An Introduction to OpenMP: Getting The Most Out of Multicore

Ben Cumming
CSCS
July 2013

Before We Start

- The source code for the exercises and a copy of these slides can be accessed on Todi
  - To get a copy on your local path

```
> ssh ela.cscs.ch
> ssh todi
> cp -Rv /project/csstaff/courses/CSCS_USI_School/Day5/openmp .
> cd openmp
```

  - A script to set up the GNU compiler is provided, so that you can compile and run the examples

```
> source setup.sh
> CC -fopenmp hello_world.cpp
> salloc -N 1
> export OMP_NUM_THREADS=8
> aprun -d 8 ./a.out
```

Before We Start

- The OpenMP website is a great source of information
  - Lots of tutorials and examples from beginner to advanced.
  - The standard, which is easy to read (for a standard!)
  - Quick reference guides

openmp.org

A collection of simple examples that are a good reference

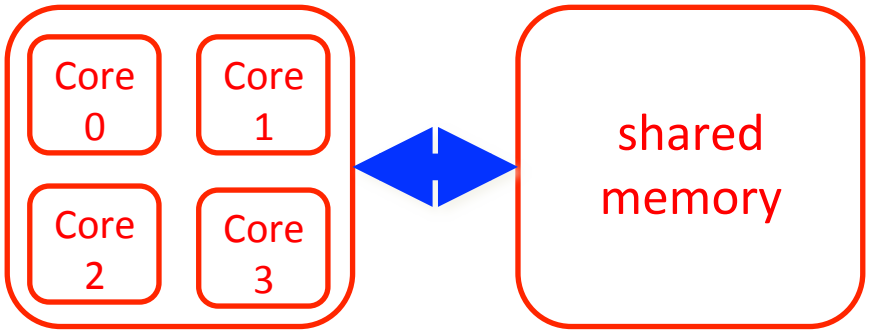
users.abo.fi/mats/PP2012/examples/OpenMP/

The Story Thus Far: MPI

- Most applications in HPC today use "flat" MPI parallelization
- Each MPI process has a separate memory space, with explicit communication between processes via the MPI library
  - `MPI_Send()`, `MPI_Recv()`, `MPI_Gather()`, etc.
- This makes sense when there are few cores on each socket/node, with separate memory spaces
  - The abstraction of the MPI model matches the underlying hardware

Multicore, Shared Memory and the Limitations of MPI

- The number of cores on multi-core processors is increasing
  - The Interlagos processors on Todi have 16 cores per socket
  - This trend will continue. For example, Intel's new Xeon Phi (MIC) architecture has 50+ cores on a single socket
- Cores on a multi-core socket have a shared memory space
- The flat MPI approach with separate memory spaces starts to break down as the number of cores per socket increases
- Performing separate domain decomposition for each core increases the ration of communication to computation
- New codes are being written with hybrid MPI-OpenMP, and existing flat MPI codes are having OpenMP directives added



What is OpenMP?

- The OpenMP standard specifies a set of compiler directives and library routines for writing parallel shared memory applications Fortran, C and C++ codes.
  - Supported by all compilers used in HPC, including GNU, Intel, Cray, PGI, IBM and Pathscale.
- OpenMP compilers allows programmers to tell the compiler where and how to parallelize using compiler directives
  - The programmer doesn't have the difficulties of dealing with threads directly, as would be the case with a low-level threading library such as pthreads.

OpenMP Compiler Directives

- In C and C++ parallel regions are started with directives of the form #pragma omp parallel
  - Applied to the block that follows, enclose with {curly braces}.

C/C++
```
#pragma omp parallel
{
  … executed in parallel
}
```

- In Fortran parallel regions are enclosed with specially-formatted comments

Fortran
```
!$omp parallel
  … executed in parallel
!$omp end parallel
```

The focus here will be on using OpenMP in C/C++. The Equivalent Fortran commands are very similar: refer to the documentation

The Fork and Join Model

- OpenMP uses the fork and join model.
- An OpenMP program starts with a master thread.
  - FORK: A team of parallel worker threads is started at the start of each parallel block.
  - The block is executed in parallel by each thread in the team.
  - JOIN: The worker threads are synchronized at the end of the block, then join the master thread.
- Threads in a block are numbered in the range [0:N-1], where N is the number of threads
- The master thread is always numbered 0

