel/mpi/scatter1.c

```
#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

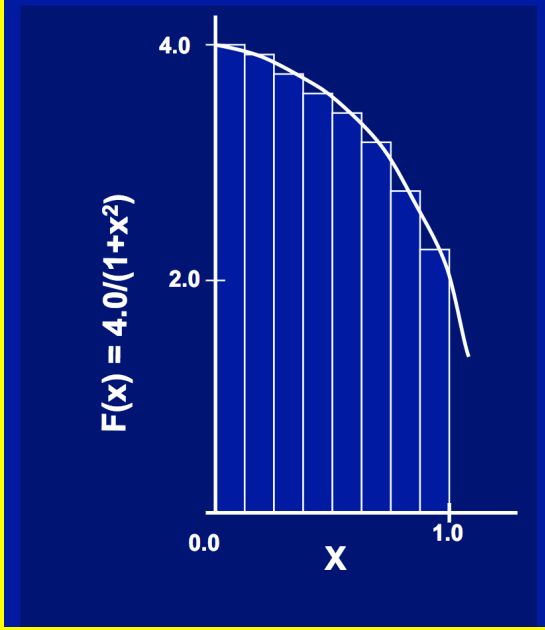
- Claim: most MPI applications can be written with only 6 functions (although which 6 may differ)
- Using point-to-point:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECEIVE`
- Using collectives:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`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

DEMO: - compiling & running hello world

1. ssh yourName@frontera.tacc.utexas.edu
2. idev
3. cp SimCenterBootcamp2020/code/parallel/mpi
4. mpicc hello1.c
5. ibrun –n 4 a.out

Exercise: Parallelize Compute PI using MPI

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

code/c/pi2.c

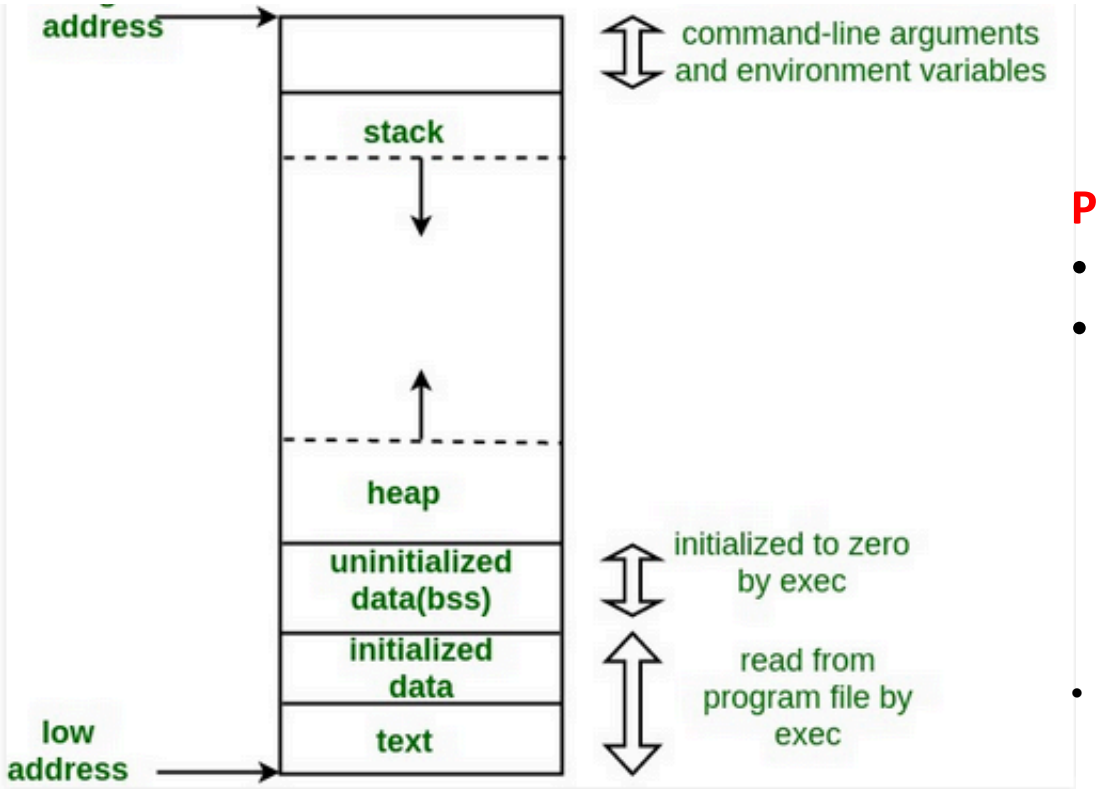
```
#include <stdio>
static int long numSteps = 100000;

