OpenMP I

B. Estrade
<estrabd@lsu.edu>
Objectives

- get exposure to the *shared memory* programming model
- get familiar with the basics of OpenMP's capabilities
- write and compile a very simple OpenMP program
- learn of sources for more information
Shared Memory

- In an ideal world, you don't want a network connecting your compute nodes (even the fastest networks are too slow!)

- 2 or more *identical* CPUs (or processing cores) that share *main* memory are called “symmetric multi-processors”, or SMP for short.

- In contrast, “asymmetric multi-processor” systems shared main memory among different, potentially special purpose, processors (i.e., an SMP computer with Cell, GPGPU, or FPGA accelerators)

- Traditionally, “supercomputers” strive to provide as much shared memory among as many cpus as possible

- As the number of cores per processor increase, clusters are trending towards distributed shared memory clusters, meaning each node is itself a shared memory machine

- However, providing a single, large shared memory machine is always the goal for companies like: Cray, IBM, Sun, and SGI
Distributed Memory

- Distributed memory systems may consist of 2 or more compute nodes, which may also be SMP themselves.

- Non-uniform Memory Access (NUMA) systems provide for the cooperation of 2 or more processors, whereby not all processors have direct (i.e., local) access to all main memory; strictly speaking, NUMA does not provide cache coherency—it only facilitates access to memory.

- NUMA clusters (e.g., Beowulf clusters) are a fairly recent phenomenon (compared to “traditional” supercomputing techniques, fueled by the availability of inexpensive x86 hardware and networking products).

- More recently, there have been efforts to provide “virtualized” SMP environments, using “ccNUMA” techniques—“cc” standing for “cache coherency,” which is a required mechanism if one wants to provide a true shared memory environment (e.g., 3leaf systems).

- Distributed memory systems always introduce latency through the methods connecting the disjoint memory; it gets even more inefficient when attempting to make the disjoint memories and CPU caches “coherent” in order to provide a virtualized global view.
Limitations

- the number of cpus on a single compute node bounds the number of threads
- memory overhead bounds performance scaling, mainly due to “cache coherency” issues
- on distributed memory clusters (i.e., Beowulf clusters), shared memory among compute nodes must be facilitated by network communication (thus introducing latency/bandwidth overhead)
- “real” supercomputers address these issues by focusing on very high bandwidth interconnects among highly integrated shared memory nodes (Sun's Ranger, IBM's Blue Gene, etc)
Local SMP Resources

- Your own desktop/laptop – if it has 2 or more “cores”!
- LSU HPC's “Santaka” (SGI Altix SSI) provides for a true 30 CPU SMP environment (Itanium)
- LONI's Power5 systems provides 8 SMP processors per compute node
- LSU HPC's Power5 system, “Pelican,” provides 16 SMP processors per compute node
- LONI's x86 cluster, “Queen Bee,” provides 8 SMP cores per compute node
- LSU HPC's newest x86 cluster, “Philip,” provides 8 cores via 2wo 2.93 GHz Quad Core Nehalem Xeon 64-bit Processors on 37 compute nodes (2 nodes up to 96 gb RAM, 3 up to 48 gb RAM, 32 nodes up to 24 gb RAM)
- ...there are more LONI and LSU HPC systems, but these are the highlights
Shared Memory Programming

- Uses the concept of “threads”
- Each thread is related to a single process, and shares the same memory space on the SMP machine
- Threads do not communicate with explicit messages (like MPI) – they communicate *implicitly* through shared variables
- OpenMP makes creating basic *multi-threaded* programs easily
Alternatives to OpenMP

- Unified Parallel C
- Co-Array Fortran
- POSIX Threads (pthreads)
- GNU Portable Threads
- Java Threads
- Win32 Threads
- Netscape Portable Runtime
- .... and many more
OpenMP's Execution Model

- OpenMP is built around a shared memory space and related, concurrent threads – this is how the parallelism is facilitated.

- Each thread is typically run on its own processor, though it is becoming common for each CPU or “core” to handle more than one thread “simultaneously”; this is called “hyper-threading” in x86-land and “symmetric multi-threading” in POWER architecture land (e.g., IBM/AIX).

