DOCTYPE html>

Introduction to OpenMP

HPC Beginner Training event

17.02.2021

Jacob Finkenrath

NCC 4 HPC

CaSToRC, The Cyprus Insitute

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)

11:30 - 11:45: Break

11:45 - 12:30: Brief introduction to Parallel Computing with OpenMP - Session 2

- OpenMP Work sharing (45 min + Hands On)

12:30 - 13:30: Lunch Break

13:30 - 15:00: Brief introduction to Parallel Computing with OpenMP - Session 3

- OpenMP Tasking
- OpenMP Vectorization

Sources for this Course:

- Slides and files can be found under

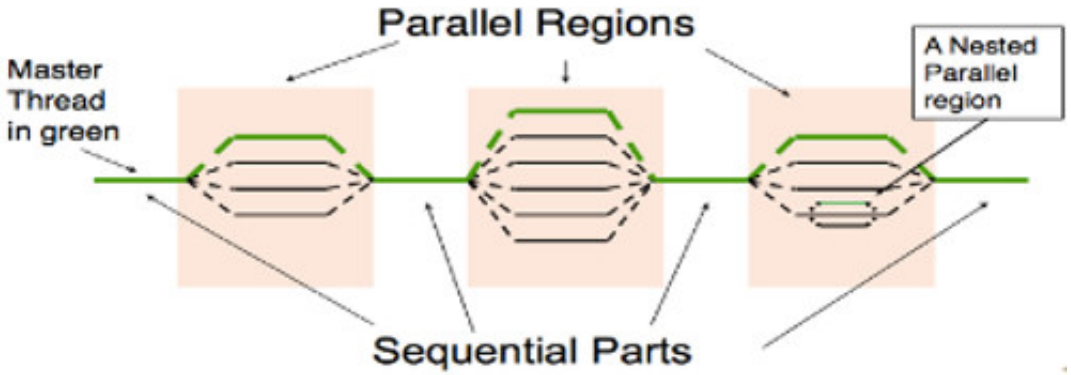
```
[front01 ~] cd /nvme/scratch/jfinkenrath/NCC_Training
[front01 NCC_Training] ls
ex01  ex02  ex03  ex04  ex05  ex06  ex07  ex_fibo  ex_trapez  slides
```

- Courses are based on
  - Assi. Prof. Giannis Koutsou, Lecture CoS501/SD402, Introduction to High Performance Computing, 09/2019 - 02/2020
  - POP CoE, Learning Material: https://pop-coe.eu/further-information/learning-material Christian Terboven, Dirk Schmidl, RWTH Aaachen
  - PRACE Training https://materials.prace-ri.eu/view/divisions/openmp.html Byckling, Mikko and Ilvonen, Sami (2014) PATC Course: Introduction to Parallel Programming @ CSC (2014). Introduction to Parallel Programming @ CSC, 2014-09-23. https://info.ornl.gov/sites/publications/files/Pub69214.pdf
- Note that POP COE and PRACE Training are providing a lot of interesting material on training/optimization/tools for HPC
  - most of Academic European Training Events on HPC are announced:
    - https://training.prace-ri.eu/index.php/training-events/
  - Tutorials and tools for Profiling and optimizing your applications:
    - https://pop-coe.eu/
- Today, we will just go through the basics of OpenMP but we will discuss this in more details at our next Workshop and for further question contact me.

Reminder on Parallel Strategies

Here we will talk about:

- Multi-threaded shared memory parallelization
  - Use multiple threads that share a common memory address space



Src: https://docs.nersc.gov/development/programming-models/openmp/openmp-resources/

To scale out of one node, parallelization has to be based on non-shared memory scheme

- need to exchange messages between processes
- MPI will be part of the next, Intermediate training event, which we will host in April

Here, we will discuss parallelization based on OpenMP to use the potential of available cores on a single node

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`
- 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/comp

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]$ more ex02.out
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?

OpenMP Summary

Data Sharing

- variables within parallel region can be shared or private
    - private: Undefined before and after the region
    - shared : all threads can write and read from it
    - firstprivate:

Work Sharing

- Let's take a look how we can parallelize the computation using OpenMP
    - Parallelization
    - how to parallize `for-loops` ?
    - First we will discuss the Loop construct of OpenMP
    - In the afternoon session we will discuss `Tasking`

OpenMP Work Sharing

- Parallel region creates an ***Single Program Multiple Data*** instance where each thread executes the same code
- we can one split the work between the threads of a parallel region?
    - Loop construct
    - Task construct

OpenMP Work Sharing

- Directive instructing compiler to share the work of a loop
  - C/C++: `#pragma omp for [clauses]`
  - Fortran: `!$omp do [clauses]`
    - The construct must followed by a loop construct. To be active it must be inside a parallel region
    - Combined construct with parallel:

      in **C/C++**:

      ```
      #pragma omp parallel for
      ```

      in **Fortran**

      ```
      $omp parallel do
      ```

- Loop index is `private` by default
- Work sharing dynamics can be controlled with the schedule `-clause`

For-loops

- in C/C++

```
#pragma omp parallel for
    for(int i=0; i<n; i++){
        ...
}
```

- in Fortran

```
!$OMP PARALLEL DO
do i = 1, n
    ...
end do
!$OMP END PARALLEL DO
```

OpenMP Work Sharing

Loop construct

```
#pragma omp parallel for
    for(int i=0; i<n; i++){
        ...
}
```

the `n` iterations will be split over the available threads accordingly

- Static scheduling, e.g.:

  ```
  #pragma omp parallel for schedule(static, 10)
  ```

  a chunk is 10 iterations. Threads receive a chunk to work in order.
- Dynamic scheduling, e.g.:

  ```
  #pragma omp parallel for schedule(dynamic, 10)
  ```

  a chunk is 10 iterations. Threads receive a chunk to work until they are exhausted.
- Guided scheduling, e.g.:

  ```
  #pragma omp parallel for schedule(guided)
  ```

  chunk size is modified as iterations are consumed.

