

CME 213, ME 339 Spring 2023

Introduction to parallel computing using MPI, openMP, and CUDA

Stanford University

Eric Darve, ICME, Stanford

“Any fool can write code that a computer can understand. Good
programmers write code that humans can understand.” (Martin Fowler)



Stanford University

ICME



- C++ threads
- How to pass arguments by value and reference
- join and detach
- promise and future
- mutex and lock_guard

OpenMP



C++ std::thread for scientific computing

- C++ thread is a powerful and efficient mechanism to program shared memory computers.
- High performance and flexible.
- But engineering codes demand a higher productivity.
- There are certain patterns of computing in engineering codes that are common and can be leveraged.

OpenMP

Key objectives:

- Simplify multithreaded programming
- Improve performance
- Allow same code to run on both sequential or multicore processors

Common patterns in parallel scientific computing

1. for loops are at the core of most computation: perform a repetitive calculation on large arrays. This form of structured parallel computation is ideal for OpenMP.
2. Hand-off a block of code to a separate thread; this is closer to C++ std::thread. OpenMP makes this process more efficient and takes care of many “boilerplate” optimizations.

OpenMP makes scientific multithreaded programming very easy!

- OpenMP simplifies the programming significantly.
- In many cases, adding one line is sufficient to make it run in parallel.
- OpenMP is the standard approach in scientific computing for multicore processors.

Goals of OpenMP

- **Standardization:**
 - Provide a **standard** among a variety of shared memory architectures/platforms
 - Jointly defined and endorsed by a **group of major** computer hardware and software vendors
- **Simple but powerful:**
 - Establish a simple and limited set of **directives** for programming shared memory machines.
 - Significant parallelism can be implemented by using just **3 or 4 directives**.
 - This goal is becoming less true with each new release, unfortunately.

Goals of OpenMP

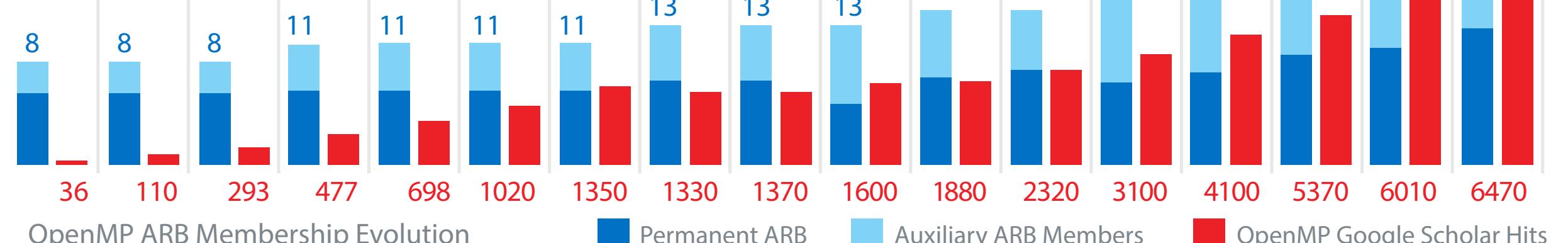
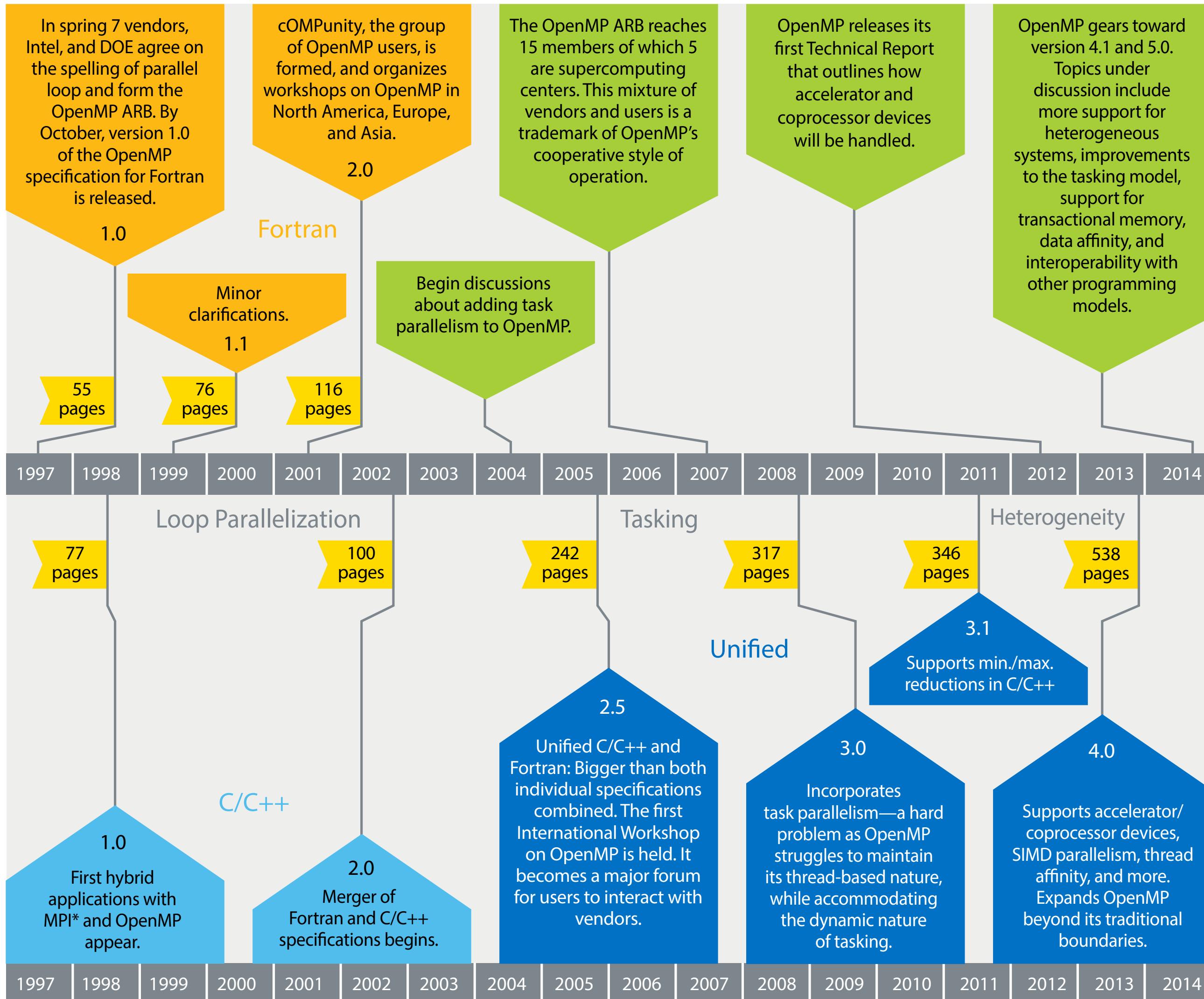
- **Ease of use:**
 - Provide capability to **incrementally** parallelize a serial program
 - Provide the capability to implement both coarse-grained and fine-grained parallelism
- **Portability:**
 - The API is specified for C/C++ and Fortran
 - Public forum for API and membership
 - **Most major platforms have been implemented including Unix/Linux platforms and Windows**

