Introduction to OpenMP and MPI

http://nci.org.au/user-support/training/

help@nci.org.au

Gaurav Mitra

nci.org.au

@NCInews

Agenda

- Serial Programming in C
- Basic Debugging with gdb
- Basic Shared Memory Programming with OpenMP
- Basic Distributed Memory Programming with MPI

All sections are hands-on!

Agenda

1. Serial Programming in C
2. Debugging with gdb
3. Basic Shared Memory Programming with OpenMP
4. Basic Distributed Memory Programming with MPI

Hello World!

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

int main(int argc, char **argv)
{
    printf(``Hello World\n'');

    return 0;
}
```

Agenda

The GNU Debugger - gdb

- A very popular command line debugger for C/C++ programs on UNIX systems
- Can introduce breakpoints
- During runtime, possible to inspect values of variables in memory
- Ability to analyze core dumps to do post-crash analysis
- Possible to step through instructions to see their effect
- Not a tool to detect memory leaks. See *valgrind* for that
- Does not help detect compilation errors

Running with gdb

```
# Load module gcc/4.7.3
# Recompile the hello world program with gcc using the '-g -ggdb -O0' options
# Run it through gdb
$ gdb ./hello

# Introduce a variable, increment it in a loop
# Run hello with gdb and add a breakpoint for a specific line number
(gdb) b <line number>
# Run the program
(gdb) r
# Upon hitting the breakpoint inspect the values of variables
(gdb) p <var>
# Continue the execution
(gdb) c
# Look at the program's stack status
(gdb) bt

# A way to attach to a running program with gdb
# Get its process ID using the 'ps' command
# Attach to it using gdb
$ gdb --pid <PID>
```

Debugging with gdb

```c
/* Simple program to generate an
    arithmetic exception */
#include <stdio.h>
#include <stdlib.h>

int divide(int a, int b)
{
 return a / b;
}

int main (int argc,char **argv)
{
 int x = 10, y = 5;
 printf("%d/%d = %d\n",x,y,divide(x, y));
 y = 0;
 printf("%d/%d = %d\n",x,y,divide(x, y));
 return 0;
}
```

```
$ qsub -I -q expressbw -l walltime
    =00:30:00 -l mem=1GB -l ncpus=1 -P
    c25
# Enable core dumps
$ ulimit -c unlimited
# Check that it is enabled
$ ulimit -a
$ gcc -g -ggdb -O0 -o div div.c
$ ./div
10/5 = 2
Floating point exception (core dumped)
$ gdb div <coredump.file>
...
# List the source code
(gdb) list
# Inspect stack status
(gdb) where
# Print value of variables
(gdb) p a
(gdb) p b
```

Agenda

OpenMP Reference Material

- http://www.openmp.org/
- *Introduction to High Performance Computing for Scientists and Engineers*, Hager and Wellein, Chapter 6 & 7
- *High Performance Computing*, Dowd and Severance, Chapter 11
- *Introduction to Parallel Computing, 2nd Ed*, A. Grama, A. Gupta, G. Karypis, V. Kumar
- *Parallel Programming in OpenMP*, R. Chandra, L.Dagum, D.Kohr, D.Maydan. J.McDonald, R.Menon

Shared Memory Parallel Programming

- Explicit thread programming is messy
  - low-level primitives
  - originally non-standard, although better since pthreads
  - used by system programmers, but $\cdots$
    $\cdots$ application programmers have better things to do!
- Many application codes can be usefully supported by higher level constructs
  - led to proprietary *directive* based approaches of Cray, SGI, Sun etc
- OpenMP is an API for shared memory parallel programming targeting Fortran, C and C++
  - standardizes the form of the proprietary directives
  - avoids the need for explicitly setting up mutexes, condition variables, data scope, and initialization

OpenMP

- Specifications maintained by OpenMP Architecture Review Board (ARB)
  - members include AMD, Intel, Fujitsu, IBM, NVIDIA $\cdots$ cOMPunity
- Versions 1.0 (Fortran '97, C '98), 1.1 and 2.0 (Fortran '00, C/C++ '02), 2.5 (unified Fortran and C, 2005), 3.0 (2008), 3.1 (2011), 4.0 (2013), 4.5 (2015)
- Comprises compiler directives, library routines and environment variables
  - C directives (case sensitive)

    `#pragma omp` *`directive_name [clause-list]`*
  - library calls begin with `omp_`

    `void omp_set_num_threads(nthreads)`
  - environment variables begin with `OMP_`

    `setenv OMP_NUM_THREADS 4`
- OpenMP requires compiler support
  - activated via `-fopenmp` (gcc) or `-openmp` (icc) compiler flags

The `parallel` Directive

- OpenMP uses a fork/join model, i.e. programs execute serially until they encounter a `parallel` directive:
  - this creates a group of threads
  - number of threads dependent on environment variable or set via function call
  - main thread becomes master with thread id 0