OpenMP Work Sharing

Loop construct

- For loops

```
#pragma omp parallel
{
   #pragma omp for
    for(int i=0; i<n; i++){
        ...
    }
}
```

  use within a parallel region, i.e. when a parallel region is already open.

- Race condition:

```
int sum_variable = 0;
#pragma omp parallel for
for(int i=0; i<n; i++){
    sum_variable += ...;
        ...
}
```

- takes place when multiple threads read and write a variable simulatneously
- random results depedning on the order of threads accessing `sum_variable`

OpenMP Work Sharing

Reductions

- Summing elements of array is an example of reduction operation

$$S = \sum_{j=1}^{N} A_j = B_1 + B_2$$

- OpenMP provides support for common reductions within parallel regions and loops with the `reduction` -clause

OpenMP Work Sharing

Reductions

- Summing elements of array is an example of reduction operation

$$S = \sum_{j=1}^{N} A_j = B_1 + B_2$$

- OpenMP provides support for common reductions within parallel regions and loops with the `reduction` -clause

reduction(operator:list)

- Performs reduction on the (scalar) variables in list
- Private reduction variable is created for each thread's partial result
- Private reduction variable is initialized to operator's initial value
- After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable

OpenMP Work Sharing

Reductions within for loop:

```
int sum_variable = 0;

#pragma omp parallel for reduction(+: sum_variable)
for(int i=0; i<n; i++){
    sum_variable += ...;
        ...
}
```

Different reductions operators available like:

Operator	Initial value
+	0
-	1
*	0

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

- Copy ex03 as before:

```
[front01 ex02]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex03 .
[front01 tomp]$ cd ex03
```

- Inspect, compile axpy.c, edit the submit script and run :

```
[front01 ex03]$ cc -std=c99 -fopenmp -o axpy axpy.c
[front01 ex03]$ vi submit_ex03.sh
...
[front01 ex03]$ sbatch submit_ex03.sh
[front01 ex03]$ more ex03.out
t0 = 0.232100 sec, t1 = 0.232260 sec, diff z norm = 0.000000e+00
```

- Use an OpenMP pragma to parallelize the second occurrence of the main for loop

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

Change:

```
for(int i=0; i<n; i++) {
    z_1[i] = a*x[i] + y[i];
}

#pragma omp parallel for
for(int i=0; i<n; i++) {
    z_1[i] = a*x[i] + y[i];
}
```

It's also useful to report the total number of threads:

```
printf(" t0 = %lf sec, t1 = %lf sec, diff z)

#pragma omp parallel
{
    int nth = omp_get_num_threads();
    #pragma omp single
    printf(" nth = %2d, t0 = %lf sec, t1 = %lf)
}
```

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

- Run for `OMP_NUM_THREADS` from 1,...,10. How does the runtime scale?

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

- Run for `OMP_NUM_THREADS` from 1,...,10. How does the runtime scale?

```
[front01 ex03]$ vi submit_ex03.sh
...

for n in 1 2 3 4 5 6 7 8 9 10
do
OMP_NUM_THREADS=$n ./axpy $((32*1024*1024))
done
...
[front01 ex03]$ sbatch submit_ex03.sh
```

OpenMP Work Sharing

Linear algebra

Operation:

$$z_i = ax_i + y_i$$

- Run for `OMP_NUM_THREADS` from 1,...,10. How does the runtime scale?

```
[front01 ex03]$ vi submit_ex03.sh
...

for n in 1 2 3 4 5 6 7 8 9 10
do
OMP_NUM_THREADS=$n ./axpy $((32*1024*1024))
done
...
[front01 ex03]$ sbatch submit_ex03.sh
```

```
[front01 ex03]$ more ex03.out
nth = 1, t0 = 0.233991 sec, t1 = 0.261533 sec,
nth = 2, t0 = 0.232231 sec, t1 = 0.131076 sec,
nth = 3, t0 = 0.235093 sec, t1 = 0.088683 sec,
...
nth = 8, t0 = 0.234414 sec, t1 = 0.053443 sec,
nth = 9, t0 = 0.230891 sec, t1 = 0.048860 sec,
nth = 10, t0 = 0.233603 sec, t1 = 0.051261 sec,
```

OpenMP Work Sharing

Linear algebra

Dot product operation:

$$r = x^T y = \sum_{i=0}^{n-1} x_i y_i$$

- Copy ex04 as before:

```
[front01 ex03]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex04 .
[front01 tomp]$ cd ex04
```

- Inspect, compile, and run xdoty.c :

```
[front01 ex04]$ cc -std=c99 -o xdoty xdoty.c
[front01 ex04]$ vi submit_ex04.sh
...
./xdoty $((32*1024*1024))
...
[front01 ex04]$ sbatch submit_ex04.sh
[front01 ex04]$ more ex04.out
t0 = 0.172530 sec, t1 = 0.171624 sec, norms = 8.387960e+06, 8.387960e+06
```

