universität innsbruck

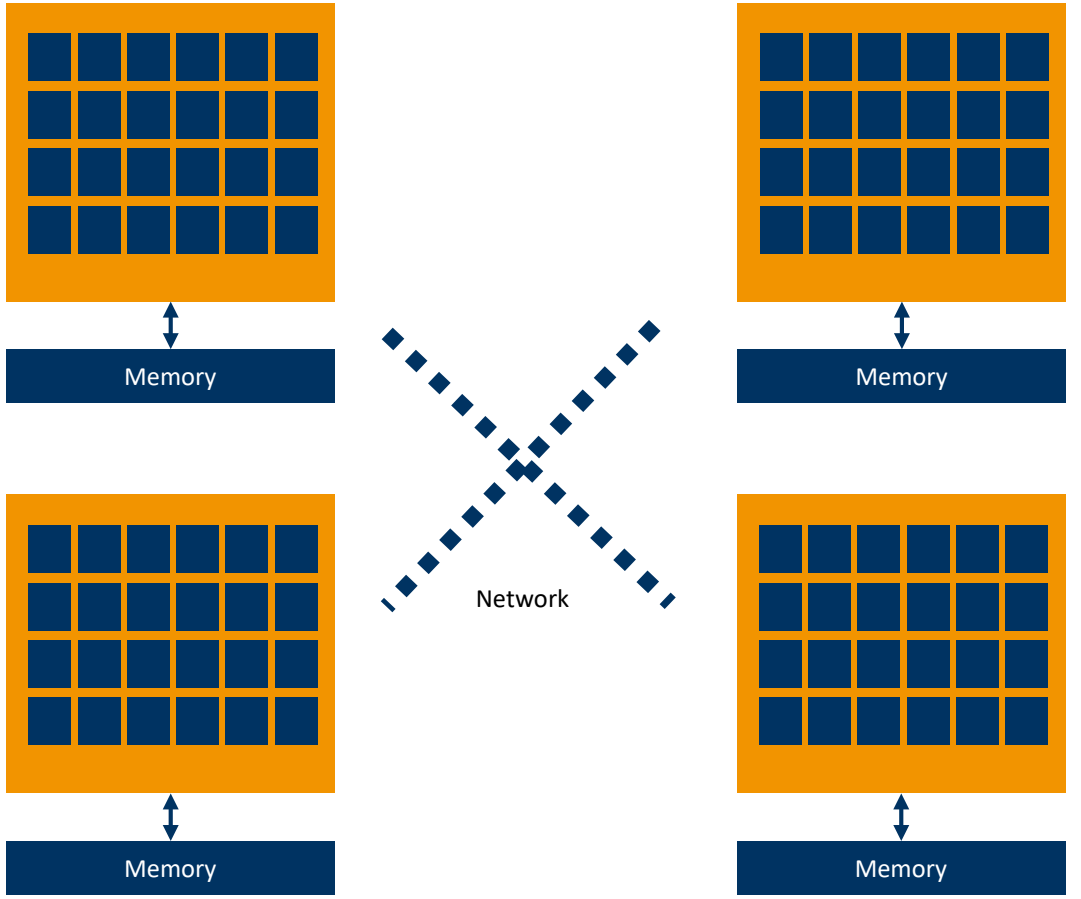
FZ HPC

1st Summer School in HPC and AI (UniBZ, June 2021)
Introduction to Shared Memory Programming with OpenMP

Philipp Gschwandtner, Research Center HPC, University of Innsbruck
with special thanks to Lukas Einkemmer and Rolf Rabenseifner on whose original slide sets parts of this course are based

Motivation: Why use Parallelism?

- Modern supercomputers are built by connecting a large number of individual compute nodes
- Each node can have multiple CPUs and cores
  - illustration on the right is simplified
- Parallelization is **essential** to exploit modern hardware!



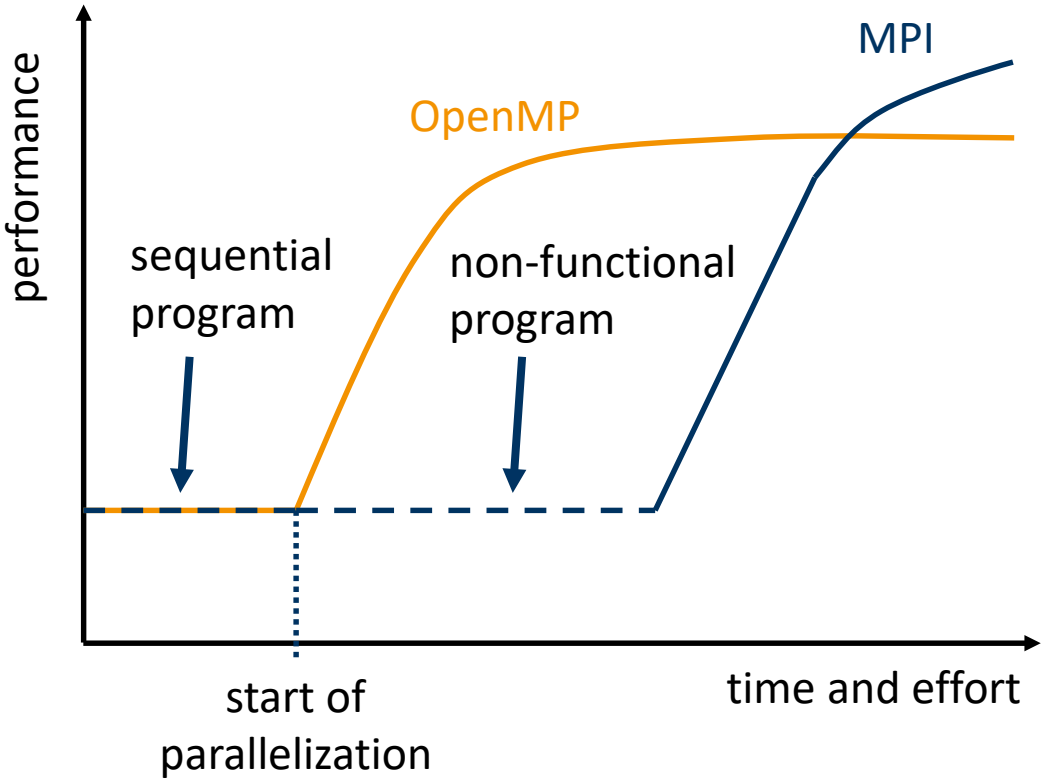
Motivation: Why consider using OpenMP?

