Introduction to OpenMP

Lukas Einkemmer
Department of Mathematics
University of Innsbruck

Shared memory parallelization with OpenMP – Day 1.
Link to slides: `http://www.einkemmer.net/training.html`

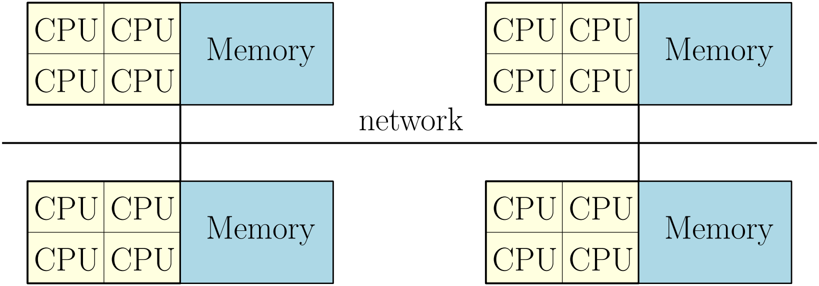
With special thanks to Rolf Rabenseifner (HLRS) on whose original slide set parts of this course are based.

An overview of OpenMP

Parallelization

Modern supercomputers are build by connecting (via a network) a large number of 'commodity computers'.

Each 'commodity computer' has multiple cores.



To exploit modern hardware parallelization is essential.

Parallelization

Parallelization is the task of dividing work among multiple execution engines.

Easy to parallelize

```
for(int i=0;i<n;i++)
    out[i] = in[i];
```

Requires a little bit more thought

```
int sum = 0;
for(int i=0;i<n;i++)
    sum += i;
```

Almost impossible (without further knowledge of f)

```
double x = 0;
for(int i=0;i<n;i++)
    x = f(x);
```

Parallel nomenclature

Thread is a set of sequential instructions that are executed in order.

Thread is a software construct. **Core** is a hardware construct.

- Often each thread in a program is mapped to a single core.

Shared memory model assumes that all threads read and write from the same memory.

Distributed memory model means that no shared memory is available. In this case communication has to be done by sending messages.

OpenMP

OpenMP

- is a common way to parallelize your code
- is a standard (since 1997)
- requires a shared memory system
- is an extension to C/C++ and Fortran (using directives, environment variables, and some library routines)
- is portable across shared memory architectures

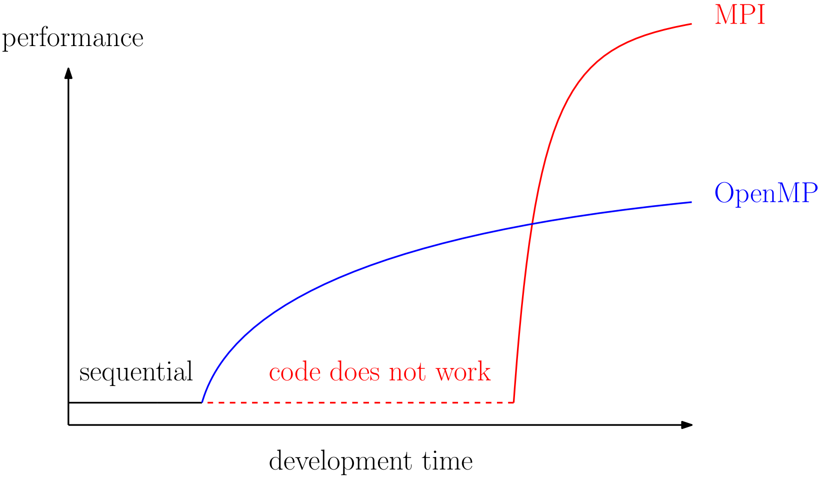
OpenMP focuses mostly on **parallelizing loops with independent iterations** (less and less true with each version).

Philosophy of OpenMP

- Parallelization with as little modification to the sequential program as possible.
- Incremental approach to parallelization.

Why should I use OpenMP?

OpenMP is the easiest technology to parallelize your application.



Where should I use OpenMP?