- Use an OpenMP pragma to parallelize the second occurrence of the main for loop

OpenMP Work Sharing

Linear algebra

Dot product operation:

$$r = x^T y = \sum_{i=0}^{n-1} x_i y_i$$

```
double norm_1 = 0;

for(int i=0; i<n; i++) {
    norm_1 += x[i]*y[i];
}
```

OpenMP Work Sharing

Linear algebra

Dot product operation:

$$r = x^T y = \sum_{i=0}^{n-1} x_i y_i$$

```
double norm_1 = 0;

for(int i=0; i<n; i++) {
    norm_1 += x[i]*y[i];
}
```

```
double norm_1 = 0;

#pragma omp parallel for reduction(+:norm_1)
for(int i=0; i<n; i++) {
    norm_1 += x[i]*y[i];
}
```

OpenMP Work Sharing

Linear algebra

Dot product operation:

$$r = x^T y = \sum_{i=0}^{n-1} x_i y_i$$

```
double norm_1 = 0;

for(int i=0; i<n; i++) {
    norm_1 += x[i]*y[i];
}
```

```
double norm_1 = 0;

#pragma omp parallel for reduction(+:norm_1)
for(int i=0; i<n; i++) {
    norm_1 += x[i]*y[i];
}
```

- Now run for OMP_NUM_THREADS from 1,...,10.
- edit `submit_ex04.sh`

```
for n in 1 2 3 4 5 6 7 8 9 10; do
    OMP_NUM_THREADS=$n ./xdoty $((32*1024*1024))
done
```

```
[front01 ex04]$ more ex04.out
nth = 1, t0 = 0.171664 sec, t1 = 0.186125 sec, norms = 8.387960e+06, 8.387960e+06
nth = 2, t0 = 0.171736 sec, t1 = 0.093612 sec, norms = 8.387960e+06, 8.387960e+06
nth = 3, t0 = 0.170798 sec, t1 = 0.062440 sec, norms = 8.387960e+06, 8.387960e+06
...
nth = 10, t0 = 0.171464 sec, t1 = 0.024965 sec, norms = 8.387960e+06, 8.387960e+06
```

OpenMP Summary

For-loop construct

Parallel region given by the

- Directive instructing compiler to share the work of a loop
  - using for C/C++: `#pragma omp for [clauses]`
  - with `clauses`:
    - schedule(static/dynamic,guided,chunk)
    - reduction(+/-/*: VARIABLE)

Tasking

If computations are not located in simple for-loops, parallelization via `for-loop` construct can become tricky

- OpenMP allows to identify tasks within parallel region
  - this simplifies to parallelize more complicated structures
  - but can increase overheads by OpenMP

OpenMP Tasking

The `task` directive

- Critical regions: where each thread should run the region one-at-a-time

```
#pragma omp parallel
{
    #pragma omp critical
    {
        ... code to be run by each thread, one-at-a-time ...
    }
}
```

  of course, critical regions are serialized, i.e. the runtime scales with the number of threads.

- Single regions: within a parallel region, run by one thread

```
#pragma omp parallel
{
#pragma omp single
    {
        printf("Hi\n");
    }
}
```

OpenMP Tasking

The `task` directive

- Tasks: define a block of code, a task to be run by a single thread:

```
#pragma omp task
{
    ...
}
```

- Usually run within a single region, to distribute work

```
int a = 1;
int b = 2;
#pragma omp parallel
{
    #pragma omp single
    {
           #pragma omp task
        {
            // This will be done by one thread
                a = a+1;
        }

        #pragma omp task
        {
            // This will be done by another thread
            b = b+1;
        }
    }
}
```

OpenMP Tasking

The task directive

- Copy ex05 as before:

```
[front01 ex04]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex05 .
[front01 tomp$ cd ex05
```

- Inspect, compile, and run a.c :

```
[front01 ex05]$ cc -std=c99 -fopenmp -o a a.c
[front01 ex05]$ vi submit_ex05.sh
...
OMP_NUM_THREADS=5 ./a
...
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
Hi, I am thread: 0 of 5
Hi, I am thread: 1 of 5
Hi, I am thread: 4 of 5
Hi, I am thread: 3 of 5
Hi, I am thread: 2 of 5
```

OpenMP Tasking

The `task` directive

- Enclose the print statement in an `omp single` region:

```
#pragma omp parallel
{
    #pragma omp single
    {
        int tid = omp_get_thread_num();
        int nth = omp_get_num_threads();
        printf("Hi, I am thread: %2d of %2d\n", tid, nth);
    }
}
```

- Compile and run a few times.

```
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
Hi, I am thread: 0 of 5
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
Hi, I am thread: 1 of 5
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
Hi, I am thread: 3 of 5
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
```

- Each time a single thread calls the `printf()` . Which thread this is, is random.