- OpenMP is one of the easiest parallel programming models & widely available
  - however, restricted to shared memory hardware (=no network)
  - there are alternatives, e.g. Intel TBB, but none as widely spread and/or mature

- modern hardware encourages use of such models
  - AMD x86 desktop: Threadripper 3990X with 64 cores and 128 threads
  - Intel x86 server: 8x Xeon Platinum 827x or 828x with 28 cores / 56 threads = 224/448
  - Marvell ARM: ThunderX3: 96 cores and 384 threads
  - exotic hardware: SGI Altix UV (“Mach 2” @ JKU in Linz, Austria) with 4096/8192

Motivation: Incremental Parallelization

- MPI
  - initially a sequential program
  - start to parallelize
  - program won't work until major parts of parallelization present
- OpenMP
  - initially a sequential program
  - parallelize incrementally
  - program remains functional throughput parallelization process



OpenMP

- thread-based programming model for shared memory parallelism
  - provides a higher level of abstraction compared to pthreads, C++ STL threads, etc.
  - e.g. “run this loop in parallel” vs. “execute this list of statements asynchronously”
- de-facto standard for C/C++ and Fortran
- maintained by the OpenMP Architecture Review Board
  - initial release in 1997 (version 1.0 for Fortran)
  - updates in 1998 (1.0 for C/C++), 2000 (2.0), 2005 (2.5), 2008 (3.0), 2011 (3.1), 2013 (4.0), 2018 (5.0), 2020 (5.1)
  - this slide set assumes at least OpenMP 3.1!
  - https://www.openmp.org/specifications/

Detour: Parallelism Terminology

- Thread
  - is a set of sequential instructions that are executed in order
  - is a software construct
  - often mapped to a single core
- Core
  - is a set of hardware components that process instructions of a thread
  - is a hardware construct
  - sometimes, cores are partially split into hardware threads (e.g. HyperThreading)
- Shared memory
  - assumes all threads have direct read and write access to the same memory
- Distributed memory
  - not all threads have direct read/write access
  - data transfers via network are required

OpenMP vs. MPI, Shared vs. Distributed Memory

- shared memory
  - single memory address space
  - usually based on threads
  - all data can be accessed directly
  - synchronization (e.g. barriers) required to ensure correctness

```
lock();
x[0] += 42;
unlock();
```

- distributed memory
  - multiple memory address spaces
  - usually based on processes
  - data cannot be accessed directly
  - message exchange required to get data and ensure synchronization

```
x = recv_data(…);
x[0] += 42;
send_data(x, …);
```

OpenMP's Main Characteristics

- compiler-based parallelization model
  - tell the compiler what should happen and when/where
  - compiler and runtime system do the rest of the job for you
- portable across many hardware architectures / platforms
- runtime system responsible for managing threads, scheduling, affinity, etc.
- C/C++ and Fortran bindings
  - even a research compiler for Java is available…
- aims at minimal changes to sequential code

```
#pragma omp parallel for
for(int i = 0; i < N; i++) {
    out[i] = in[i];
}
```

OpenMP's Caveats

- compiler-based parallelization model
  - tell compiler what code to run in parallel and when to synchronize
  - tell compiler whether to share data among threads or create private copies
  - but compiler cannot/will not check semantic correctness
    - Won't somebody please think of the compiler developers?

Helen Lovejoy

- only works in shared memory
- no guaranteed performance portability
  - it will run on any hardware, but maybe not as fast

OpenMP Implementations

- many implementations available
  - GCC, LLVM, Intel, Microsoft, etc.
  - allow to run OpenMP virtually on every platform
  - compiler and runtime support required
    - sometimes interchangeable components
    - check https://www.openmp.org/resources/openmp-compilers-tools/ for compiler support
- do not confuse implementation adherence with specification adherence
  - many minor semantics in OpenMP are implementation-defined

How to Choose between OpenMP and MPI?

- all considerations on the last two slides, plus:
  - OpenMP is a language extension, hence requires compiler support
  - MPI is a library, hence compiler-independent

- often used together, referred to as “hybrid” parallelism, e.g.
  - one MPI process per shared memory node, CPU, or NUMA domain
  - multiple OpenMP threads per MPI process for the individual cores / hardware threads