```
#pragma omp parallel [clause-list]
    /* structured block */
```

- Each thread executes a *structured block*

Fork Join Model

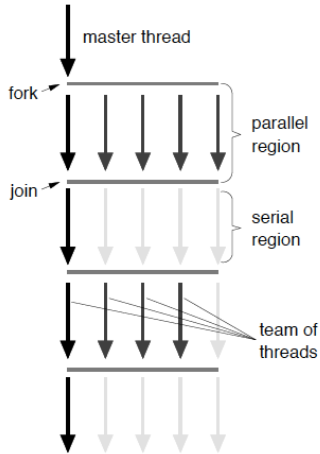

Figure 6.1: Model for OpenMP thread operations: The master thread "forks" team of threads, which work on shared memory in a parallel region. After the parallel region, the threads are "joined," i.e., terminated or put to sleep, until the next parallel region starts. The number of running threads may vary among parallel regions.

Introduction to High Performance Computing for Scientists and Engineers, Hager and Wellein, Figure 6.1

Parallel Clauses

Clauses are used to specify

- **Conditional Parallelization:** to determine if parallel construct results in creation of threads

```
if (scalar expression)
```

- **Degree of concurrency:** explicit specification of the number of threads created

```
num_threads (integer_expression)
```

- **Data handling:** to indicate if specific variables are local to thread (allocated on the stack), global, or "special"

```
private (variable_list)
shared (variable_list)
firstprivate (variable_list)
```

Compiler Translation: OpenMP to Pthreads

OpenMP code

```
int a,b;
main(){
       // serial segment
    #pragma omp parallel num_threads
        (8) private(a) shared(b)
    {
       // parallel segment
    }
       // rest of serial segment
}
```

Pthread equivalent (structured block is *outlined*)

```
int a, b;
main(){
       //serial segment
    for (i=0; i<8; i++)pthread_create
        (....., internal_thunk,...);
    for (i=0; i<8; i++)pthread_join
        (........);
       //rest of serial segment
}
void *internal_thunk(void *
    packaged_argument){
    int a;
       // parallel segment
}
```

Parallel Directive Examples

```
#pragma omp parallel if (is_parallel == 1) num_threads(8) \
                     private(a) shared(b) firstprivate(c)
```

- If value of variable `is_parallel` is one, eight threads are used
- Each thread has private copy of a and c, but shares a single copy of b
- Value of each private copy of c is initialized to value of c before parallel region

```
#pragma omp parallel reduction(+:sum) num_threads(8) default(private)
```

- Eight threads get a copy of variable `sum`
- When threads exit the values of these local copies are accumulated into the `sum` variable on the master thread
  - other `reduction` operations include *, -, &, |, ^, && and ||
- All variables are private unless otherwise specified

Hello World with OpenMP!

```c
/* Source: https://computing.llnl.gov/tutorials/openMP/samples/C/omp_hello.c */

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

int main (int argc, char *argv[])
{
    int nthreads, tid;

    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtain thread number */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }  /* All threads join master thread and disband */
}
```

Example: Computing Pi

Compute π by generating random numbers in square with side length of 2 centered at (0,0) and counting numbers that fall within a circle of radius 1

- area of square $= 4$, area of circle $= \pi r^2 = \pi$
- ratio of points in circle to outside approaches $\pi/4$

```
#pragma omp parallel default(private) shared(npoints) \
                     reduction(+: sum) num_threads(8)
{
  num_threads = omp_get_num_threads();
  sample_points_per_thread = npoints/num_threads;
  sum = 0;
  for (i = 0; i < sample_points_per_thread; i++){
    rand_x = (double) (rand_range(&seed,-1,1);
    rand_y = (double) (rand_range(&seed,-1,1);
    if ((rand_x * rand_x + rand_y * rand_y) <= 1.0) sum++;
  }
}
```

OpenMP code very simple - try writing equivalent pthread code

The `for` Worksharing Directive

- Used in conjunction with `parallel` directive to partition the `for` loop immediately afterwards