- Thread (process) communication is implicit, and uses variables pointing to shared memory locations; this is in contrast with MPI, which uses explicit messages passed among processes.

- OpenMP simply makes it easier to manage the traditional fork/join paradigm using special “hints” or directives given to an OpenMP enabled compiler

- These days, most major compilers do support OpenMP directives for most platforms (IBM's XL suite, Intel, even GCC ≥ 4.2).
OpenMP's Execution Model

- Any parallel program may have its execution expressed as a directed acyclic graph.
- A fork is when a single thread is made into multiple, concurrently executing threads.
- A join is when the concurrently executing threads synchronize back into a single thread.
- During the overall execution, threads may communicate with one another through shared variables.
- OpenMP programs essentially consist of a series of forks and joins.
A Simple OpenMP Example

```c
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[]) {
  int id, nthreads;
  #pragma omp parallel private(id)
  {
    id = omp_get_thread_num();
    printf("hi from %d\n", id);
    #pragma omp barrier
    if ( id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("%d threads say hi!\n",nthreads);
    }
  }
  return 0;
}
```

Output using 5 *threads*:

```
hi from 0
hi from 4
hi from 2
hi from 3
hi from 1
5 threads say hi!
```
Trace of Execution

0

hi from 0

W

5 threads say hi!

0

1

hi from 1

W

1 != 0

1

2

hi from 2

W

2 != 0

2

3

hi from 3

W

3 != 0

3

4

hi from 4

W

4 != 0

4

W = wait for all threads before progressing.
Controlling Data Access

- Data access refers to the sharing of variables among threads
- **shared(var1, var2, ..., varN)**
  - specifies variables that may *safely* be shared among threads
- **private(var1, var2, ..., varN)**
  - specifies *uninitialized* variables that are only accessible to a specific thread
- **default(shared|private|none)**
  - allows one to define the default scoping of the threads' variables
Data Access Examples

**PRIVATE**

**C/C++**

```c
int id, nthreads;
    #pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
    }
```

**Fortran**

```fortran
integer:: id, nthreads
    !$omp parallel private(id)
    id = omp_get_thread_num()
```

**SHARED**

**C/C++**

```c
int id, nthreads, A, B;
    A = getA();
    B = getB();
    #pragma omp parallel
    #pragma omp default(private)
    #pragma omp shared(A,B)
    {
        id = omp_get_thread_num();
    }
```

**Fortran**

```fortran
integer:: id, nthreads, A, B
    call getA(A)
    call getB(B)
    !$omp parallel
    !$omp default(private)
    !$omp shared(A,B)
    id = omp_get_thread_num()
```
Variable Initialization

- Variable initializations controls how private and shared variables get assigned an initial value.

- `firstprivate(var1, ..., varN)`
  - allows a variable to be globally initialized, but private to each thread once program execution has entered the parallel section

- `threadprivate(/BLOCK1/, ..., /BLOCKN/)`
  - allows named globally defined COMMON blocks (in Fortran) to be private to each thread, but global within the thread itself
Reductions

- Reductions allow for a variable used privately by each thread to be aggregated into a single value;
- Variable specification implies a private variable (error if specified in both)
- Always initialize your reduced variables properly!
- Reduction operations (C/C++):
  - Arithmetic: + – * /
  - Bitwise: & ^ |
  - Logical: && ||
- Specified in main OMP clause, or a work sharing construct:

```c
int i = 0; //important!
#pragma omp parallel reduction(+:i)
{
    // ... code
}
```
Trace of Reduction Execution

i = 0 → 0 → i₀ = 1
i = 1 → 1 → i₁ = 1
i = 2 → 2 → i₂ = 1
i = 3 → 3 → i₃ = 1
i = 4 → 4 → i₄ = 1

reduction(+:i)

i = i₀ + i₁ + i₂ + i₃ + i₄

i = 5
Reduction Limits & Traps!

- Reduced variable must be a scalar, i.e., a single value (no arrays, data structures, etc)
- (trap!) Initialized value *does* affect the outcome; for example, initializing a variable to 0 (zero) will make the outcome of an aggregate multiplication 0 (zero) as well!

```c
int i = 0;
...reduction(*:i)
```

Means: \(0 \times i_0 \times i_1 \times \ldots \times i_n\), which is \(\equiv 0\)
Synchronization

- **barrier**
  - Provides a place in the code for *all* threads to wait for the others
  - No thread may progress until all threads have made it to this point
Trace of Execution

W = wait for all threads before progressing.