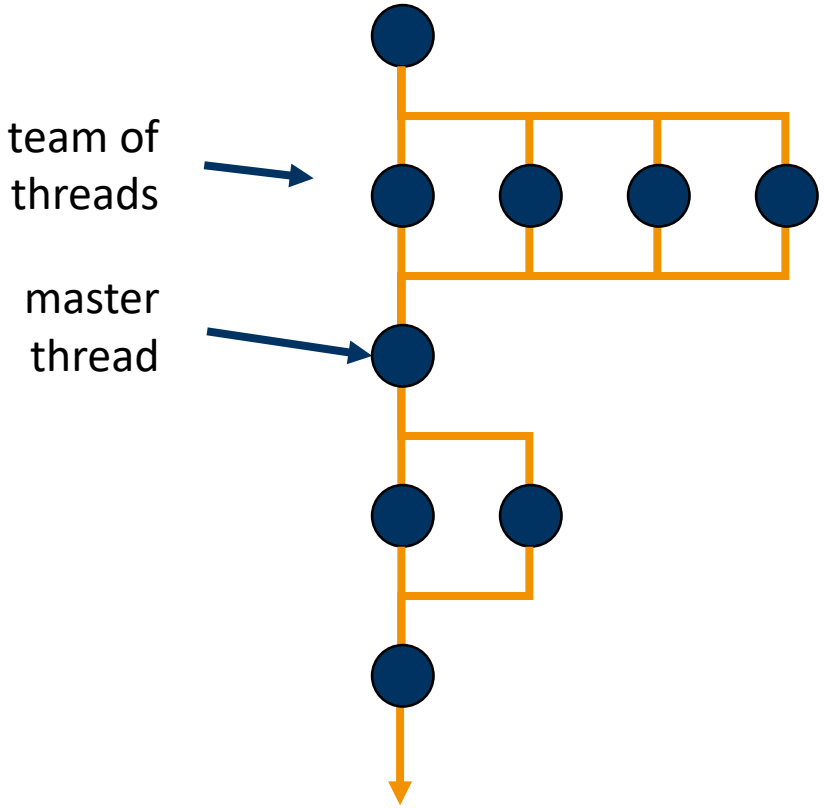
- Choose your weapon wisely!

OpenMP Programming and Execution Model

Execution Model

- fork-join parallelism
  - program starts sequentially
  - parallel regions can be opened, which spawn new threads
  - end of parallel regions synchronize threads
  - afterwards, execution continues sequentially

- There is no guarantee in which order the threads are executed
  - specific order can be enforced, but this is **very expensive and usually not desired**

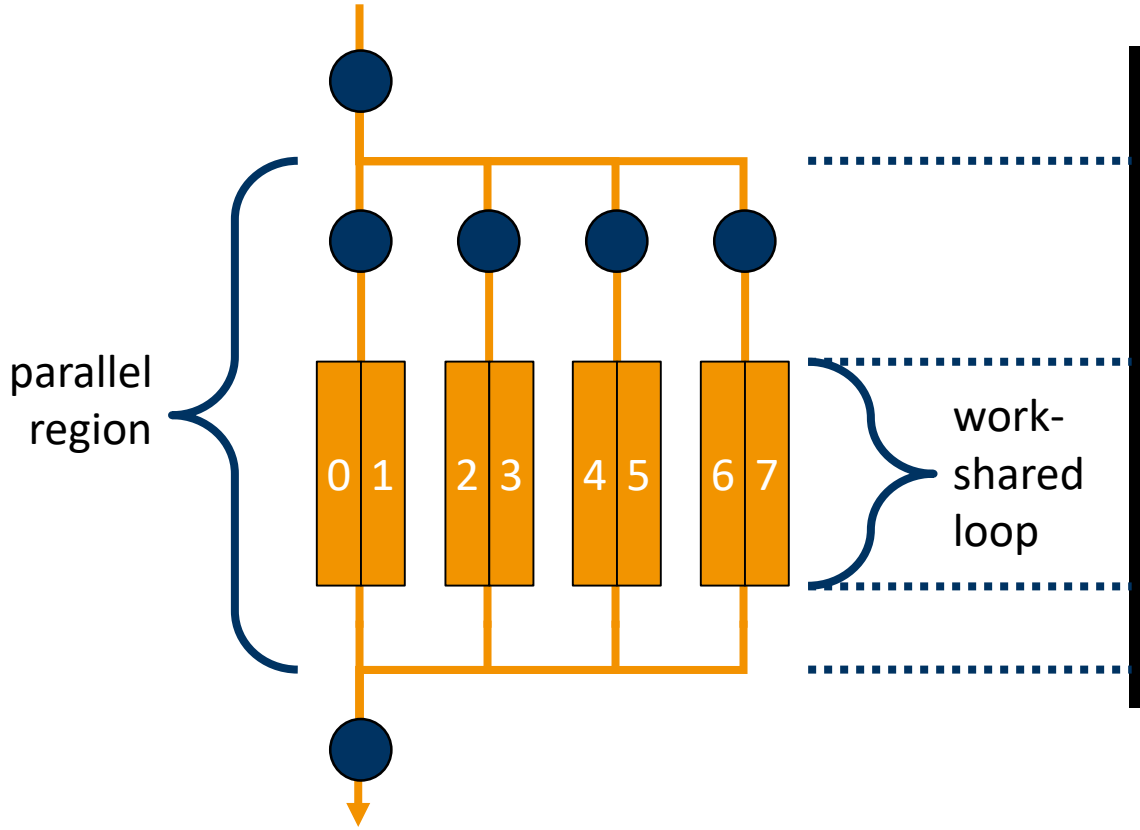


Programming Model

- mark code regions with *directives* or *pragmas* (version 5.1: also C++ attributes), e.g.
  - parallel regions
  - work to be distributed
  - thread synchronization
- add *clauses* for further information, e.g.
  - which variables to share, which not to
  - scheduling strategies
- any valid OpenMP program must be a valid sequential program if all pragmas are removed!
  - easy to do: remove OpenMP compiler flag
  - facilitates debugging

```
int f = ...
#pragma omp parallel shared(a,b,c,f)
  default(none)
{
  #pragma omp for
  for(int i = 0; i < 8; ++i) {
    c[i] = a[i] + b[i] * f;
  }
}
```