Use cases

- **Exploit node level parallelism** (i.e. parallelize across cores and for vectorization).
- Write programs for a shared memory supercomputer (such as the MACH-2 in Linz).
- Write programs for multiple GPUs on a single node.

Traditional clusters are distributed memory systems.
Predominant parallelization technology in this context is **MPI**.

Hybrid parallelization for **large supercomputers**.

- MPI (between nodes) + OpenMP (on each node)
- MPI (between nodes) + OpenMP + CUDA (on each node)

Simple example

To start with OpenMP is easy

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

```
!$OMP PARALLEL DO
do i=1,n
    out(i) = in(i)
end do
!$OMP END PARALLEL DO
```

Divides the loop iterations into pieces that are then executed in parallel by different threads.

OpenMP versions

OpenMP 2

- basic features, loop level parallelism

OpenMP 3

- task level parallelization
- additional features for loop level parallelism

OpenMP 3.1

- additional features for loop level parallelism
- thread affinity support (`OMP_PROC_BIND`)

Supported by reasonably recent version of all common compilers (e.g. gcc and icc).

OpenMP versions

OpenMP 4.0

- additional support for thread affinity (`OMP_PLACES`)
- support for vectorization
- support for accelerators (GPUs)

OpenMP 4.5

- taskloop construct

OpenMP 5.0

- reduction for tasks
- C++ for range loops

Most recent compilers support OpenMP 4.5 (except for the accelerator focused features).

At the moment, widespread support for accelerators is lacking.

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/`

OpenMP homepage: `http://www.openmp.org`

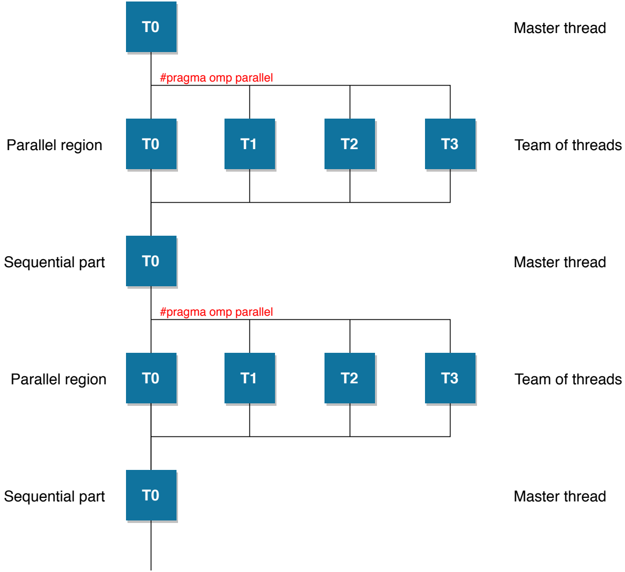
OpenMP programming and execution model

OpenMP execution model

- Program begins as a single process (master thread).
- At the beginning of a parallel region a team of threads is created.
- At the end of a parallel region threads synchronize (implied barrier).
- At the end of a parallel region execution continues sequentially.

OpenMP execution model

Fork-join model of parallel execution



OpenMP directive format

C/C++

```
#pragma omp directive_name [clause, [[,] clause] ... ]
{
    // code
}
```

Conditional compilation

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

Fortran

```
!$OMP directive_name  [clause, [[,] clause] ... ]
! code
!$OMP END directive_name
```

Conditional compilation: enable preprocessor with the -cpp flag (gcc) and -fpp (icc). Alternative: comments starting with !$ are executed only if OpenMP is available.

OpenMP parallel region

A **parallel region** creates a team of threads that (potentially) execute the workload.

```
#pragma omp parallel
{
    cout << "Hello World!" << endl;
}
```

prints (assuming OpenMP uses 3 threads)

```
$ g++ helloworld.cpp -o helloworld -fopenmp
$ ./helloworld
Hello World!
Hello World!
Hello World!
```

Code is executed redundantly.

Intel compiler uses -qopenmp (instead of -fopenmp).

