Parallel Programming for Multicore and Distributed Systems

10th Summer School in Statistics for Astronomers

Pierre-Yves Taunay

Research Computing and Cyberinfrastructure
224A Computer Building
The Pennsylvania State University
University Park
py.taunay@psu.edu

June 2014

PENNSTATE

Introduction

Objectives

1. Have a good understanding of
   1.1 Shared memory programs executed on multicore machines, and
   1.2 Distributed programs executed on multiple multicore machines
2. Get familiar with programming syntax for
   2.1 Shared memory programming, and
   2.2 Distributed programming

PENNSTATE

Recap from last session – 1/2

- Multicore programming
  - ↪ Possible to have 2 or more cores in a processor cooperate through **threads**
  - ↪ Thread libraries: **OpenMP**, POSIX, Boost
- Distributed programming
  - ↪ Multiple compute nodes are cooperating
  - ↪ MPI for communication paradigm

PENNSTATE

Recap from last session – 2/2

Shared memory	**Distributed memory**
Limited to 1 machine	No limit
1 process = multiple cores	1 process = 1 core
Multiple cores can access same memory	Memory local to processor core
Data exchange through memory bus	Data exchange through network

PENNSTATE

Disclaimers

- C for multicore programming
- C for distributed programming

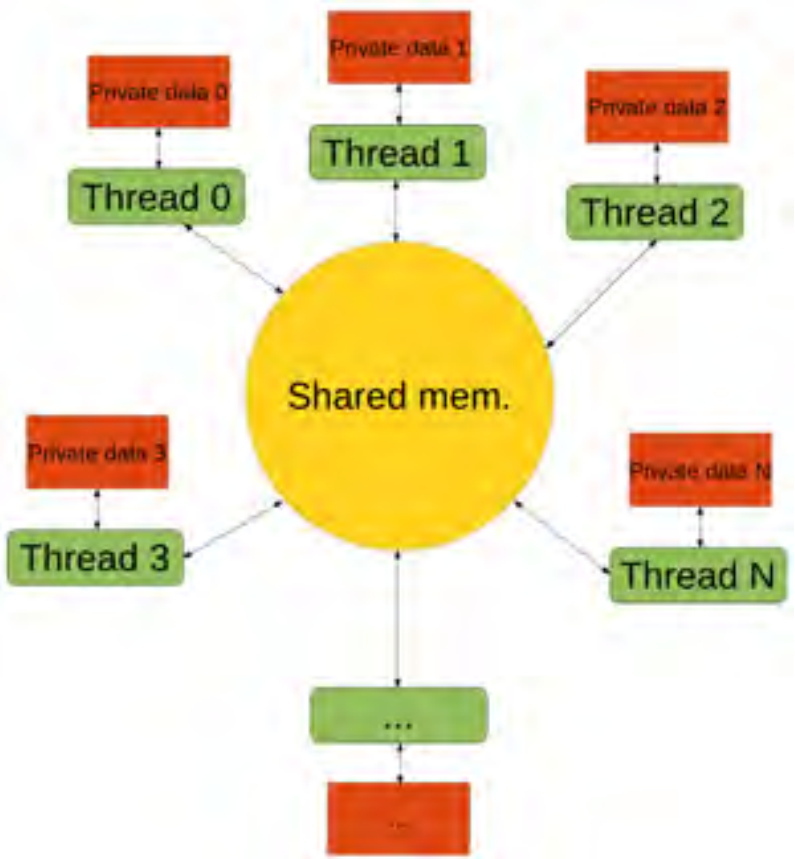
PENNSTATE

Multicore programming with OpenMP

OpenMP

General concepts – 1/2

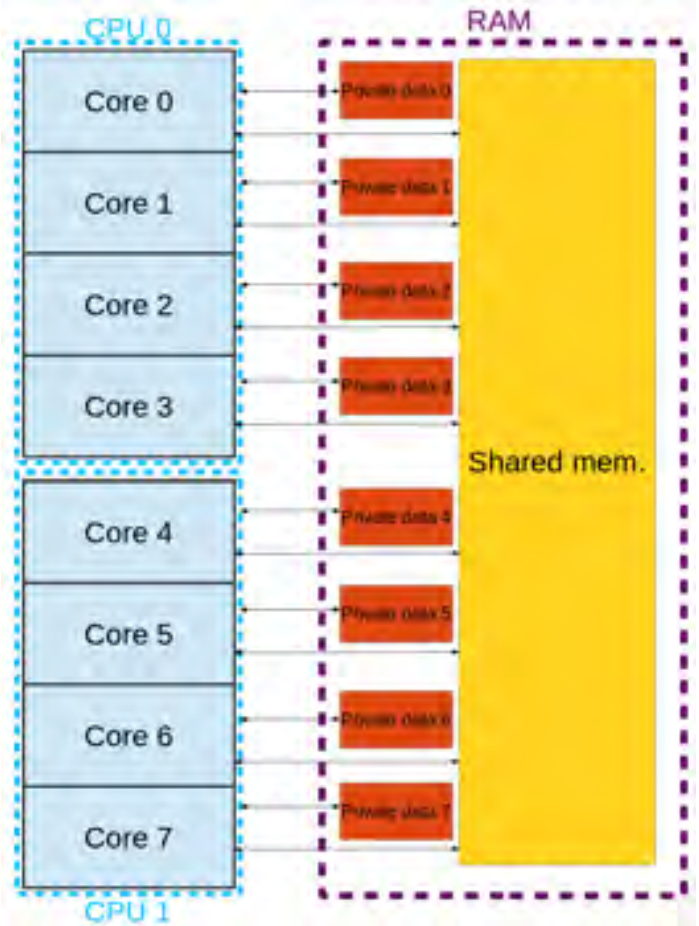
- Shared memory: all threads can access same data pool
- Still possible to have private data for each thread