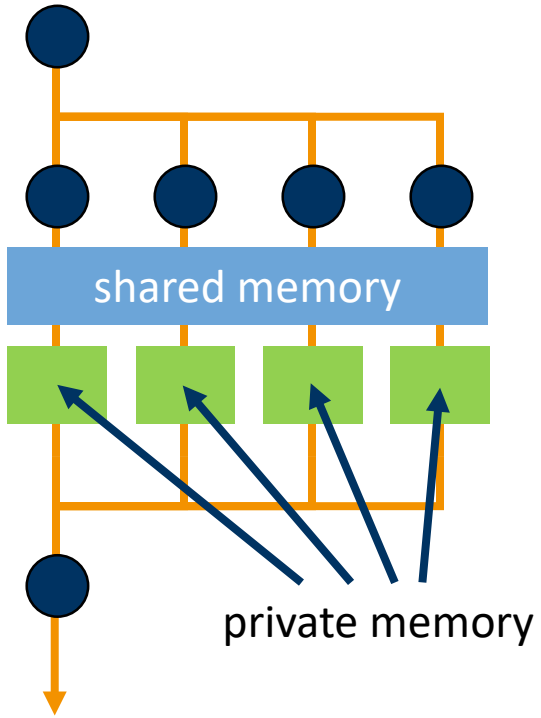
Programming Model cont'd




```
int f = ...
#pragma omp parallel shared(a,b,c,f)
  default(none)
{
  #pragma omp for
  for(int i = 0; i < 8; ++i) {
    c[i] = a[i] + b[i] * f;
  }
}
```

Memory Model

- OpenMP is based on threads
  - all threads have access to global, shared data
  - each thread has additional local, private data
  - modifications to private data are not visible across threads
  - modifications to shared data are visible across threads and need to be done carefully



OpenMP API

OpenMP API

- pragmas (also “directives”)
  - control constructs
    - parallelism & work sharing
  - data sharing
    - private & shared variables, initialization
  - synchronization
    - critical & atomic sections, barriers
- library functions
  - querying/controlling runtime system
  - timing
  - locking
- environment variables
  - degree and nesting of parallelism
  - loop scheduling
  - thread mapping and binding

Pragmas

- `#pragma omp` *`directive`* `[` *`clause`* `[, clause ] ]` *`(newline)`*
- pragmas must be on their own source code line and end with a newline
- OpenMP directives can often take a number of optional clauses, possibly with parameters
- pragmas have dynamic and lexical extent
  - e.g. `#pragma omp for` must always be nested in `#pragma omp parallel`
  - but not necessarily statically (see example on the right)

```
void bar() {
    #pragma omp for
    for(...) { ... }
}

void foo() {
    #pragma omp parallel
    bar();
}
```

Combined Pragmas

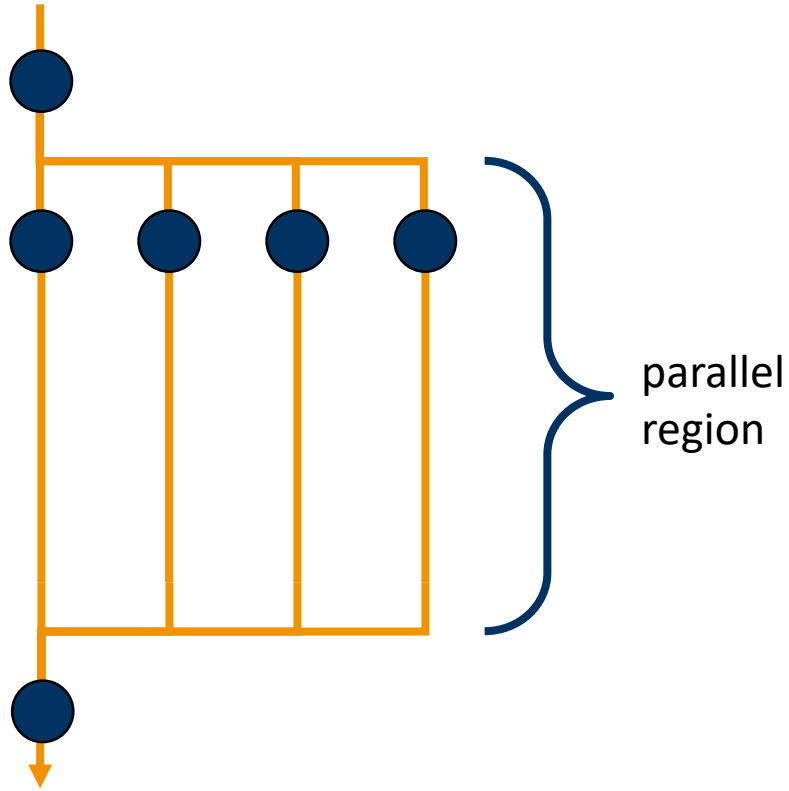
```
#pragma omp parallel
{
  #pragma omp for
  for(int i = 0; i < 8; ++i) {
    c[i] = a[i] + b[i] * f;
  }
}
```

```
#pragma omp parallel for
for(int i = 0; i < 8; ++i) {
  c[i] = a[i] + b[i] * f;
}
```

Most Important Directive: `parallel`

- `#pragma omp parallel`
  - must be followed by a statement or another OpenMP construct
  - master thread creates a team of threads, each executing the code **redundantly**
  - implicit **"barrier"** at the end (threads in team synchronize), only master continues

- `parallel` may also be nested
  - but with great power comes great responsibility…
  - Do not nest unless explicitly required!



Hello World in OpenMP

```c
#include <stdio.h>

int main() {
  #pragma omp parallel
  {
    printf("Hello World\n");
  }
  return 0;
}
```

```
gcc hello.c –o hello -fopenmp
./hello
Hello World!
Hello World!
Hello World!
```

Library Functions

- The header file `omp.h` provides library functions for various aspects
  - e.g. querying the number of threads and individual thread IDs

```
#include <omp.h>
...
#pragma omp parallel
{
  if(omp_get_thread_num() == 0) {
    count << "Number of threads: "
      << omp_get_num_threads() << endl;
  }

  count "Hello world from thread "
    << omp_get_thread_num() << endl;
}
```

Controlling the Number of Threads

- Using environment variables
  - OMP_NUM_THREADS=4 ./program
- Using clauses
  - #pragma omp parallel num_threads(4)
- Using library functions
  - omp_set_num_threads(4);