```
#pragma omp parallel default(private) shared(npoints) \
            reduction(+: sum) num_threads(8)
{
  sum = 0;
#pragma omp for
  for (i = 0; i < npoints; i++){
    rand_x = (double) (rand_range(&seed,-1,1);
    rand_y = (double) (rand_range(&seed,-1,1);
    if ((rand_x * rand_x + rand_y * rand_y) <= 1.0) sum++;
  }
}
```

  - loop index (`i`) is assumed to be private
  - only two directives plus sequential code (code is easy to read/maintain)
- Implicit synchronization at end of loop
  - can add `nowait` clause to prevent synchronization

Agenda

Message Passing Interface

- Parallel program is launched as set of identical but independent processes
  - The same program code and instructions
  - Can reside in different computing nodes
- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages
- MPI is an application programming interface (API) for communication between separate processes
- The most widely used approach for distributed parallel computing
- MPI programs are portable and scalable
- MPI standardization by `mpi-forum.org`

The MPI Standard

- MPI 1 (1994)
  - Basic point-topoint communication, collective, datatypes, etc
- MPI 2 (1997)
  - Added parallel I/O, Remote Memory Access (one-sided operations), dynamic processes, thread support, C++ bindings, ...
- MPI-2.1 (2008)
  - Minor clarifications and bug fixes to MPI-2
- MPI 2.2 (2009)
  - Small updates and additions to MPI 2.1
- MPI 3 (2012)
  - Major new features and additions to MPI Non-blocking collectives
  - Neighbour collectives improved one-sided communication interface
  - Tools interface
  - Fortran 2008 bindings

The MPI Standard

The MPI standard includes:

- Point-to-point message passing
- Collective communication
- One-sided communication and PGAS memory model
- Parallel I/O routines
- Group and communicator concepts
- Process topologies (e.g. graph)
- Environment management (e.g., timers, error handling)
- Process creation and management
- Profiling interface

The MPI API

```
MPI_Init (*argc, ***argv)
MPI_Comm_size(comm, size)
MPI_Comm_rank(comm, rank)
MPI_Send (buf, count, datatype, dest, tag, comm)
MPI_Recv (buf,  count, datatype, source, tag, comm, status)
MPI_Finalize()
```

Parameters:

- *buf* - The data that is sent/received
- *count* - Number of elements in buffer
- *datatype* - Type of each element in buf
- *source* - The rank of sender
- *dest* - The rank of receiver
- *tag* - An integer identifying the message
- *comm* - Communicator
- *status* - Information on the received message

MPI Hello World!

```c
/* C Version */
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv)
{
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf("Hello world %d of %d\n",rank,
       size);
   MPI_Finalize();
   return 0;
}
```

```
$ module load openmpi/1.10.2
$ mpicc mpi_hello.c -o mpi_hello
$ mpirun -np 8 ./mpi_hello
```

```
! Fortran Version
program mpi_hello
include 'mpif.h'
   integer rank, size, ierror;

   call MPI_INIT (ierror);
   call MPI_COMM_RANK (MPI_COMM_WORLD,
       rank, ierror);
   call MPI_COMM_SIZE (MPI_COMM_WORLD,
       size, ierror);

   print*, "Hello world", rank, "of",
       size;

   call MPI_FINALIZE(ierror);
end
```

```
$ module load openmpi/1.10.2
$ mpifort mpi_hello.f -o mpi_hello
$ mpirun -np 8 ./mpi_hello
```

MPI Point-to-Point Communication

```c
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv)
{
    int rank, size, i, number;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    if (rank==0)
    {
        for (i=1; i<size; i++) MPI_Send(&i, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    } else
    {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process %d received number %d from process 0\n", rank, number);
    }
    MPI_Finalize();
    return 0;
}
```

```
$ mpicc sendrecv.c -o sendrecv
$ mpirun -np 4 ./sendrecv
Process 1 received number 1 from process 0
Process 3 received number 3 from process 0
Process 2 received number 2 from process 0
```

MPI has a number of predefined datatypes to represent data as well as support for custom datatypes

MPI type	C type
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	8 binary digits


NCI
NCI
NCI

Feedback Survey

Please give us your feedback on topics covered today!

`https://usersupport.nci.org.au/report/training_survey`

Conclusion

```
# Thank You!
# Questions?
# Wiki:      opus.nci.org.au
# Helpdesk:  help.nci.org.au
# Email:     help@nci.org.au
```