

# Computer Architecture

Jens Teubner, TU Dortmund  
jens.teubner@cs.tu-dortmund.de

Summer 2019

# Part I

## Programming for Parallelism

**Moore's Law**  $\rightsquigarrow$  parallelism  $\nearrow$

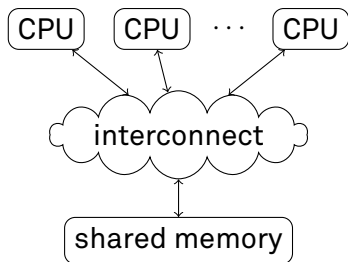
**But:**

- Parallelism  $\rightarrow$  communication/synchronization
- Amdahl's Law:
  - $\rightarrow$  Must keep sequential code parts **really** small.

**Therefore:**

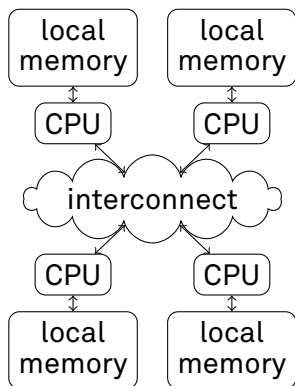
- Need **efficient communication/synchronization mechanisms**
  - $\rightarrow$  What are the ways to realize that?
- Parallelize **aggressively**
  - $\rightarrow$  Tool support?

## Shared Memory:



- Memory accessible by all CPUs/threads.
- Threads communicate through memory.

## Message Passing:



- Threads communicate through messages.

# Shared Memory ↔ Message Passing

 **Pros/Cons of the two mechanisms?**

## OpenMP: An API for Shared Memory Multiprocessing

- Non-profit industry consortium
- First version: October 1997 (for Fortran)
- Latest version: November 2018 (version 5.0; Fortran/C/C++)
- Supported by many modern compilers (e.g., gcc, LLVM, icc)

### Idea:

- **Shared memory**, parallelism through **threads**
- Start from **sequential code**, introduce parallelism through **pragmas** (`#pragma` in C/C++)

**Pragmas** are directives for the **compiler**.

E.g., C/C++: (here: ignore warning for uninitialized variable)

```
#pragma diagnostic ignored "-Wuninitialized"  
foo (b);
```

OpenMP pragmas begin with `#pragma omp`:

```
#pragma omp parallel for  
for (unsigned int i=0; i<N; i++)  
    a[i]=b[i] + k*c[i];
```

Pragmas are **ignored** if the compiler does not understand them.

# “Hello World!” of OpenMP

```
#pragma omp parallel for
for (unsigned int i=0; i<N; i++)
    a[i] = b[i] + k*c[i];
```

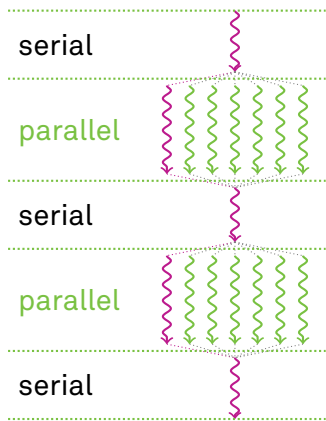
- Compiler will automatically assign loop iterations to parallel threads.
  - Number of threads created is (by default) at the compiler's discretion.
- Pragma is ignored by a non-OpenMP-aware compiler.
  - Above code will run just fine as sequential code.
- Strategy:
  - Identify performance-critical loops.
  - Parallelize using OpenMP pragmas.



# OpenMP Fork/Join Model

The OpenMP `parallel` clause declares a **parallel region**.

```
printf ("Initializing...\n");  
  
#pragma omp parallel for  
for (int i=0; i<1000; i++)  
    a[i] = random ();  
  
printf ("Incrementing...\n");  
  
#pragma omp parallel for  
for (int i=0; i<1000; i++)  
    a[i] = a[i] + 1;  
  
printf ("Done.\n");
```



↪ `pthread_create () / pthread_join ()`

## Actually...

- Most `#pragma omp` directives apply to the following **structured block of code**.
  - *E.g.*, a C statement or a block inside `{/}`
  - Block must have **single entry** and **single exit**.
- `#pragma omp parallel` **only** tells the master thread to fork and create a team of threads
  - **All** threads will execute all of the code in the block.
  - Threads will share (almost<sup>1</sup>) **all** data.

 **What will be the output of the following code?**

```
#pragma omp parallel
printf("Hello World!\n");
```

---

<sup>1</sup>Each thread will have its own program counter and stack.

# Library Functions

Library functions help to interface with OpenMP.

```
#include <omp.h>
int
main(int argc, char **argv)
{
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        unsigned int id = omp_get_thread_num();
        unsigned int cnt = omp_get_num_threads();
        printf("I'm thread number %u (out of %u).\n",
              id, cnt);
    }
}
```

#pragma omp parallel:

- Implicit **barrier** at the end of the structured block.
  - Execution continues only after all threads have joined.
- Threads may or may not actually terminate at block end.
  - Thread creation is expensive.
  - Compiler might decide to rather put threads to sleep and wake them up again later.