int main() {
  double pi   = 0;
  double stepSize = 1.0/(double) numSteps;

  for (int i=0; i<numSteps; i++) {
      double x = (i+0.5)*stepSize;
       pi += 4./(1.+x*x);
  }

  pi *= stepSize;

  printf("PI = %16.14f\n",pi);
  return 0;
}
```

Outline

- Parallel Machines & Parallel Machine Models
- Parallel Programming
  - Message Passing With MPI
  - Shared Memory Programming with OpenMP

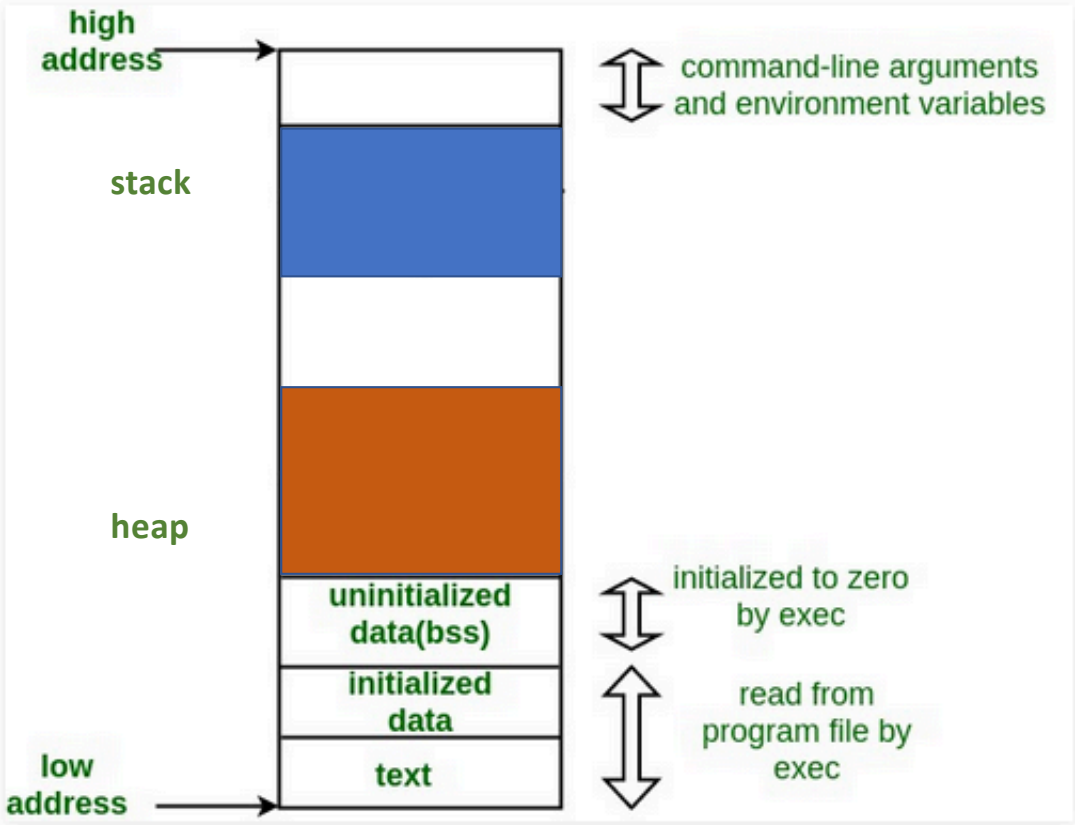
Ignoring co-processors and GPUs

many slides source: CS267, Jim Demmell



PROCESS:

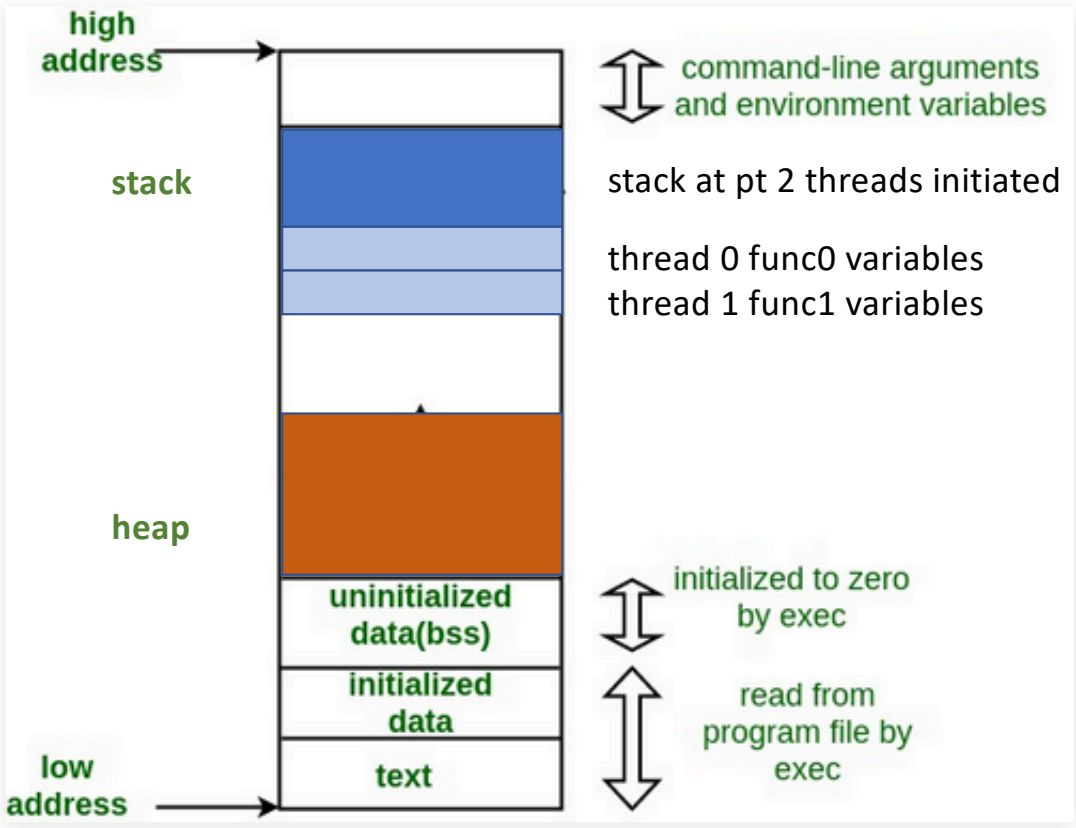
- An instance of a program execution.
- A process executes a program, you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- you can have multiple processes executing 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.



State of Memory at some point during program execution

