DOCTYPE html>

Agenda

10:00 - 11:30: Brief introduction to Parallel Computing with OpenMP - Session 1

- OpenMP Introduction (45 min + Hands On)
- OpenMP Data sharing (45 min + Hands On)

Reminder - Cyclone Enviroment

Compute Nodes

- 17 CPU compute nodes
  - equiped with Intel(R) Xeon(R) Gold 6248 CPU, 40 cores
- 16 GPU compute nodes
  - equiped each with 4 Nvidia Volta GPU
- see for more info's

  ```
  [cn01 ~] less /proc/cpuinfo
  ```

- *default*

  ```
  [front01 ~]$ which gcc
  /usr/bin/gcc
  [front01 ~]$ gcc --version
  gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-39)
  Copyright (C) 2015 Free Software Foundation, Inc.
  [front01 ~]$ gfortran --version
  GNU Fortran (GCC) 4.8.5 20150623 (Red Hat 4.8.5-39)
  Copyright (C) 2015 Free Software Foundation, Inc.
  ```

  check out `module avail` for more

Reminder - Preparation

vim editor

- Open file

  ```
  [front01 ~] vim example.txt
  ```

- Use `Insert` to switch between "Insert"- and "Replace"-mode
- To exit and write open command-line via: `Ctr + c`
  - save and exit via `:wq` and `Enter`
  - exit without save via

    `:q` and `Enter`
  - write content to file "New.txt" via

    `:w New.txt` and `Enter`

slurm basics

- submit job via script `submit_script.sh`:
  - `[front01 ~] sbatch submit_script.sh Submitted batch job "JOB ID"`
- check job status:
  - `[front01 ~] squeue -u $USER`
- cancel job:
  - `[front01 ~] scancel "JOB ID"`

Motivation - Why OpenMP ?

- OpenMP parallelized program can be run on your many- core workstation or on a node of a cluster
- Enables to parallelize one part of the program without re-building your software
    - Get some speedup with a limited investment in time
    - Efficient and well scaling code still requires effort
- Serial and OpenMP versions can easily co-exist
- Hybrid programming: OpenMP parallelization on top of MPI-tasks
    - e.g. can enable to optimize the on-node performance

OpenMP

- Multi-threaded shared memory parallelization
    - Use multiple threads that share a common memory address space
- Fortran 77/9X/03 and C/C++ are supported
- Pragma-based, i.e. uses directives rather than functions (mostly)
- Also an API, i.e. some simple functionality through function calls

Three components of OpenMP

- Compiler directives, i.e., language extensions for shared memory parallelization
  - Syntax: *directive*, **construct**, `clauses`
  - C/C++: *#pragma* **omp parallel** `shared(data)`
  - Fortran: *!$omp* **parallel** `shared(data)`
- Runtime library routines (Intel: libiomp5, GNU: libgomp)
  - Conditional compilation to build serial version
- Environment variables
  - Specify the number of threads, thread affinity,
  - like `OMP_NUM_THREADS`, other are important in Hybrid parallelization approaches
  - see for more https://www.ibm.com/support/knowledgecenter/SSXVZZ_13.1.5/com.ibm.xlcpp1315.lelinux.doc/compiler_ref/ruomprun.html

OpenMP introduction

- Starts with a single thread
- Define parallel regions
- More than one parallel regions can be defined
- So-called fork-join concept

```
int
main()
{
    ...
    work to do outside parallel region
    ...
    #pragma omp parallel
    {
        ...
        work to do in parallel
        ...
    }
    ...
    more work outside parallel region
    ...
    return 0;
}
```

OpenMP introduction

Parallel regions:

- No jumping in or out (e.g. `goto`)
- No branching in or out (e.g. inside `if-else` block)
- A thread can terminate the program from within a block

OpenMP OpenMP runtime takes care of

- thread management, forking, joining, etc.
- Specify number of threads via environment variable `OMP_NUM_THREADS`

```
int
main()
{
    ...
    work to do outside parallel region
    ...
    #pragma omp parallel
    {
        ...
        work to do in parallel
        ...
    }
    ...
    more work outside parallel region
    ...
    return 0;
}
```

OpenMP introduction

Parallel regions:

- No jumping in or out (e.g. `goto`)
- No branching in or out (e.g. inside `if-else` block)
- A thread can terminate the program from within a block

OpenMP OpenMP runtime takes care of

- thread management, forking, joining, etc.
- Specify number of threads via environment variable `OMP_NUM_THREADS`

parallel region

- use: `omp_get_thread_num()` and `omp_get_num_threads()`

```
int
main()
{
    ...
    work to do outside parallel region
    ...
    #pragma omp parallel
    {
        ...
        work to do in parallel
        ...
    }
    ...
    more work outside parallel region
    ...
    return 0;
}
```

```
#include <omp.h>
...
/* Return a unique thread if for each thre
int tid = omp_get_thread_num();
...
/* Return the total number of threads */
int nth = omp_get_num_threads();
```