- The default is implementation-defined and often not a good choice
  - Intel: Single thread
  - GCC: All hardware threads in the system (incl. hyperthreads)
- Rule of thumb:
  - number of threads = number of cores

Time Measurements

- OpenMP provides wall clock timers
- Precision can be queried using `omp_get_wtick()`
- Note that timers such as `clock()` return “CPU time”
  - accumulated execution time across all threads used by the program

```
double time_start = omp_get_wtime();
// ... do something ...
double time_end = omp_get_wtime();

double duration = time_end - time_start;
```

Compilation and Execution

- compile as usual but include OpenMP-specific flag
  - e.g. gcc/clang/ARM: `-fopenmp`, Intel: `-qopenmp`, IBM: `-qsmp=omp`
- execute as usual, but set required environment variables
  - e.g. `OMP_NUM_THREADS` for controlling degree of parallelism
- be sure to properly set up your job submission on clusters
  - e.g. parallel environments for SLURM, SGE, etc.

Detour: Sequential Debugging Made Easy

- Want to test your code without OpenMP parallelization?
  - simply omit the OpenMP-specific compiler flag
  - pragmas will be ignored
  - greatly facilitates debugging

```
#ifdef _OPENMP
// code that requires OpenMP
// header/library
#endif
```

- only need to take care of code that requires library functions

Data Sharing in OpenMP

- Two types of variables
  - **shared:** global variables, read-only data, usually arrays, etc.
  - **private:** local or temporary variables, loop counters, etc.

```
// shared integer
int n = 10;
// shared array
vector<double> in(n);

#pragma omp parallel for
for(int i = 0; i < n; i++) {
    // private double
    double x = 3 * in[i];
    in[i] = x;
}
```

Data Sharing Clauses

- `private`
  - each thread gets a private copy of variable, independent of original variable
  - private copy is **not** initialized (C++: default constructor is called)
  - default for variables declared inside parallel region and loop counters of parallel loops
  - often better to declare variables inside parallel region, reduces amount of code (also minimizes *"vertical distance"* in source code)

- `shared`
  - each thread references the same, global copy
  - data races if access is not synchronized
  - default for variables declared outside parallel region and global variables, often used for read-only access

- `default`
  - can be set to `shared`, or `none` for C/C++
  - `default(none)` helpful for detecting missing variables in clauses (compiler will complain!)

Data Sharing Clauses cont'd

```
int f = ...
#pragma omp parallel shared(a,b,c,f)
  private(temp) default(none)
{
  #pragma omp for
  for(int i = 0; i < n; ++i) {
    temp = b[i] * f
    c[i] = a[i] + temp;
  }
}
```

```
double x = 3;
#pragma omp parallel private(x)
{
    // <- here x is NOT equal to 3
    x = 5;
}
// <- here x is NOT equal to 5
```

Data Sharing Clauses cont'd

- `firstprivate`
  - like `private`, but private copies are initialized with value of copy outside of parallel region
  - C++: copy constructor is called

- `lastprivate`
  - like `private`, but outside copy is set to the private copy of the final iteration (`for` loops) or last section (`sections`), **NOT** the iteration/section that was chronologically executed last

- `threadprivate`
  - like private, but will persist across parallel regions
  - master thread variable is storage-associated with original variable (not the case for `private`!)

Race Conditions

- OpenMP is easy to write, but easy to get wrong
- Most responsibility is delegated to the application developer
  - compiler will only do very basic checks for you

```
int x = 0;

#pragma omp parallel
{
    // short form for x = x + 1
    // x is read and written by all
    // threads! Race condition!
    x += 1;
}
```

