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
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
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
# Recap from last session – 2/2

<table>
<thead>
<tr>
<th>Shared memory</th>
<th>Distributed memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited to 1 machine</td>
<td>No limit</td>
</tr>
<tr>
<td>1 process = multiple cores</td>
<td>1 process = 1 core</td>
</tr>
<tr>
<td>Multiple cores can access same memory</td>
<td>Memory local to processor core</td>
</tr>
<tr>
<td>Data exchange through memory bus</td>
<td>Data exchange through network</td>
</tr>
</tbody>
</table>
Disclaimers

- C for multicore programming
- C for distributed programming
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
OpenMP

General concepts – 1/2
OpenMP

General concepts – 2/2

- Fork-join model of execution

- Directive based: give “hints” to the compiler
OpenMP

Content

OpenMP language extensions

Parallel control structure
- Create a team of threads and start parallel execution
- parallel

Work sharing
- Split work amongst threads
- for sections, section single reduction

Data control
- Access rules of variables
- shared, private default

Synchronization and locks
- Coordinate thread execution; avoid race conditions
- Directives:
  - critical, master, atomic barrier, nowait, ordered
- Runtime functions:
  - omp_init_lock()
  - omp_set_lock()

Runtime
- Env. variables:
  - OMP_NUM_THREADS
  - OMP_SCHEDULE
  - OMP_PROC_BIND
- Runtime functions:
  - omp_get_rtime()
  - omp_get_num_threads()
  - omp_get_thread_num()
OpenMP

Compiling a program

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

```
gcc -fopenmp -O3 myprogram.c -o myprogram
```
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
```
OpenMP
Timing parts of a program

- Function `omp_get_wtime()`

```c
double tic = omp_get_wtime();
// Code to time...
double toc = omp_get_wtime();
printf("Result: %f\n", toc-tic);
```
Parallel control structure

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

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

int main() {
    #pragma omp parallel
    printf("Hello world ! \n");
    return 0;
}
```
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

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

```c
#pragma omp parallel
#pragma omp for reduction (+:out)
for (int i = 0; i<N; i++)
  out = out + a[i];
```
Work sharing

for directive – 3/3

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

#pragma omp for
    for (int i = 0; i < N; i++)
        z[i] = x[i] + y[i];
} /* End of parallel */
```
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.
#pragma omp parallel
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (int i = 0; i < N; i++)
            c[i] = a[i] + b[i];

        #pragma omp section
        printf("Hello !\n");
    } /* End of sections */
} /* End of parallel */
Work sharing

single directive

★ Single: only one thread executes that block

```
#pragma omp parallel
{
    #pragma omp single
    {
        printf("Starting operations...\n");
    } /* End of single */

    #pragma omp sections nowait
    {
        #pragma omp section
        for(int i = 0; i < N; i++)
            c[i] = a[i] + b[i];

        #pragma omp section
        for(int i = 0; i < N; i++)
            z[i] = x[i] + y[i];
    } /* End of sections */
} /* End of parallel */
```
Parallel work sharing

- Combine directives

```c
#pragma omp parallel for
for(int i = 0; i < N; i++)
c[i] = a[i] + b[i];
```
Multicore programming with OpenMP

Data control
Data model

- Threads can access global shared data pool
- Data can be also \textit{private} to a thread
Data control

**private directive**

- Control of data is made with pragmas too

```c
int my_thread_num = 0;

#pragma omp parallel private (my_thread_num)
{
    // Get the thread number
    my_thread_num = omp_get_thread_num();

    printf("Hello from thread %d
", my_thread_num);
}

/* End parallel */
```

- **Remark:** `private` creates a private variable for each thread, but does not copy the data. Use `firstprivate` or `lastprivate` to do so.
Data control
Dangers of shared resources

```c
#pragma omp parallel shared(a, b, c, x, y, z)
{
    #pragma omp for nowait
    for (int i = 0; i < N; i++)
        a[i] = b[i] + c[i];

    #pragma omp for
    for (int i = 0; i < N; i++)
        z[i] = a[i] * x[i] + y[i];
} /* End parallel */
```

- Can not ensure that “a” was updated before being used again: 
  race condition
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
Multicore programming with OpenMP

Control flow
Controlling the execution flow

**barrier directive**