PENNSTATE

OpenMP

General concepts – 1/2

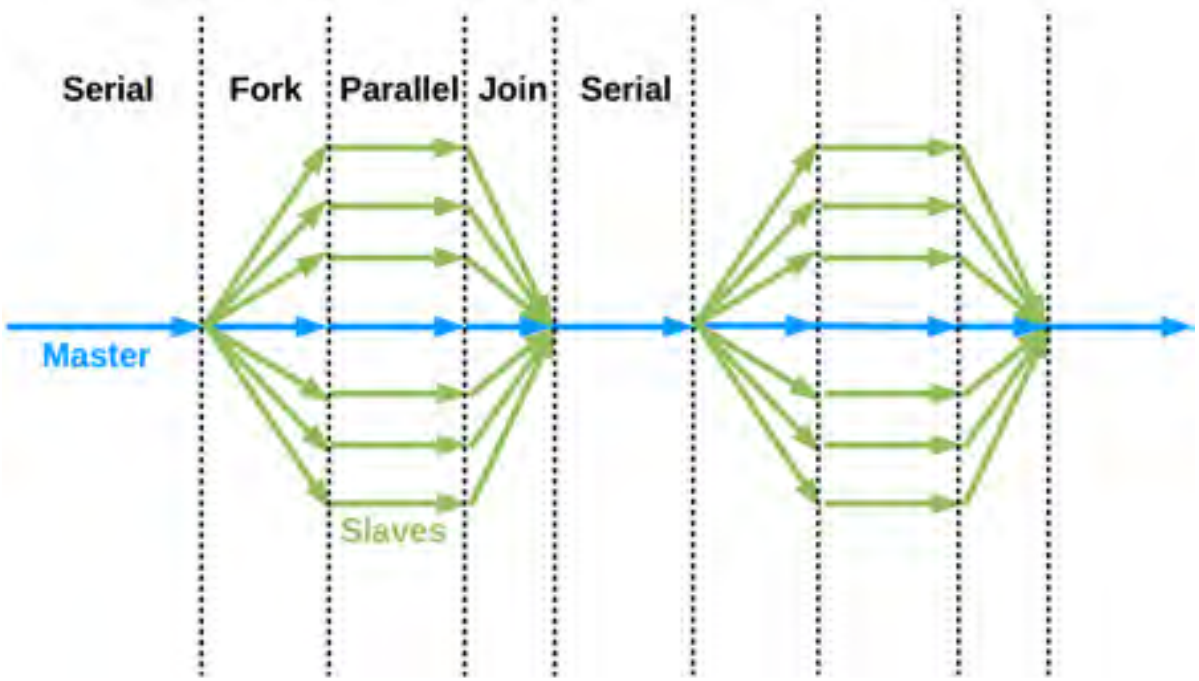


PENNSTATE

OpenMP

General concepts – 2/2

- Fork-join model of execution

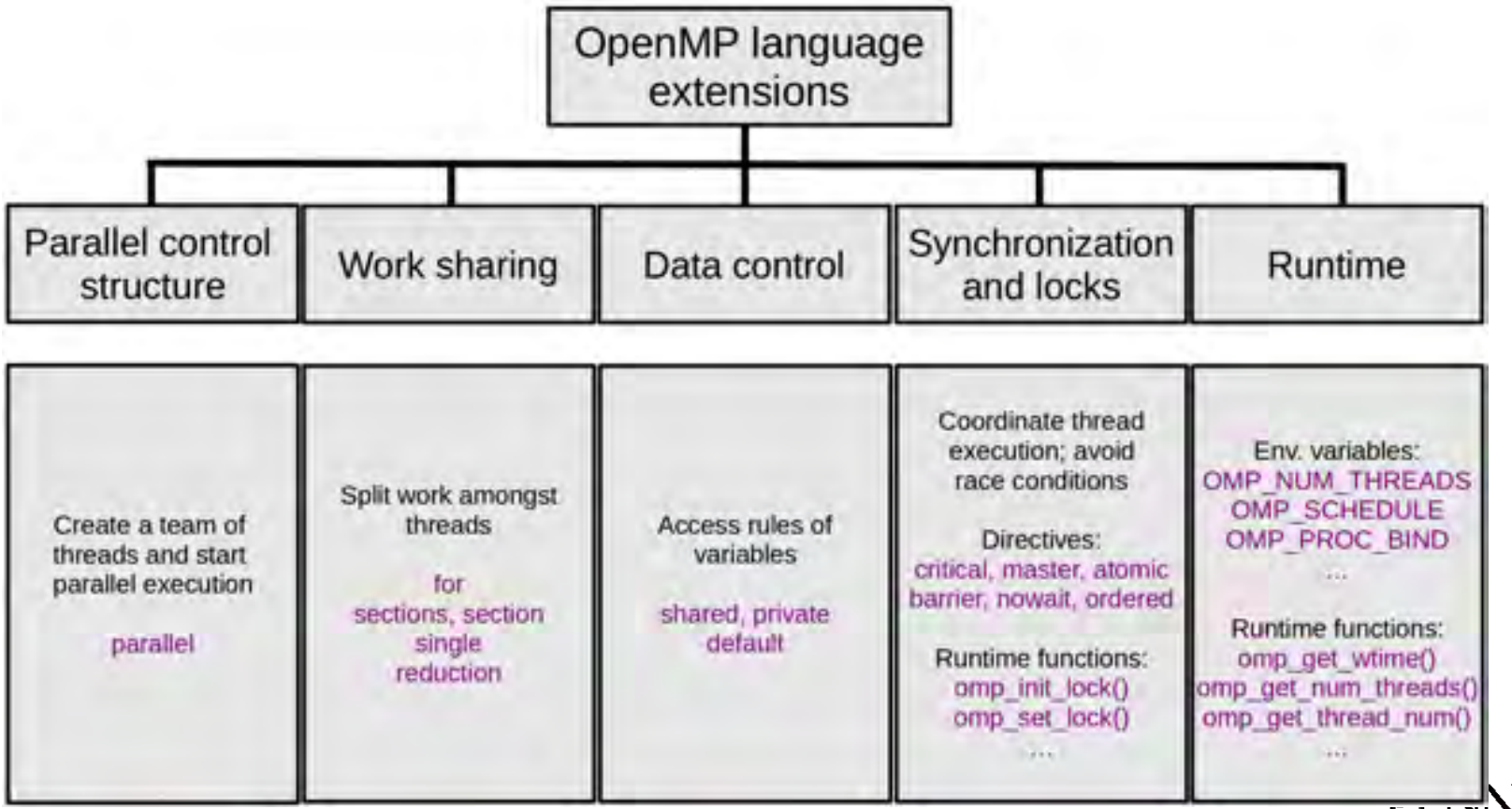


- Directive based: give "hints" to the compiler

PENNSTATE

OpenMP

Content



OpenMP

Compiling a program

- Intel compiler: -openmp flag
- GCC: -fopenmp flag

```
gcc -fopenmp -O3 myprogram.c -o myprogram
```

PENNSTATE

OpenMP

Running a program

- Environment variables are set to control program
- Number of threads to launch: `OMP_NUM_THREADS`

```
export OMP_NUM_THREADS = 4
```

```
setenv OMP_NUM_THREADS 4
```

PENNSTATE

OpenMP

Timing parts of a program

- Function **omp_get_wtime()**

```
1 double tic = omp_get_wtime();
2 // Code to time...
3 double toc = omp_get_wtime();
4 printf("Result: %f\n", toc-tic);
```

PENNSTATE

Parallel control structure

- In a **parallel** block, code is executed by all threads

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5 #pragma omp parallel
6   printf("Hello world ! \n");
7
8   return 0;
9 }
```

PENNSTATE

Multicore programming with OpenMP

Work sharing

Work sharing

for directive – 1/3

- Multiple ways to share work
- Simple "for" loop: iterations are distributed amongst threads

```
1 #pragma omp parallel
2   #pragma omp for
3   for(int i = 0;i<N;i++)
4     a[i] = b[i] + c[i];
```

PENNSTATE

Work sharing

for directive – 2/3

- Additional options:
  - ↪ **reduction**: to perform a reduction
  - ↪ **ordered**: iterations are executed in same order as a serial core
  - ↪ **nowait**: threads do not synchronize at end of loop

```
1 #pragma omp parallel
2   #pragma omp for reduction(+:out)
3   for(int i = 0;i<N;i++)
4     out = out + a[i];
```

PENNSTATE

Work sharing

for directive – 3/3

```
1  #pragma omp parallel
2  {
3    #pragma omp for nowait
4    for(int i = 0;i<N;i++)
5      c[i] = a[i] + b[i];
6
7    #pragma omp for
8    for(int i = 0;i<N;i++)
9      z[i] = x[i] + y[i];
10 } /* End of parallel */
