703650 VO Parallel Systems WS2019/2020
OpenMP Basics

Philipp Gschwandtner

Overview

- main characteristics

- programming, execution and memory models

- directives

- Tales From the Proseminar

Motivation for Using OpenMP

▸ **OpenMP is one of the easiest parallel programming models & widely available**

  ▸ however, restricted to shared memory and multi-threaded hardware

▸ **modern hardware encourages use of such models**

  ▸ desktop: AMD 3990X with 64 cores and 128 threads expected 2020…

  ▸ AMD server: 2x EPYC 77xx with 64 cores / 128 threads = 128/256

  ▸ Intel server: 8x Xeon Platinum 827x or 828x with 28 cores / 56 threads = 224/448

  ▸ exotic hardware: SGI Altix UV ("Mach 2" @ JKU in Linz) with 4096/8192

▸ **also enables access to up to e.g. 64 TB of RAM on SGI Altix UV systems**
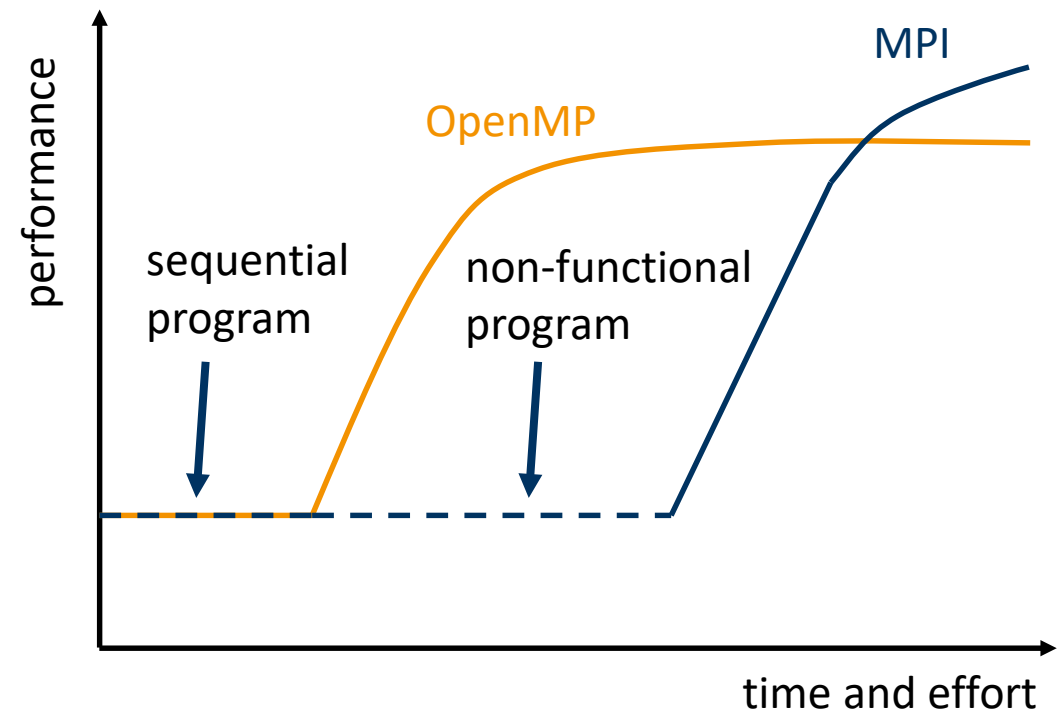
The Advantage of Incremental Parallelization

▶ **MPI**

    ▸ initially a sequential program

    ▸ start to parallelize

    ▸ program non-functional 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

▸ 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)

  ▸ this slide set assumes at least OpenMP 3.1!

OpenMP Implementations

▶ **many implementations available**

- ▶ GCC, LLVM, Intel, Microsoft, etc.
- ▶ allow to run OpenMP basically on every platform (portability)
  - ▶ that provides decent performance…
- ▶ 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

Main Characteristics

▸ **compiler-based parallelization model**

  ▸ tell compiler what code to parallelize and where to put synchronization points

  ▸ tell compiler whether to share data among threads or create private copies

  ▸ but compiler cannot/will not check semantic correctness

    ▸ was designed with compiler developers in mind…

▸ **allows incremental parallelization**

▸ **C/C++ and Fortran bindings**

  ▸ even a research compiler for Java is available…

How to Choose between OpenMP and MPI?

