Programming with OpenMP

Parallel Programming

2023-11-09

Prof. Dr. Michael Kuhn
michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de
Programming with OpenMP

Review

Introduction

Overview

Parallelism

Synchronization

Summary
What is the difference between SPMD and MPMD?

1. SPMD is based on a single process and uses threads
2. MPMD starts the same application multiple times
3. SPMD distributes data, MPMD distributes functionality
What is the best way to distribute a matrix using SPMD?

1. Each process holds one element
2. Each process holds one row/column
3. Each process holds several rows/columns
4. Each process holds a sub-matrix
Why are we using semi-automatic instead of automatic parallelization?

1. Semi-automatic parallelization allows more control
2. Automatic parallelization is very limited
3. Automatic parallelization is only available for esoteric languages
4. Automatic parallelization only works for MPMD
Why should a critical region not be used in an inner loop?

1. The application could deadlock
2. Critical regions have too much overhead
3. It could result in race conditions due to too much synchronization
4. Critical regions can always be replaced by other constructs
Programming with OpenMP

Review

Introduction

Overview

Parallelism

Synchronization

Summary
Motivation

- Automatic parallelization still does not work in all cases
  - Compilers require hints from developers for efficient parallelization
  - We have to settle for semi-automatic parallelization
- OpenMP is a compiler-based high-level approach for shared memory systems
  - It is easy to use and does not require in-depth knowledge
  - Available for the most widely used languages in HPC (C/C++ and Fortran)
- Alternatives are often based on libraries and require manual parallelization
  - POSIX Threads can be used for shared memory systems
  - MPI can be used for distributed memory systems
  - Recent versions of C and C++ include native support for threads
• OpenMP is an abbreviation for Open Multi-Processing
  • Independent standard supported by several compiler vendors
• Parallelization is done via so-called compiler pragmas
  • Compilers without OpenMP support can simply ignore the pragmas
  • There is a small runtime library for additional functionality
• Compiler translates pragmas into a parallelized application using threads
  • Actual translation is implementation-specific and can change
  • Performance also depends on the compiler’s implementation
• Very convenient for users since no internals have to be known
  • Reduced feature set in comparison to low-level approaches
• Compilation requires -fopenmp flag during building and linking
  • Translation during building, runtime library during linking
    • For instance, GCC uses libgomp.so as the runtime library
    • Without the flag, no parallelization and linking happens
• Applications have to be written carefully for serial compilation
  • Compiler simply ignores pragmas but library functions are problematic
  • Typically requires preprocessor handling to substitute unavailable functionality
Programming with OpenMP

Review

Introduction

Overview

Parallelism

Synchronization

Summary
• OpenMP only takes care of computation
  • No support for parallel I/O or explicit communication
• OpenMP is a standard that is still updated frequently
  • Developed by the OpenMP Architecture Review Board
  • Compiler vendors can implement it differently, sometimes not fully supported
  • Performance and behavior can differ from compiler to compiler and version to version
• OpenMP does not check applications for conformity
  • Developers have to take care not to produce undefined behavior
  • OpenMP does not detect dependencies, races, deadlocks etc.
• Standardization and portability
  • Applications compile on all supported platforms
    • For instance, applications can be written on laptop but run on HPC machine
    • Works with C/C++ and Fortran, widely used languages in HPC
      • There is also a research project to support OpenMP in Java

• Ease of use
  • Application can be parallelized incrementally
    • That is, it is possible to parallelize single parts of the code
  • High-level and abstract approach, suited for scientists
• Directives to express parallelism
  • That is, running code within multiple threads
• Work sharing
  • Automatically splitting up loops for parallel processing
• Synchronization
  • Threads can communicate with each other and coordinate work
• Accelerator offloading
  • Code can be offloaded to GPUs and other accelerators for more performance
• Vectorization
  • Loops can be annotated for improved SIMD support
• OpenMP 1.0 in 1997/1998
  • Mainly for regular loops for numerical applications
  • Number of iterations known at time of entry

• OpenMP 2.0 in 2000/2002 and OpenMP 2.5 in 2005
• OpenMP 3.0 in 2008
  • Support for tasks via the task directive, more general parallelization
• OpenMP 4.0 in 2013
  • Support for offloading to accelerators, atomics, error handling, thread affinity, user-defined reduction, SIMD and more

• OpenMP 5.0 in 2018
  • Support for task reductions, more loop forms, memory ordering and more