OpenMP Tasking

The `task` directive

- Now try the following:

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

int
main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            {
                int tid = omp_get_thread_num();
                int nth = omp_get_num_threads();
                printf("1: Hi, I am thread: %2d of %2d\n", tid, nth);
            }
            #pragma omp task
            {
                int tid = omp_get_thread_num();
                int nth = omp_get_num_threads();
                printf("2: Hi, I am thread: %2d of %2d\n", tid, nth);
            }
        }
    }
    return 0;
}
```

- Compile and run a few times

OpenMP Tasking

The `task` directive, another example

- Run a few times:

```
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
1: Hi, I am thread: 2 of 5
2: Hi, I am thread: 0 of 5
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
1: Hi, I am thread: 4 of 5
2: Hi, I am thread: 2 of 5
[front01 ex05]$ sbatch submit_ex05.sh
[front01 ex05]$ more ex05.sh
2: Hi, I am thread: 2 of 5
1: Hi, I am thread: 4 of 5
```

Explanation:

- The thread within the omp single region encounters a task and dispatches it to idle threads to run
- Which thread picks up the task is random
- Which task of the two is run first is also random

OpenMP Tasking

The `task` directive, another example

- The `taskwait` directive can be used to ensure the order in which tasks are run:

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

int
main()
{
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        {
            int tid = omp_get_thread_num();
            int nth = omp_get_num_threads();
            printf("1: Hi, I am thread: %2d of %2d\n", tid, nth);
        }
        #pragma omp taskwait
        #pragma omp task
        {
            int tid = omp_get_thread_num();
            int nth = omp_get_num_threads();
            printf("2: Hi, I am thread: %2d of %2d\n", tid, nth);
        }
    }
}
return 0;
}
```

- This way, the first `printf()` is always run first

OpenMP Tasking

The `task` directive, another example

- Consider the three words: one , two , and three
- Write a program that, each time it is run, prints a random permutation of these three words

The regular procedure

- Copy ex06 as before:

```
[front01 ex05]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex06 .
[front01 tomp]$ cd ex06
```

- Inspect, compile, and submit:

```
[n001 ex06]$ vi submit_ex06.sh
...
./a
...
[n001 ex06]$ more ex06.out
one two three
```

- Add task directives, so that the three words appear in a random permutation

OpenMP Tasking

The task directive, yet another example

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

int
main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            printf("one ");

            #pragma omp task
            printf("two ");

            #pragma omp task
            printf("three ");
        }
    }
    printf("\n");
    return 0;
}
```

Edit `submit_ex06.sh` to

```
[front01 ex06]$ vi submit_ex06.sh
...
OMP_NUM_THREADS=5 ./a
...
```

and run a few times:

```
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
three one two
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
one two three
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
two one three
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
one two three
```

OpenMP Tasking

The `task` directive, yet another example

- Now add an additional `task` to print `done!` , but insure that its *always* as the last word

OpenMP Tasking

The `task` directive, yet another example

- Now add an additional `task` to print `done!` , but insure that its *always* as the last word

```
    #include <stdio.h>
#include <omp.h>
int
main()
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            printf("one ");

            #pragma omp task
            printf("two ");

            #pragma omp task
            printf("three ");

            #pragma omp taskwait

            #pragma omp task
            printf("done!");
        }
    }
    printf("\n");
    return 0;
}
```

Compile and run for a few times:

```
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
two three one done!
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
one two three done!
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
two one three done!
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
one two three done!
[front01 ex06]$ sbatch submit_ex06.sh
[front01 ex06]$ more ex06.out
one two three done!
```

OpenMP Tasking

The `task` directive, yet another example

- Copy ex07 as before:

```
[front01 ex06]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex07 .
[front01 tomp]$ cd ex07
```

OpenMP Tasking

The task directive, yet another example

- Inspect `a.c` , in particular `main()`
  - Initializes an array `a[N]`
    - The first `N-1` elements are set to a random integer, up to 8 digits long
    - The last element `a[N-1]` is set to `-1`
  - Sets a pointer p to the first element: `*p = &a[0]`
  - And then enters a loop:
    - Calls the function `process()` with argument the pointer `p` : `process(p)`
    - After `process()` returns, sets p to the next element in `a[]` : `p=p+1`
    - The loop terminates when the value in p is -1 , i.e. when it reaches the end of array `a[]`
- Inspect function `process()`
  - Takes the value pointed to by `*x` (p in the main program)
  - Sums all integers from one up to the value of `*x`
  - Sets `*x` to be equal to the sum

OpenMP Tasking

The `task` directive, yet another example

- main of `ex07.c`

```
int
main()
{
  srand(2147483641);
  int a[N];
  for(int i=0; i<N-1; i++)
    a[i] = irand();

  a[N-1] = -1;

  double t0 = stop_watch(0);
  int *p = &a[0];
  while(*p >= 0) {
      process(p);
      p = p+1;
  }
  t0 = stop_watch(t0);

  printf(" t = %lf\n", t0);
  return 0;
}
```

- and function `process()`

```
  void
  process(int *x)
  {
      int sum = 0;
      for(int i=0; i<*x; i++) {
          sum += i;
      }
  *x = sum;
  return;
  }
```

OpenMP Tasking

The `task` directive, yet another example

- Currently, the program uses an array length of 10, and takes about 1.0 seconds to complete
- Our goal is to use `omp task` directives so that the sums are distributed to different threads
- Use `omp parallel` and `omp single` to define a `parallel` region and a `scalar` region within that parallel region
- Use `omp task` to specify which block is to be distributed to different threads. Be careful, you will need to use a `firstprivate` directive somewhere
- There are many ways in which this code could enter an infinite loop, including:
  - Corrupting the array. For example by writing, in `process()` to the wrong memory address, thus corrupting the last element of `a[]` , causing the `while()` loop to loop forever
  - Not incrementing correctly (*p) . For example if the thread that increments p is not the thread that checks the `while()` condition

OpenMP Tasking

The task directive, yet another example