▶ OpenMP

  ▶ language extension, requires compiler support

  ▶ data transfer happens implicitly through shared memory

  ▶ restricted to shared memory parallelism

  ▶ incremental parallelization

▶ MPI

  ▶ library, hence fully compiler-independent

  ▶ data transfer requires explicit message passing

  ▶ supports distributed memory parallelism

  ▶ no incremental parallelization

often: both (*"hybrid"* parallelism)!

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

team of threads →

master thread →
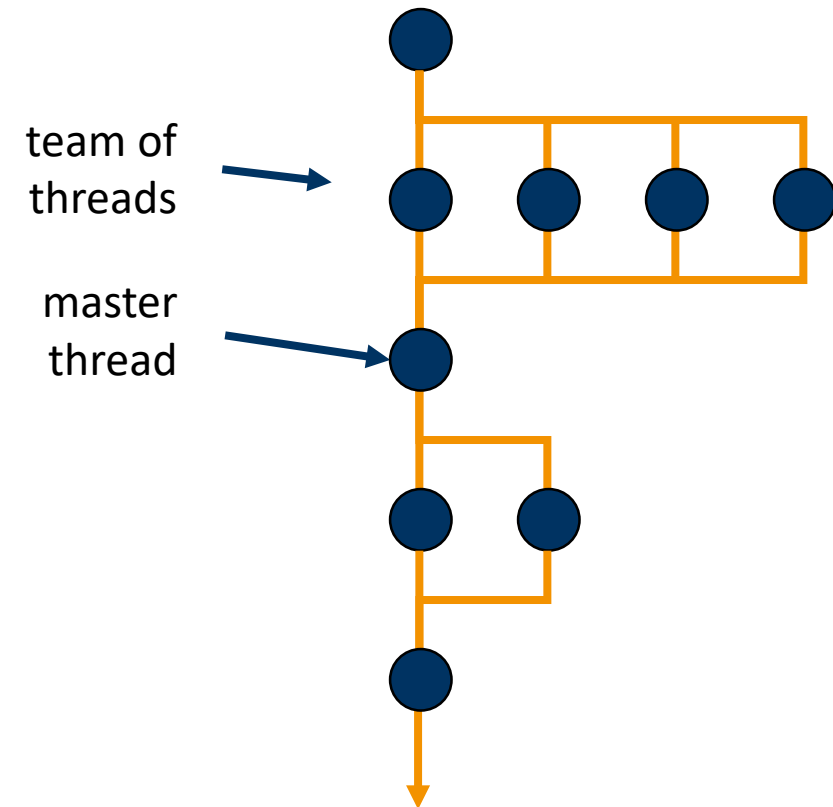
Programming Model

▶ mark code regions with directives or *pragmas*, e.g.
  ▶ parallel / sequential regions
  ▶ work to be distributed
  ▶ thread synchronization

▶ add *clauses* for further information, e.g.
  ▶ which variables to share, which not to
  ▶ scheduling strategies
  ▶ forced execution order