```

PENNSTATE

Work sharing

sections directive

- Distributes a set of structured blocks amongst a "team" of threads
- Does not have to have iterations, as in the **for** directive
- Synchronization is implied at the end of **sections**
- Stand-alone **section** directives can be nested in a **sections**
- Each **section** is executed **once** by a thread.

PENNSTATE

Work sharing

sections directive

```
1  #pragma omp parallel
2  {
3    #pragma omp sections nowait
4    {
5      #pragma omp section
6      for(int i = 0;i<N;i++)
7          c[i] = a[i] + b[i];
8
9      #pragma omp section
10     printf("Hello !\n");
11   } /* End of sections */
12 } /* End of parallel */
```

Work sharing

single directive

- **Single**: only one thread executes that block

```
1  #pragma omp parallel
2  {
3    #pragma omp single
4    {
5      printf("Starting operations...\n");
6    } /* End of single */
7
8    #pragma omp sections nowait
9    {
10     #pragma omp section
11     for(int i = 0;i<N;i++)
12         c[i] = a[i] + b[i];
13
14     #pragma omp section
15     for(int i = 0;i<N;i++)
16         z[i] = x[i] + y[i];
17   } /* End of sections */
18 } /* End of parallel */
```

PENNSTATE

Parallel work sharing

- Combine directives

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

PENNSTATE

Multicore programming with OpenMP

Data control

Data model

- Threads can access global shared data pool
- Data can be also **private** to a thread

PENNSTATE

Data control

private directive

- Control of data is made with pragmas too

```
1  int my_thread_num = 0;
2
3  #pragma omp parallel private(my_thread_num)
4  {
5    // Get the thread number
6    my_thread_num = omp_get_thread_num()
7
8    printf("Hello from thread %d\n",my_thread_num);
9  } /* End parallel */
```

- **Remark**: **private** creates a private variable for each thread, but does not copy the data. Use **firstprivate** or **lastprivate** to do so.

PENNSTATE

Data control

Dangers of shared resources