Reference material

- OpenMP website <https://openmp.org>
- Wikipedia <https://en.wikipedia.org/wiki/OpenMP>
- LLNL tutorial <https://hpc-tutorials.llnl.gov/openmp>
- Intel [https://www.intel.com/content/dam/www/public/apac/xa/en/pdfs/ssg/Programming with OpenMP-Linux.pdf](https://www.intel.com/content/dam/www/public/apac/xa/en/pdfs/ssg/Programming%20with%20OpenMP-Linux.pdf)

History of OpenMP

1996 Vendors provide similar but different solutions for loop [parallelism](#), causing portability and maintenance problems.
 Kuck and Associates, Inc. (KAI) | SGI | Cray | IBM | High Performance Fortran (HPF) | Parallel Computing Forum (PCF)



[https://www.openmp.org/uncategorized/
openmp-timeline/](https://www.openmp.org/uncategorized/openmp-timeline/)

OpenMP release history

Month/Year	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2018	OpenMP 5.0
Nov 2020	OpenMP 5.1
Nov 2021	OpenMP 5.2

How to use the icme-gpu cluster

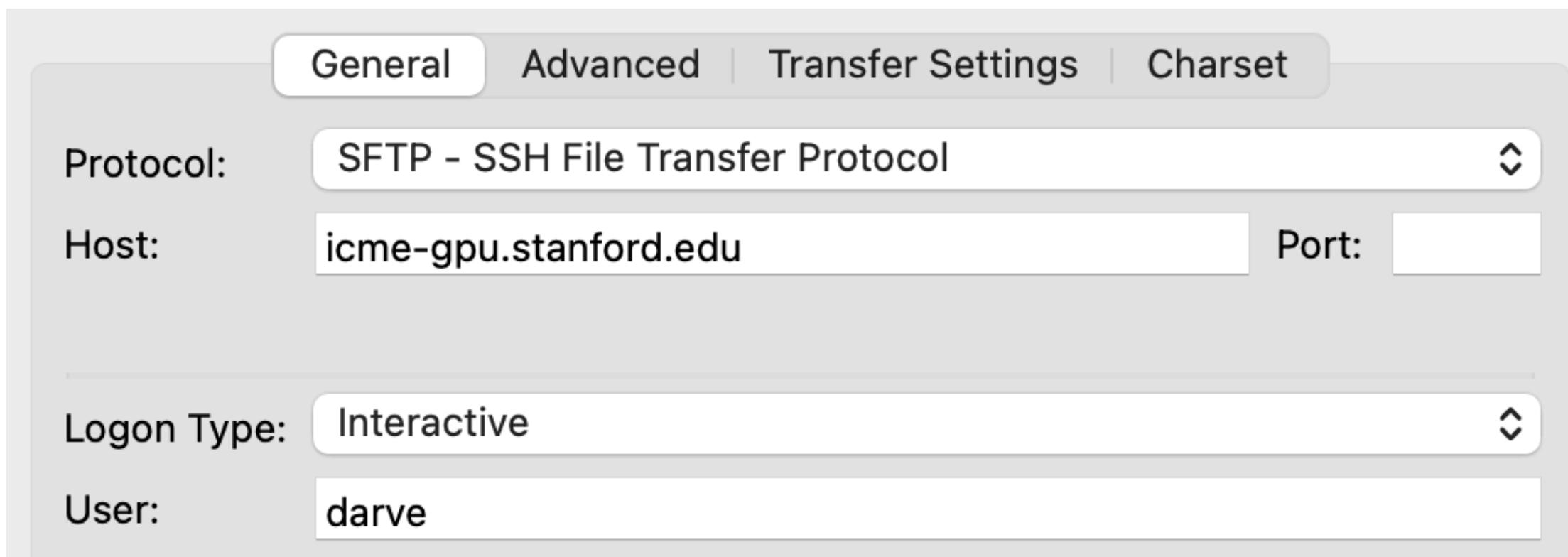
Connecting to the cluster

- Use scp or ssh.
- Example:
 - FileZilla
 - VSCode

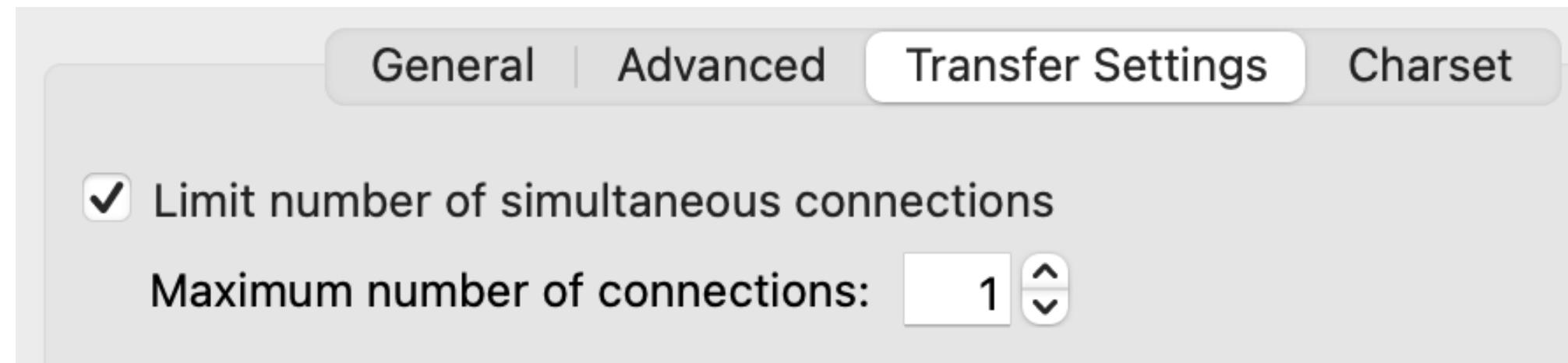
FileZilla



- Use Site Manager with these options.
- Make sure the Logon type is **interactive**.



- Select only **one connection** at a time to avoid having to enter your password repeatedly.