• OpenMP 5.1 in 2020
  • Better support for accelerator devices, additional hints for optimization, C++ attribute syntax and more

• OpenMP 5.2 in 2021
  • Makes syntax more consistent, minor improvements
Outline

Programming with OpenMP

Review
Introduction
Overview
Parallelism
Synchronization
Summary
Threads vs. Processes

- Processes are instances of an application
  - Applications can be started multiple times
  - Processes are isolated from each other by the operating system for security reasons
  - Resources like allocated memory, opened files etc. are managed per-process

- Threads are lightweight processes
  - Threads have their own stacks but share all other resources
  - Shared access to resources has to be synchronized
  - Uncoordinated access can lead to errors very easily

- OpenMP takes care of thread management and scheduling
  - Support for loops, tasks, synchronization and more
  - Synchronization via barriers, critical regions, atomic operations etc.
• Threads share a common address space
  • OpenMP supports shared and private variables for communication
  • Threads are processed independently, that is, in parallel
• Processes have their own address spaces
  • Typically have to start multiple processes for distributed memory
  • Overhead is normally higher than with shared memory
• Hybrid approaches with MPI + OpenMP in HPC
  • A few processes per node (e.g., one per socket)
  • Many threads per process (e.g., one per core)
• Processes start with one main thread
  • In the serial case, main thread executes everything
• Threads use a fork-join model
  • New threads are forked from the main thread
  • Threads are joined, that is, terminated
• Thread creation takes some time
  • Should not be done in an inner loop
• OpenMP typically takes care of overhead
  • For instance, using a thread pool
Functionality

- Compiler pragmas for main functionality
  - #pragma omp directive clauses

- Library functions for additional functionality
  - omp_get_thread_num
  - omp_get_num_threads
  - omp_set_num_threads
  - Locks, time measurement and more

- Environment variables to influence behavior
  - OMP_NUM_THREADS
  - OMP_SCHEDULE
  - Thread affinity, stack size and more
Parallel Regions

- General parallel regions
  - Started using `#pragma omp parallel`
  - Parallelizes the following statement
    - Statement can be a block of code (`{ . . . }`)
  - Creates a number of threads
    - Forked at the beginning and joined at the end
    - Main thread becomes master thread
    - In reality, thread pools might be used for performance reasons
  - Implicit barrier at the end of region
    - All threads wait at barrier

```c
int main(void) {
    #pragma omp parallel
    printf("Hello world.\n");
    return 0;
}
```
Parallel Regions

- General parallel regions
  - Started using #pragma omp parallel
  - Parallelizes the following statement
    - Statement can be a block of code (\{ ... \})
  - Creates a number of threads
    - Forked at the beginning and joined at the end
    - Main thread becomes master thread
    - In reality, thread pools might be used for performance reasons
  - Implicit barrier at the end of region
    - All threads wait at barrier

Hello world.

Hello world.

Hello world.

Hello world.

Hello world.

Hello world.

Hello world.

Hello world.

Hello world.

Hello world.
Parallel Regions…

- Directives apply to next statement
  - Single statement or block of code
- No atomicity for block of code
  - Single statements are run independently
  - Problematic for split statements

```
int main(void) {
    #pragma omp parallel
    {
        printf("Hello ");
        printf("world.\n");
    }

    return 0;
}
```
Parallel Regions...

- Directives apply to next statement
  - Single statement or block of code
- No atomicity for block of code
  - Single statements are run independently
  - Problematic for split statements
- Output differs every time
  - Depending on timing and other factors
  - Typical race condition due to missing synchronization

Hello world.
Hello world.
Hello world.
Hello world.
Hello world.
Hello Hello world.
Hello world.
world.
Hello world.
Hello world.
Parallel Regions…

- Threads are assigned an ID
  - IDs are numbers from 0 to \(N-1\)
- ID and count useful for coordination
  - Have to be retrieved using functions
- Can be used to perform actions once
  - For instance, first thread coordinates
- Number of threads can be influenced

```c
int main(void) {
    #pragma omp parallel
    {
        int id, num;
        id = omp_get_thread_num();
        num = omp_get_num_threads();

        printf("Hello world from "
                "thread %02d/%02d.\n",
                id, num);
        if (id == 0)
            printf("I am zero.\n");
    }
    return 0;
}
```
Parallel Regions...