OpenMP library functions

The header file omp.h/module omp_lib provides library functions.

```
// returns the thread id (0 to omp_get_num_threads()-1)
omp_get_thread_num()
// returns the number of threads in the current team
omp_get_num_threads()
```

```
#pragma omp parallel
{
    if(omp_get_thread_num()==0)
        cout << "Number of threads: "
             << omp_get_num_threads() << endl;

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

prints

```
Number of threads: 3
Hello world from thread 2
Hello world from thread 0
Hello world from thread 1
```

OpenMP execution model

There is no guarantee in which order the threads are executed.

If a specific order is desired this has to be enforced (might be **very expensive**).

The thread id can be used to divide the work among threads. But this is a lot of boilerplate. **OpenMP provides facilities to automatically divide the work among the threads in a team.**

- The corresponding directives are called worksharing directives.

Controlling the number of threads

The number of threads used by OpenMP can be set as follows:

By using environment variables:

```
# Set number of threads for the entire session
export OMP_NUM_THREADS=4; ./program
# or only for one execution of the program
OMP_NUM_THREADS=4 ./program
```

By appending a clause to the OpenMP directive:

```
#pragma omp parallel num_threads(4)
```

By calling an OpenMP library function:

```
#include <omp.h>
omp_set_num_threads(10);
```

The default, often the number of hyperthreads in the system, is usually **not** an optimal choice.

- **Rule of thumb:** number of threads = number of cores.

Time your code

OpenMP provides wall clock timers

```
double t1 = omp_get_wtime();
// code
double t2 = omp_get_wtime();
cout << "Execution took " << t2-t1 << " s" << endl;
```

Precision of the timer can be queried by using `omp_get_wtick`.

Note that std::clock and many other timers return the CPU time.

- CPU time is the accumulated execution time of all threads that are used by the program.

OpenMP data environment

Shared memory model: All threads can write and read from main memory.

There are two types of variables:

- **shared** variables are common to all threads (usually arrays, global variables, ...).
- **private** variables are duplicated on each thread (local variables, loop counters, ...).

Example:

```
int n=10;                // shared integer
vector<double> in(n); // shared array
#pragma omp parallel for
for(int i=0;i<n;i++) {
    double x = 3*in[i]; // private double
    in[i] = x;
}
```

OpenMP data environment

By default all variables are shared.

Exceptions

- local variables defined inside an OpenMP directive
- Loop control variables for a parallel for loop
- Variables that are declared in a called function

A variable can be explicitly declared as private or shared

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

In C++ this is almost never necessary. Good practice: define variables where they are used.

OpenMP data environment

Be careful: private variables inside and outside an OpenMP directive are not storage associated.

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

If this behavior is desired we can use

```
double x=3;
double y;
#pragma omp parallel for firstprivate(x) lastprivate(y)
for(int i=0;i<10;i++)
    y = i + x; // here x is equal to 3
// y is equal to whatever value is set in the last
// iteration of the loop (here 9+3=12)
```

Exercise

Goals:

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

A sequential hello world program is provided.

- **C/C++:** hello.c
- **Fortran 90:** hello.f90

Exercise 1a

Compile the program and run

```
export OMP_NUM_THREADS=4
./hello
```

Expected result:

- program is not parallelized so nothing changes

Exercise 1b

Tasks:

- Add a parallel region that prints the id of each thread and the total number of threads.
- Compile and run with 4 threads.

Expected result:

```
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
```

Why does the order of the output change from run to run?

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 the Intel compiler and gcc;
- with optimization turned on (-O3) and optimization turned off;
- by increasing the number of threads?

Expected result:

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

Why do you observe correct results for some configurations, even though there is a race condition in the program?

Exercise 1d

Tasks:

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

Expected result:

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

OpenMP worksharing directives

Worksharing directives

Worksharing directives **distribute work among the threads in a team**.

Worksharing directives do **not create new threads** and thus must be enclosed within a parallel region.

Commonly used worksharing directives in OpenMP:

- sections
- for/do
- task
- single/master
- workshare (Fortran only)

Worksharing: sections

The OpenMP **sections** directive can be used for very coarse grained parallelism.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            a0 = 10;
            b0 = 20;
        }
        #pragma omp section
        {
            a1 = 15;
            b1 = 22;
        }
    }
}
```

Worksharing: for

OpenMP **for** directive.