```
1  #pragma omp parallel shared(a,b,c,x,y,z)
2  {
3    #pragma omp for nowait
4    for(int i = 0; i < N; i++)
5      a[i] = b[i] + c[i];
6
7    #pragma omp for
8    for(int i = 0; i < N; i++)
9      z[i] = a[i]*x[i] + y[i];
10 } /* End parallel */
```

- Can not ensure that "a" was updated before being used again: **race condition**

PENNSTATE

Detecting race conditions

- valgrind with the helgrind tool
- Compile with -g

```
gcc -g race.c -o race -fopenmp -std=c99
```

- Run valgrind

```
valgrind --tool=helgrind ./race
```

```
==18495==
==18495== Possible data race during read of size 4 at 0x4C75300 by thread #2
==18495== Locks held: none
==18495==    at 0x3B9A409CAD: ??? (in /usr/lib64/libgomp.so.1.0.0)
==18495==    by 0x3B9A408515: ??? (in /usr/lib64/libgomp.so.1.0.0)
==18495==    by 0x4A0C0D4: mythread_wrapper (hg_intercepts.c:219)
==18495==    by 0x3B95C079D0: start_thread (in /lib64/libpthread-2.12.so)
==18495==    by 0x3B954E8B6C: clone (in /lib64/libc-2.12.so)
==18495==
==18495== This conflicts with a previous write of size 4 by thread #1
==18495== Locks held: none
==18495==    at 0x3B9A409BF7: ??? (in /usr/lib64/libgomp.so.1.0.0)
==18495==    by 0x3B9A4089C9: ??? (in /usr/lib64/libgomp.so.1.0.0)
==18495==    by 0x4006C5: main (race.c:17)
==18495==
==18495== Address 0x4C75300 is 128 bytes inside a block of size 192 alloc'd
==18495==    at 0x4A069BE: malloc (vg_replace_malloc.c:270)
==18495==    by 0x3B9A403768: ??? (in /usr/lib64/libgomp.so.1.0.0)
==18495==    by 0x3B9A408C25: ??? (in /usr/lib64/libgomp.so.1.0.0)
==18495==    by 0x4006C5: main (race.c:17)
==18495==
```

PENNSTATE

Multicore programming with OpenMP

Control flow

PENNSTATE

Controlling the execution flow

barrier directive

```
1  #pragma omp parallel shared(a,b,c,x,y,z)
2  {
3    #pragma omp for nowait
4    for(int i = 0; i < N; i++)
5      a[i] = b[i] + c[i];
6
7    #pragma omp barrier
8
9    #pragma omp for
10   for(int i = 0; i < N; i++)
11     z[i] = a[i]*x[i] + y[i];
12 } /* End parallel */
```

- Expensive
- Wasted resources
- ...but sometimes necessary !

PENNSTATE

Controlling the execution flow

Other directives

- **master**
- **critical**
- **atomic**

Worked example I

- Want to calculate log–likelihood on several processes, at once

```
1  int main(int argc, char *argv[]) {
2    ...
3    // Parse the command line
4    ret = parse_command_line(argc,argv,
5      &nobs,&sizex,&nsample,&location);
6
7    // Parse the data on master
8    double *buffer_X = (double*)malloc(nobs*sizex*sizeof(double));
9    double *isigma = (double*)malloc(sizex*sizex*sizeof(double));
10   double *mu = (double*)malloc(sizex*sizeof(double));
11   double det_sigma = 0.0;
12
13   ret = read_data(buffer_X, isigma, &det_sigma, mu,
14     &nobs, &sizex, location);
15
16   // Thread variables
17   int nthreads = 1;
18   int th_num = 0;
19   int th_nobs = nobs;
20   nthreads = get_num_threads();
21   // Timing variables
22   double tic, toc, tot_time = 0.0;
23
```

PENNSTATE

Worked example II

```
24  //// Arrays for all threads
25  // The pool is allocated inside the shared memory
26  double *pool_LV = (double*)malloc(nobs*sizex*sizeof(double)); //
        Left hand side vector (X-mu)
27  double *pool_tmp = (double*)malloc(nobs*sizex*sizeof(double)); //
        Temporary holder for (X-mu)*SIG
28  double *pool_ones = (double*)malloc(nobs*sizeof(double)); //
        Temporary holder to create LV
29  double *pool_res = (double*)malloc(nthreads*sizeof(double)); //
        Each thread puts its result in pool_res
30
31  // Use pointers to get the correct location in the array
32  double *LV = NULL;
33  double *tmp = NULL;
34  double *ones = NULL;
35  double *X = NULL;
36
37  // Holder for final sum
38  double final_sum = 0.0;
39
40
41
42
43
44  // Main driver
```

PENNSTATE

Worked example III

```
45  #pragma omp parallel private(ones,LV,tmp,X,th_num, th_nobs)
        default(shared)
46  {
47    // Get thread number
48    th_num = omp_get_thread_num();
49    // Total number of observations for that thread
50    th_nobs = nobs/nthreads;
51
52    // Use the address to point to the correct location in the
          vector
53    X = &buffer_X[th_num*nobs*sizex/nthreads];
54    LV = &pool_LV[th_num*th_nobs*sizex];
55    tmp = &pool_tmp[th_num*th_nobs*sizex];
56    ones = &pool_ones[th_num*th_nobs];
57
58    // Each process can now calculate the term in the
59    // exponent for a subset of random vectors
60    log_likelihood(X,isigma,mu,det_sigma,th_nobs,
61      sizex,&pool_res[th_num],LV,tmp,ones);
62
63    #pragma omp barrier
64
65    // Reduction: sum all the intermediary results
66    #pragma omp for reduction(+:final_sum)
67    for(int i = 0; i < nthreads; i++)
68      final_sum = final_sum + pool_res[i];
```

PENNSTATE

Worked example IV