▶ any valid OpenMP program must be a valid sequential program if all pragmas are removed!

```
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

parallel region

work-shared loop
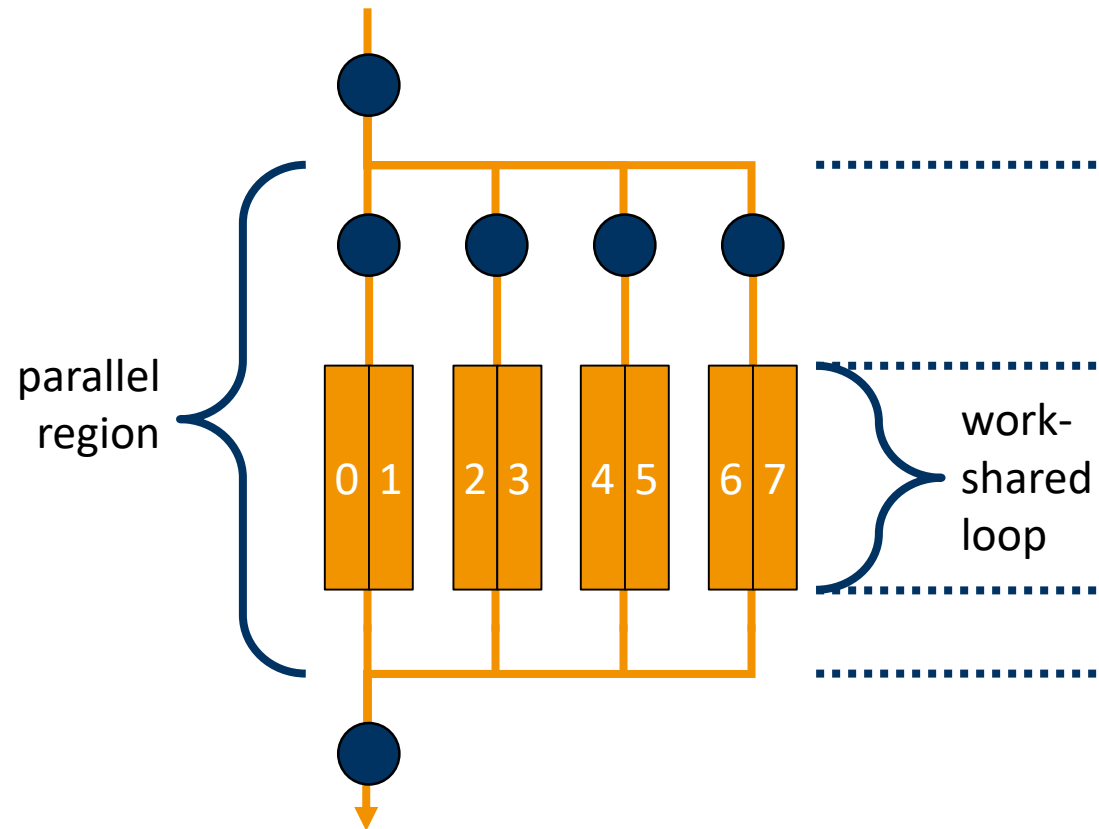
```
int f = ...
#pragma omp parallel shared(a,b,c,f)
  default(none)
{

  #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 not necessarily immediately visible across threads

shared memory

private memory
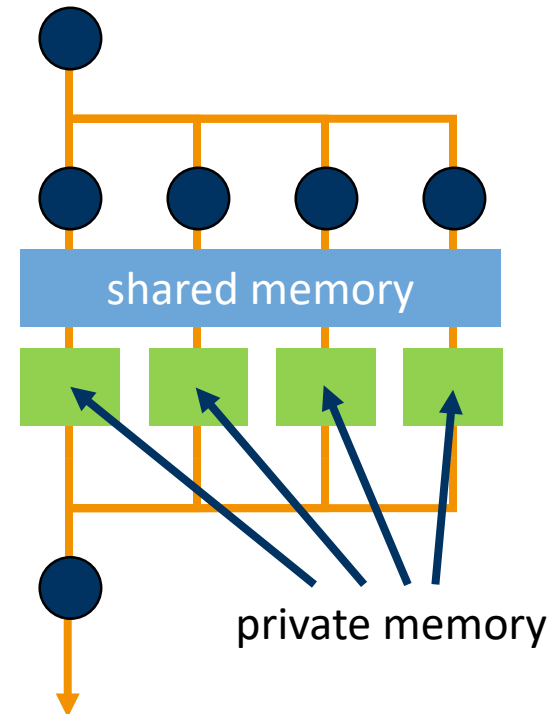
Compilation and Execution

- compile as usual but include OpenMP-specific flag
  - e.g. gcc: `-fopenmp`

- 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
  - e.g. for SGE @ LCC2, use `-pe openmp`

OpenMP API

OpenMP API cont'd

- **pragmas**
  - control constructs
    - parallelism & work sharing
  - data sharing
    - private & shared variables, initialization
  - synchronization
    - critical & atomic sections, barriers

- **library functions**
  - querying/controlling environment
  - 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

```
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;
  }
}
```