```
int *p = &a[0];
#pragma omp parallel
{
    #pragma omp single
    while(*p >= 0) {
        #pragma omp task firstprivate(p)
        {
            process(p);
        }
        p = p+1;
    }
}
```

OpenMP Tasking

The `task` directive, yet another example

```
int *p = &a[0];
#pragma omp parallel
{
    #pragma omp single
    while(*p >= 0) {
        #pragma omp task firstprivate(p)
        {
            process(p);
        }
        p = p+1;
    }
}
```

- A single thread executes the `while()` loop

OpenMP Tasking

The `task` directive, yet another example

```
int *p = &a[0];
#pragma omp parallel
{
    #pragma omp single
    while(*p >= 0) {
        #pragma omp task firstprivate(p)
        {
            process(p);
        }
        p = p+1;
    }
}
```

- A single thread executes the `while()` loop
- A task is dispatched to call `process()` in each iteration. The pointer p is copied each time and is private to each thread

OpenMP Tasking

The `task` directive, yet another example

```
int *p = &a[0];
#pragma omp parallel
{
    #pragma omp single
    while(*p >= 0) {
        #pragma omp task firstprivate(p)
        {
            process(p);
        }
        p = p+1;
    }
}
```

- A single thread executes the `while()` loop
- A task is dispatched to call `process()` in each iteration. The pointer p is copied each time and is private to each thread
- Only the thread in the single region increments p . Only this thread checks the `while()` statement termination condition

OpenMP Tasking

The task directive, yet another example

```
int *p = &a[0];
#pragma omp parallel
{
    #pragma omp single
    while(*p >= 0) {
        #pragma omp task firstprivate(p)
        {
            process(p);
        }
        p = p+1;
    }
}
```

```
[front01 ex07]$ vi submit_ex07.sh
..
for n in 1 2 3 4 5 6 7 8 9 10 11 12; do OMP_NUM_THREADS=$n ./a ; done
..
[front01 ex07]$ sbatch submit_ex07.sh
[front01 ex07]$ more ex07.sh
nth:   1 t = 1.445933
nth:   2 t = 0.722370
nth:   3 t = 0.513235
nth:   4 t = 0.418930
nth:   5 t = 0.368200
nth:   6 t = 0.309495
nth:   7 t = 0.257213
nth:   8 t = 0.255306
nth:   9 t = 0.256733
nth: 10 t = 0.256414
nth: 11 t = 0.255445
nth: 12 t = 0.281543
```

OpenMP Tasking

The `task` directive, reduction of overheads

Recursive approach to compute Fibonacci

```
int main(int argc,
char* argv[])
{
    [...]
    fib(input);
    [...]
}
```

```
int fib(int n)
{
    if (n < 2) return n;
    int x = fib(n - 1);
    int y = fib(n - 2);
    return x+y;
}
```

- On the following slides we will discuss three approaches to parallelize this recursive code with Tasking.

OpenMP Tasking

The `task` directive, reduction of overheads

First version parallelized with Tasking (omp-v1)

```
int main(int argc,
char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
    }
    [...]
}
```

```
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
    return x+y;
}
```

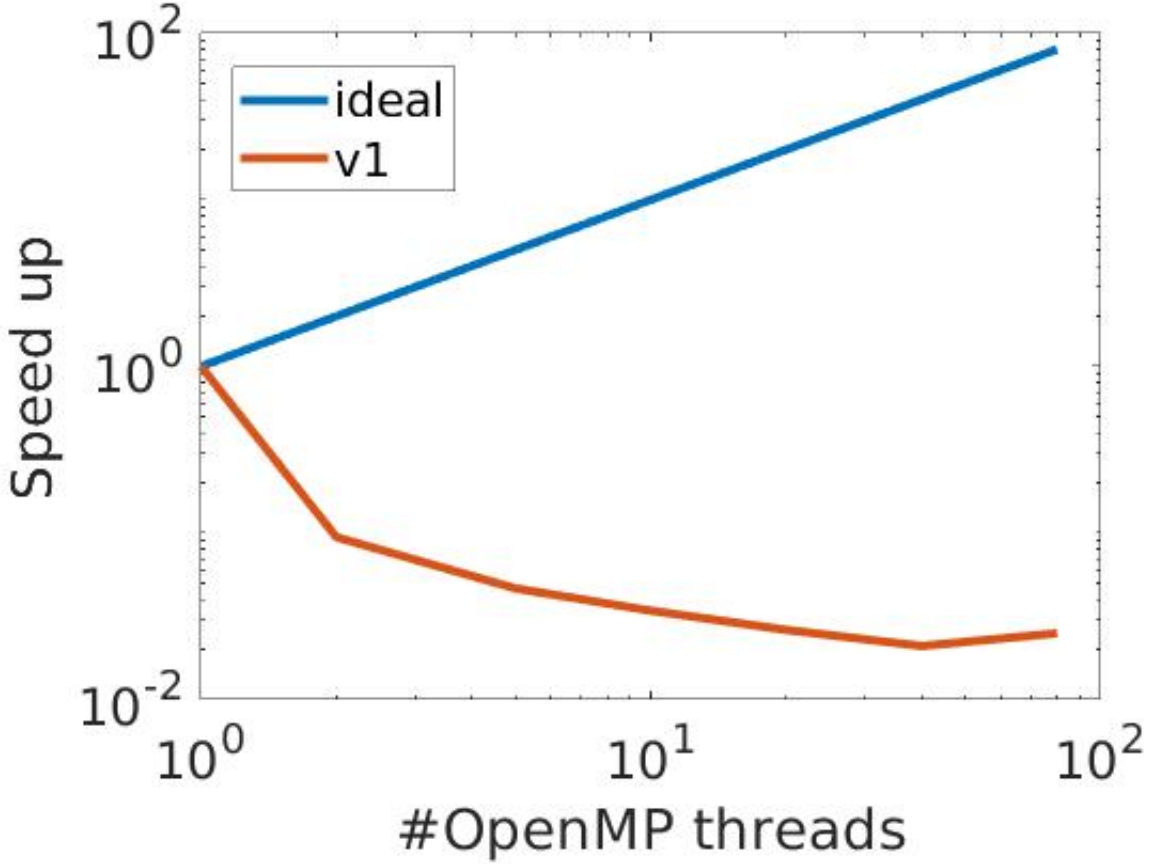
- Only one Task / Thread enters `fib()` from `main()`, it is responsable for creating the two initial work tasks
- Taskwait is required, as otherwise `x` and `y` would be lost

OpenMP Tasking

The `task` directive, reduction of overheads

Scalability measurements (1/3)

Overhead of task creation prevents better scalability



OpenMP Tasking

The `task` directive, reduction of overheads *`if` Clause*

- If the expression of an `if` clause on a task evaluates to `false`
  - The encountering task is suspended
  - The new task is executed immediately
  - The parent task resumes when the new task finishes
- Used for optimization, e.g., avoid creation of small tasks

OpenMP Tasking

The `task` directive, reduction of overheads

Second version parallelized with Tasking (omp-v2)

- Improvement: Don‘t create yet another task once a certain (small enough) n is reached