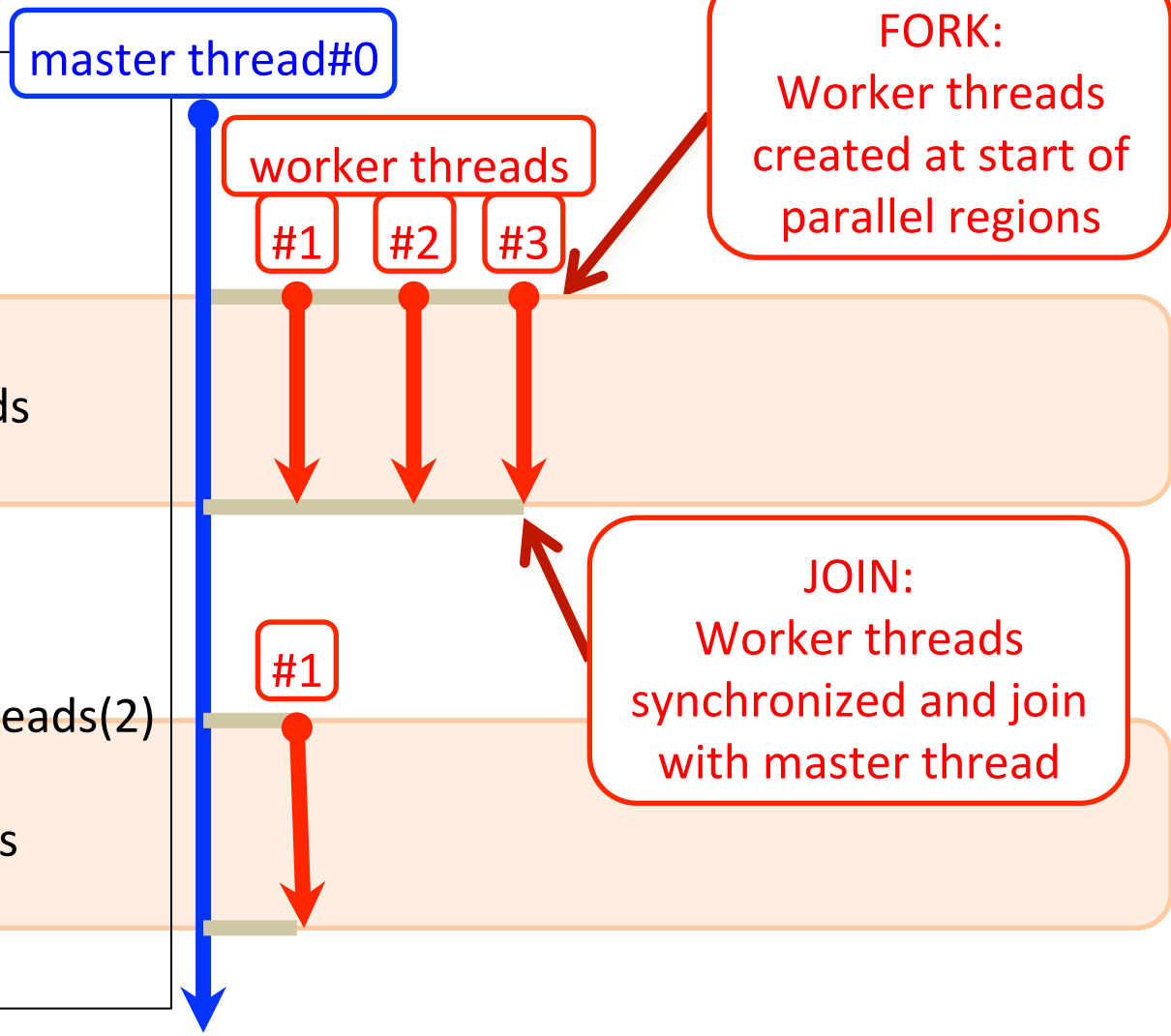
The Fork and Join Model

```
// … serial work

#pragma omp parallel
{
   // parallel work with 4 threads
}

// … serial work

#pragma omp parallel num_threads(2)
{
   // paralle work with 2 threads
}
```