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

Library Functions

▶ 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()`

Environment Variables

▶ `OMP_NUM_THREADS`

  ▶ sets the number of threads during execution of parallel regions

  ▶ if dynamic adjustment is enabled, sets the maximum number of threads

  ▶ default is implementation-defined

▶ `OMP_SCHEDULE`

  ▶ applies to `for` work sharing only

  ▶ supports `static`, `dynamic`, and `guided` with optional chunk size

  ▶ e.g. `OMP_SCHEDULE=static,4`

▶ `OMP_NESTED`

  ▶ enables/disables nested parallelism

▶ `OMP_PROC_BIND`

  ▶ enables/disables thread binding

▶ `OMP_PLACES`

  ▶ controls mapping threads to hardware resources (e.g. cores or sockets)

`parallel` Directive

▶ **`#pragma omp parallel`**

    ▶ must be followed by a single-entry single-exit statement or an OpenMP construct

    ▶ master thread creates a team of threads, each executing the same 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…
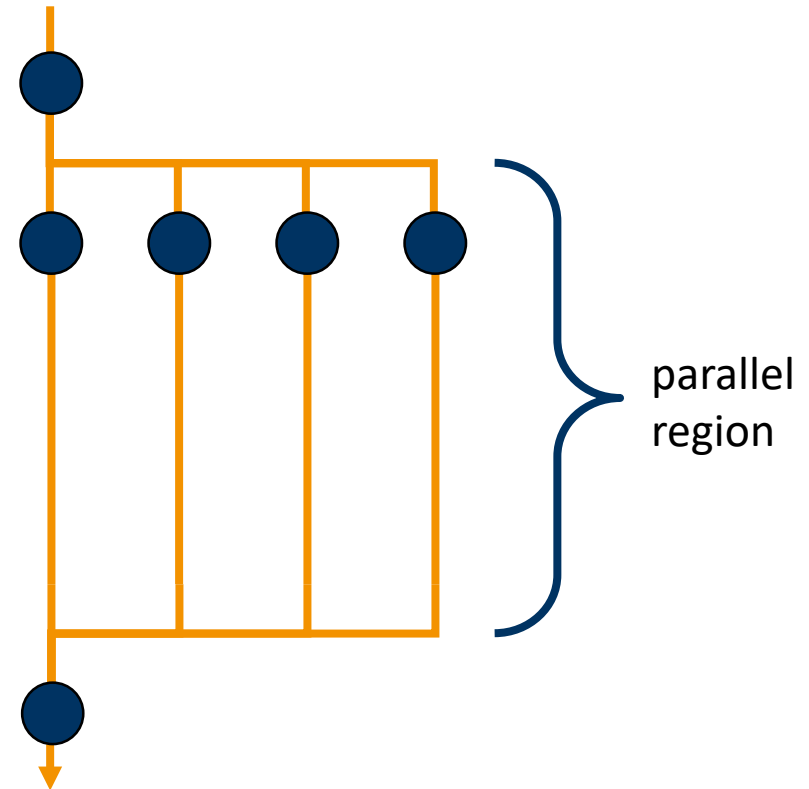
parallel
region

Hello World in OpenMP

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

int main(int argc, char** argv) {
  #pragma omp parallel
  {
    int myThreadID = omp_get_thread_num();
    int numThreads = omp_get_num_threads();
    printf("Hello from thread %d of %d\n",
      myThreadID, numThreads);
  }
  return 0;
}
```

```
$ OMP_NUM_THREADS=4 ./hello
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
```

Data Sharing Clauses

- `private`
  - each thread gets a private copy of variable, no longer *"storage-associated"* with original variable
  - private copy is not initialized
  - often better to declare variables inside parallel region, reduces amount of code (also minimizes *"vertical distance"* in source code)
  - default for variables declared inside parallel region

- `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 `private`, `shared`, or `none`
  - none helpful for detecting missing variables in clauses (compiler will complain!)

Data Sharing Clauses cont'd

```c
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;
  }
}
```

```c
int f = ...
#pragma omp parallel
{

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

Data Sharing Clauses cont'd

▸ `firstprivate`

   ▸ like `private`, but private copies are initialized with value of copy outside of parallel region

▸ `lastprivate`

   ▸ like `private`, but outside copy is set to the private copy of the final iteration (`for` loops) or last section (`sections`)

▸ `threadprivate`

   ▸ like private, but will persist across parallel regions

   ▸ master thread variable is storage-associated with original variable (not the case for `private`!)