```
int main(int argc,
char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
    }
    [...]
}
```

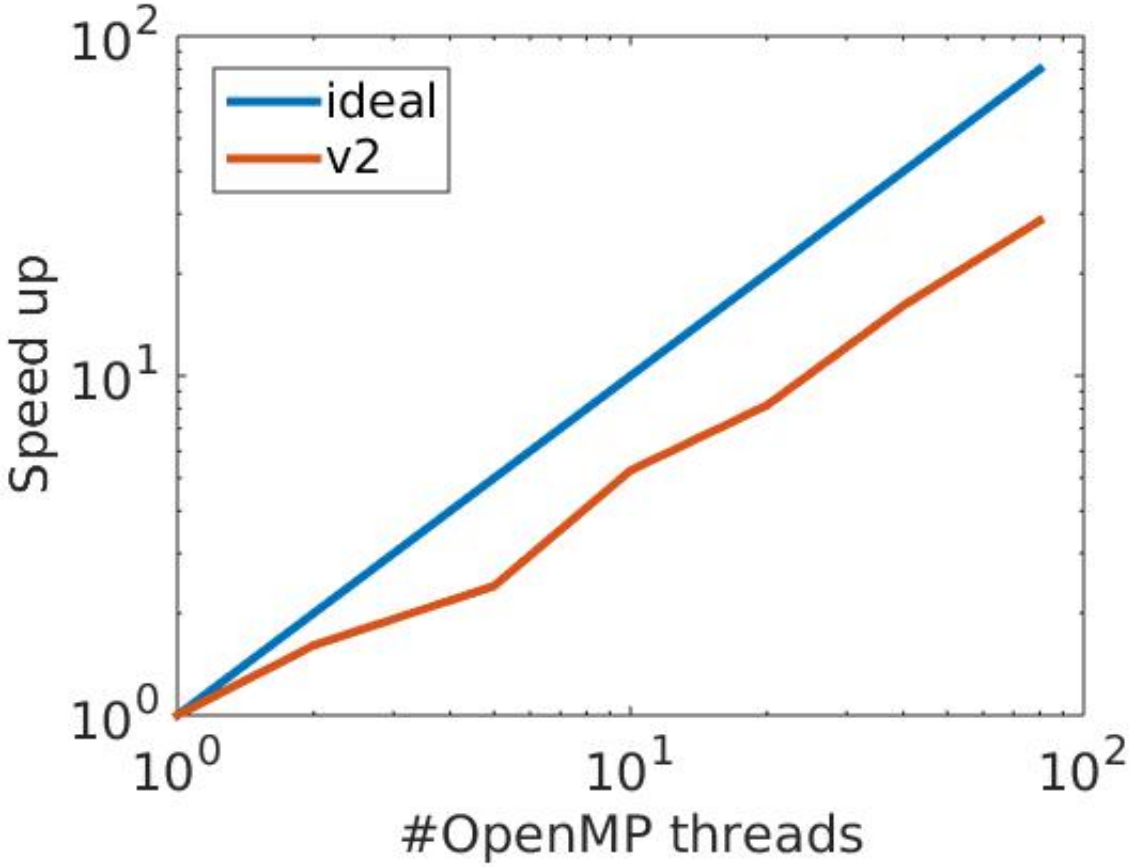
```
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x) \
    if(n > 30)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y) \
    if(n > 30)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
    return x+y;
}
```

OpenMP Tasking

The `task` directive, reduction of overheads

Scalability measurements (2/3)

Speedup is ok, but we still have some overhead when running with 4 or 8 threads



OpenMP Tasking

The `task` directive, Third version parallelized with Tasking (omp-v3)

- Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)

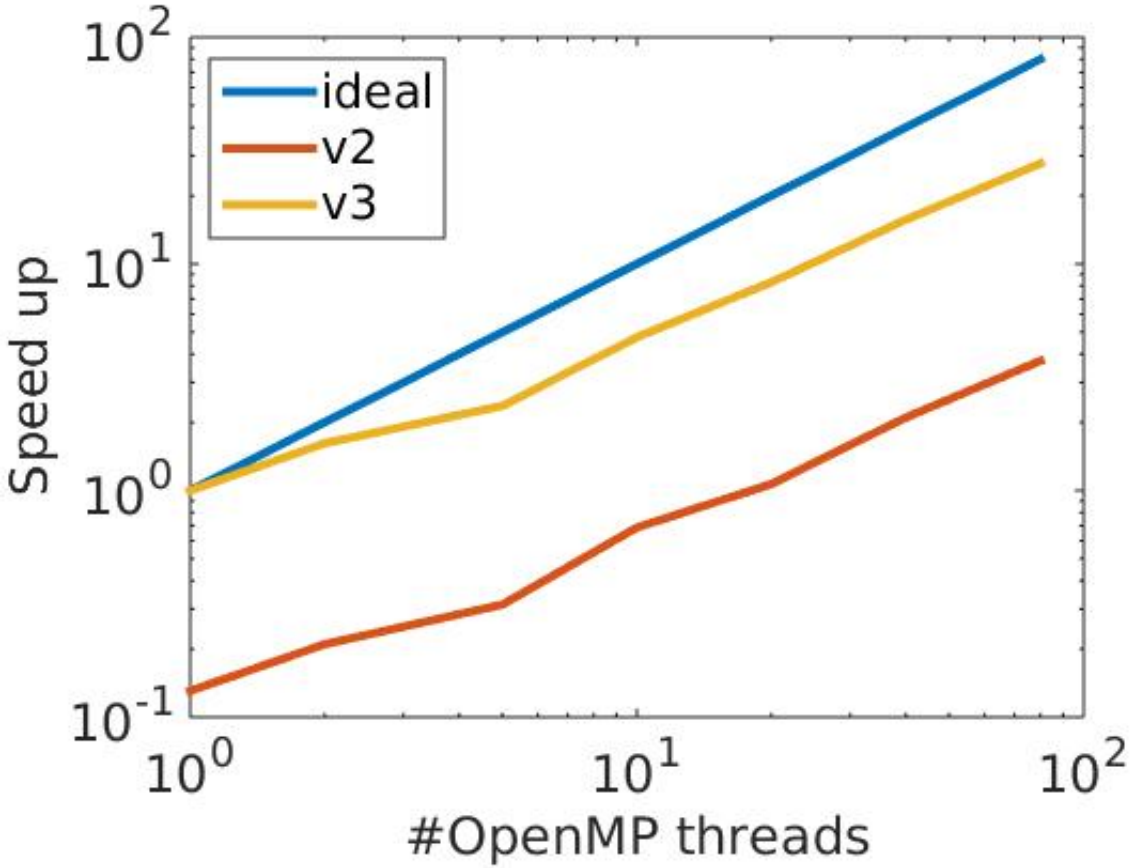
```
int main(int argc,
char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
    }
    [...]
}
```