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

Worksharing construct and parallel regions can be combined.

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

Canonical form of a for loop

For loop

```
for(type var=lb; var<b; var+=incr)
```

must be in **canonical form.**

- Type must be an integer type, a pointer type, or a random access iterator.
- Comparison must be $<$, $<=$, $>$, or $>=$.
- lb, b, incr must not change during the execution of the loop.

Essentially, it must be possible to determine the number of iterations at the time the loop starts execution.

Nested for loops

Nested for loops, of canonical form, can be parallelized as follows

```
#pragma omp parallel for collapse(2)
for(int i=0;i<n;i++)
    for(int j=0;j<m;j++)
        // code
```

The **collapse** clause takes one integer argument, the number of nested for loops to parallelize.

Recommendation: Only use collapse if the outermost loop does not expose enough parallelism (i.e. if `OMP_NUM_THREADS` $> n$).

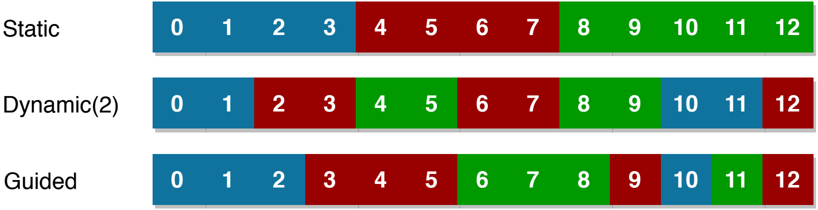
Recommendation: Make sure that each loop iteration has as much work to do as possible (for nested loops parallelize the outermost loop).

Loop scheduling

How the loop iteration are divided among the threads can be influenced by specifying the **schedule clause**. The form is (specifying a chunk size is optional)

```
#pragma omp for schedule(type,chunk)
```

The basic scheduling strategies



Loop scheduling

static: divides the loop into pieces of size specified by chunk. If no chunk size is given the pieces are chosen as large as possible.

- has the smallest overhead and should be used in most cases.

dynamic: functions are broken into pieces of a size specified by chunk. If a thread finishes a new chunk is assigned to that thread. Default chunk size is 1.

guided: Similar to dynamic but the chunk size is decreased in an exponential manner. The variable chunk sets the smallest possible piece (default is 1).

Dynamic and **guided** are used to perform load balancing. E.g. for cases where different loop iterations have different computational cost.

auto: OpenMP chooses the 'best' scheduling strategy.

runtime: A scheduling strategy can be chosen by setting the OMP_SCHEDULE environment variable.

Worksharing: workshare (Fortran only)

The OpenMP **workshare** directive can be used to parallelize array expressions and `FORALL` statements.

```
!$OMP WORKSHARE
A=B+C
!$OMP END WORKSHARE
```

Combining multiple parallel regions

For efficiency reasons we can combine multiple parallel regions.

```
#pragma omp parallel for
for(int i=0;i<n-1;i++)
    out[i] = in[i] + in[i+1]

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

becomes

```
#pragma omp parallel
{
    #pragma omp for
    for(int i=0;i<n-1;i++)
    out[i] = in[i] + in[i+1]

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

Combining multiple parallel regions

The **single** or **master** directive can be used to embed sequential code inside a parallel region.

```
#pragma omp parallel
{
    #pragma omp for
    for(int i=0;i<n-1;i++)
        out[i] = in[i] + in[i+1]

    #pragma omp single
    out[n-1] = 3.0;

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

Master requires that the block be executed by thread 0; there is no implied synchronization at the end of master.

Worksharing: task

The OpenMP **task** worksharing construct allows us to parallelize code that is more irregular.

The idea is that at certain points in the code a **task is created**. The task can either

- **execute immediately** (if idle threads are available);
- or **defer execution** until later.

Advantages

- No need to know a priori how many tasks will be created.
- Tasks provide automatic load balancing.

Disadvantages

- Increased overhead compared to loop level parallelism.
- Difficult to synchronize or exchange data between tasks.

Apply a function to each node in a tree