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.



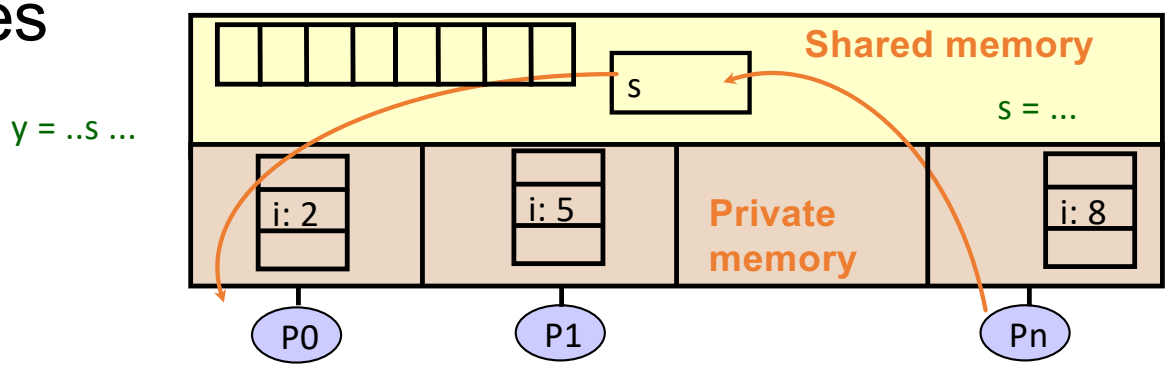
THREAD:

- 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.

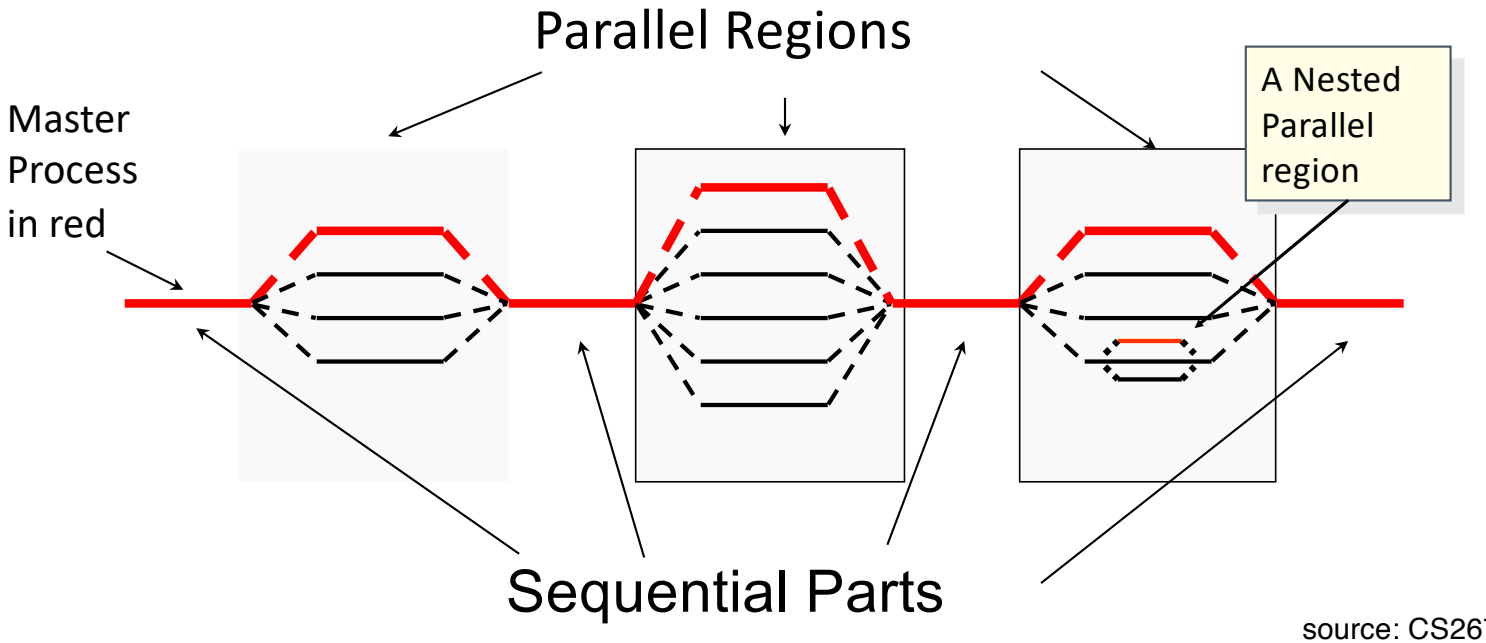
Threads

- 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

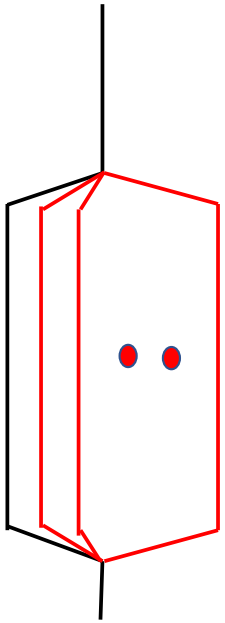
```
omp_num_threads(4);
```

- 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