```
int fib(int n)
{
    if (n < 2) return n;
    if (n <= 30)
        return serfib(n);
    int x, y;

    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }

    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }

    #pragma omp taskwait
    return x+y;
}
```

OpenMP Tasking

The `task` directive, reduction of overheads

Scalability measurements (3/3)



OpenMP Tasking

The `task` directive, reduction of overheads

- Can you achieve better scaling ?
- Check the dependence on the `if clause`
- Can you come up with a faster version for calculating Fibonaci numbers ?

Vectorization in OpenMP

In OpenMP 4.0, SIMD directives were added to help compilers generate efficient vector code.

- SIMD directives explicitly enable vectorization in the compiler
- can support the autovectorization of the compiler

SIMD loop directives

- can be placed above for loops with the syntax

  ```
  #pragma omp simd [clause[[,]clause] ...]
  ```

  which marks the loop as a SIMD enabled loop or SIMD region.
- OpenMP loop directives only apply to for loops that are in a canonical form, where the number of iteration is known

Vectorization in OpenMP

SAXPY example

- Now, lets declare a `simd` region for the second loop

```
...
#pragma omp simd
    for(int i=0; i<n; i++) {
    z_1[i] = a*x[i] + y[i];
}
...
```

compile with `-O1` and run

```
[front01 ex08]$ more ex03.out
t0 = 0.099114 sec, t1 = 0.098404 sec, diff z norm = 0.000000e+00
```

Vectorization in OpenMP

SAXPY example

- Now, lets declare a `simd` region for the second loop

```
...
#pragma omp simd
    for(int i=0; i<n; i++) {
    z_1[i] = a*x[i] + y[i];
}
...
```

compile with `-O1` and run

```
[front01 ex08]$ more ex03.out
t0 = 0.099114 sec, t1 = 0.098404 sec, diff z norm = 0.000000e+00
```

- compile with `-O2` and check the change
- what happens with `-Ofast` ? Be very careful with `-Ofast`, it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules for math function

Vectorization in OpenMP

SAXPY example

`Lets try to parallelize it`

```
...
#pragma omp parallel for simd
 for(int i=0; i<n; i++) {
    z_1[i] = a*x[i] + y[i];
}
 ...
```

- Note that the default gnu compiler will not work, reload another version