`reduction` Clause

▶ **performs reduction to a single variable in parallel or loop context**
  - ▶ arithmetic ops: +, -, *, max, min
  - ▶ logical ops: &, &&, |, ||, ^
  - ▶ same issues as with MPI regarding associativity of operations!

▶ **user-defined reductions are possible**
  - ▶ need to be declared with #pragma omp declare reduction

```cpp
#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();
```

Work Sharing

Work Sharing Directives

▸ distribute execution of following code region among existing threads of the team

▸ must be enclosed in parallel region

▸ cannot be directly nested

▸ do not launch new threads

▸ no barrier on entry

▸ implicit barrier on exit

    ▸ unless `nowait` clause

▸ `for`

▸ `sections`

▸ `single`

▸ `task`

▸ `simd`

for Directive

▸ **loop iterations may be executed in parallel**
  ▸ requires loop iterations to be independent
  ▸ matches SPMD patterns

▸ **most common form of data parallelism in OpenMP**

▸ **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**
  - ▸ defined by the OpenMP specification
  - ▸ 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++: iterator must use random access iterators**

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

for: schedule Clause

▶ 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
  ▶ 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

- 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
  - 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`
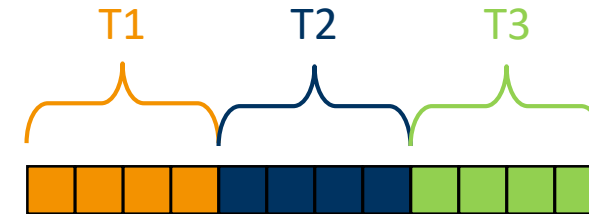
Comparing OpenMP Loop Scheduling Strategies

- ▸ incrementing a 64 bit integer in 10^{11} loop iterations on LCC2, gcc 8.2.0, 8 threads

- ▸ static, dynamic and guided loop scheduling for increasing chunk sizes 2^n with $0 \leq n \leq 10$

Wall Time Comparison
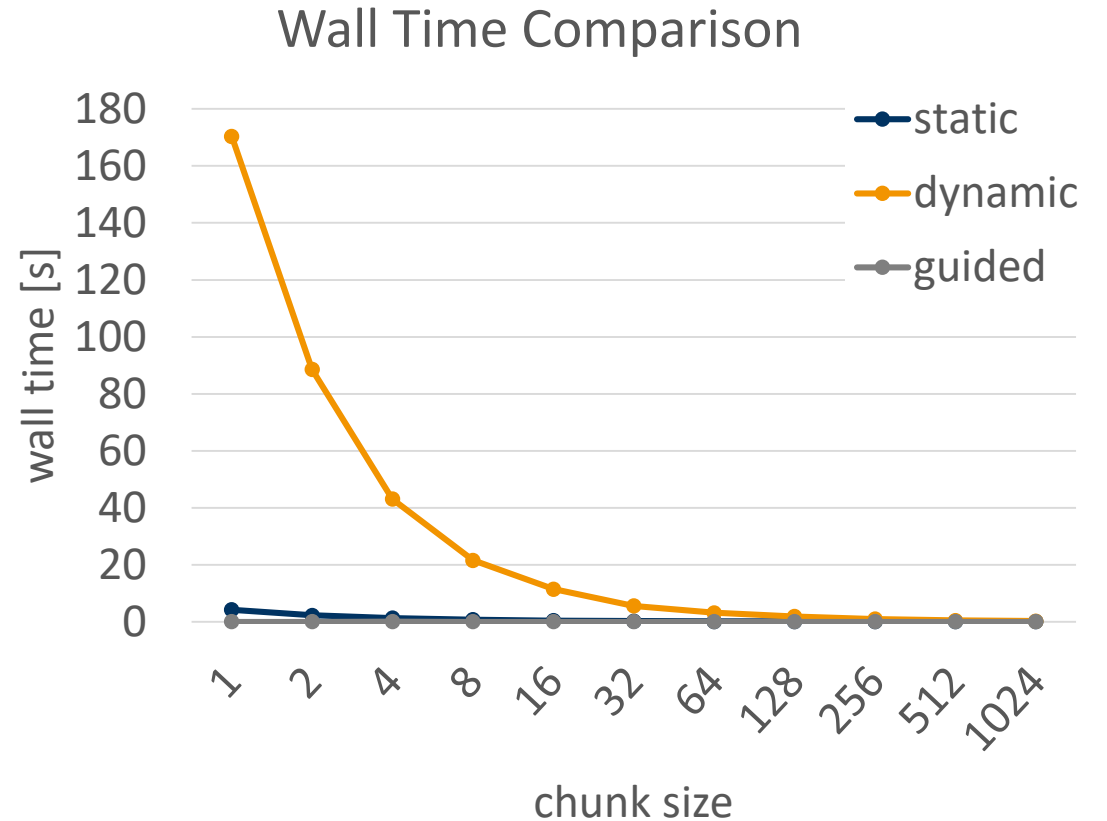
for: `collapse` Clause