```
#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;
}
```

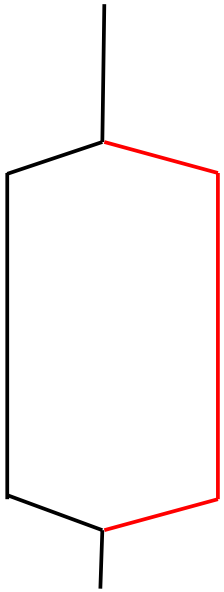
Code/Parallel/openmp/hello1.c

```
openmp >gcc-7.2 hello1.c -fopenmp;
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 >
```

Hello World – changing num threads



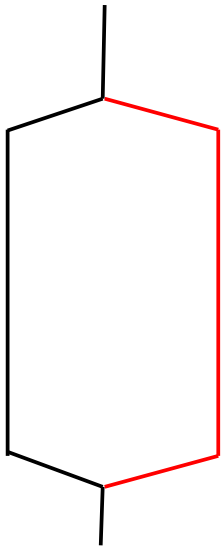
```
#include <openmp.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
  omp_set_num_threads(2);
  #pragma omp parallel
   {
        int id = omp_get_thread_num();
        int numP = omp_get_num_threads();
        printf("Hello World, I am  %d of
   }
  return 0;
}
```

Code/Parallel/openmp/hello2.c

Actually an upper limit ..
You might not get all requesetd

Runtime function to request a certain number of threads

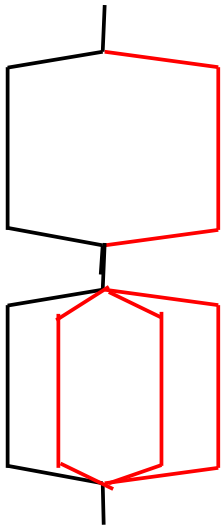
Runtime function to return actual number of threads in the team



Code/Parallel/openmp/hello3.c

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

int main( int argc, 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);
   }
  return 0;
}
```

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 >
```

For Those Who Need to Know Why Stack

- It's Implementation under the hood – **THUNKS**!

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

=

```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
   pthread_create (
        &tid[i],0,thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

RACE CONDITIONS (can get different answers)
a consequence of variable sharing

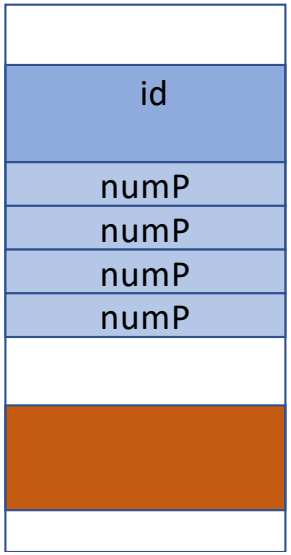
Code/Parallel/openmp/hello5.c

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

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

 int id;

#pragma omp parallel num_threads(4)
  {
    id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World from %d of %d th
  }

  return(0);
}
```

```
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 1 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
Hello World from 0 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 3 of 4 threads
Hello World from 2 of 4 threads
Hello World from 1 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 0 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
```

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

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

 int id;

#pragma omp parallel num_threads(4)
  {
    id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World from %d of %d threads\n",id,numP);
  }

  return(0);
}
```

```
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 1 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
Hello World from 0 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 3 of 4 threads
Hello World from 2 of 4 threads
Hello World from 1 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 0 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
```



Threads 0-4 share id

thread 0 has private var numP
thread 1 has private var numP
thread 2 has private var numP
thread 3 has private var numP

Between time thread 1 sets and prints the variable id, Process 0 has come in and changed it's value