Compiling OpenMP

- Most compilers require a compiler flag to enable OpenMP compilation
  - Without the flag, the OpenMP directives are ignored, producing a sequential application

```
Cray    : on by default for –O1 and greater, disable with "-h noomp"
Intel   : off by default, enable with "-openmp"
GNU     : off by default, enable with "-fopenmp"
PGI     : off by default, enable with "-mp"
```

Running OpenMP Applications

- The default number of threads is set via an environment variable OMP_NUM_THREADS
- When executing the application, we want to have at least one core per thread
  - Otherwise multiple threads have to share resources on a core.
  - On Cray systems (like Todi) the number of cores is specified with the -d flag for aprun

```
> CC -fopenmp –o app app.cpp
> export OMP_NUM_THREADS=8
> aprun –d 8 ./app
… or:
> aprun –d $OMP_NUM_THREADS ./app
```

Before We Start

- The source code for the exercises and a copy of these slides can be accessed on Todi
  - To get a copy on your local path

```
> ssh ela.cscs.ch
> ssh todi
> cp -Rv /project/csstaff/courses/CSCS_USI_School/Day5/openmp .
> cd openmp
```

  - A script to set up the GNU compiler is provided, so that you can compile and run the examples

```
> source setup.sh
> CC -fopenmp hello_world.cpp
> salloc -N 1
> export OMP_NUM_THREADS=8
> aprun -d 8 ./a.out
```

Exercise 1: Compiling and Running

- open the test program `hello_world.cpp`
  - What do you expect the output of the program to be?
- Compile `hello_world.cpp`

```
> source setup.sh
> CC -fopenmp hello_world.cpp
```

- Run the program

```
> export OMP_NUM_THREADS=8
> aprun -d 8 ./a.out
...
```

  - Does it produce the output you expected?

Library Calls

- The OpenMP runtime library
  - `omp_get_thread_num()`
    - returns unique integer id for the current thread
  - `omp_get_num_threads()`
    - returns number of threads in the current region
  - `omp_get_max_threads()`
    - returns the default number of threads in parallel regions (corresponds to the environment variable OMP_NUM_THREADS)
  - `omp_get_wtime()`
    - returns a double precision wall time in seconds.
- There are many more however these are the ones you use 99% of the time.
- You have to include a header to use the libraries:

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

Synchronization

- OpenMP provides directives that can be used to synchronize thread execution inside a parallel block. These include:
  - `#pragma omp master`
  - `#pragma omp critical`
  - `#pragma omp single`
  - `#pragma omp barrier`
- WARNING: synchronization can have a detrimental impact on performance.
  - Try to write your code to avoid the need for synchronization

`master` directive

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp master
    std::cout << "I am thread " << tid << std::endl;
}
```

- Only the master thread will execute the code
  - Recall that the master thread always has id==0
- In this case only one message will be printed by thread 0

`single` directive

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp single
    std::cout << "I am thread " << tid << std::endl;
}
```

- Executed only by the first thread to reach the block
  - This will vary from one execution to the next
- Again one message is printed, but the message may differ between executions.

`critical` directive

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp critical
    std::cout << "I am thread " << tid << std::endl;
}
```

- Only one thread in the team can be inside a critical section at a time
  - All threads wait at start of critical section until their turn
- If there are 8 threads, then 8 messages will be printed in the example above.
  - They won't overwrite one-another because there is one thread at a time writing to the screen

`barrier` directive

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp barrier
    std::cout << "I am thread " << tid << std::endl;
}
```

- All threads wait at the barrier until the last thread has arrived at the barrier
- In the example above no thread will start output before all the other threads have finished the call to omp_get_thread_num()
  - But the output will still be random!

Exercise 2

- Go back to the hello_world.cpp example and add the appropriate synchronization directive
  - Do you see the expected behavior now?
- Look at the sum_threads.cpp source code
  - What do you think the expected output of this program is?
  - When you run the example, do you get the expected results?
  - Can you add a synchronization directive to get the expected result?

```
> source setup.sh
> CC -fopenmp hello_world.cpp
> export OMP_NUM_THREADS=8
> aprun -d 8 ./a.out
```

Shared Memory Model

- OpenMP uses a shared memory model
- All threads "see" the same memory/variables
- The results of computation where multiple threads try to read from/write to the same variable at the same time are undefined
  - See the sum_threads.cpp example in the previous code for an example of such undefined behavior.

