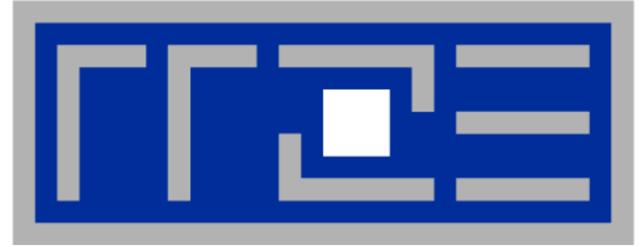


A very, very quick introduction to Shared-memory parallel processing with OpenMP

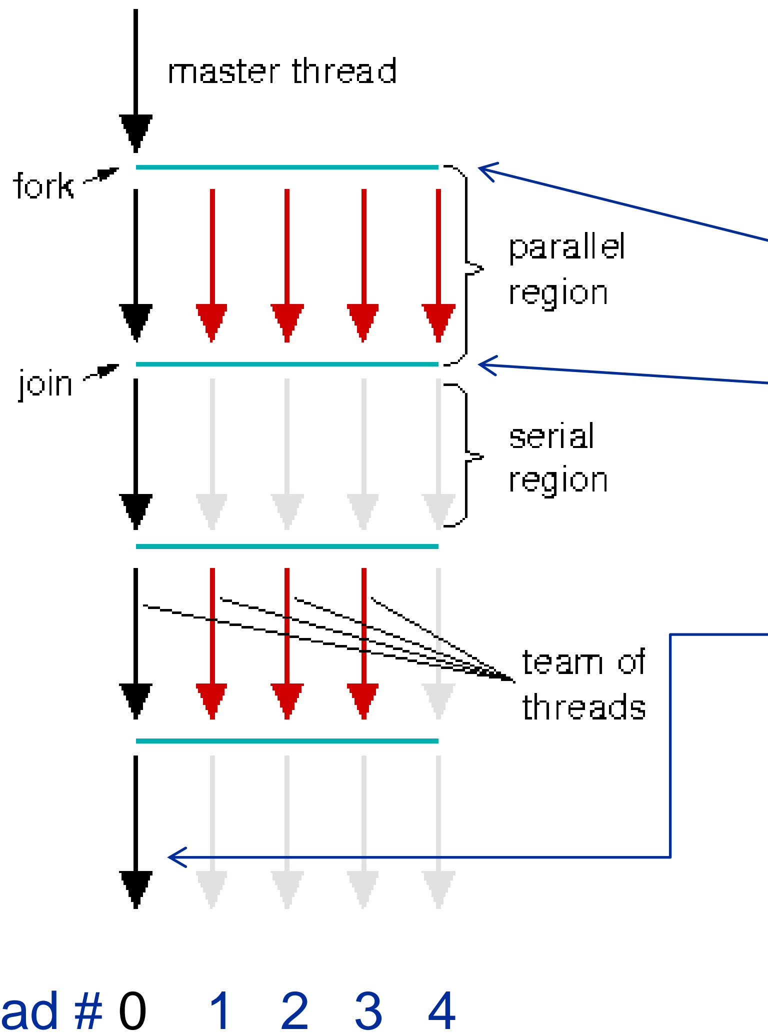
- **“Easy”, incremental and portable parallel programming of shared-memory computers: OpenMP**
- **Standardized set of compiler directives & library functions: `http://www.openmp.org/`**
  - FORTRAN, C and C++ interfaces are defined
  - Supported by all free and commercial compilers
  - Few free tools are available

- Books
  - B. Chapman, G. Jost, R. v. d. Pas: Using OpenMP. MIT Press, 2007, ISBN 978-0262533027
  - R. v.d. Pas, E. Stotzer, C. Terboven: Using OpenMP – The Next Step. MIT Press, 2017, ISBN 978-0262534789