```c
#pragma omp parallel shared (a, b, c, x, y, z)
{
    #pragma omp for nowait
    for (int i = 0; i < N; i++)
        a[i] = b[i] + c[i];

    #pragma omp barrier

    #pragma omp for
    for (int i = 0; i < N; i++)
        z[i] = a[i] * x[i] + y[i];
}
/* End parallel */
```

- Expensive
- Wasted resources
- ...but sometimes necessary!
Controlling the execution flow

Other directives

- master
- critical
- atomic
Worked example I

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

```c
int main(int argc, char *argv[]) {
    ...
    // Parse the command line
    ret = parse_command_line(argc, argv,
                              &nobs,&sizex,&nsample,&location);

    // Parse the data on master
    double *buffer_X = (double*)malloc(nobs*sizex*sizeof(double));
    double *isigma = (double*)malloc(sizex*sizex*sizeof(double));
    double *mu = (double*)malloc(sizex*sizeof(double));
    double det_sigma = 0.0;
    ret = read_data(buffer_X, isigma, &det_sigma, mu,
                    &nobs, &sizex, location);

    // Thread variables
    int nthreads = 1;
    int th_num = 0;
    int th_nobs = nobs;
    nthreads = get_num_threads();
    // Timing variables
    double tic, toc, tot_time = 0.0;
```
Intro Multicore Distributed Conclusion

Worked example II

```c
// Arrays for all threads
// The pool is allocated inside the shared memory
double *pool_LV = (double*)malloc(nobs*sizex*sizeof(double)); // Left hand side vector (X−mu)
double *pool_tmp = (double*)malloc(nobs*sizex*sizeof(double)); // Temporary holder for (X−mu)*SIG
double *pool_ones = (double*)malloc(nobs*sizeof(double)); // Temporary holder to create LV
double *pool_res = (double*)malloc(nthreads*sizeof(double)); // Each thread puts its result in pool_res

// Use pointers to get the correct location in the array
double *LV = NULL;
double *tmp = NULL;
double *ones = NULL;
double *X = NULL;

// Holder for final sum
double final_sum = 0.0;

// Main driver
```

PENNSTATE
#pragma omp parallel private (ones, LV, tmp, X, th_num, th_nobs) default (shared) 
{
    // Get thread number
    th_num = omp_get_thread_num();
    // Total number of observations for that thread
    th_nobs = nobs / ntreads;

    // Use the address to point to the correct location in the vector
    X = &buffer_X[th_num*nobs*sizex/ntreads];
    LV = &pool_LV[th_num*th_nobs*sizex];
    tmp = &pool_tmp[th_num*th_nobs*sizex];
    ones = &pool_ones[th_num*th_nobs];

    // Each process can now calculate the term in the exponent for a subset of random vectors
    log_likelihood(X, isigma, mu, det_sigma, th_nobs, sizex, &pool_res[th_num], LV, tmp, ones);

    #pragma omp barrier

    // Reduction: sum all the intermediary results
    #pragma omp for reduction(:final_sum)
    for (int i = 0; i < ntreads; i++)
        final_sum = final_sum + pool_res[i];
Worked example IV

```c
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
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
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
MPI

General concepts – 2/3

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

General concepts – 2/3

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

Running a program

- Use the command “mpirun”

> mpirun ./hello
MPI

Program structure – 1/2

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

2. Program start...
   
   ```
   int main(int argc, char* argv[]) {
   ```

3. Initialize the MPI environment
   
   ```
   MPI_Init(&argc,&argv);
   ```

4. Do stuff...

5. “Finalize” the MPI environment
   
   ```
   MPI_Finalize();
   ```
MPI

Program structure – 2/2

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

int main(int argc, char* argv[]) {
    MPI_Init(&argc,&argv);
    printf("Hello world !\n");
    MPI_Finalize();

    return 0;
}

Hello world !
Hello world !
```
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);`
MPI
Size and rank – 2/2

```c
MPI_Init(&argc,&argv);
int my_rank, number_proc;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&number_proc);

printf("Hello from process %d of %d !\n", my_rank,
        number_proc);
MPI_Finalize();
```

Hello from process 1 of 2
Hello from process 0 of 2
Message Passing: point-to-point