Simple Vector Sum

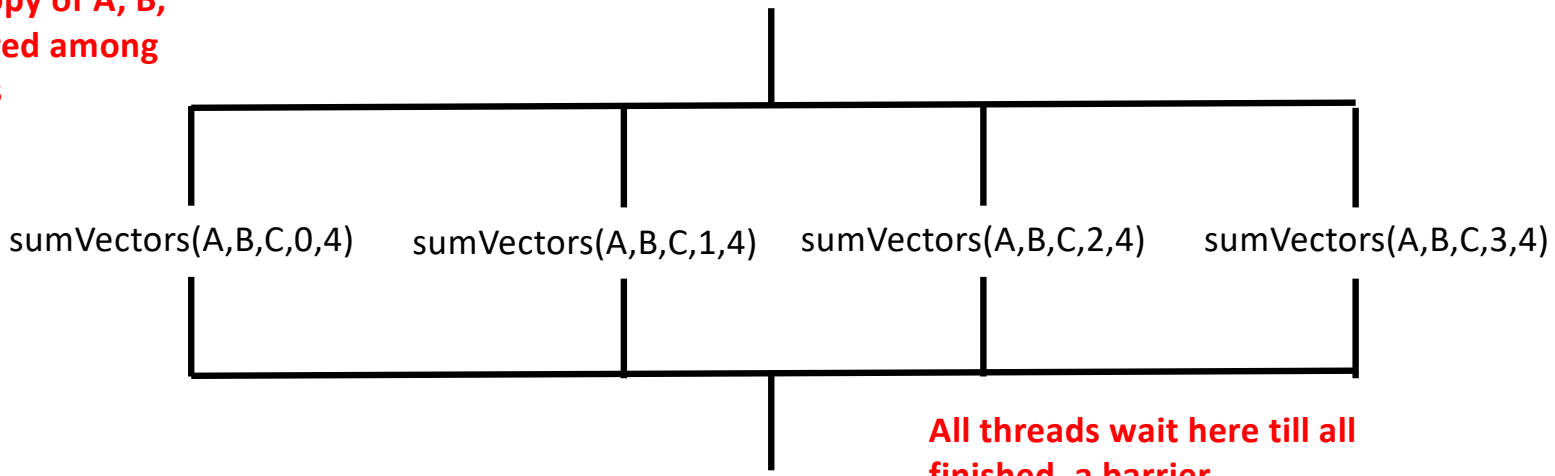
Code/Parallel/openmp/sum1.c

```
#include <omp.h>
#include <stdio.h>
#define DATA_SIZE 10000

void sumVectors(int N, double *A, double *B, double *C, int tid, int numT);
int main(int argc, const char **argv) {
  double a[DATA_SIZE], b[DATA_SIZE], c[DATA_SIZE];
  int num;
  for (int i=0; i<DATA_SIZE; i++) { a[i] = i+1; b[i] = i-
  double tdata = omp_get_wtime();
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    num = numT;
    sumVectors(DATA_SIZE, a, b, c, tid, numT);
  }
  tdata = omp_get_wtime() - tdata;
  printf("first %f last %f in time %f using %d threa
  return 0;
}
```

```
void sumVectors(int N, double *A, double *B, double *C, int tid, int numT) {
  // determine start & end for each thread
  int start =   tid * N / numT;
  int end = (tid+1) * N / numT;
  if (tid == numT-1)
    end = N;


  // do the vector sum for threads bounds
  for(int i=start; i<end; i++) {
    C[i] = A[i]+B[i];
  }
}
```

Implicit Barrier in Code

A single copy of A, B, and C shared among all threads




```
openmp >export env OMP_NUM_THREADS=1; ./a.out
first 2.000000 last 200000.000000 in time 0.000902 using 1 threads
openmp >export env OMP_NUM_THREADS=2; ./a.out
first 2.000000 last 200000.000000 in time 0.000678 using 2 threads
openmp >export env OMP_NUM_THREADS=4; ./a.out
first 2.000000 last 200000.000000 in time 0.000652 using 4 threads
openmp >export env OMP_NUM_THREADS=8; ./a.out
first 2.000000 last 200000.000000 in time 0.000693 using 8 threads
```

The for is such an obvious candidate for threads:

code/Parallel/openmp/sum2.c

```
#include <omp.h>
#include <stdio.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double a[DATA_SIZE], b[DATA_SIZE], c[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) { a[i] = i+1; b[i] = i+1; }
  double tdata = omp_get_wtime();
#pragma omp parallel
  {
   #pragma omp for
   for (int i=0; i<DATA_SIZE; i++)
      c[i] = a[i]+b[i];
  }
  tdata = omp_get_wtime() - tdata;
  printf("first %f last %f in time %f \n",c[0], c[DATA_SIZE-1], tdata);
  return 0;
}
```

Code/Parallel/openmp/sum3.c

```
#pragma omp parallel for
    for (int i=0; i<DATA_SIZE; i++)
     c[i] = a[i]+b[i];
```

How About Dot Product?