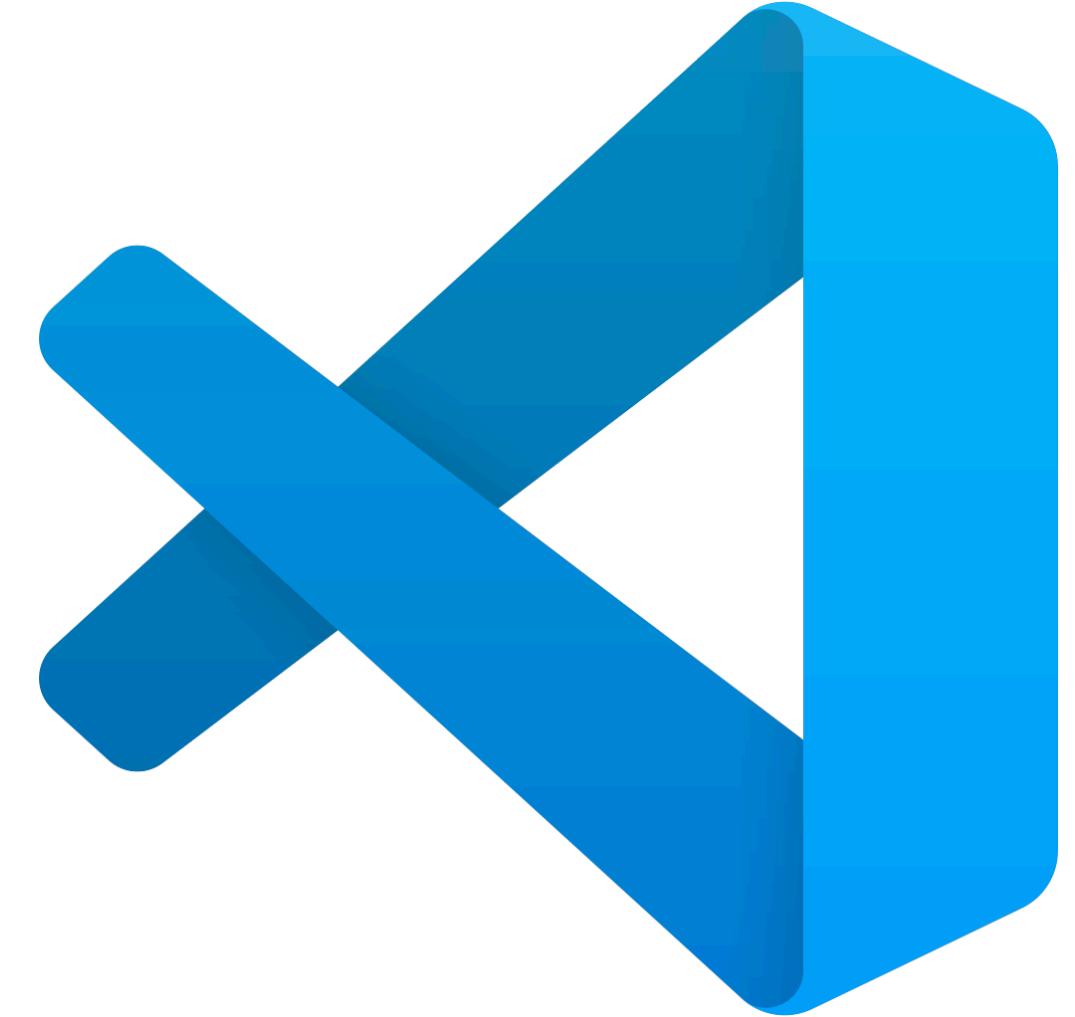


- Use graphical interface to transfer files.

FileZilla transfer window

Filename	Filesize	Filetype	Last modified	Filename	Filesize	Filetype	Last modified	Permissions	Owner/Group
..				..					
.vscode		Directory	01/17/2020 12:4...	.vscode		Directory	04/05/2022 1...	drwxrwxr-x	darve darve
Makefile	537	File	04/05/2022 14:3...	Makefile	537	File	04/05/2022 ...	-rw-rw-r--	darve darve
for_loop_openmp.cpp	825	cpp-file	04/06/2021 20:5...	for_loop_openmp.cpp	825	cpp-file	04/05/2022 1...	-rw-rw-r--	darve darve
hello_world_openmp.cpp	2,310	cpp-file	04/06/2021 20:5...	hello_world_openmp.cpp	2,310	cpp-file	04/05/2022 1...	-rw-rw-r--	darve darve
matrix_prod_openmp.cpp	2,714	cpp-file	04/06/2021 20:5...	matrix_prod_openmp.cpp	2,714	cpp-file	04/05/2022 1...	-rw-rw-r--	darve darve
matrix_prod_openmp_solution.cpp	2,755	cpp-file	04/06/2021 21:0...	matrix_prod_openmp_solution.cpp	2,755	cpp-file	04/05/2022 1...	-rw-rw-r--	darve darve
script.sh	70	Shell script	04/05/2022 14:...	script.sh	70	Shell script	04/05/2022 1...	-rw-rw-r--	darve darve
shared_private_openmp.cpp	938	cpp-file	04/06/2021 20:5...	shared_private_openmp.cpp	938	cpp-file	04/05/2022 1...	-rw-rw-r--	darve darve

Visual Studio Code



Open the Command Palette (F1, ⌘P) and enter:

Remote-SSH: Connect to Host

Enter your credentials.

Edit your files as if they were local.

```
$ source /etc/profile
```

may be needed

Visual Studio Code remote interface

The screenshot shows the Visual Studio Code interface with a remote workspace. The Explorer sidebar on the left lists files in the 'LECTURE_04' folder, including '.vscode', 'for_loop_openmp.cpp', 'hello_world_openmp.cpp' (which is currently selected), 'Makefile', 'matrix_prod_openmp_solution.cpp', 'matrix_prod_openmp.cpp', 'script.sh', and 'shared_private_openmp.cpp'. The main editor area displays the 'hello_world_openmp.cpp' file content:

```
1 #include <omp.h>
2 #include <cstdio>
3 #include <vector>
4 #include <sstream>
5 #include <iomanip>
6 #include <chrono>
7
8 using namespace std;
9
10#define SCALE 10000
11#define ARRINIT 2000
12
13 void DoWork(long tid, int &ndigits, long &etime)
14 {
15     auto start = chrono::system_clock::now();
16
17     const int digits = (2 + tid) * 28;
18     ndigits = (2 + tid) * 8;
19
20     ostringstream pi;
21     pi.fill('0');
22     std::vector<long> arr(digits + 1);
23
24     for (int i = 0; i <= digits; ++i)
25         arr[i] = ARRINIT;
```

At the bottom, the terminal window shows the command: `darve@icme-gpu:~/Lecture_04$`.