OpenMP introduction

Compiling and running

- Using GNU compile via:

```
[front01 ~]$ cc -fopenmp program.c -o program
```

- Note that, depending on the compiler, the `#pragma` may not cause an error if you accidentally omit `-fopenmp` . You will just produce a scalar code.
- On your local resources via:

```
[PC ~]$ export OMP_NUM_THREADS=10
[PC ~]$ ./program
```

  or

```
[PC ~]$ OMP_NUM_THREADS=10 ./program
```

- On Cylcone use submit-scripts via `slurm`

```
[front01 ~] more submit_script.sh
#!/bin/bash
#SBATCH --nodes=1 # 1 node
#SBATCH --ntasks-per-node=10 # Number of tasks to be invoked on each node
#SBATCH --time=00:02:00 # Run time in hh:mm:ss
OMP_NUM_THREADS=10 ./a
```

OpenMP introduction

Example: every thread says hi

- Make a directory for this session:

```
[front01 ~]$ mkdir tomp
[front01 ~]$ cd tomp
```

- Copy first example (/nvme/scratch/jfinkenrath/NCC_Training/ex01):

```
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex01 .
[front01 tomp]$ cd tomp
```

- Edit the submit file `submit_ex01.sh`

```
[front01 tomp]$ vi submit_ex01.sh
..
```

- Inspect file a.c , compile it, and run:

```
[front01 ex01]$ more a.c
...
[front01 ex01]$ cc -o a a.c
[front01 ex01]$ sbatch submit_ex01.sh
[front01 ex01]$ less ex01.out
```

OpenMP introduction

Example: every thread says hi

Now, let's add a parallel region around the print statement:

- Add the parallel region:

```
#include <stdio.h>
int
main()
{
#pragma omp parallel
    {
        printf("Hi\n");
    }
    return 0;
}
```

OpenMP introduction

Example: every thread says hi

Now, let's add a parallel region around the print statement:

- Add the parallel region:

```
#include <stdio.h>
int
main()
{
#pragma omp parallel
    {
        printf("Hi\n");
    }
    return 0;
}
```

- Compile, adding the -fopenmp option, then run:

```
[front01 ex01]$ cc -fopenmp -o a a.c
[front01 ex01]$ sbatch submit_ex01.sh
[front01 ex01]$ more ex01.out
```

- you should see 10 Hi s

OpenMP introduction

Example: every thread says hi

The default number of threads depends on the requested tasks, here

```
#SBATCH --nodes=1 # 1 nodes
#SBATCH --ntasks-per-node=10 # Number of tasks to be invoked on each node
```

here (nodes * ntasks-per-node) = 10 but we can control this with `OMP_NUM_THREADS` :

- Set `OMP_NUM_THREADS` before running. No need to compile again. Edit the submit-script:

```
for ((n=1;n<11;n++)) do
    # printout number of Threads
    echo Number of OMP Threads = $n

    # run program a with OMP_NUM_THREADS
    OMP_NUM_THREADS=$n ./a
done
```

- You can also set `OMP_NUM_THREADS` to something larger than 10. You will simply be over-subscribing the cores, i.e. more than one thread will run per core. Edit the submit script to

```
for ((n=10;n<81;n+=10)) do
    # printout number of Threads
    echo Number of OMP Threads = $n

    # run program a with OMP_NUM_THREADS
    OMP_NUM_THREADS=$n ./a
done
```

OpenMP introduction

Example: every thread says hi

Now let's see how to use the OpenMP API. Additional to that every thread is printing Hi , to also write its thread id and the total number of threads. For that

- Add the following:
    1. Include `<omp.h>` in the beginning of the source code
    2. Get the thread id with `omp_get_thread_num()`
    3. Get the number of threads with `omp_get_num_threads()`

OpenMP introduction

Example: every thread says hi

```c
#include <stdio.h>
#include <omp.h>
int
main()
{
#pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nth = omp_get_num_threads();
        printf("Hi, I am thread: %2d of %2d\n", tid, nth);
    }
    return 0;
}
```

- Compile, edit the submit file and submit as usual. You should see something like:

```
[front01 ex01]$ less ex01.out
Hi, I am thread: 0 of 5
Hi, I am thread: 3 of 5
Hi, I am thread: 4 of 5
Hi, I am thread: 1 of 5
Hi, I am thread: 2 of 5
```

- Note that the order by which each thread reaches the `printf()` statement is non-deterministic
- Indeed, you should make no assumptions on the order by which each thread runs

OpenMP - Overview

Introduction

- Use exercise, see /nvme/scratch/jfinkenrath/NCC_Training/ex01
- Compiling with `-fopenmp` or `-qopenmp`
- Running using `OMP_NUM_THREADS=10 ./a`
- Using `<omp.h>` to use OpenMP functions `omp_get_thread_num()` and `omp_get_num_threads()`

Data sharing

- how to interact with data in parallel region ?
- how the different threads are interacting ?