Variable Scoping

- There are two basic choices for scoping variables inside parallel regions
  - shared: all threads read and write from the same variable.
    - Variables are shared by default.
    - WARNING: take care when writing to shared variables.
  - private: each thread gets its own private copy of the variable.

```
int tid;
const int num_threads = omp_get_max_threads();
#pragma omp parallel private(tid) shared(num_threads)
{
  tid = omp_get_thread_num();
  #pragma omp critical
  std::cout << "Hello World from thread " << tid
            << " of " << num_threads << std::endl
}
```

Private Example

Problem: all threads try to write to variable tid

```
int tid;
#pragma omp parallel
{
  tid = omp_get_thread_id();
  std::cout << “Hello World from thread ” << tid << std::endl
}
```

Solution 1: use the private clause

```
int tid;
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_id();
  std::cout << “Hello World from thread ” << tid << std::endl
}
```

Solution 2: make the variable’s scope local to the block

```
#pragma omp parallel
{
  int tid = omp_get_thread_id();
  std::cout << “Hello World from thread ” << tid << std::endl
}
```

Only for C/C++

Exercise 3

- You now have all the tools you require to fix the hello_world.cpp program

Work Sharing: for loops

- A very common target for parallelization is for loops like the following:

```
void vec_add(double* x, double* y, int n){
  for(int i=0; i<n; i++)
    x[i] += y[i];
}
```

- One approach to solve this with what we have learnt so far is:

```
void vec_add(double* x, double* y, int n){
  #pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    int work = n/num_threads;
    int s = tid*work;
    int e = (tid==num_threads-1) ? n : s+work;
    for(int i=s; i<e; i++)
      x[i] += y[i];
  }
}
```

What a mess!
And error prone too:
are you sure this will work if n<num_threads?

parallel for

- OpenMP provides a directive for for loops:

Loop variable `i` is private by default

```
void vec_add(double* x, double* y, int n){
  int i;
  #pragma omp parallel
  {
    #pragma omp for
    for(i=0; i<n; i++)
      x[i] += y[i];
  }
}
```

- OpenMP performs partitioning of the loop for you
- Compact form that combines parallel and for:

```
void vec_add(double* x, double* y, int n){
  int i;
  #pragma omp parallel for
  for(i=0; i<n; i++)
    x[i] += y[i];
}
```

Example: Vector Normalize

- Open `vector_normalize.cpp`
  - Your job is to write a parallel version of the function `normalize_vector()`
  - This routine first finds the 2-norm of v
  - Then scales the vector

$$\|v\|_2 = \sqrt{\sum_{i=1}^{n} v_i}$$

$$v_i \leftarrow \frac{v_i}{\|v\|_2} \quad \forall i$$

- Open `dot.cpp`
  - This program computes the dot product of two vectors
  - Understand the code, and add OpenMP directives to parallelize it

Reductions

- Reduction operations are very common with the form

```
Intialize a
for( i=0; i<N; i++ )
  a = a op expr;
```

for example

```
double x[N];
double sum = 0.;
for( i=0; i<N; i++ )
  sum = sum + x[i];
```

- OpenMP provides `reduction(op:list)` for such operations with the following criteria
  - a is a scalar variable in list
  - expr is a scalar expression that does not reference a
  - Only certain expressions are allowed (+,*,-,/, and binary operations)

```
double x[N];
double sum = 0.;
#pragma omp parallel for reduction(+:sum)
for( i=0; i<N; i++ )
  sum = sum + x[i];
```

Exercise

- Can you change your solution to the dot product example in `dot.cpp` to use a parallel reduction?
  - Use the test.sh script to see how it scales from 1 to 8 threads
- Extension: can you apply everything that we have learnt to the example in `pi.cpp`, which computes π using the trapezoidal rule.
  - Use the test.sh script to see how it scales from 1 to 8 threads.
  - Is its scaling better or worse than dot.cpp. Why do you think this is?

The End