- Threads are assigned an ID
  - IDs are numbers from 0 to N-1
- ID and count useful for coordination
  - Have to be retrieved using functions
- Can be used to perform actions once
  - For instance, first thread coordinates
- Number of threads can be influenced

Hello world from thread 08/12.
Hello world from thread 11/12.
Hello world from thread 02/12.
Hello world from thread 07/12.
Hello world from thread 05/12.
Hello world from thread 04/12.
Hello world from thread 06/12.
Hello world from thread 00/12.
I am zero.
Hello world from thread 10/12.
Hello world from thread 09/12.
Hello world from thread 01/12.
Hello world from thread 03/12.
• Number of threads can be specified in a number of ways
  1. Implementation’s default thread count
  2. Environment variable OMP_NUM_THREADS
  3. Function omp_set_num_threads
  4. Directive’s num_threads clause
  5. Directive’s if clause

• Default thread count is typically the number of logical cores
  • That is, what nproc prints

• if determines whether region should be executed in parallel
  • Otherwise, region is executed by main thread only
• How many threads run in this example?
  1. Twelve (nproc output)
  2. Four
  3. Three
  4. Two
  5. One

```c
int main(void) {
    omp_set_num_threads(3);

    #pragma omp parallel
        num_threads(4) if(0)
    printf("Hello world from "
        "thread %02d/%02d.\n",
        omp_get_thread_num(),
        omp_get_num_threads());

    return 0;
}
```
How many threads run in this example?

1. Twelve (nproc output)
2. Four
3. Three
4. Two
5. One

Hello world from thread 00/01.
Variables are shared by default
  That is, all threads can access them
  There are no warnings when doing so
Access has to be coordinated
  There are several directives for synchronization
Visibility can be changed using clauses
  Also called data sharing clauses
  default, shared, private, firstprivate, lastprivate, reduction etc.

```c
int main(void) {
    int id = -1;
    #pragma omp parallel
    id = omp_get_thread_num();
    printf("id=%d\n", id);
    return 0;
}
```
Variables are shared by default
  - That is, all threads can access them
  - There are no warnings when doing so
Access has to be coordinated
  - There are several directives for synchronization
Visibility can be changed using clauses
  - Also called data sharing clauses
    - default, shared, private, firstprivate, lastprivate, reduction etc.
• The default clause changes default visibility of variables
  • C/C++ only allow shared or none
  • none is useful to avoid mistakes

```
parallel4.c: In function 'main':
parallel4.c:9:5: error: 'id' not specified in enclosing 'parallel'
  9 |   id = omp_get_thread_num();
      | ^~~~~~~~~~~~~~~~~~~~~~~
parallel4.c:8:10: error: enclosing 'parallel'
  8 |   #pragma omp parallel default(none)
      |    ^~~
```
private clause makes variables private
  • Has no connection to global version
  • Value of global version is not inherited
    • That is, variable will be uninitialized
    • Can be done using firstprivate
Value of private version is not propagated
  • Can be done using lastprivate (only for loops and sections)

```c
int main(void) {
    int id = -1;

    #pragma omp parallel
        private(id)
    id = omp_get_thread_num();
    printf("id=%d\n", id);
    return 0;
}
```
• private clause makes variables private
  • Has no connection to global version
• Value of global version is not inherited
  • That is, variable will be uninitialized
  • Can be done using firstprivate
• Value of private version is not propagated
  • Can be done using lastprivate (only for loops and sections)
• The for directive distributes loops
  • Must be inside a parallel region
  • Can be combined as parallel for
  • Only applies to the immediately following loop
• Loop variable is automatically private
  • Otherwise, threads would interfere
• Distribution is configurable
  • There are static, dynamic and guided scheduling strategies

```c
int main(void) {
    int i;

    omp_set_num_threads(2);

    #pragma omp parallel for
    for (i = 0; i < 10; i++) {
        printf("i=%d, id=%d\n", i, omp_get_thread_num());
    }

    return 0;
}
```
The for directive distributes loops
  - Must be inside a parallel region
  - Can be combined as parallel for
  - Only applies to the immediately following loop