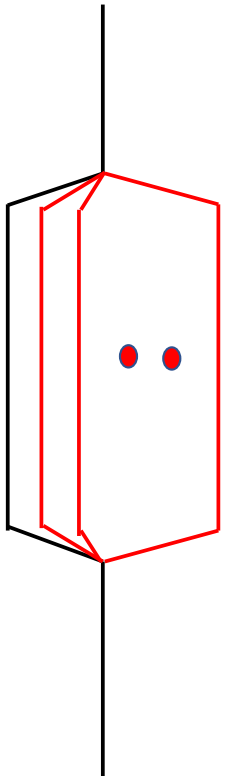
code/Parallel/openmp/dot1.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  int nThreads = 0;
  double dot = 0, a[DATA_SIZE], sum[64];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;
  for (int i=0; i<64; i++) sum[i] = 0;

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    if (tid == 0)  nThreads = numT;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum[tid] += a[i]*a[i];
  }
  for (int i=0; i<nThreads; i++)
      dot += sum[i];
  dot = sqrt(dot);
  printf("dot %f\n", dot);
  return 0;
}
```

Create a shared array to store data

Iterate over big array
using thread id
and number of threads

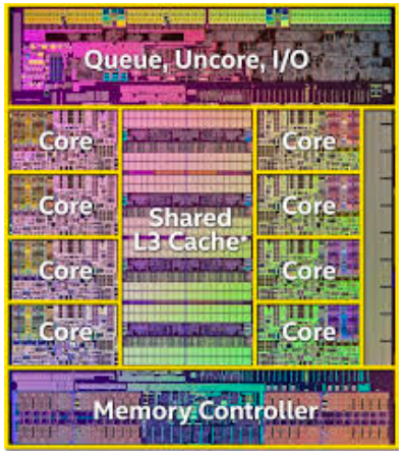
Combine sequentially



Poor Performance?

- Want high performance for shared memory: Use Caches!
  - Each processor has its own cache (or multiple caches)
  - Place data from memory into cache
  - Writeback cache: don't send all writes over bus to memory

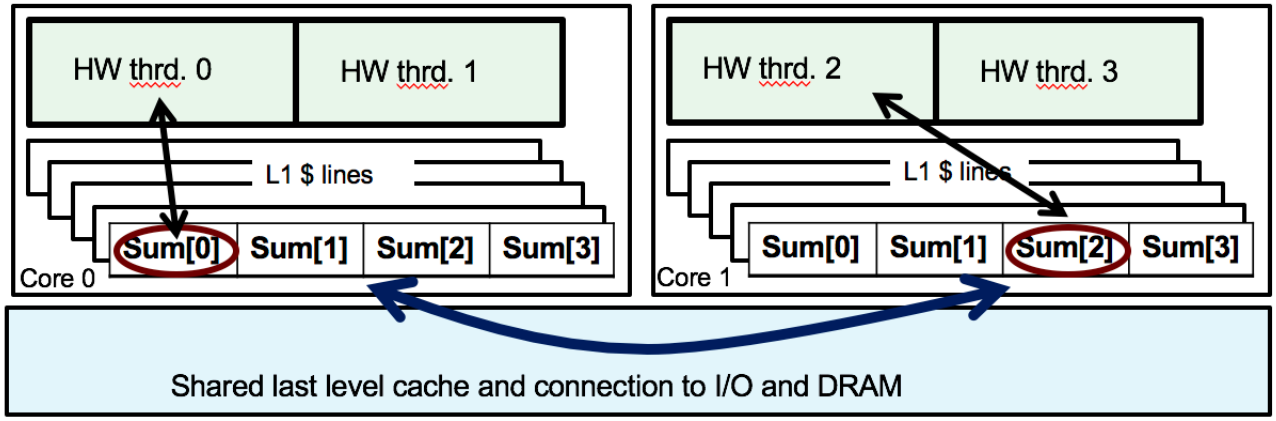
L3 Cache Actually Shared



- Problem is in multi-threaded model with all threads wanting to WRITE same spatially temporal data we have contention at the cache line in the L3 cache

False Sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads … This is called **false sharing** or sequential **consistency**.



- Sequential Consistency problem is pervasive and performance critical in shared memory

Solution?

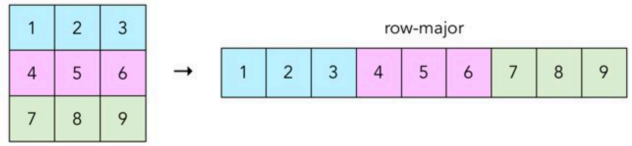
source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

AVOID FALSE SHARING

code/Parallel/openmp/dot2.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000
#define PAD 64

int main(int argc, const char **argv) {
  int nThreads = 0;
  double dot = 0, a[DATA_SIZE], sum[64][PAD];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;
  for (int i=0; i<64; i++) sum[i][0]= 0;

#pragma omp parallel
  {
     int tid = omp_get_thread_num();
     int numT = omp_get_num_threads();
     if (tid == 0)  nThreads = numT;
     for (int i=tid; i<DATA_SIZE; i+= numT)
       sum[tid][0]+= a[i]*a[i];
  }
  for (int i=0; i<nThreads; i++)
      dot += sum[i][0];
  dot = sqrt(dot);
  printf("dot %f \n", dot);
  return 0;
```