Central concept of OpenMP programming: Threads

- Threads access globally shared memory
- Data: shared or private
  - shared data available to all threads (in principle)
  - private data only to thread that owns it
- Data transfer transparent to programmer
- Synchronization takes place, is mostly implicit
- Tailored to data parallel execution

Introduction to OpenMP: Fork and join execution model



Thread # 0 1 2 3 4

Program start:
only master thread runs

Parallel region: team of threads is generated (“fork”)

Synchronize when leaving parallel region (“join”)

Serial region:
only master executes

worker threads usually sleep

Task (and data) distribution possible via directives

Often best choice 1 thread/core

Application		User
Compiler Directives		Environment Variables
Runtime Library		
Threads in OS Cores in hardware		

- **Programmer's view:**
  - **directives/pragmas** in application code
  - (a few) library routines
- **User's view:**
  - **environment variables** determine:
    resource allocation
    scheduling strategies and other (implementation-dependent) behavior
- **Operating system view:**
  - parallel work done by **threads**

Introduction to OpenMP: Syntax in C/C++

- Include file: `#include <omp.h>`

- Compiler directive:

```
#pragma omp [directive [clause ...]]
   structured block
```

- Conditional compilation: Compiler's OpenMP switch sets preprocessor macro (acts like `-D_OPENMP`)

```
#ifdef _OPENMP

 ... do something

#endif
```

Introduction to OpenMP: Parallel execution

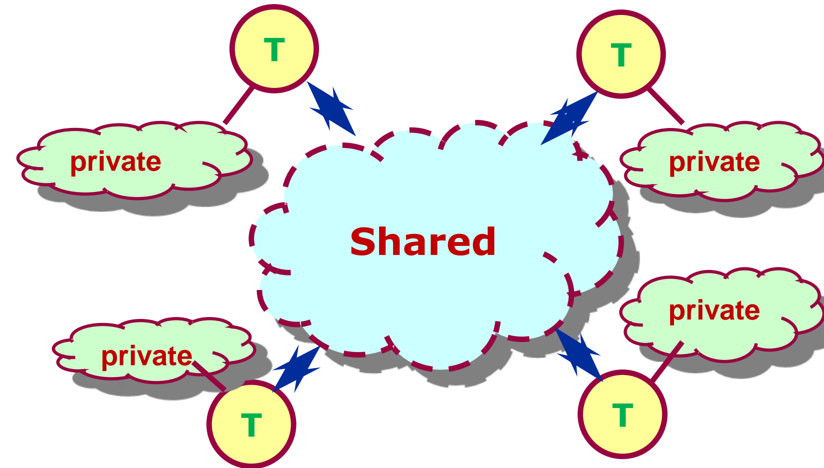
- `#pragma omp parallel`
  `    structured block`

  - Makes structured block a **parallel region**: **All code executed** between start and end of this region is executed **by all threads**.

  - This includes subroutine calls within the
  - Local variables inside the block are automatically private to each thread

```
#pragma omp parallel
{
  double f;   // one instance of f per thread
  f = do_work(omp_get_thread_num());
}
```

The OpenMP memory model



Data in a parallel region can be:

- private to each executing thread
  → each thread has its own local copy of data
- shared between threads
  → there is only one instance of data available to all threads
  → this does not mean that the instance is always visible to all threads!
- OMP clause specifies scope of variables:
  - Default: **shared**
  - Specify private variables in a parallel region:

```
#pragma omp parallel private(var1, tmp)
```

Introduction to OpenMP: Simplest program example

```
#include <omp.h>

int main(...) {
  int nthr, myth;

#pragma omp parallel private(myth, nthr)
{
  nthr = omp_get_num_threads();

  myth = omp_get_thread_num();

  cout << "Hello from" << myth
       << " of " << nthr ;
}

// ... serial execution here
}
```

- **Parallel region directive:**
  - enclosed code executed by **all** threads („lexical construct“)
  - may include subprogram calls („dynamic region“)
- **Special function calls:**
  - header omp.h
  - here: get number of threads and index of executing thread
- **Data scoping:**
  - uses a **clause** on the directive
  - `myth, nthr` thread-local: **private**