OpenMP Data Sharing

Data Sharing between Threads

can be set by the causes:

private(list)

- Private variables are stored in the private stack of each thread
- Undefined initial value
- Undefined value after parallel region

shared(list)

- All threads can write to, and read from a shared variable
- Variables are shared by default

default(private/shared/none)

- Sets default for variables to be shared, private or not defined
- In C/C++ default(private) is not allowed
- default(none) can be useful for debugging as each variable has to be defined manually

OpenMP Data Sharing

Data sharing attributes

```
int a = 1;
int b = 2;
#pragma omp parallel private(a) shared(b)
{
    ...
}
```

- Each thread will have a local copy of `a` . `a` can be modified by each thread independently
- The variable `b` is shared between threads. Each thread can modify it and all threads will see the same data
- You can also set a default attribute for data sharing

```
int a = 1, b = 2, c = 3, d = 4, e = 5;
# pragma omp parallel default(shared) private(b)
{
    ...
}
```

- All variables are shared, except `b` which is `private`

OpenMP Data Sharing

Data sharing example

- Copy ex02 as before:

```
[front01 ex01]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex02 .
[front01 tomp]$ cd ex02
```

- Inspect, compile, and run a.c :

```
[front01 ex02]$ cc -fopenmp -o a a.c
[front01 ex02]$ more submit_ex02.sh
..
OMP_NUM_THREADS=5 ./a
[front01 ex02]$ sbatch submit_ex02.sh
..
..
[front01 ex02]$ more ex02.out
Thread: 2 of 5, some_var = 42
Thread: 4 of 5, some_var = 42
Thread: 0 of 5, some_var = 42
Thread: 1 of 5, some_var = 42
Thread: 3 of 5, some_var = 42
```

all threads have `some_var` set to the value 42

OpenMP Data Sharing

Data sharing example

- Now change the code so that the variable is modified within the parallel block, for example:

```c
#include <stdio.h>
#include <omp.h>
int
main()
{
    int some_var = 42;
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        some_var = tid;
        int nth = omp_get_num_threads();
        printf("Thread: %2d of %2d, some_var = %d\n", tid, nth, some_var);
    }
    return 0;
}
```

OpenMP Data Sharing

Data sharing example

- Now change the code so that the variable is modified within the parallel block, for example:

```c
#include <stdio.h>
#include <omp.h>
int
main()
{
    int some_var = 42;
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        some_var = tid;
        int nth = omp_get_num_threads();
        printf("Thread: %2d of %2d, some_var = %d\n", tid, nth, some_var);
    }
    return 0;
}
```

- The output is non-deterministic, for example:

```
[cn01 ex02]$ OMP_NUM_THREADS=5 ./a
Thread: 3 of 5, some_var = 3
Thread: 4 of 5, some_var = 4
Thread: 2 of 5, some_var = 4
Thread: 1 of 5, some_var = 1
Thread: 0 of 5, some_var = 3
```

OpenMP Data Sharing

Data sharing example

- Set the variable to private, to avoid this race condition

```c
#include <stdio.h>
#include <omp.h>
int
main()
{
    int some_var = 42;
    #pragma omp parallel private(some_var)
    {
        int tid = omp_get_thread_num();
        some_var = tid;
        int nth = omp_get_num_threads();
        printf("Thread: %2d of %2d, some_var = %d\n", tid, nth, some_var);
    }
    return 0;
}
```

```
[front01 ex02]$ sbatch submit_ex02.sh
...
[front01 ex02]$ more ex02.out
Thread: 0 of 5, some_var = 0
Thread: 4 of 5, some_var = 4
Thread: 2 of 5, some_var = 2
Thread: 3 of 5, some_var = 3
Thread: 1 of 5, some_var = 1
```

What is the value of `some_var` after the parallel region ends?

OpenMP Data Sharing

Data sharing example

- Initial value of a private variable

```c
#include <stdio.h>
#include <omp.h>
int
main()
{
    int some_var = 42;
    #pragma omp parallel private(some_var)
    {
        int tid = omp_get_thread_num();
        some_var = some_var+tid;
        int nth = omp_get_num_threads();
        printf("Thread: %2d of %2d, some_var = %d\n", tid, nth, some_var);
    }
    return 0;
}
```

- What do you expect this code to produce?
- Now try with firstprivate(`some_var`)

OpenMP Data Sharing

Data sharing example

- Shared vs private array (add `-std=c99` to your compiler flag)

```c
#include <stdio.h>
#include <omp.h>
int
main()
{

    int arr[12] = {0,0,0,0,0,0,0,0,0,0,0,0};

    #pragma omp parallel shared(arr)
    {
        int tid = omp_get_thread_num();
        int nth = omp_get_num_threads();
        arr[tid] = tid;
        printf("Thread: %2d of %2d, some_var = %d\n", tid, arr[tid]);
    }

    for(int i=0; i<12; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    return 0;
}
```

- What do you expect the output of this program to be?