

LAB : OpenMP Lonestar



Kent Milfeld, Lars Koesterke

Texas Advanced Computing Center
The University of Texas at Austin

February 6th , 2012

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 (Lonestar) 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##@lonestar.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. Login in a separate window, cd to your working directory, and execute the command:

```
login1% idev
```

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)

```
c201-112% ifort hello.f90 -o hello
```

```
c201-112% ./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 Lonestar 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 π
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: π Integration Kernel Check

- cd exercise_1
- Codes: f_kernel.f90/c_kernel.c
- Number of intervals is varied (Trial loop)

Kernel

Trial Loop: i_{trial}
Calculation of n and deltax
Loop over i
make sure $\text{area} > 0.0$

- 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

- 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

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

Exercise II: π Integration

- cd exercise_2
- Codes: f_pi.90/c_pi.c
- Number of intervals is varied (Trial loop)

π calculation

Trial Loop: i trial
Calculation of n and 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

- 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
- Try **static** scheduling

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?