

# LAB : OpenMP Stampede



TACC

Texas Advanced Computing Center  
The University of Texas at Austin

September 19, 2013

# Introduction

## What you will learn

- How to compile Code (C and Fortran) with OpenMP
- How to parallelize code with OpenMP
  - Use the correct header declarations
  - Parallelize simple loops
- How to effectively hide OpenMP statements

## What you will do

- Modify example code **READ the CODE COMMENTS**
- Compile and execute the example
- Compare the run-time of the serial codes and the OpenMP parallel codes with different scheduling methods

# Accessing Lab Files

- Log on to **Stampede** using your train## account.
- Untar the file lab\_OpenMP.tar file (in ~train00).
- The new directory (lab\_openmp) contains sub-directories for exercises 1-3.
- cd into the appropriate subdirectory for an exercise.

You will be assigned this number.

```
ssh train##@stampede.tacc.utexas.edu
tar -xvf ~train00/lab_OpenMP.tar
cd lab_openmp
```

# Running on compute nodes Interactively

YOU CAN DO THE LAB WITHOUT RUNNING ON COMPUTE NODES!!!

- You can compile\* and execute your code on the login node (login1); or you can use one of the compute nodes (c###-###). Here is how to do that.

1.

```
login2$ srun -t 60 -p development -n 16 --pty /bin/bash -l
--> Verifying availability of home dir (/home1/00770/
milfeld)...OK
--> Verifying access to desired queue (devel)...OK
c559-802$
```

1hr, development queue, 16 cores.  
If asked, put an account (-A ...) here.

2. Once you have a command prompt, you are ready to go (you own the node- it isn't shared with any other user). E.g. compile and execute - note the login prompt is the node name.

(this is only an example)

```
c559-001% ifort hello.f90 -o hello
c559-001% ./hello
```

# Compiling

- All OpenMP statements are activated by the OpenMP flag:
  - Intel compiler: `icc/ifort -openmp -fpp source.<c,f90>`
  - PGI compiler: `pgcc/pgf90 -mp source.<c,f90>`
- **On Stampede we will be using the Intel compiler**
- Compilation with the OpenMP flag (`-openmp`):  
Activates OpenMP comment directives (...):
  - Fortran: `!$OMP ...`
  - C: `#pragma omp ...`Enables the macro named `_OPENMP`
  - `#ifdef _OPENMP` evaluates to true  
(Fortraners: compile with `-fpp`)Enables "hidden" statements (Fortran only!)
  - `!$ ...`

# Exercises – Lab 1

- Exercise 1: Kernel check  
f\_kernel.f90/c\_kernel.c  
Kernel of the calculation (see exercise 2)  
Parallelize one Loop
- Exercise 2: Calculation of  $\pi$   
f\_pi.f90/c\_pi.c  
Parallelize one Loop with a reduction
- Exercise 3: daxpy ( $a * x + b$ )  
f\_daxpy.f90/c\_daxpy.c  
Parallelize one Loop

# Exercise I: $\pi$ Integration Kernel Check

- cd exercise\_1
- Codes: f\_kernel.f90/c\_kernel.c
- Number of intervals is varied (Trial loop)

## Kernel

**Trial Loop: *itrial***  
**Calculation of *n* and *deltax***  
**Loop over *i***  
**make sure *area* >0.0**

- 1 Parallelize the Loop over *i* :  
Use **omp parallel do/for**  
Set appropriate variables to private
- 2 Compile with:  
**ifort -openmp f\_kernel.f90**  
**icc -openmp c\_kernel.c**

- 1 Parallelize the code
- 2 Compile
- 3 Run with 1, 2, 4, 8,12, 16 threads  
e.g. export OMP\_NUM\_THREADS=4  
. /a.out
- 4 Compare the timings

- ✓ Timings decrease with more threads.
- ✓ If you execute with more threads than cores the timings will NOT decrease. Why?

# Exercise II: $\pi$ Integration

- cd exercise\_2
- Codes: f\_pi.90/c\_pi.c
- Number of intervals is varied (Trial loop)

## $\pi$ calculation

**Trial Loop:  $i_{\text{trial}}$**   
**Calculation of  $n$  and  $\text{deltax}$**   
**Loop over  $i$**

- Parallelize the code

### 1 Complete OpenMP statements

- Initialization
- omp get max threads
- omp get thread num

1 Parallelize the Loop over  $i$  :  
Use **omp parallel do/for**  
with the default(none) clause

2 Compile with:  
**make f\_pi**  
or  
**make c\_pi**

3 Run with 1, 2, 4, 8, 12 threads

e.g. export OMP\_NUM\_THREADS=4  
./c\_pi or ./f\_pi

4 Compare timings

- ✓ Timings decrease with more threads
- ✓ What is the scale up at 12 threads?



# Exercise III: daxpy

- cd exercise\_3
- Codes: f\_daxpy.f90/c\_daxpy.c
- Number of intervals is varied (Trial loop)

daxpy

**Trial Loop: *itrial***  
**Loop over *i***

1 Parallelize the Loop over ***i*** :  
Use **omp parallel do/for**  
with the default(none) clause

2 Compile with:  
**make f\_daxpy**  
or  
**make c\_daxpy**

3 Run with 1 and 12

4 Compare timings

- Why is performance only doubled?