```
struct node {
    node *left, *right;
};
void traverse(node* p) {
    if(p->left)
        #pragma omp task
        traverse(p->left); // this is created as a task
    if(p->right)
        #pragma omp task
        traverse(p->right); // this is created as a task
    process(p);
}
int main() {
    node tree;
    #pragma omp parallel // create a team of threads
    {
        #pragma omp single
        traverse(&tree); // executed sequentially
    }
}
```

A word of warning

OpenMP is easy to write, but it is also easy to get wrong.

OpenMP **delegates a lot of responsibility to the programmer**.

Ensure that the code can be parallelized

We have to make sure that the loop iterations are independent.

WRONG!

```
#pragma omp parallel for
for(int i=0;i<n-1;i++)
    in[i] = in[i] + in[i+1]
```

Correct alternative.

```
#pragma omp parallel for
for(int i=0;i<n-1;i++)
    out[i] = in[i] + in[i+1]

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

Race conditions

WRONG!

```
double s=0; // shared variable
#pragma omp parallel for
for(int i=0;i<n;i++)
    s += in[i];
```

A **race condition** occurs when multiple threads are allowed to access the same memory location and at least one access is a write.

A program with a race condition is always wrong.

Here the race condition is hidden

```
// s is written and read from all threads
s = s + in[i];
```

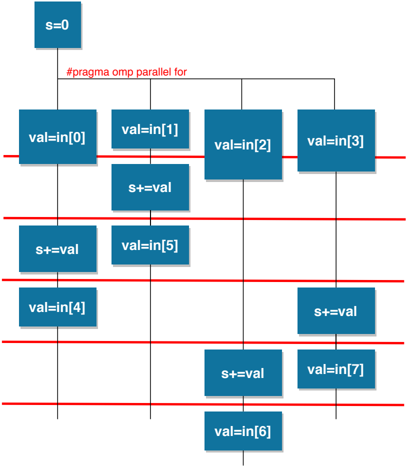
Reduction

Reduction, combining multiple values into a single one, is a common pattern. For example, to compute the norm of a vector.

The race condition can be avoided by using critical.

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

The **critical** directive ensures that only one thread enters the critical region at any one time.



Unfortunately, a critical region is extremely expensive. **In the above program we require *n* critical regions.**

Faster reduction

Reduce the number of critical regions to increase performance.

```
double s=0; // shared variable
#pragma omp parallel
{
    // local_s is a private variable in parallel region
    double local_s = 0;

    // each thread computes a local sum and stores it
    // in its local_s
    #pragma omp for
    for(int i=0;i<n;i++) {
        double val = in[i];
        local_s += val;
    }

    // perform the reduction
    #pragma omp critical
    s += local_s;
}
```

This requires only `OMP_NUM_THREADS` many critical regions.

Reduction using atomic

Many computer architectures provide hardware support for so-called **atomics.**

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

Usually results in improved performance, but the form the update statement is allowed to take is much more restricted than with critical.

OpenMP replaces atomic with critical if no hardware support is available.

Exercise 2a

Goal:

- for/do worksharing construct
- critical directive

Add parallel region and for/do directive in pi.[c|f90|f] and compile the program.

Expected result: Output of 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

OpenMP reduction clause

OpenMP provides built in support for performing reductions.

```
double s=0;
#pragma omp parallel for reduction(+:s)
for(int i=0;i<n;i++)
    s += in[i];
```

This keeps within the philosophy of OpenMP: the parallel code should be as close as possible to the sequential code.

The reduction variable s must be shared and can be an array. In C++ the length of the array has to be specified (OpenMP 4.5)

```
#pragma omp parallel for reduction(+:pointer_to_s[:n])
```

Thread local variables

It is often useful to have global

```
double x;
#pragma omp threadprivate(x)
```

or static variables

```
void test() {
    static double x;
    #pragma omp threadprivate(x)
}
```

private to each thread.

Display environment variables

```
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.

Summary