0
hi from 0
W
5 threads say hi!

1
hi from 1
W
1 != 0

2
hi from 2
W
2 != 0

3
hi from 3
W
3 != 0

4
hi from 4
W
4 != 0

B. Estrade <estrabd@lsu.edu>, Louisiana Optical Network Initiative/HPC @ LSU
(some) OpenMP Runtime Routines

- `omp_get_num_threads`
- `omp_set_num_threads`
- `omp_in_parallel`
OpenMP Environmental Variables

- **OMP_NUM_THREADS**
  - required, informs execution of the number of threads to use

- **OMP_SCHEDULE**
  - The OMP_SCHEDULE environment variable applies to PARALLEL DO and work-sharing DO directives that have a schedule type of RUNTIME.

- **OMP_DYNAMIC**
  - The OMP_DYNAMIC environment variable enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions
Compiling and Executing

- **IBM p5 575s:**
  - xlf_r, xlf90_r, xlf95_r, xlc_r, cc_r, c89_r, c99_r, xlc128_r, cc128_r, c89_128_r, c99_128_r

  ```bash
  %xlc_r -qsmp=omp test.c && OMP_NUM_THREADS=5 ./a.out
  ```

- **x86 Clusters:**
  - ifort, icc

  ```bash
  %icc -openmp test.c && OMP_NUM_THREADS=5 ./a.out
  ```
The “Hard” Part

• The difficult aspect of creating a shared memory program is translating what you want to do into a multi-threaded version.

• It is even more difficult to make this multi-threaded version optimally efficient.

• The most difficult part of this is verifying that your multi-threaded version is correct, and that there are no issues with shared variables over writing one another (race conditions, dead locks, etc)

• Program verification and detecting/debugging race conditions (and other run time issues) are beyond the scope of this tutorial, and they will be covered in future talks on advanced OpenMP.
Lab 1

- Using the following description, design on paper (visually) how a multi-threaded version may look.

- Description:
  - for a threaded application with \( N \) threads, return the number \( N^2 \)
  - each thread may use only a single private (or firstprivate) variable
  - this variable must be reduced using addition, so all threads must contribute to the answer
Lab 1- Solution

- declare private variable, `int i`
- for each thread, set `i = #threads`
- reduce `i` to the sum of all threads' `i` values
Lab 2

- Implement the solution to Lab 1 in either C or Fortran
- Compile, then run the code using the examples provided in the presentation
Lab 2 – A Solution in C

```c
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    int c = 0; // required on AIX (xlc_r)
    #pragma omp parallel reduction(+:c)
    {
        c=omp_get_num_threads();
    }
    printf("%d\n",c);
    return 0;
}
```

On AIX, p5 575

```bash
%xlc_r -qsmp=omp test.c && OMP_NUM_THREADS=5 ./a.out
25
```

On Linux, x86

```bash
%icc -openmp test.c && OMP_NUM_THREADS=5 ./a.out
test.c(5) : (col. 3) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
25
```
Lab 2 – A Solution in Fortran

```fortran
program numthreadssquared
use omp_lib
integer::c=0
   !$omp parallel default(private) reduction(+:c)
       c = omp_get_num_threads()
   !$omp end parallel
   WRITE (*,*) c
end program
```

On Ducky (AIX, p5 575)
```
xlf90_r -qsmp=omp test.f90 && OMP_NUM_THREADS=5 ./a.out
25
```

On Eric (Linux, x86)
```
ifort -openmp test.f90 && export OMP_NUM_THREADS=5 ./a.out
25
```
Additional Resources

- https://docs.loni.org/wiki/Using_OpenMP
- help: otrs@loni.org, sys-help@loni.org
- https://docs.loni.org/wiki/Introduction_to_OpenMP
- http://kallipolis.com/openmp/1.html
References

- https://docs.loni.org/wiki/Using_OpenMP
- http://www.nersc.gov/nusers/help/tutorials/openmp/
- http://www.llnl.gov/computing/tutorials/openMP/
- http://publib.boulder.ibm.com