Parallelization of for loops is a common situation.

→ OpenMP for directive

```
#pragma omp for
for (unsigned int i=0; i<1000; i++)
    do_something(i);
```

If #pragma omp for is found **outside parallel region**:

→ Directive has no effect.

If #pragma omp for is found **inside parallel region**:

→ **Distribute** iterations over threads.

→ Only allowed for “simple” loops where number of iterations can be counted before the loop evaluation starts.

## Example:

```
#pragma omp parallel
{
    #pragma omp for
    for (unsigned int i=0; i<1000; i++)
        do_something(i);
}
```

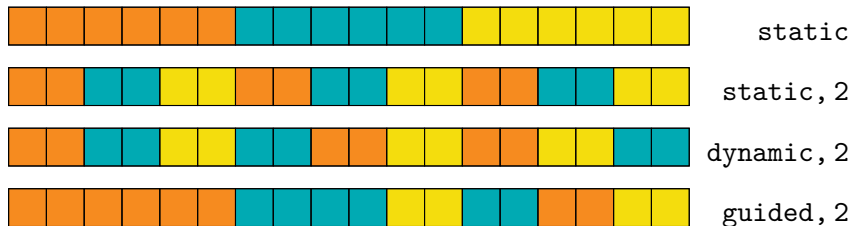
## More convenient shortcut:

```
#pragma omp parallel for
for (unsigned int i=0; i<1000; i++)
    do_something(i);
```

## Scheduling of the following loop?

```
#pragma omp parallel for  
for (unsigned int i=0; i<18; i++)  
    do_something (i);
```

**Possible strategies:** (■: Thread 1; ■: Thread 2; ■: Thread 3)



# Loop Scheduling (cont.)

Loop scheduling can be controlled using the `schedule` **clause**:

```
#pragma omp parallel for schedule(dynamic,2)
for (unsigned int i=0; i<18; i++)
    do_something(i);
```

- If no `schedule` clause is given, the compiler will use its **implementation-dependent default**.
- The `static` policy may be helpful when you need to know which thread processes which subset of the data.
- Use `dynamic` when you expect that execution time varies between iterations.



## Shared Memory programming model:

- By default, all resources are **shared** between threads.
- **Except:**
  - Program counter, register contents, etc. are private.
  - Each thread has its private stack.  
Locally created variables are thus private, too.
- Sharing can be controlled using **clauses** in the `parallel/for` constructs.
- **Private** variables:
  - Each thread gets its own copy.
  - These copies are **not initialized** by OpenMP.
- `for` construct → **loop variable** becomes **private**

# Data Sharing Clauses

`shared ( list of variables )`

- All threads read/write to **same memory location**.
- No (automatic) protection by OpenMP.

`private ( list of variables )`

- Each thread gets an **uninitialized copy** of the variable.
- Only visible to the respective thread.

`default ( shared | private2 | none )`

- `shared`: All variables shared by default.
- `private`: All variables private by default.
- `none`: Sharing mode for each variable must be explicitly declared.

More sharing modes:

- `lastprivate`, `firstprivate`, `reduction`, `copyin`; `threadprivate`, ...

---

<sup>2</sup>Only allowed in Fortran.

# Synchronized Access to Shared Variables

**Critical sections** can be specified using the `critical` construct:


```
unsigned int aggregate=0;

#pragma omp parallel for
for (unsigned int i=0; i<100; i++)
{
    unsigned int foo;
    foo=compute_something (a[i]);

    #pragma omp critical
    { aggregate+=foo; }
}
```

For some simple assignments, `atomic` can be used instead:

```
#pragma omp atomic
aggregate += foo;
```

 **Advantage?**

reduction helps to handle a common task:

```
#pragma omp parallel for reduction(+:sum)
for (unsigned int i=0; i<100; i++)
    sum += a[i];
```

- **Local variable** (copied from shared during fork)
- At join, local variables are combined and assigned to shared variable.
- The reduction clause contains
  - an **operator** using which local variables are combined and
  - a **variable name** (as for the other data sharing clauses).

# Barrier Synchronization

By default, parallel threads will synchronize at **end of structured block**.

To synchronize in-between, use `#pragma omp barrier`:

→ Threads will wait until **all** threads have reached the barrier.

```
#pragma omp parallel
{
    do_this ();

    #pragma omp barrier
    /* Execute only after all threads have
       finished do_this () */
    do_that ();
}
```

(For performance reasons,) automatic barrier synchronization can be skipped.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (unsigned int i=0; i<N; i++)
        do_this (i);

    /* Some threads will still be running do_this (),
       while others will already be doing do_that () */
    do_that ();
}
```

# Ordering

The **order** in which threads execute and finish loop iterations is unspecified.

The `ordered` clause can be used to control the order of some operations within a loop.