Compile

```
darve@icme-gpu1:~/2023/openMP$ make
g++ -std=c++11 -Wall -O2 -fopenmp -o for_loop_openmp for_loop_openmp.cpp
g++ -std=c++11 -Wall -O2 -fopenmp -o hello_world_dowork hello_world_dowork.cpp
g++ -std=c++11 -Wall -O2 -fopenmp -o hello_world_openmp hello_world_openmp.cpp
g++ -std=c++11 -Wall -O2 -fopenmp -o matrix_prod_openmp matrix_prod_openmp.cpp
g++ -std=c++11 -Wall -O2 -fopenmp -o matrix_prod_openmp_solution matrix_prod_openmp_solution.cpp
g++ -std=c++11 -Wall -O2 -fopenmp -o matrix_prod_simple matrix_prod_simple.cpp
g++ -std=c++11 -Wall -O2 -fopenmp -o shared_private_openmp shared_private_openmp.cpp
darve@icme-gpu1:~/2023/openMP$ █
```

run code

- The cluster has two types of nodes:
 - **the login node:** where you land when you log in; this node must not be used to run CPU intensive jobs.
 - **the compute nodes:** these are the nodes where you run your code.
- It's OK to compile on the login node.
- To run a code on the cluster, you need to use a queuing system.

The batch queue

- To run a code on the cluster, you need to use the queue.
- This will reserve a compute node and will run your program on that node.
- To do this, you need to write a SHELL script.
- This script will be run on the compute node by SLURM.
- SLURM: a cluster management and job scheduling system.

Example of batch script

- script.sh

```
1  #!/bin/bash
2  #SBATCH -p CME
3
4  ./hello_world_openmp
```

Fancier version

```
1 #!/bin/bash
2 #SBATCH --partition=CME
3 #SBATCH --cpus-per-task=16
4 #SBATCH --time=00:00:10
5 #SBATCH --nodes=1
6 #SBATCH --job-name=slurm
7 #SBATCH --output=slurm-%j.out
8 #SBATCH --error=slurm-%j.err
9
10 echo "Starting at `date`"
11 echo "Current working directory is $SLURM_SUBMIT_DIR."
12 echo "The master node of this job is: $SLURMD_NODENAME."
13 echo "Running on hosts: $SLURM_JOB_NODELIST"
14 echo "Using $SLURM_CPUS_PER_TASK CPUs per task."
15 echo "Number of CPUs on node: $SLURM_CPUS_ON_NODE."
16 echo
17
18 ./hello_world_openmp
19
20 echo
21 echo "Environment variables"
22 export
```

Run your script

```
$ sbatch script.sh
```

- This puts your job in the queue.
- When a resource is available (a compute node), SLURM will run your script on the node.
- The output goes to:

`slurm_[JOBID].out`

Example of output

```
1 Starting at Mon Apr 17 05:44:43 PM UTC 2023
2 Current working directory is /home/darve/2023/openMP.
3 The master node of this job is: icmet01.
4 Running on hosts: icmet01
5 Using 8 CPUs per task.
6 Number of CPUs on node: 8.
7
8 Let's compute pi =      3.
14159265358979323846264338327950288419716939937510582097494459230781640628620899
86280348253421170679
9 Hello World from thread = 4
10 Hello World from thread = 2
11 Hello World from thread = 6
12 [info] Number of threads = 8
13 Hello World from thread = 0
14 Hello World from thread = 3
15 Hello World from thread = 5
16 Hello World from thread = 1
17 Hello World from thread = 7
18 Thread 0 approximated Pi as      3141592653589793
19 Thread 2 approximated Pi as      31415926535897932384626433832795
20 Thread 6 approximated Pi as
3141592653589793238462643383279502884197169399375105820974944592
21 Thread 4 approximated Pi as
314159265358979323846264338327950288419716939937
22 Thread 3 approximated Pi as      3141592653589793238462643383279502884197
23 Thread 7 approximated Pi as
314159265358979323846264338327950288419716939937510582097494459230781640
24 Thread 1 approximated Pi as      314159265358979323846264
25 Thread 5 approximated Pi as
31415926535897932384626433832795028841971693993751058209
26 Thread 0 computed 16 digits of pi in 143 musecs ( 8.938 musec per digit)
27 Thread 1 computed 24 digits of pi in 375 musecs ( 15.625 musec per digit)
28 Thread 2 computed 32 digits of pi in 270 musecs ( 8.438 musec per digit)
29 Thread 3 computed 40 digits of pi in 363 musecs ( 9.075 musec per digit)
30 Thread 4 computed 48 digits of pi in 323 musecs ( 6.729 musec per digit)
31 Thread 5 computed 56 digits of pi in 387 musecs ( 6.911 musec per digit)
32 Thread 6 computed 64 digits of pi in 313 musecs ( 4.891 musec per digit)
33 Thread 7 computed 72 digits of pi in 358 musecs ( 4.972 musec per digit)
```

sbatch vs srun

- The `srun` command is designed for interactive use, with someone monitoring the output.
- The output of the application is seen as output of the `srun` command, typically at the user's terminal.
- The `sbatch` command is designed to submit a script for later execution and its output is written to a file.
- Command options used in the job allocation are almost identical.