```
69    } /* End of parallel */
70    toc = omp_get_wtime();
71    tot_time += toc-tic;
72    ...
73 }
```

Summary

- Multicore / shared memory: use OpenMP
- Language extensions for C,C++, and Fortran
- Compiler directives

\+ Relatively easy to use: speedup with not too much effort
\+ Good scalability on 1 node
\+ No network communication – low latency, high BW
\- Threads are heavy
\- Limited to one compute node
\- Overhead if not enough work for threads

PENNSTATE

Going further with OpenMP

- OpenMP documentation
  http://openmp.org/wp/resources/
- LLNL OpenMP tutorial
  http://computing.llnl.gov/tutorials/openMP/
- RCC HPC Essentials
  http://rcc.its.psu.edu/education/seminars/pages/hpc_essentials/HPC2.pdf

PENNSTATE

Distributed programming with MPI

MPI

General concepts – 1/3

- **Process** work unit on a single core
- **Rank** process number
- **Distributed memory** multiple compute nodes
- **Message Passing** communication paradigm
- **Send / Receive**
- **Buffer** space where data is stored for send / receive ops
- **Synchronous / Asynchronous** wait / don't wait for transfer to be complete (ACK from receiver)
- **Blocking / Non-blocking** completion of comm. is (in)dependent of events (e.g. buffer that contained data is available for reuse)
- **Communicators, groups** can specify topology – MPI_COMM_WORLD

PENNSTATE

MPI

General concepts – 2/3

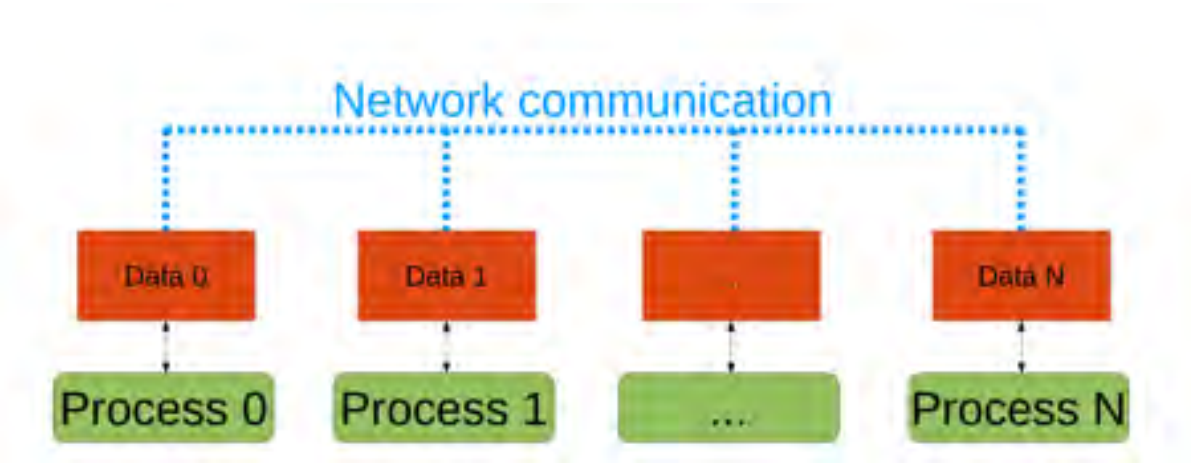
- Distributed memory: each process has its own data
- Collaboration through message exchange

PENNSTATE

MPI

General concepts – 2/3

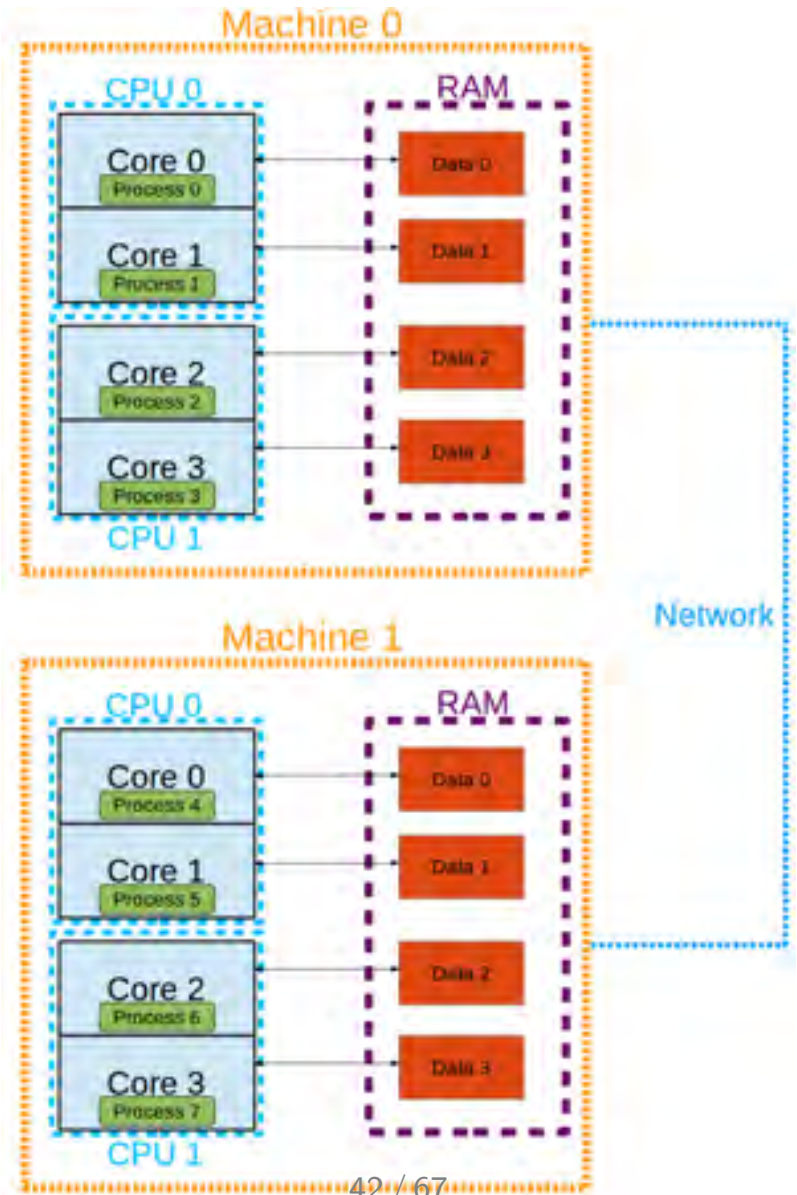
- Distributed memory: each process has its own data
- Collaboration through message exchange



PENNSTATE

MPI

General concepts – 2/3



MPI

General concepts – 3/3

- Process is started on each specified core

MPI

Compiling a program

- Choice 1: Use the MPI wrappers to compile and link