▸ given a loop nest, the goal is usually to parallelize and distribute outermost loop
  ▸ minimizes overhead

▸ What if outermost loop has low iteration count?
  ▸ insufficient parallelism for modern systems

▸ `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 patterns

- 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

▸ 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
  {

    ...

  }

}
```

Synchronization

Synchronization Directives

▸ **constructs discussed so far are fairly high-level in terms of synchronization**

    ▸ e.g. unable to enforce specific order or simply wait for all threads

▸ **OpenMP also offers more fine-grained synchronization directives**

▸ `barrier`

▸ `critical`

▸ `atomic`

▸ `ordered`

▸ `flush`

▸ `master`

barrier Directive

▸ explicit barrier requested by user

▸ threads are not allowed to continue until all heave reached the barrier

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

`critical` Directive

▸ code region executed by all threads but only one at a time

▸ can be named to allow finer-grained mutual exclusion

  ▸ only critical regions with the same name enforce mutual exclusion

  ▸ all regions without a name have the same name

```
#pragma omp parallel
{
  #pragma omp critical(foo)
  { ... }
  #pragma omp critical(foo)
  { ... }
  #pragma omp critical(bar)
  { ... }
  #pragma omp critical
  { ... }
  #pragma omp critical
  { ... }
}
```

`atomic` Directive

▸ same as `critical`, but restricted to a single memory location and certain operations

▸ restriction allows mapping to fast hardware mechanisms
  ▸ optional specialization clauses `read`, `write`, `update`, and `capture`

▸ keeps code hardware- and compiler-independent compared to using intrinsics
  ▸ but may just be a wrapper for `critical` e.g. when lacking hardware support

```
int count = 0;
#pragma omp parallel
{
    #pragma omp atomic
    count++;
}
```

ordered Directive + Clause

▸ **enforces critical region with sequential execution order**

▸ **required for strong sequential equivalence**
  ▸ but at high performance cost

▸ **only efficient if code outside the ordered region is expensive enough**
  ▸ remember guidelines about strong / weak equivalence
  ▸ check whether your numerical method really needs strong sequential equivalence

```cpp
double total = 0.0;
#pragma omp parallel for ordered
for(int i = 0; i < N; ++i) {
    double part = f(i);
    #pragma omp ordered
    total += part;
}
```

flush Directive

▸ problem: writes by one thread are not immediately visible to another in the same parallel region without synchronization

▸ example on the right: the second section could hang in while loop
  ▸ But why?

```
#pragma omp parallel sections
{
    #pragma omp section
    { // producer section
        // produce some data
        flag = 1;
    }
    #pragma omp section
    { // consumer section
        while (flag == 0) { }
        // use data
    }
}
```

flush Directive cont'd

RAM

L2 Cache

L1 Cache   L1 Cache

Core 0 Reg.   Core 1 Reg.

flag = 1   flag = 0