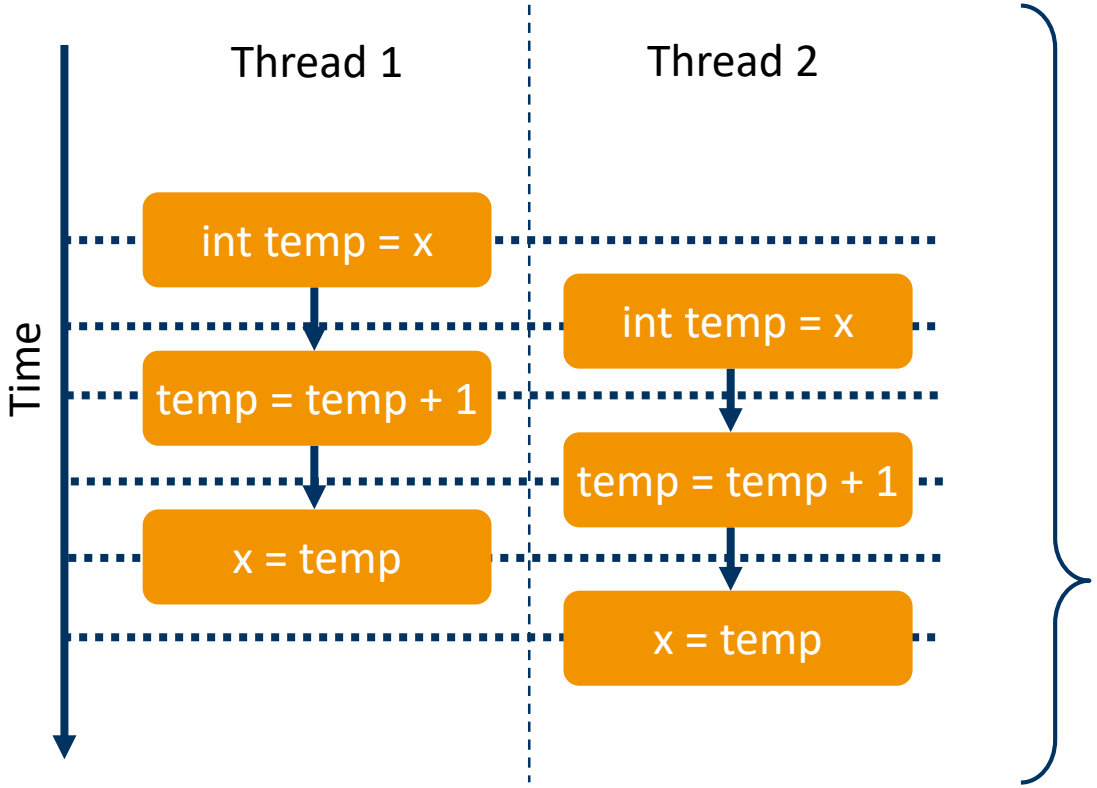
Race Conditions cont'd

- a race condition occurs when
  - **multiple threads** can access
  - the **same memory location**
  - at the **same time** and
  - at least one access is a **write** operation
- a program with a race condition is **always** incorrect
  - even if it manages to (sometimes) compute correct results
  - result is non-deterministic (depends on execution order)

```
int x = 0;

#pragma omp parallel
{
    // short form for x = x + 1
    // x is read and written by all
    // threads! Race condition!
    x += 1;
}
```

Race Conditions cont'd




```
int x = 0;

#pragma omp parallel
{
    // short form for x = x + 1
    // x is read and written by all
    // threads! Race condition!
    x += 1;
}
```

Detour: Data Sharing, Arrays and Pointers in C

- Arrays in C are accessed via a pointer
  - copying the pointer does **not** copy the underlying array but only the reference to it
- copying array pointer usually not required
  - array elements are accessed via individual indices
  - pointer itself is usually only read, even when writing to array

```
// shared integer
int n = 10;
// shared array
int a[n]

#pragma omp parallel for
for(int i = 0; i < n; i++) {
    a[i] += 3; // no race condition!
}
```

Exercises

- Connect to the cluster and copy the exercise from to your working directory
  - e.g. `cp -r /home/clusterusers/sc/Day_3 ~/Day_3`
- `Day_3/openmp/exercises` contains template source code used for the following exercises
- `Day_3/openmp/solutions` contains possible implementations
  - Try yourself first, otherwise you'll miss out on the learning experience!
- `Day_3/openmp.tar.gz` contains today's exercises and additional code examples

Exercise 1

- Goals:
  - runtime library functions
  - conditional compilations
  - environment variables
  - parallel regions with private and shared clauses

- A sequential hello world program is provided
  - `exercises/hello/hello.c`

Exercise 1a

- Compile the program and run as shown on the right side

```
gcc hello.c –o hello -fopenmp
export OMP_NUM_THREADS=4
./hello
```

- Expected result:
  - program is not parallelized so nothing changes

Exercise 1b

- Add a parallel region that prints the ID of each thread and the total number of threads
- Compile and run with 4 threads
- Example output shown on the right
- Why does the order of the output change from run to run?

```
OMP_NUM_THREADS=4 ./hello
I am thread 0 of 4 threads
I am thread 2 of 4 threads
I am thread 3 of 4 threads
I am thread 1 of 4 threads
```

Exercise 1c

- Introduce a race condition by “forgetting” to put a `private` clause on the `omp` parallel directive. Can you observe the race condition
  - with optimization turned on (`-O3`) and turned off (`-O0`)?
  - by increasing the number of threads?
  - by adding a `sleep(1)` just before the `printf()`?
- Example output shown on the right
- Why do you observe correct results for some configurations/runs even though there is a race condition in the program?

```
OMP_NUM_THREADS=4 ./hello
I am thread 2 of 4 threads
I am thread 2 of 4 threads
I am thread 2 of 4 threads
I am thread 2 of 4 threads
```

Exercise 1d

- Check that the program still works if OpenMP is turned off
- Add a statement that informs the user that OpenMP is not used.
- Example output shown on the right

```
g++ hello.c -o hello
./hello
The program is not compiled
with OpenMP
```

Work Sharing

Work Sharing Directives

- distribute execution of following code region among existing threads
- must be enclosed in parallel region, cannot be directly nested
- do not launch new threads but assign work to existing threads
- no barrier on entry
- implicit barrier on exit
  - unless `nowait` clause specified

- `for`
- `sections`
- `single`
- `task`
- `simd`

`for` Directive

- loop iterations may be executed in parallel
  - requires loop iterations to be independent (dependence analysis)
  - matches single program multiple data (SPMD) paradigm
  - arbitrary iteration-to-thread mapping!
- most common form of data parallelism in OpenMP
  - but OpenMP also offers task parallelism
- can also take clauses
  - `reduction, schedule, collapse`

```
#pragma omp parallel
{
  #pragma omp for
  for(/*init*/; /*test*/; /*inc*/) {
        ...
  }
}
```

`for` Directive cont'd

- `for` loops must have canonical form
  - requires number of iterations to be known upon loop entry
    - `init`, `test`, and `inc` expressions must be loop invariant
    - `test` only allows `<`, `<=`, `>`, `>=`
    - `inc` only allows common patterns such as `++var`, `var++`, `--var`, `var--`, `var+=step`, `var-=step`, …
    - loop variable must not be written to in loop body
- C: iterator must be integer or pointer
- C++: must be a random access iterator
  - range-based `for` only with OpenMP ≥ 5.0

```
#pragma omp parallel
{
  #pragma omp for
  for(/*init*/; /*test*/; /*inc*/) {
        ...
  }
}
```