srun example

```
darve@icme-gpu1:~/2023/openMP$ srun --cpus-per-task=2 -p CME ./hello_world_openmp
Hello World from thread = 0
Number of threads = 2
Hello World from thread = 1
darve@icme-gpu1:~/2023/openMP$ srun --cpus-per-task=4 -p CME ./hello_world_openmp
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 0
Number of threads = 4
darve@icme-gpu1:~/2023/openMP$ sbatch ./script.sh
Submitted batch job 7269
darve@icme-gpu1:~/2023/openMP$ █
```

Other SLURM commands

- **sbatch/srun**: run a script on a compute node
- **squeue**: list the jobs currently in the queue
- **sinfo**: view information about Slurm nodes and partitions
- **scancel**: used to cancel a pending or running job or job step

SLURM documentation

- <https://slurm.schedmd.com/overview.html>
- <https://slurm.schedmd.com/tutorials.html>

Let's compile our first OpenMP code

- Header file:

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

Compiler flag

Compiler / Platform	Compiler	Flag
Intel Linux Opteron/Xeon	icc icpc ifort	-openmp
PGI Linux Opteron/Xeon	pgcc pgCC pgf77 pgf90	-mp
GNU Linux Opteron/Xeon IBM Blue Gene	gcc g++ g77 gfortran	-fopenmp

OpenMP version supported by the compiler

```
darve@icme-gpu:~$ echo | cpp -fopenmp -dM | grep OPENMP
#define _OPENMP 201511
darve@icme-gpu:~$ █
```

Mapping table

Year	OpenMP version
200505	2.5
200805	3.0
201107	3.1
201307	4.0
201511	4.5
201811	5.0

Structure of an OpenMP program

- OpenMP breaks the code into sequential and parallel regions.
- **Sequential region:** only a single thread executes that block of code.
- **Parallel regions:**
 - OpenMP creates a pool of threads available to do work.
 - This is more efficient than creating threads. The pool of threads is reused throughout the program and efficiently managed by openMP.
 - All threads in the pool run the parallel region.

Hello World example

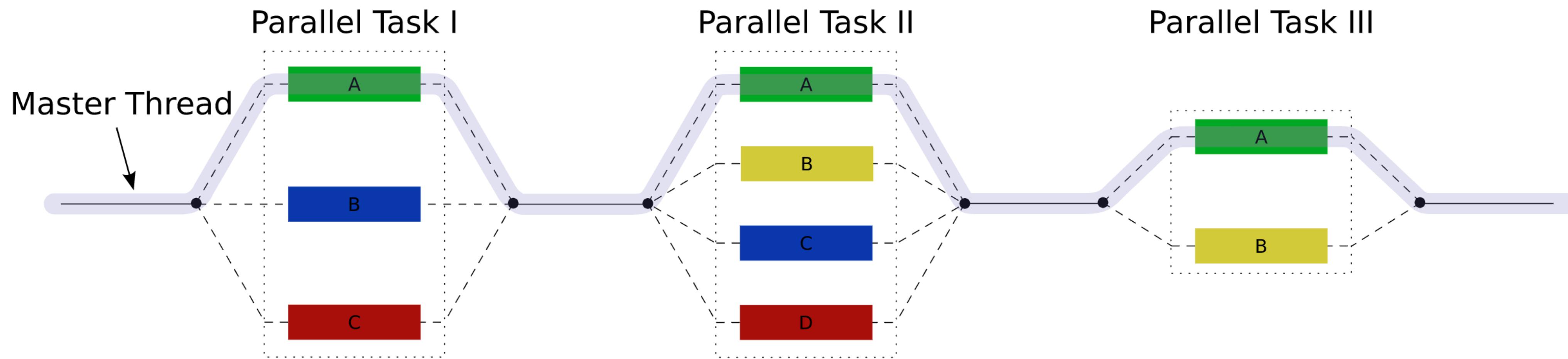
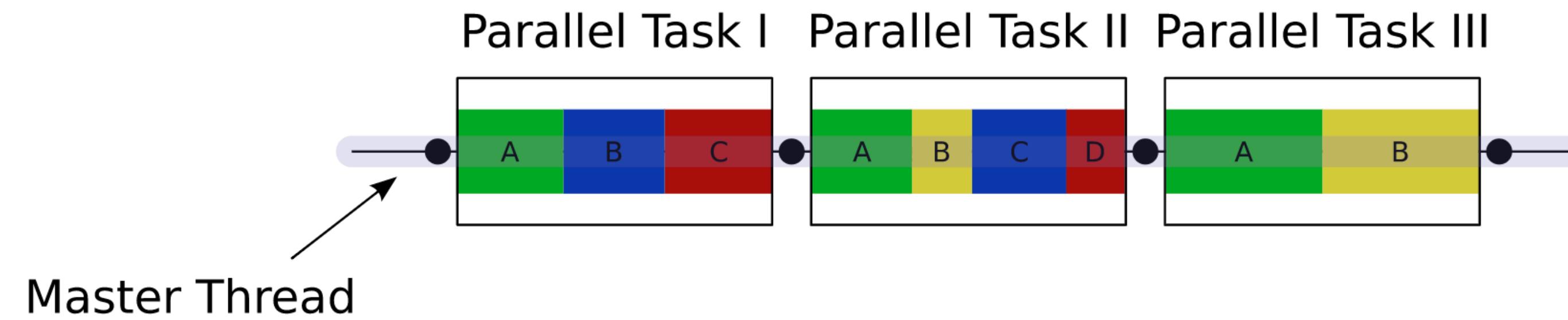
File `hello_world_openmp.cpp`

```
darve@icme-gpu1:~/2023/openMP$ srun --cpus-per-task=8 -p CME ./hello_world_openmp
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 2
Hello World from thread = 5
Hello World from thread = 4
Hello World from thread = 7
Hello World from thread = 3
Hello World from thread = 6
Hello World from thread = 1
darve@icme-gpu1:~/2023/openMP$ █
```