```c
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);
```
```
int my_rank, number_proc;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
// Process 0 wants to send 10 integers to process 1
if (my_rank == 0) {
    // Send 10 integers from vector V
    MPI_Send(V,10,MPI_INT,1,0,MPI_COMM_WORLD);
}
else if (my_rank == 1) {
    // Receive 10 integers from vector V; store in U
    MPI_Recv(U,10,MPI_INT,0,0,MPI_COMM_WORLD);
}
```
if (my_rank == 0) {
    int *V = (int*) malloc(NELEM*sizeof(int));
    for (int i = 0; i < NELEM; i++)
        V[i] = rand();

    print_vector(V,NELEM, my_rank);

    int dest = 1;
    MPI_Send(V,NELEM, MPI_INT, dest, 0, MPI_COMM_WORLD);
    printf("New vector V\n");
    print_vector(V,NELEM, my_rank);

    free(V);
}

} else if (my_rank == 1) {
    int *U = (int*) malloc(NELEM*sizeof(int));

    print_vector(U,NELEM, my_rank);

    int src = 0;
    MPI_Status status;
    MPI_Recv(U,NELEM, MPI_INT, src, 0, MPI_COMM_WORLD,&status);
    printf("New vector U\n");
    print_vector(U,NELEM, my_rank);

    free(U);
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
Watch where you allocate memory!

```c
int *V;
if (my_rank == 0) {
    V = (int*)malloc(N*sizeof(int));
}
...
// Seg. fault error — only the process 0 has allocated memory!
if (my_rank == 1) {
    V[0] = 1;
}
```
MPI

Collectives

- Involves all processes in a communicator
- Examples: broadcast, gather, scatter
MPI
Collectives
**Message Passing**: broadcast

1. `MPI_Bcast__(*data, count, type, root, comm*)`

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

Collectives

- **Message Passing**: scatter

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

```c
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);
```
Message Passing: gather

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

```c
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);
```
MPI

Collectives

- Other efficient collective: reductions
Worked example I

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

```c
int main(int argc, char *argv[]) {
  // Start MPI
  MPI_Init(&argc, &argv);
  // Get rank and number of proc.
  int myrank, nproc = 0;
  MPI_Comm_rank(COMM, &mymrank);
  MPI_Comm_size(COMM, &nproc);

  int nobs, sizeX, nsample = 0;
  char *location = NULL;
  int ret = 0;

  // Parse the command line and set proc_nobs
  // Number of observations per process
  int proc_nobs;
  ret = parse_command_line(argc, argv, &nobs, &sizeX, &nsample,
                            &location, &proc_nobs, myrank, nproc);

  // Allocate the data for ALL processes.
  double *isigma = (double*)malloc(sizeX*sizeX*sizeof(double));
  double *mu = (double*)malloc(sizeX*sizeof(double));
  double det_sigma = 0.0;
```
Worked example II

24 // Only the first process will parse the data. Others will wait
double *buffer_X = NULL;
25 if ( myrank == 0 ) {
26   int ret = 0;
27   buffer_X = (double*)malloc(nobs*sizeX*sizeof(double));
28   ret = read_data(buffer_X, isigma, &det_sigma, 
29                   mu, &nobs, &sizeX, location);
30 }
31 // Wait for the process 0 to finish parsing the files
32 MPI_Barrier(COMM);

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));

48 // Process 0 sends data to everyone with collectives
Worked example III

```c
MPI_Bcast(isigma, sizeX*sizeX, MPI_DOUBLE, 0, COMM);
MPI_Bcast(mu, sizeX, MPI_DOUBLE, 0, COMM);
MPI_Bcast(&det_sigma, 1, MPI_DOUBLE, 0, COMM);
MPI_Scatter(buffer_X, proc_nobs*sizeX, MPI_DOUBLE, X, proc_nobs*sizeX, MPI_DOUBLE, 0, COMM);

tic = omp_get_wtime();
final_sum = 0.0;
log_likelihood(X, isigma, mu, det_sigma, proc_nobs, sizeX, &my_sum, LV, tmp, ones);

// Combine all the intermediary sums in a single one
MPI_Reduce(&my_sum, &final_sum, 1, MPI_DOUBLE, MPI_SUM, 0, COMM);
MPI_Barrier(COMM);
toc = omp_get_wtime();
tot_time += toc - tic;
...
if (myrank == 0) {
    free(buffer_X);
}
// Etc.
MPI_Finalize();
return EXIT_SUCCESS;
```

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
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
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
Questions ?