Loop variable is automatically private
  - Otherwise, threads would interfere

Distribution is configurable
  - There are static, dynamic and guided scheduling strategies

\[
\begin{align*}
i &= 5, \quad id = 1 \\
i &= 6, \quad id = 1 \\
i &= 7, \quad id = 1 \\
i &= 8, \quad id = 1 \\
i &= 9, \quad id = 1 \\
i &= 0, \quad id = 0 \\
i &= 1, \quad id = 0 \\
i &= 2, \quad id = 0 \\
i &= 3, \quad id = 0 \\
i &= 4, \quad id = 0
\end{align*}
\]
For Loops...

```c
for (int i = 0; i < 5; i++) {
    printf("i=%d, id=%d\n", i, omp_get_thread_num());
}
```

```c
for (int i = 5; i < 10; i++) {
    printf("i=%d, id=%d\n", i, omp_get_thread_num());
}
```

- The for loop is split up across two threads due to `omp_set_num_threads`
  - Compiler automatically distributes loop indices to threads
  - Distribution is static by default but can be changed to be dynamic
- More convenient than calculating distribution manually
  - OpenMP can also collapse multiple loops for better distribution
• What happens in this example?
  1. The same as with parallel for
  2. Compiler exits with an error
  3. Both threads calculate the whole loop
  4. Race condition because i is shared

```c
int main(void) {
    omp_set_num_threads(2);

    #pragma omp parallel
    for (int i = 0; i < 10; i++) {
        printf("i=%d, id=%d\n", i, omp_get_thread_num());
    }

    return 0;
}
```
• What happens in this example?
  1. The same as with parallel for
  2. Compiler exits with an error
  3. Both threads calculate the whole loop
  4. Race condition because $i$ is shared

\[
\begin{align*}
i &= 0, \quad id = 0 \\
&\quad \ldots \\
i &= 9, \quad id = 0 \\
i &= 0, \quad id = 1 \\
&\quad \ldots \\
i &= 9, \quad id = 1
\end{align*}
\]
• The schedule clause allows specifying how data should be distributed
  • static divides loop iterations into chunks and assigns them statically
  
  ![Static Schedule Diagram]

  • dynamic divides loop iterations into chunks and assigns them dynamically
  
  ![Dynamic Schedule Diagram]

  • guided divides loop iterations and assigns them dynamically
    • Similar to dynamic but shrinks the chunk size towards the end
  
  • runtime gets the scheduling strategy from OMP_SCHEDULE
  
  • auto lets the compiler decide which scheduling strategy to use
• The sections directive divides work
  • Must be inside a parallel region
  • Contains individual section directives
• Each section is executed exactly once
  • Threads may execute multiple sections
  • Workload of sections might differ
  • Not every thread might be involved
• Can be used for more general applications
  • For instance, one thread each for computation, communication and I/O

```c
int main(void) {
    #pragma omp parallel sections
    {
        #pragma omp section
        printf("section0 id=%d\n", omp_get_thread_num());
        #pragma omp section
        printf("section1 id=%d\n", omp_get_thread_num());
        #pragma omp section
        printf("section2 id=%d\n", omp_get_thread_num());
    }
    return 0;
}
```
• The sections directive divides work
  • Must be inside a parallel region
  • Contains individual section directives
• Each section is executed exactly once
  • Threads may execute multiple sections
  • Workload of sections might differ
  • Not every thread might be involved
• Can be used for more general applications
  • For instance, one thread each for computation, communication and I/O

section1 id=0
section0 id=4
section2 id=6
• Thread affinity becomes important with OpenMP
  • Otherwise threads might be re-scheduled by the operating system
  • Migration to other core causes cache invalidations etc.
• Threads can be bound to specific cores
  • Available ways depend on operating system, compiler etc.
• OpenMP provides an environment variable OMP_PROC_BIND
  • Setting it to true enables binding by default
GDB supports multiple threads
- Shows when threads are created and destroyed
- `thread apply` can be used to apply commands to threads
- For instance, `thread apply all backtrace`

`ps` shows an overview of currently active processes
- `ps -Lf` and `ps -T` show processes and threads

`top` displays a live view of currently running processes
- `top -H` shows threads in addition to processes
Programming with OpenMP

Review

Introduction

Overview

Parallelism

Synchronization

Summary
The single directive causes only one thread to execute the code
- Not specified which thread
- Typically the one that reaches it first
- Can force master thread with master
  - Ensures it is always the same thread
- single has an implicit barrier
  - All threads will wait for completion

```c
int main(void) {
    #pragma omp parallel
    {
        #pragma omp single
        printf("Single %02d/%02d\n", 
            omp_get_thread_num (), 
            omp_get_num_threads ());

        printf("Thread %02d/%02d\n", 
            omp_get_thread_num (), 
            omp_get_num_threads ());
    }
    return 0;
}
```
- The single directive causes only one thread to execute the code
  - Not specified which thread
  - Typically the one that reaches it first
- Can force master thread with master
  - Ensures it is always the same thread
- single has an implicit barrier
  - All threads will wait for completion