```
#pragma omp parallel sections
{

    #pragma omp section
    { // producer section
        // produce some data
        flag = 1;
    }
    #pragma omp section
    { // consumer section
        while (flag == 0) { }
        // use data
    }
}
```
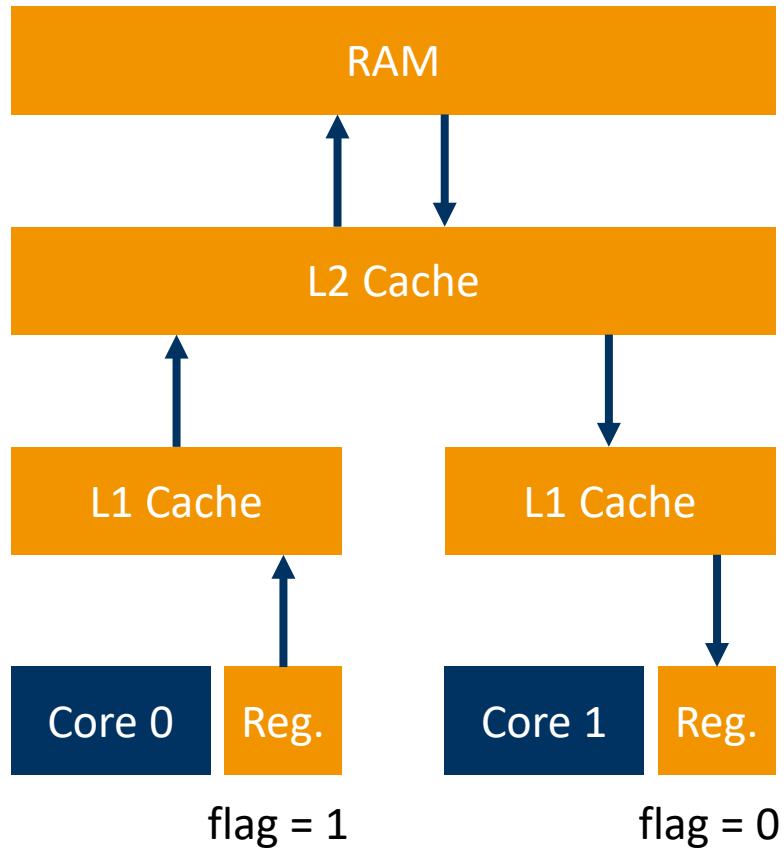
flush Directive cont'd

```
#pragma omp parallel sections
{

  #pragma omp section
  { // producer section
    // produce and flush data
    #pragma omp flush
    #pragma omp atomic write
    flag = 1;
    #pragma omp flush(flag)
  }
```

```
  #pragma omp section
  { // consumer section
    while (1) {
      #pragma omp flush(flag)
      #pragma omp atomic read
      int temp_flag = flag;
      if(temp_flag == 1) break;
    }
    // use data
  }
} // end sections
```

flush Directive cont'd

▸ **`flush` is implied at**
  - ▸ barrier
  - ▸ entry and exit of `critical`
  - ▸ exit of
    - ▸ parallel
    - ▸ for
    - ▸ sections
    - ▸ single
  - ▸ set/unset of locks

▸ **if otherwise required, use `flush` directive explicitly**

Tales From the Proseminar

Tales From the Proseminar

▸ Which code version is faster? (yes, both have a race condition…)

```
int foo(int dummy) {
    int i, j;
    #pragma omp parallel for private(j)
    for(i = 0; i < 10; ++i) {
        for(j = 0; j < 10; ++j) {
            dummy += i + j;
        }
    }
    return dummy;
}
```

```
int foo(int dummy) {
    #pragma omp parallel for
    for(int i = 0; i < 10; ++i) {
        for(int j = 0; j < 10; ++j) {
            dummy += i + j;
        }
    }
    return dummy;
}
```

Tales From the Proseminar

▸ the assembly code is exactly the same

  ▸ screenshot shows output for gcc 9.2 and -O0 (also works for -O3)

▸ https://godbolt.org/z/SKx8hA

Summary

▸ **main characteristics**
  ▸ incremental parallelization

▸ **programming, execution and memory models**
  ▸ based on threads and shared data access
  ▸ mainly relies on pragmas as programmer interface

▸ **directives**
  ▸ parallelism, work sharing, data sharing, synchronization

▸ **Tales From the Proseminar**
  ▸ minimize vertical code distance
  ▸ specifically: declare `for` loop iterators in loop header whenever possible