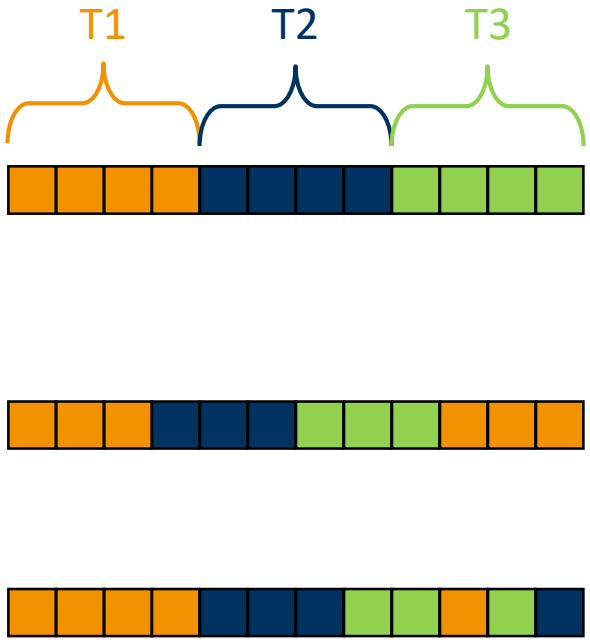
`for`: `schedule` Clause

- specifies method of dividing iteration space into chunks and assigning chunks to threads
- static: equally-sized chunks, fixed round-robin assignment
  - optional: chunk size (default is “approximately equal in size & at most one chunk per thread”)
- dynamic: equally-sized chunks assigned turn-by-turn, at runtime
  - optional: chunk size (default is 1)
- guided: like dynamic, but chunk size decreases proportionally to no. of unassigned iterations
  - optional: minimum chunk size (default is 1)
- also available: `auto`, `runtime`

```
#pragma omp parallel
{
  #pragma omp for schedule(dynamic,2)
  for(/*init*/; /*test*/; /*inc*/) {
        ...
  }
}
```

`for`: `schedule` Clause cont'd

- specifies method of dividing and assigning chunks
- static: equally-sized chunks, fixed round-robin assignment
  - optional: chunk size (default is "approximately equal in size & at most one chunk per thread")
- dynamic: equally-sized chunks assigned turn-by-turn, at runtime
  - optional: chunk size (default is 1)
- guided: like dynamic, but chunk size decreases proportionally to no. of unassigned iterations
  - optional: minimum chunk size (default is 1)
- also available: `auto`, `runtime`



`for`: `collapse` Clause

- given a loop nest, the goal is usually to parallelize and distribute the outermost loop
  - minimizes management overhead
- What if the outermost loop has few iterations?
  - insufficient parallelism for modern systems
  - nesting `parallel` pragmas runs the risk of oversubscription (exponential growth)
- `collapse` combines multiple iteration spaces into a single, larger one
  - allows to exploit more parallelism

```
#pragma omp parallel for collapse(3)
for(int i = 0; i < 3; ++i) {
  for(int j = 0; j < 4; ++j) {
    for(int k = 0; k < 5; ++k) {
      ...
    }
  }
}
```

`sections` Directive

- sections may be executed concurrently, each by an arbitrary thread of the team
- matches MPMD programming patterns
  - coarse-grained parallelism
- easily leads to load imbalance if individual sections not equally work-intensive
  - also, maximum degree of parallelism limited by number of sections

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    { ... }
    #pragma omp section
    { ... }
    #pragma omp section
    { ... }
  }
}
```

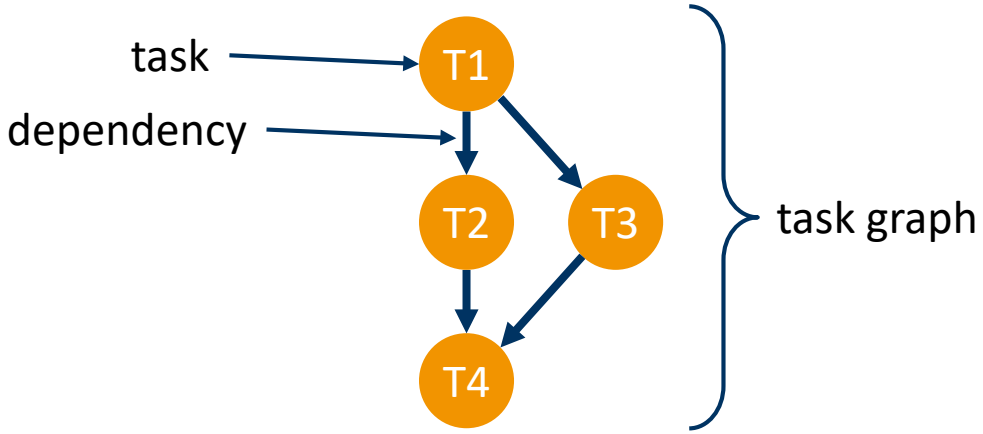
`single` Directive

- code region will only be executed by a single, arbitrary thread
  - useful for interacting with libraries, that are not multi-threading-aware
- implicit barrier at the end for all threads in the team
- also available as `master` variant
  - like single, but for master thread
  - no implicit barrier at the end

```
#pragma omp parallel
{
  #pragma omp single
  {
    ...
  }
}
```

Task-based Parallelism

- allows to work with more irregular problems than e.g. using flat arrays
  - trees, linked lists, unstructured meshes, etc.
- can be created on-demand
  - no need to know the total number of tasks before execution (contrary to loops)
  - automatic load balancing (threads that are idle will fetch a task to work on)



`task` Directive

- allows explicit specification of tasks
  - careful, `firstprivate` is the default
- whenever a thread encounters a `task` directive, a task is generated
  - task may be immediately executed
  - or execution may be deferred
- wait for completion using `taskwait`
  - waits for child tasks spawned by the current task

```
int fib(int n) {
  int i, j;
  if (n < 2)
    return n;

  #pragma omp task shared(i)
  i = fib(n-1);

  #pragma omp task shared(j)
  j = fib(n-2);

  #pragma omp taskwait
  return i + j;
}
```

Example: Traversing a Linked List

```
struct Node {
   struct Node *next;
   struct Data *data;
};

void traverse(struct Node *p) {
  if (p->next) {
    #pragma omp task
    traverse(p->next);
  }
  process(p); // do work
}
```