```
[front01 ex03]$ module load GCC/8.2.0-2.31.1
[front01 ex03]$ cc -std=c99 -fopenmp -O2 -o axpy axpy.c
```

- Checkout OpenMP enviroment variables:
  - Set `OMP_DYNAMIC=true` and compare it with `OMP_NUM_THREADS=$n`

Vectorization in OpenMP

SAXPY example

- Now does it work ?
- Intel compiler can provide a report

```
[front01 ex08]$ module load icc
[front01 ex08]$ icc -std=c99 -qopt-report=2  -qopenmp -O2 -o axpy axpy.c
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location
[front01 ex08]$ more axpy.optrpt
...
LOOP BEGIN at axpy.c(76,3)
remark #15300: LOOP WAS VECTORIZED
LOOP END
...
LOOP BEGIN at axpy.c(83,3)
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END
...
```

- in more complex code, the auto-vectorization of the compiler is likely to fail, for that OpenMP provides causes which will help the compiler to identify vectorizable regions, like loop and delvare directives (see next pages for an overview)

Vectorization in OpenMP

SIMD loop directives

- SIMD aligned `#pragma omp simd aligned([ptr] : [alignment], . . . )`
  - data aligned speed up memory access and help the compiler to determine property of data
  - programmer has to ensure the data alignment

- SIMD reduction `#pragma omp simd reduction([operation] : [variable],. . . )`
  - instructs the compiler to perform a vector reduction on a variable
  - some compilers have difficulties to detecting reductions automatically

- SIMD safelen `#pragma omp simd safelen([value])`
  - for data dependency within the loop
  - assure that only data get accessed which are not exceeding the specified value

- SIMD collapse `#pragma omp simd collapse([value]`
  - try collapse nested loops into one, suitable for auto-vectorization

- SIMD private/lastprivate `#pragma omp simd private([variable],...)`
  - `private` and `lastprivate` clauses control data privatization and sharing of variables for a SIMD Loop
  - `private` clause creates an uninitialized vector inside the SIMD loop for the given variable
  - `lastprivate` clause provides the same semantics but also copies out the values produced from the last iteration to outside the loop

Vectorization in OpenMP

SIMD declare directives

SIMD enabled functions can be declared by

```
#pragma omp declare simd [clause[[,] clause] ...]
```

- compiler will create several versions of SIMD declared functions
  - different vectorization used depend on from which region function is called
  - function has its own type of vectorized arguments, `uniform`, `vector` and `linear`
- SIMD declare aligned `#pragma omp declare simd aligned([argument] : [alignment],. . . )`
  - `aligned clause` instructs the compiler that the pointers passed as function arguments are aligned by the given alignment value
- SIMD declare simdlen `#pragma omp declare simd simdlen([value])`
  - `simdlen clause` specifies the number of packed arguments the vectorized function will execute
- SIMD declare uniform `#pragma omp declare simd uniform([argument],. . . )`
  - indicates that the valuel not change and is shared between the SIMD lanes of the loop
- SIMD declare linear `#pragma omp declare simd linear([argument] : [linearstep],. . . )`
  - `linear` clause will be increased by linear between each successive function call

Vectorization in OpenMP

SIMD declare directives

Lets take a look to our last exercise

```
[front01 ex07]$ cd ../
[front01 tomp]$ cp -r /nvme/scratch/jfinkenrath/NCC_Training/ex08 .
[front01 tomp$ cd ex08
```

The file is based on exercise 3, only that the function is outsourced. Compile it via:

```
[front01 ex08]$ gcc -fopenmp -O0 -c fnt.c
 [front01 ex08]$ gcc -fopenmp -O0 -o axpy fnt.o axpy.c
```

and check the timings.

- check if compiler optimization can speed up the problem
- does something change if you compile the different files with different flags
- does it help to the function axpy as `declare simd`

SUMMARY

- OpenMP Introduction
  - Compiler directives, i.e., language extensions for shared memory parallelization

    ```
    - Syntax: *directive*, **construct**, `clauses`
    - C/C++: *#pragma* **omp parallel** `shared(data)`
    ```

  - OpenMP API functions can give back informations on the thread ID with `omp_get_thread_num()` and `omp_get_num_threads()`
- OpenMP Data sharing
  - Private variables can be indicated as `private(list)`
  - Shared variables can be indicated as `shared(list)`
- OpenMP Work sharing
  - `for-loop` construct
    - use `#pragma omp parallel for` to parallize a for-loop
    - use reduction clauses to avoid race-conditions like `#pragma omp parallel for reduction(sum_var:+)`
  - tasking
    - task can be assigned to a single thread in a parallel region
    - simplies to paralellize code but still need care to avoid overheads
- OpenMP Vectorization
  - openMP can indicate vectorizable regions for auto-vectorization of the compiler
    - regions with `#pragma omp simd` or for function `#pragma omp declare simd`

FINAL REMARKS

Programme of the NCC for HPC

Upcoming events in the first half of 2021

- **Industrial week**: 1st March - 4th March
- **Intermediate Training Event**: 19th - 21th April
    - which includes MPI, OpenMP and Hybrid Programming, Python for HPC, Optimization etc.
- **Hackathon**: 19th - 21th May
    - we are looking for projects, call for application is not open
- Academic and industrial call for High Level Support (will be announced soon)

```
- to leverage access to High Performance Computing
- includes High Level Support for projects
```

For news, please subscribe to mailing list https://castorc.cyi.ac.cy/national-hpc-competence-centre

For any question regarding the academic support, send me an email **j.finkenrath@cyi.ac.cy**

Conclusion

Thank you all for your interest !!!

Hope to see you all at the Intermediate Training in April