```
mpicc -c myfile1.c -o myfile1.o
mpicc -c myfile2.c -o myfile2.o
mpicc myfile1.o myfile2.o -o myprogram
```

- Choice 2: Use any compiler; have to provide include file location and libraries location as well

```
gcc -c myfile1.c -o myfile1.o -I/path/to/mpi/include
gcc -c myfile2.c -o myfile2.o -I/path/to/mpi/include
gcc myfile1.o myfile2.o -o myprogram
    -L/path/to/mpi/libs -lmpi
```

PENNSTATE

MPI

Running a program

- Use the command "mpirun"

```
> mpirun ./hello
```

PENNSTATE

MPI

Program structure – 1/2

1. Include file

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

2. Program start...

```
1 int main(int argc, char* argv[]) {
```

3. Initialize the MPI environment

```
1 MPI_Init(&argc,&argv);
```

4. Do stuff...
5. “Finalize” the MPI environment

```
1 MPI_Finalize();
```

PENNSTATE

MPI

Program structure – 2/2

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char* argv[]) {
5
6     MPI_Init(&argc,&argv);
7     printf("Hello world !\n");
8     MPI_Finalize();
9
10    return 0;
11 }
```

```
Hello world !
Hello world !
```

PENNSTATE

MPI

Size and rank – 1/2

- What is the process ID ?

```
1  MPI_Comm_rank(comm,*rank);
2  // Usage:
3  int my_rank;
4  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
```

- How many processes did we launch ?

```
1  MPI_Comm_size(comm,*size);
2  // Usage:
3  int number_proc;
4  MPI_Comm_size(MPI_COMM_WORLD,&number_proc);
```

PENNSTATE

MPI

Size and rank – 2/2

```
1  MPI_Init(&argc,&argv);
2  int my_rank, number_proc;
3  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
4  MPI_Comm_size(MPI_COMM_WORLD,&number_proc);
5
6  printf("Hello from process %d of %d !\n", my_rank,
       number_proc);
7  MPI_Finalize();
```

```
Hello from process 1 of 2
Hello from process 0 of 2
```

PENNSTATE

MPI

Send / Recv – 1/4

- **Message Passing**: point-to-point

```
1 // Sending data:
2 MPI_Send(*buffer,count,type,destination,tag,comm);
3 // Receiving data:
4 MPI_Recv(*buffer,count,type,source,tag,comm,status);
```

PENNSTATE

MPI

Send / Recv – 2/4

```
1  int my_rank, number_proc;
2  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
3  // Process 0 wants to send 10 integers to process 1
4  if(my_rank == 0) {
5    // Send 10 integers from vector V
6    MPI_Send(V,10,MPI_INT,1,0,MPI_COMM_WORLD);
7  } else if (my_rank == 1) {
8    // Receive 10 integers from vector V; store in U
9   MPI_Recv(U,10,MPI_INT,0,0,MPI_COMM_WORLD);
10 }
```

PENNSTATE

MPI

Send / Recv – 3/4

```
1  if(my_rank == 0) {
2    int *V = (int*) malloc(NELEM*sizeof(int));
3    for(int i = 0; i < NELEM; i++)
4      V[i] = rand();
5
6    print_vector(V,NELEM,my_rank);
7
8    int dest = 1;
9    MPI_Send(V,NELEM,MPI_INT,dest,0,MPI_COMM_WORLD);
10   printf("New vector V \n");
11   print_vector(V,NELEM,my_rank);
12
13   free(V);
14
15 } else if(my_rank == 1) {
16   int *U = (int*) malloc(NELEM*sizeof(int));
17
18   print_vector(U,NELEM,my_rank);
19
20   int src = 0;
21   MPI_Status status;
22   MPI_Recv(U,NELEM,MPI_INT,src,0,MPI_COMM_WORLD,&status);
23   printf("New vector U\n");
24   print_vector(U,NELEM,my_rank);
25
26   free(U);
27 }
```