```
#pragma omp for ordered schedule(dynamic)
for (unsigned int i=0; i<100; i++)
{
    compress (files[i]);

    #pragma omp ordered
    send (files[i]);
}
```

- The `compress ()` part will be executed in **unspecified** order.
- Order **will be enforced** for the `send ()` call.

# Work-Sharing Constructs

**for** is also called a **work-sharing** construct.

Another work-sharing construct is `sections`:

```
#pragma omp parallel sections
{
    work_one ();

    #pragma omp section
    { work_two ();
      work_three (); }

    #pragma omp section
    work_four ();
}
```

- All `work_x` calls will be executed **exactly once**.
- The tasks `work_one ()`, `work_two () + work_three ()`, and `work_four ()` may be run in parallel.



# Running Code by Just One Thread

## The directives

- `#pragma omp single` and
- `#pragma omp master`

can be used to enforce execution of code blocks by just one thread.

→ `single` → **some** thread will run it

→ `master` → the **master** thread will run it

```
#pragma omp parallel
{
    work_one ();

    #pragma omp single
    { work_two ();
      work_three (); }

    work_four ();
}
```

# MPI—Message Passing Interface

- De facto standard for **communication protocol** in large-scale computing systems.
- Driven mostly by **researchers** (from academia and industry).
- Designed to be **portable**.
- Effort started in 1991, current version is 3.1 (June 2015)<sup>3</sup>.
- Several free implementations available.

## Programming Model:

- MPI defines an **API** for **message passing**.
- Implementations available for numerous communication substrates
  - “Real networks”
  - Shared memory

---

<sup>3</sup>This is a book of 868 pages!

# MPI—Message Passing Interface

The core of MPI are the `send` and `receive` functions.

`send ( “buffer”, “dest”, tag )`

- Send data (specified by “buffer”) to the machine given by “dest”.
- *tag*: message “type”

`receive ( “buffer”, “source”, tag )`

- Listen for messages of type *tag*, coming from “source”
- Put received data into “buffer”

## **Two-sided communication:**

- There must be matching processes that `send` and `receive` data.

More specifically:

`MPI_Send ( address , count , datatype , destination , tag , comm )`

- Send *count* elements of type *datatype*, starting from memory address *address*.
- *destination* is an integer, also called **rank**
  - Processes in an MPI group are identified by integers  $0 \dots n$ .
- *tag* is also an integer, used for message matching.
- *comm*: communication context
  - Identifies a group of MPI processes.

`MPI_Recv ( address , maxcount , datatype , source , tag , comm , status )`

- The receive buffer at *address* must be large enough to hold *maxcount* objects of type *datatype*.
- *source* is the rank of the node that we want to receive from  
→ or use wildcard `MPI_ANY_SOURCE`
- *status* is a return parameter with information about actual size, source, and tag.

## Note:

- For both, send and receive, **blocking** and **non-blocking** versions exist.

# Data Types

All data in MPI is **typed**.

- Types are **machine- and language-independent**.
- This increases **portability**.

MPI base types:

MPI type	C type
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
⋮	⋮

**Derived datatype:** sequence of basic data types.

→ Represented as a **type map**:

$$\text{type map} = \{ \langle \text{type}_0, \text{disp}_0 \rangle, \dots, \langle \text{type}_{n-1}, \text{disp}_{n-1} \rangle \} .$$

Displacement  $\text{disp}_i$ :

→ Data elements need **not** be contiguous in memory.

 **Why is this useful, even important?**

The **type signature** is a list of type, without the displacements:

$$\text{type signature} = \{ \text{type}_0, \dots, \text{type}_{n-1} \} .$$

# MPI “Hello World!”

```
#include <mpi.h>
int main(int argc, char **argv) {
    char msg[40];
    int myrank;
    MPI_Status status;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
    if (myrank==0) {
        strcpy(msg, "Hello, there");
        MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else if (myrank==1) {
        MPI_Recv(msg, 40, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received: %s\n", msg);
    }
    MPI_Finalize();
}
```



API functions exist for typical patterns; e.g.:

- `MPI_Bcast ()`
  - Replicate data (broadcast) to a group of processes
- `MPI_Reduce ()`
  - Collect data and reduce (↔ OpenMP's reduce)
- `MPI_Scatter ()`
  - Distribute an array of data; one piece to every process
- `MPI_Gather ()`
  - Opposite of `MPI_Scatter ()`
- `MPI_Allgather ()`, `MPI_Allreduce ()`
  - `MPI_Gather ()/MPI_Reduce ()`, plus `MPI_Bcast ()`
- `MPI_Barrier ()`
  - Wait for all processes to reach the barrier.