Directives

- `#pragma omp parallel`
- Creates a parallel region with the default number of threads.
- All threads execute the following block of code.

fork-join model



Example: computing π using openMP

- Formula used to compute π :

$$\frac{\pi}{2} = 1 + \frac{1}{3} \left(1 + \frac{2}{5} \left(1 + \frac{3}{7} \left(1 + \frac{4}{9} \left(1 + \dots \right) \right) \right) \right)$$

- Each thread will compute a varying number of digits in π .
- File `hello_world_dowork.cpp`

```
3.141592653589793238462433832795028841971693993751058209749445
9230716406286208998628034825342117067982148086513282306647938
4460955058223172359408128481117450284102701938521105559644
6229489549303819644288109756659334461284756482337867831652
7120190914564856692346034861045432664821339360726024914127
37245870660631558817488152092096282925409175364367892590
3600113305305488204665213841469519415116094330572703657595
919530921861738193261179310511854807446237996274956735188575
2724891227938183011949129833673362440656643086021394946395224
73719070217986094370277053921717629317675238467481846766940513
20005681271452635608277857713427577896091736371787214684409012
2495343014654958537105079227968925892354201956112129021960864
0344181598136297747713099605187072113499999837297804995105973
17328160963185950244594553469083026425223082533446850352619311
88171010003137838752886587533208381420617177669147303598253490
42875546873115956286388235378759375195778185778053217122680661
30019278766111959092164201989380952572010654858632788659361533
81827968230301952035301852968995773622599413891249721775283479
13151574857242454150695950829533116861727858890750983875463
7464939192550604009277016711390098488240128583616035637076601
0471018194295559619894676783749448255379774726847104047534646
2080466842590694912933136770289891521047521620569660240580381
5019351125338243003558764024749647326391419927260426992279678
2354781636009341721641219924586315030286182974555706749938505
494588586926995690927210797509302955321165344987202759602364
80665499119881834797753566369807426542527862551818417546728
909777279380008164706001614524919217321721477235014419735
6854816136115735252133475741849468438523323907394143345477
624168625189835694855620992192218427255025425688767179049460
1653466804988627237917860857843838279679766814541009538837863
6095068064225125020511739284896084128486269456042419652850222
106611863067442786220391949450471237137869609563643719172874677
```

Output

```
darve@icme-gpu1:~/2023/openMP$ srun --cpus-per-task=8 -p CME ./hello_world_dowork
Let's compute pi = 3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679
Hello World from thread = 4
Hello World from thread = 1
Hello World from thread = 2
[info] Number of threads = 8
Hello World from thread = 0
Hello World from thread = 7
Hello World from thread = 6
Hello World from thread = 3
Hello World from thread = 5
Thread 0 approximated Pi as 3141592653589793
Thread 4 approximated Pi as 314159265358979323846264338327950288419716939937
Thread 7 approximated Pi as 314159265358979323846264338327950288419716939937510582097494459230781640
Thread 2 approximated Pi as 31415926535897932384626433832795
Thread 1 approximated Pi as 314159265358979323846264
Thread 3 approximated Pi as 3141592653589793238462643383279502884197
Thread 5 approximated Pi as 31415926535897932384626433832795028841971693993751058209
Thread 6 approximated Pi as 3141592653589793238462643383279502884197169399375105820974944592
Thread 0 computed 16 digits of pi in 122 musecs ( 7.625 musec per digit)
Thread 1 computed 24 digits of pi in 343 musecs ( 14.292 musec per digit)
Thread 2 computed 32 digits of pi in 303 musecs ( 9.469 musec per digit)
Thread 3 computed 40 digits of pi in 312 musecs ( 7.800 musec per digit)
Thread 4 computed 48 digits of pi in 228 musecs ( 4.750 musec per digit)
Thread 5 computed 56 digits of pi in 321 musecs ( 5.732 musec per digit)
Thread 6 computed 64 digits of pi in 366 musecs ( 5.719 musec per digit)
Thread 7 computed 72 digits of pi in 225 musecs ( 3.125 musec per digit)
darve@icme-gpu1:~/2023/openMP$ █
```