<table>
<thead>
<tr>
<th>Single</th>
<th>06/12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>00/12</td>
</tr>
<tr>
<td>Thread</td>
<td>10/12</td>
</tr>
<tr>
<td>Thread</td>
<td>04/12</td>
</tr>
<tr>
<td>Thread</td>
<td>02/12</td>
</tr>
<tr>
<td>Thread</td>
<td>06/12</td>
</tr>
<tr>
<td>Thread</td>
<td>08/12</td>
</tr>
<tr>
<td>Thread</td>
<td>05/12</td>
</tr>
<tr>
<td>Thread</td>
<td>11/12</td>
</tr>
<tr>
<td>Thread</td>
<td>01/12</td>
</tr>
<tr>
<td>Thread</td>
<td>09/12</td>
</tr>
<tr>
<td>Thread</td>
<td>07/12</td>
</tr>
<tr>
<td>Thread</td>
<td>03/12</td>
</tr>
</tbody>
</table>
Synchronization

- `nowait` disables implicit barriers
  - Has to be used carefully
  - Can lead to race conditions
- Can be used for several directives
  - `for` and `sections` have implicit barriers
- `master` has no implicit barrier

```c
int main(void) {
    #pragma omp parallel
    {
        #pragma omp single nowait
        printf("Single %02d/%02d\n",
               omp_get_thread_num(),
               omp_get_num_threads());

        printf("Thread %02d/%02d\n",
               omp_get_thread_num(),
               omp_get_num_threads());
    }

    return 0;
}
```
- `nowait` disables implicit barriers
  - Has to be used carefully
  - Can lead to race conditions
- Can be used for several directives
  - `for` and `sections` have implicit barriers
- `master` has no implicit barrier
• Reminder: Variables are shared by default
  • Access has to be synchronized

```c
int main(void) {
    int i, j, iters;

    for (i = 0; i < 10; i++) {
        iters = 0;
        #pragma omp parallel for
        for (j = 0; j < 100; j++)
            iters ++;
        printf("iters=%d\n", iters);
    }

    return 0;
}
```
• Reminder: Variables are shared by default
  • Access has to be synchronized
• Race condition causes wrong result
  • Three operations are performed
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable
  • Have to be performed atomically
• Several possibilities to solve the problem
  • Add a lock around the operation (slow)
  • Use atomic instructions (fast)
Synchronization

- critical protects statement with a lock
  - Can be entered by one thread at a time
- All threads execute the critical region
  - Have to wait for others to finish
  - Critical region is serialized
- Use atomic for simple instructions
  - Atomic operations are faster than locks

```c
int main(void) {
    int i, j, iters;

    for (i = 0; i < 10; i++) {
        iters = 0;
        #pragma omp parallel for
        for (j = 0; j < 100; j++)
            #pragma omp critical
                iters++;
        printf("iters=%d\n", iters);
    }

    return 0;
}
```
Critical...

- critical protects statement with a lock
  - Can be entered by one thread at a time
- All thread execute the critical region
  - Have to wait for others to finish
  - Critical region is serialized
- Use atomic for simple instructions
  - Atomic operations are faster than locks
Atomic Synchronization

- Only applies to a single statement
  - Critical regions can contain multiple
- Only specific statements
  - Assignments, increment, decrement, binary operations etc.
- Compiler uses atomic instructions
  - Guaranteed by the CPU to be atomic

```c
int main(void) {
    int i, j, iters;

    for (i = 0; i < 10; i++) {
        iters = 0;
        #pragma omp parallel for
        for (j = 0; j < 100; j++)
            #pragma omp atomic
            iters++;
        printf("iters=%d\n", iters);
    }
    return 0;
}
```
• Only applies to a single statement
  • Critical regions can contain multiple
• Only specific statements
  • Assignments, increment, decrement, binary operations etc.
• Compiler uses atomic instructions
  • Guaranteed by the CPU to be atomic
Reduction