PENNSTATE

MPI

Send / Recv – 4/4

Output

```
Vector V
Process 0
V[0] = 1918581883
V[1] = 1004453085
V[2] = 786820889

Vector U
Process 1
V[0] = 24259120
V[1] = 0
V[2] = 24265808

New vector V
Process 0
V[0] = 1918581883
V[1] = 1004453085
V[2] = 786820889

New vector U
Process 1
V[0] = 1918581883
V[1] = 1004453085
V[2] = 786820889
```

PENNSTATE

MPI

A word on distributed memory

- Watch where you allocate memory !

```
1  int *V;
2  if(my_rank == 0) {
3   V = (int*)malloc(N*sizeof(int));
4  }
5  ...
6  // Seg. fault error -- only the process 0 has
       allocated memory !
7  if(my_rank == 1) {
8   V[0] = 1;
9  }
```

MPI

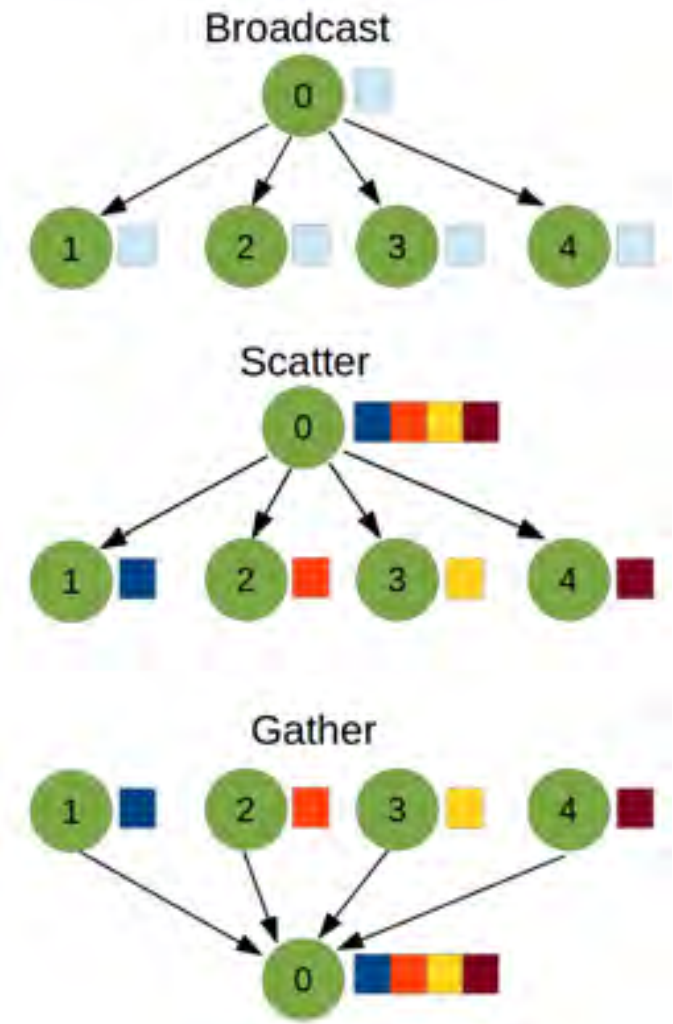
Collectives

- Involves all processes in a communicator
- Examples: broadcast, gather, scatter

PENNSTATE

MPI

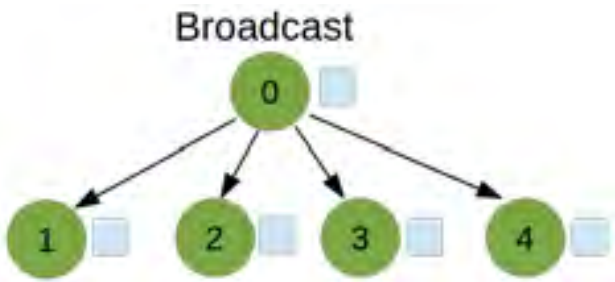
Collectives



PENNSTATE

MPI

Collectives



- **Message Passing**: broadcast

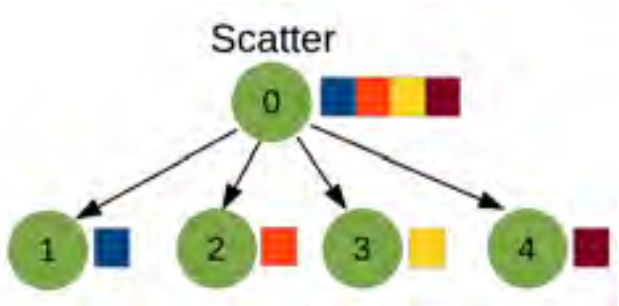
```
1 MPI_Bcast(*data,count,type,root,comm)
```

```
1 ...
2 int data = rand();
3 // Process 0 sends its value to everyone
4 MPI_Bcast(&data,1,MPI_INT,0,MPI_COMM_WORLD);
5 ...
```

PENNSTATE

MPI

Collectives



- **Message Passing**: scatter

```
MPI_Scatter(*send_data, send_count, send_datatype,
            *recv_data, recv_count, recv_datatype,
            root, comm)
```

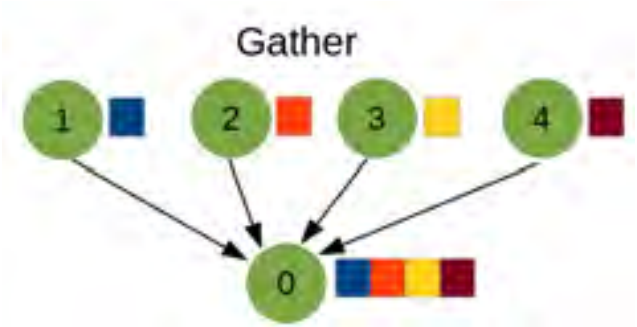
```
...
int * data = NULL; // Size = 2*N elements
int * recv_buffer = NULL; // Size = 2 elements
...
for(int i = 0;i<2*N;i++)
  data[i] = rand();

// Process 0 sends 2 values to everyone
// 0: [0,1], 1: [2,3], etc.
MPI_Scatter(data,2,MPI_INT,recv_buffer,2,MPI_INT,0,MPI_COMM_WORLD);
...
```