Using Google Colab

▼ OpenMP

Hello World in OpenMP

File
openMP.ipynb

File: `hello_world_openmp.cpp`

```
[ ] 1 !name=hello_world_openmp; g++ -I. -o $name $name.cpp gtest_main.a -fopenmp && ./$name

Running main() from googletest-1.13.0/googletest/src/gtest_main.cc
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from openmp
[ RUN      ] openmp.hello_world
Hello World from thread = 0
Number of threads = 2
Hello World from thread = 1
[       OK ] openmp.hello_world (0 ms)
[-----] 1 test from openmp (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

for loops

- So far we have not done much with OpenMP.
- The syntax was simpler than `std::thread` but the coding is still cumbersome.
- In most cases, scientific codes can be parallelized by partitioning for loops across multiple threads.

Manual partitioning

- Following our previous examples, we can manually partition the for loop across threads.
- The code runs correctly and will use all the threads.
- See `matrix_prod_simple.cpp`

```
#pragma omp parallel
{
    const long tid = omp_get_thread_num();
    const int n_threads = omp_get_num_threads();
    for (int i = tid; i < size; i += n_threads)
```

Output

```
darve@icme-gpu1:~/2023/openMP$ srun -c 2 -p CME ./matrix_prod_simple
Running main() from /home/darve/googletest-1.13.0/googletest/src/gtest_main.cc
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from matrix_prod
[ RUN    ] matrix_prod.simple
Size of matrix = 512
Number of threads to create = 2
[      OK  ] matrix_prod.simple (221 ms)
[-----] 1 test from matrix_prod (221 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (221 ms total)
[  PASSED  ] 1 test.
darve@icme-gpu1:~/2023/openMP$ █
```

OpenMP directives

- But openMP provides a much more concise and simple mechanism to do this.
- This is a very common operation in scientific computing.
- OpenMP has many mechanisms to optimize this type of parallelism.

pragma omp for

- `#pragma omp parallel` creates a parallel region
- Adding `for` at the end results in a parallel execution of the for loop that follows.
- See `matrix_prod_openmp.cpp`

```
#pragma omp parallel for
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j) {
            float c_ij = 0;
            for (int k = 0; k < size; ++k) {
                c_ij += MatA(i, k) * MatB(k, j);
            }
            mat_c[i * size + j] = c_ij;
        }
```

Syntax options

```
#pragma omp parallel  
#pragma omp for  
|   for (int i = 0; i < size; ++i)
```

```
#pragma omp parallel for  
|   for (int i = 0; i < size; ++i)
```

Performance and scalability

```
darve@icme-gpu1:~/2023/openMP$ srun -c 16 -p CME ./matrix_prod_openmp
Running main() from /home/darve/googletest-1.13.0/googletest/src/gtest_main.cc
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from matrix_prod
[ RUN      ] matrix_prod.parallel_for
Size of matrix = 512
Number of threads to create = 2
[       OK  ] matrix_prod.parallel_for (183 ms)
[ RUN      ] matrix_prod.nthreads
Number of threads: 1; elapsed time [millisec]: 117
Number of threads: 2; elapsed time [millisec]: 59
Number of threads: 4; elapsed time [millisec]: 37
[       OK  ] matrix_prod.nthreads (213 ms)
[-----] 2 tests from matrix_prod (397 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (397 ms total)
[ PASSED  ] 2 tests.
darve@icme-gpu1:~/2023/openMP$ █
```

Loop scheduling

- Loop scheduling is important for performance.
- Some fine-tuning is often required.
- A parallel for loop often has a barrier at the end.
- At the barrier, the other members of the team must wait for the last thread to arrive.
- To minimize this wait time, shared work should be distributed so that all threads arrive at the barrier at about the same time.
- The schedule clause allows optimizing this execution.

General syntax

```
#pragma omp for schedule(kind,chunk_size)
```

kind = static, dynamic, guided

chunk_size the exact meaning varies depending on kind but generally this is the number of iterations in a given chunk assigned to a thread.

static

- This is the simplest.
- chunk size is fixed (and equal `chunk_size`).
- The chunk assignment is done using a round-robin assignment.
- Pro: low-overhead
- Con: will not work well if the runtime of each iteration varies significantly.