C:



Pad the shared array to store data to avoid false sharing

SYNCHRONIZATION

code/Parallel/openmp/dot3.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    double sum = 0.;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum += a[i]*a[i];
#pragma omp_critical
    dot += sum;
  }
  dot = sqrt(dot);
    printf("dot %f  \n", dot);
  return 0;
}
```

REDUCTION

code/Parallel/openmp/dot4.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel reduction(+:dot)
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    double sum = 0.;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum += a[i]*a[i];

    dot += sum;
  }
  dot = sqrt(dot);
   printf("dot %f  \n", dot);
  return 0;
}
```

dot5.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel reduction(+:dot)
  {
  #pragma parallel for
      for (int i=tid; i<DATA_SIZE; i+= numT)
      dot += a[i]*a[i];
  }
  dot = sqrt(dot);
   printf("dot %f  \n", dot);
  return 0;
}
```

Additional Reduction Operators

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

Pitfalls to Parallel Loops

might only occur for experienced programmer!

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
x = 0.5*dx;
for (int i=0; i<numSteps; i++) {
    pi += 4./(1.+x*x);
    x += dx;
  }
```

Note: loop index "i" is private by default

```
#pragma omp parallel for reduction(+:pi)
  for (int i=0; i<numSteps; i++) {
        x = (i+0.5)*dx;
        pi += 4./(1.+x*x);
  }
```

Remove loop carried dependence

~~#pragma omp parallel for reduction(+:pi)~~

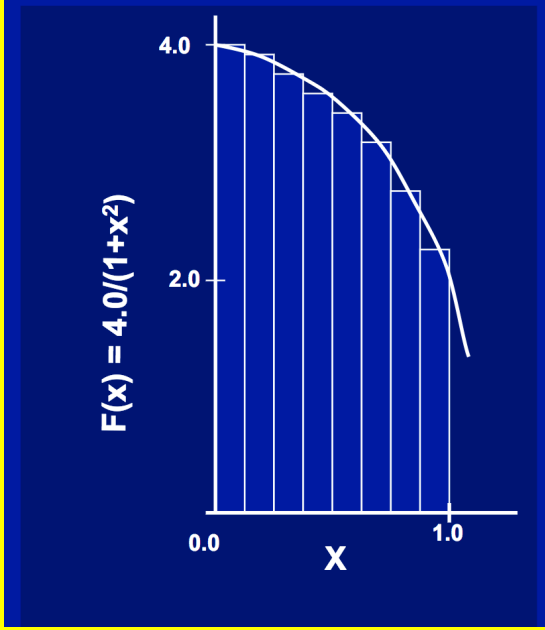
```
  for (int i=0; i<numSteps; i++) {
        pi += 4./(1.+x*x);
        x+=dx;
  }
```

DEMO: - compiling & running hello world

1. ssh yourName@frontera.tacc.utexas.edu
2. idev
3. cp SimCenterBootcamp2020/code/parallel/openmp
4. icc -qopenmp hello1.c
5. export OMP_NUM_THREADS=4
6. ibrun a.out

Exercise: Parallelize Compute PI using OpenMP

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

code/c/pi2.c

```
#include <stdio>
static int long numSteps = 100000;

int main() {
  double pi   = 0;
  double stepSize = 1.0/(double) numSteps;

  for (int i=0; i<numSteps; i++) {
      double x = (i+0.5)*stepSize;
       pi += 4./(1.+x*x);
  }

  pi *= stepSize;

  printf("PI = %16.14f\n",pi);
  return 0;
}
```