PENNSTATE

MPI

Collectives



- **Message Passing**: gather

```
MPI_Gather(*send_data, send_count, send_datatype,
           *recv_data, recv_count, recv_datatype,
           root, comm)
```

```
...
int * data = NULL; // Size = 2 elements
int * recv_buffer = NULL; // Size = 2*N elements
...
for(int i = 0;i<2;i++)
  data[i] = rand();

// Process 0 receives 2 elements from N processes
MPI_Gather(data,2,MPI_INT,recv_buffer,2*N,MPI_INT,0,MPI_COMM_WORLD);
...
```

PENNSTATE

MPI

Collectives

- Other efficient collective: reductions

Worked example I

- Want to calculate log–likelihood on several processes, at once

```
1  int main(int argc, char *argv[]) {
2    // Start MPI
3    MPI_Init(&argc,&argv);
4    // Get rank and number of proc.
5    int myrank, nproc = 0;
6    MPI_Comm_rank(COMM,&myrank);
7    MPI_Comm_size(COMM,&nproc);
8
9    int nobs, sizex, nsample = 0;
10   char *location = NULL;
11   int ret = 0;
12
13   // Parse the command line and set proc_nobs
14   // Number of observations per process
15   int proc_nobs;
16   ret = parse_command_line(argc,argv,&nobs,&sizex,&nsample,
17     &location,&proc_nobs,myrank,nproc);
18
19   // Allocate the data for ALL processes.
20   double *isigma = (double*)malloc(sizex*sizex*sizeof(double));
21   double *mu = (double*)malloc(sizex*sizeof(double));
22   double det_sigma = 0.0;
23
```

Worked example II

```
24  // Only the first process will parse the data. Others will wait
25  double *buffer_X = NULL;
26  if( myrank == 0 ) {
27    int ret = 0;
28    buffer_X = (double*)malloc(nobs*sizex*sizeof(double));
29    ret = read_data(buffer_X, isigma, &det_sigma,
30      mu, &nobs, &sizex, location);
31  }
32  // Wait for the process 0 to finish parsing the files
33  MPI_Barrier(COMM);
34
35  // Timing variables
36  double tic, toc,tot_time = 0.0;
37
38  // Sums
39  double final_sum = 0.0;
40  double my_sum = 0.0;
41
42  // Individual arrays
43  double *X = (double*)malloc(proc_nobs*sizex*sizeof(double));
44  double *LV = (double*)malloc(proc_nobs*sizex*sizeof(double));
45  double *tmp = (double*)malloc(proc_nobs*sizex*sizeof(double));
46  double *ones = (double*)malloc(proc_nobs*sizeof(double));
47
48
49  // Process 0 sends data to everyone with collectives
```

PENNSTATE

Worked example III

```
50    MPI_Bcast(isigma, sizex*sizex, MPI_DOUBLE, 0, COMM);
51    MPI_Bcast(mu, sizex, MPI_DOUBLE, 0, COMM);
52    MPI_Bcast(&det_sigma, 1, MPI_DOUBLE, 0, COMM);
53    MPI_Scatter(buffer_X, proc_nobs*sizex, MPI_DOUBLE, X, proc_nobs*sizex,
          MPI_DOUBLE, 0, COMM);
54
55    tic = omp_get_wtime();
56    final_sum = 0.0;
57    log_likelihood(X, isigma, mu, det_sigma, proc_nobs, sizex, &my_sum, LV,
          tmp, ones);
58
59    // Combine all the intermediary sums in a single one
60    MPI_Reduce(&my_sum, &final_sum, 1, MPI_DOUBLE, MPI_SUM, 0, COMM);
61    MPI_Barrier(COMM);
62    toc = omp_get_wtime();
63    tot_time += toc-tic;
64    ...
65    if ( myrank == 0 ) {
66      free(buffer_X);
67    }
68    // Etc.
69    MPI_Finalize();
70    return EXIT_SUCCESS;
71 }
```

Summary

- Distributed programming: use MPI
- Library for C, C++, and Fortran
- Programmer responsible for a lot of stuff !

\+ Standard
\+ Optimized collective communication
\+ Can overlap communication and computation
\- Strong learning curve
\- Difficult (?) to debug
\- Communication through network = optimization issues

PENNSTATE

Going further with MPI

- OpenMPI documentation
  http://www.open-mpi.org/doc/
- MPI tutorial
  http://mpitutorial.com/
- LLNL MPI tutorial
  https://computing.llnl.gov/tutorials/mpi/
- RCC HPC Essentials
  http://rcc.its.psu.edu/education/seminars/pages/hpc_essentials/HPC3.pdf

PENNSTATE

Conclusion

- Covered two approaches for shared and distributed mem.
- Shared memory or distributed memory ?
  - ↪ Depends on pb. size
  - ↪ Amdahl's law !
- Combine best of both worlds: hybrid approach
- Other easier (?) distributed approaches
  - ↪ Python – MPI4py
  - ↪ R – RMPI
  - ↪ Julia

PENNSTATE

Questions ?