- reduction variables are hybrid
  - Each thread has a private copy
  - All variables are reduced to shared result
- Several pre-defined reductions
  - Addition, multiplication, subtraction, and, or, xor, min, max
  - User-defined since OpenMP 4.0
- Reduction determines initialization value
  - 0 for addition etc.
- Order might change result
  - Especially for floating-point values

```c
int main(void) {
    int i, j, iters;

    for (i = 0; i < 10; i++) {
        iters = 0;
        #pragma omp parallel for reduction(+:iters)
        for (j = 0; j < 100; j++)
            iters++;
        printf("iters=%d\n", iters);
    }

    return 0;
}
```
Reduction

- reduction variables are hybrid
  - Each thread has a private copy
  - All variables are reduced to shared result
- Several pre-defined reductions
  - Addition, multiplication, subtraction, and, or, xor, min, max
  - User-defined since OpenMP 4.0
- Reduction determines initialization value
  - 0 for addition etc.
- Order might change result
  - Especially for floating-point values

Synchronization

```
iter = 100
iter = 100
iter = 100
iter = 100
iter = 100
iter = 100
iter = 100
iter = 100
iter = 100
```
Reduction

• reduction variables are hybrid
  • Each thread has a private copy
  • All variables are reduced to shared result

• Several pre-defined reductions
  • Addition, multiplication, subtraction, and, or, xor, min, max
  • User-defined since OpenMP 4.0

• Reduction determines initialization value
  • 0 for addition etc.

• Order might change result
  • Especially for floating-point values

<table>
<thead>
<tr>
<th>V0</th>
<th>Thread 0</th>
<th>V1</th>
<th>Thread 1</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Load 0</td>
<td>0</td>
<td>Load 0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>Inc 1</td>
<td>0</td>
<td>Inc 1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Store 1</td>
<td>1</td>
<td>Store 1</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>50</td>
<td>+</td>
<td>50</td>
<td>=</td>
<td>100</td>
</tr>
</tbody>
</table>
Locks

- OpenMP generates locks implicitly
  - For instance, for critical regions
- Library functions can be used for locking
  - Enables more general applications
- Locks have to be initialized and destroyed
  - Can be set and unset
  - Threads will wait for a set lock

```c
int main(void) {
    int i, iters = 0;
    omp_lock_t lock[1];

    omp_init_lock(lock);
    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
        omp_set_lock(lock);
        iters++;
        omp_unset_lock(lock);
    }
    printf("iters=%d\n", iters);
    omp_destroy_lock(lock);
    return 0;
}
```
• OpenMP generates locks implicitly
  • For instance, for critical regions
• Library functions can be used for locking
  • Enables more general applications
• Locks have to be initialized and destroyed
  • Can be set and unset
  • Threads will wait for a set lock
Barriers

- Reminder: No atomicity
  - Single statements are run independently
  - Problematic for split statements
- Ordering is more or less random
  - Depending on timing and other factors

```c
int main(void) {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        printf("Hello top.\n");
        printf("Hello bottom.\n");
    }
    return 0;
}
```
Barriers

• Reminder: No atomicity
  • Single statements are run independently
  • Problematic for split statements
• Ordering is more or less random
  • Depending on timing and other factors
• Barriers can also be used explicitly
  • Allow waiting at a synchronization point

Hello top.
Hello top.
Hello top.
Hello bottom.
Hello bottom.
Hello bottom.
Hello top.
Hello bottom.
Barriers...

- barrier waits until all threads reach it
  - Defines a common synchronization point
- All threads have to reach the barrier
  - Or all threads have to skip it
  - Could potentially produce a deadlock
- Useful to ensure previous work is finished
  - Can also become a performance problem

```c
int main(void) {
    omp_set_num_threads(4);
    
    #pragma omp parallel
    {
        printf("Hello top.\n");
        #pragma omp barrier
        printf("Hello bottom.\n");
    }
    
    return 0;
}
```
Barriers...

- barrier waits until all threads reach it
  - Defines a common synchronization point
- All threads have to reach the barrier
  - Or all threads have to skip it
  - Could potentially produce a deadlock
- Useful to ensure previous work is finished
  - Can also become a performance problem

Hello top.
Hello top.
Hello top.
Hello top.
Hello bottom.
Hello bottom.
Hello bottom.
Hello bottom.
OpenMP is a standard for shared memory programming
- It is widely supported by compiler vendors for C/C++ and Fortran
- It enables convenient and portable parallel programming

OpenMP includes common functionality for thread parallelism
- Thread management, work sharing, synchronization, offloading and vectorization

OpenMP uses compiler pragmas, library functions and environment variables
- Most functionality is provided by compiler pragmas, which can be turned off easily