```
int main(int argc, char **argv) {
  struct Node *head;
  head = ... // produce list
  #pragma omp parallel
  {
    #pragma omp single
    {
      traverse(head);
    }
  }
}
```

Combining Multiple Parallel Regions

```
#pragma omp parallel for
for(int i = 0; i < N-1; i++) {
    y[i] = x[i] + x[i+1];
}

#pragma omp parallel for
for(int i = 0; i < N-1; i++) {
    x[i] = y[i];
}
```

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N-1; i++) {
        y[i] = x[i] + x[i+1];
    }

    #pragma omp for
    for(int i = 0; i < N-1; i++) {
        x[i] = y[i];
    }
}
```

Reduction

- Reductions combine multiple values into a single one
  - sum, product, max, min, etc.
- inherently cause race conditions that need to be avoided
- `critical` ensures that only one thread executes the “critical region” at a time
  - solves the race condition, but is very expensive (requires N critical regions)

```
double s = 0;
#pragma omp parallel for
for(int i = 0; i < N; i++) {
    double val = in[i];
    #pragma omp critical
    s += val;
}
```

Faster Reduction

- Reduce number of critical regions
  - Only requires [number of threads] critical regions
  - potentially large performance gain, depending on N
- `critical` can also take multiple statements or function calls

```
double s = 0; // shared
#pragma omp parallel
{
  double local_s = 0; // private

  #pragma omp for
  for(int i = 0; i < N; i++) {
    double val = in[i];
    local_s += val;
  }

  #pragma omp critical
  s += local_s;
}
```

Alternative: Atomic

- same as `critical`, but restricted to a single memory location and certain operations
- restriction allows mapping to fast hardware mechanisms
- keeps code hardware- and compiler-independent compared to using intrinsics
  - but may just be a wrapper for `critical` e.g. when lacking hardware support

```
double s = 0;
#pragma omp parallel for
for(int i = 0; i < N; i++) {
    double val = in[i];
    #pragma omp atomic
    s += val;
}
```

Exercise 2a

- Goal:
  - for workshare construct
  - critical directive

- A sequential program that computes π is provided in `exercises/pi/pi.c`
  - add `parallel` region and `for` directive

- Expected result: Result (π) is unpredictable when used with `OMP_NUM_THREADS` > 1

- Find and fix the two race conditions in the code!

Exercise 2b

- Run the program multiple times and compare the result
- What do you observe?
- Investigate the run time as a function of `OMP_NUM_THREADS`!
- How can we improve the performance?

More OpenMP

`reduction` Clause

- performs reduction to a single variable in parallel or loop context
  - arithmetic ops: +, -, *, max, min
  - logical ops: &, &&, |, ||, ^
  - careful with associativity of floating-point operations!
- user-defined reductions are possible (version 4.0)
  - need to be declared with `#pragma omp declare reduction`

```
#pragma omp parallel
{
  #pragma omp for reduction(+:x)
  for(int  i = 0; i < 10; ++i) {
    x += i;
  }
}

// or

#pragma omp parallel reduction(-:x)
x -= omp_get_thread_num();
```

`barrier` Directive

- explicit barrier requested by user
- threads are not allowed to continue until all have reached the barrier
- Implicit barrier at the end of `for`, `sections`, `single`, `task`, `simd` unless `nowait` specified
  - explicit barrier usually not required except for debugging

```
#pragma omp parallel
{
  ...
  #pragma omp barrier
  ...
}
```

Library Functions and Environment Variables

- querying/controlling environment
  - `omp_get_num_threads()`
  - `omp_get_thread_num()`
  - `omp_get_nested()`
  - `omp_in_parallel()`
  - and a few others, also setters!
- timing
  - `omp_get_wtime()`
  - `omp_get_wtick()`
- locking
  - `omp_init_lock()`
  - `omp_set_lock()`
  - `omp_unset_lock()`
  - `omp_test_lock()`
  - `omp_destroy_lock()`

```
export OMP_DISPLAY_ENV=true
./a.out

OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP = '201511'
  OMP_DYNAMIC = 'FALSE'
  OMP_NESTED = 'FALSE'
  OMP_NUM_THREADS = '8'
  OMP_SCHEDULE = 'DYNAMIC'
  OMP_PROC_BIND = 'FALSE'
  OMP_PLACES = ''
  OMP_STACKSIZE = '0'
  OMP_WAIT_POLICY = 'PASSIVE'
  OMP_THREAD_LIMIT = '4294967295'
  OMP_MAX_ACTIVE_LEVELS = '2147483647'
  OMP_CANCELLATION = 'FALSE'
  OMP_DEFAULT_DEVICE = '0'
  OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
```

Exercise 3

- Goal:
  - Usage of the reduction clause.
- Replace the critical directive in favor of a reduction clause!
- Investigate the performance as a function of `OMP_NUM_THREADS`!
- Expected result: almost linear scaling

Additional OpenMP Features not Covered Today

- flushes
  - low-level, fine-grained synchronization constructs
- affinity
  - OS-independent control over thread-core mapping
- vectorization
  - hardware- and compiler-independent use of SIMD instructions
- accelerator support
  - use e.g. NVIDIA GPUs without writing CUDA code
- Fortran
- debuggging
  - gdb, valgrind, Intel Inspector, etc.

Summary

- main characteristics
  - incremental parallelization
- programming, execution and memory models
  - based on threads and shared data access
  - mainly relies on compiler directives as programmer interface
- directives
  - parallelism, data sharing, work sharing, synchronization
- exercises

Additional Resources

- Introduction to High Performance Computing for Scientists and Engineers, Georg Hager and Gerhard Wellein. 2010, CRC Press.
- OpenMP, Blaise Barney, Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/openMP/
- “Parallel Programming for Science and Engineering” by Victor Eijkhout, https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/EijkhoutParallelProgramming.pdf
- OpenMP homepage: http://www.openmp.org

Image Sources

- Helen Lovejoy: https://simpsons.fandom.com/wiki/Helen_Lovejoy