Introduction to OpenMP: Compile and run

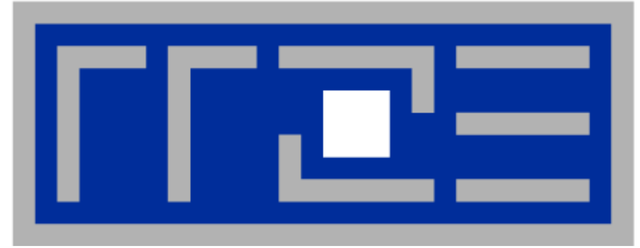
- Compiler must be instructed to recognize OpenMP directives (Intel compiler: `-openmp`)
- Number of threads: Determined by shell variable `OMP_NUM_THREADS`

```
$ export OMP_NUM_THREADS=4
$ ./a.out
 Hello from  0 of  4
 Hello from  2 of  4
 Hello from  3 of  4
 Hello from  1 of  4
```

Ordering not reproducible

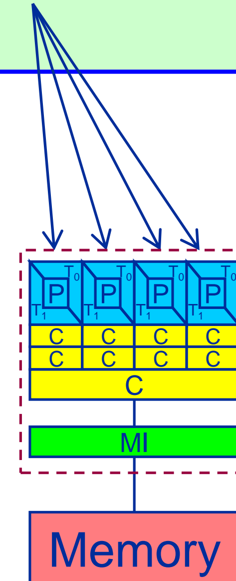
- More environment variables available:
  - Loop scheduling: `OMP_SCHEDULE`, Stacksize: `OMP_STACKSIZE`
  - Dynamic adjustment of threads: `OMP_DYNAMIC`
- Executable should be able to run with any number of threads!
- Thread pinning via `OMP_PROC_BIND`/`OMP_PLACES` or LIKWID tools (later)

```
$ export OMP_NUM_THREADS=4
$ likwid-pin -c 0-3 ./a.out
```



Loop worksharing as one example of parallelization in OpenMP

```
integer i, N
dp, dimension(N):: a,b,c,d
    …
do i=1,N
    a(i)=b(i)+c(i)*d(i)
enddo
```



`omp for` declares the loop following to be divided between threads if within a parallel region ("sliced")

- Loop counter of parallel loop is declared private implicitly
- No impact in serial region ("orphaned directive")

```
#pragma omp parallel
#pragma omp do        ! Parallelize loop
  for(int i=0; i<N; i++) {
    a[i] = b[i] + c[i] * d[i];
  }
```

`omp parallel` and `omp for` can be combined into `omp parallel for`

Implicit thread synchronization at end of worksharing construct and at end of parallel region! (suppress barrier after worksharing w/ `nowait`)

Introduction to OpenMP: `schedule` clause

Within `schedule(` *type* `[,` *chunk* `])` *type* can be one of the following:

- `static`: Iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number.
  Default chunk size: one contiguous piece for each thread.
- `dynamic`: Iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. *Default chunk size: 1.*
- `guided`: The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  chunk specifies the smallest piece (except possibly the last).
  Default chunk size: 1. Initial chunk size is implementation dependent.
- `runtime`: The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable.
- `auto`: Compiler decides

Default `schedule`: implementation dependent.

Introduction to OpenMP: `schedule` clause

Iteration

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

`STATIC` `STATIC,3` `DYNAMIC[,1]` `DYNAMIC,3` `GUIDED[,1]`

T0
T1
T2

- **Dense matrix-vector multiplication**

```
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=0; n<size; n++){
          y[m]+=a[m*size+n]*x[n];}
    }
    if(y[size>>1]<0){
      dummy(a,x,y,0);
    }

  }
}
```

Parallel execution starts here!

Workload Distribution: Every Thread receives “some” m-iterations

Full inner loop is executed by every thread for it’s m-iterations

Where is the private data????