- Parallelize the code

- 1 complete OpenMP statements
  - Initialization
  - **omp get max threads**

- ✓ Hint: Parallel performance can be limited by memory bandwidth– what is happening for every daxpy operation? (Is there cache reuse?)

# Exercises – Lab 2

- Exercise 4: Update from neighboring cells (2 arrays)  
f\_neighbor.f90/c\_neighbor.c  
Create a **Parallel Region**  
Use a **Single** construct to initialize  
Use a **Critical** construct to update  
Use **dynamic** or **guided** scheduling
- Exercise 5: Update from neighboring cells (same array)  
f\_red\_black.f90/c\_red\_black.c  
Parallelize 3 individual loops, use a reduction  
Create a **Parallel Region**  
Combine loops 1 and 2  
Use a **Single** construct to initialize

# Exercise IV: Neighbor Update; Part 1

- cd exercise\_4
- Codes: f\_neighbor.f90/c\_neighbor.c

## neighbor update

### Parallel Region

Initialization: j\_update

### Parallelize loop i

Loop i

    Loop j

        increment j\_update

        Loop k

            b is calculated from a

- Try different schedules:  
**static, dynamic, guided**

Compile with: **make f\_neighbor**  
**make c\_neighbor**

- Parallelize the Loop over **i**
- Use a **single** construct for initialization
- Would a **master** construct work, too?
- Use **critical** for increment of **j\_update**
- Use omp parallel do/for with the default(none) clause

# Exercise IV: Neighbor Update; Part 2

## neighbor update

Parallel Region

Initialization: `j_update`

Parallelize loop i

Loop i

Loop j

`single` or `master`

increment `j_update`

`end single` or `end master`

Loop k

b is calculated from a

Compile with: `make f_neighbor`  
`make c_neighbor`

- Change the `single` to a `master` construct
- Run with 1 **and** 12 threads
- How does the number of `j_update` change?

# Exercise V: Red-Black Update; Part 1

- cd exercise\_5
- Codes: f\_red\_black.f90/c\_red\_black.c
- make a **copy** and create f\_red\_black\_v1.f90/c\_read\_black\_v1.c

## red-black update

Iteration Loop: `niter`

Loop: Update even elements

Loop: Update odd elements

Initialize `error`

Loop-summation: `error`

Compile with: `make f_red_black_v1`  
`make c_red_black_v1`

## Part 1

- Try `static` scheduling

- Parallelize each loop separately
- Use `omp parallel do/for` for the ‘‘Update’’-loops
- Use a `reduction` for the ‘‘Error’’-calculation with the `default(none)` clause

# Exercise V: Red-Black Update; Part 2

- cd exercise\_5
- Start from **version 1**
- Codes: f\_red\_black.f90/c\_red\_black.c
- make a **copy** and create **f\_red\_black\_v2.f90/c\_read\_black\_v2.c**

## red-black update

Iteration Loop: **niter**

Loop:

Update even and odd el.

Initialize **error**

Loop-summation: **error**

- Try **static** scheduling

Compile with: **make f\_red\_black\_v2**  
**make c\_red\_black\_v2**

## Part 1

- Can the loops be combined?
- Why can the “update” loops be combined?
- Why can the “error” loop not be combined?
- Task:

Combine the “update” loops

# Solution V: Red-Black Update; Part 2

## red-black update

```
!*** Update even elements
do i=2, n, 2
  a(i) = 0.5 * (a(i) + a(i-1))
enddo
!*** Update odd elements
do i=1, n-1, 2
  a(i) = 0.5 * (a(i) + a(i+1))
enddo
```

## red-black update

```
!*** Update even and odd
!*** in one loop
do i=2, n, 2
  a(i) = 0.5 * (a(i) + a(i-1))
  a(i-1) = 0.5 * (a(i-1) + a(i))
enddo
```

# Exercise V: Red-Black Update; Part 3

- cd exercise\_5
- Start from **version 2**
- Codes: f\_red\_black.f90/c\_red\_black.c
- make a **copy** and create f\_red\_black\_v3.f90/c\_read\_black\_v3.c

## red-black update

```
Iteration Loop: niter
  parallel region
  Loop:
    Update even and odd el.
  single
  Initialize error
end single
Loop-summation: error
end parallel region
```

Compile with: **make f\_red\_black\_v3**  
**make c\_red\_black\_v3**

### Part 1

- Make **one parallel region** around both loops: “update” and “error”.
- The initialization of error has to be done by one thread
- Use a **single** construct
- Would a **master** construct work?



# Exercise VI: Orphaned work-sharing

- cd exercise\_6
- Codes: f\_print.f90/c\_print.c
- make a **copy** and create f\_red\_black\_v3.f90/c\_read\_black\_v3.c

## Orphaned work-sharing

```
parallel region
  print 1
parallel Loop
  print 2
call printer_sub
master
  print 5

subroutine print_sub
parallel Loop
  print 3
Loop
  print 4
```

Compile with: **make f\_print**  
**make c\_print**

- Inspect the code
- Run with 1, 2, ... threads
- Explain the output
- How often are the 5 print statements executed?
- Why?