



# Introduction to OpenMP

Dr. Stefan Albensoeder

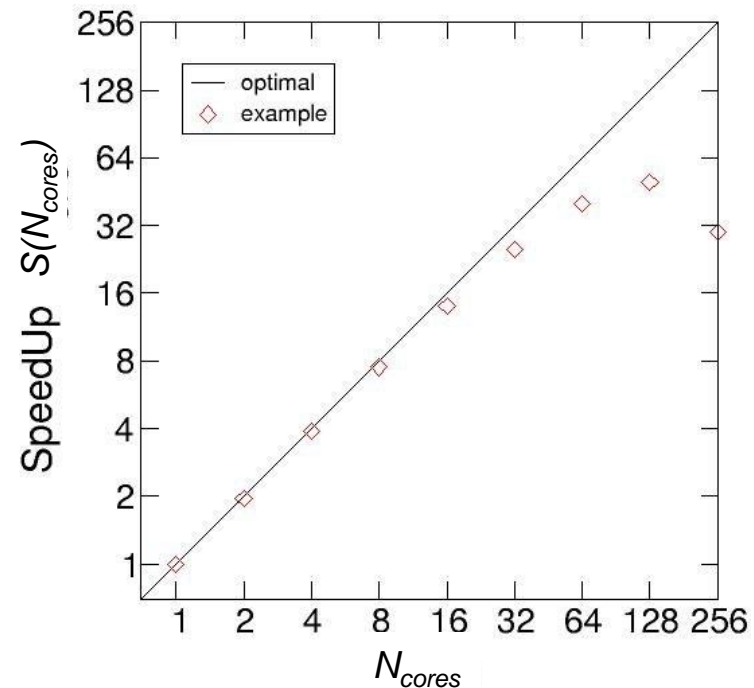
Contact: [Stefan.Albensoeder@uni-oldenburg.de](mailto:Stefan.Albensoeder@uni-oldenburg.de)

# INTRODUCTION PARALLELIZATION



# Limit of parallelization - Scaling

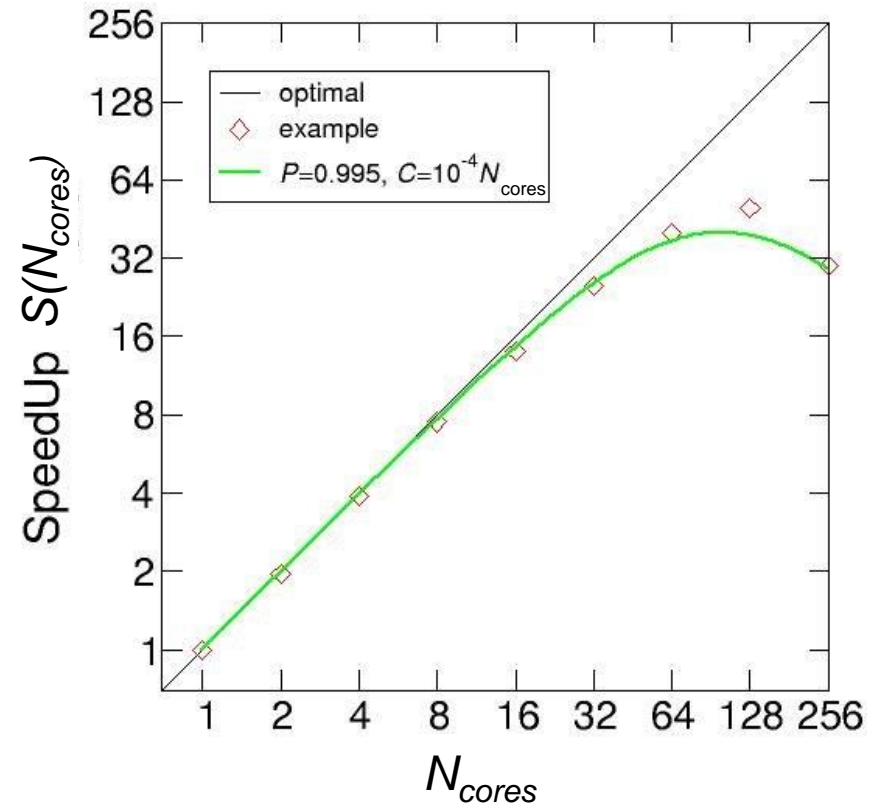
- Speedup of the program when increasing number of processes / used cores
- Dependencies
  - Fraction of parallelization code  $P$
  - Amount of communication / synchronization  $C \sim O(N_{cores})$
  - Size / complexity of problem (e.g. number of unknown of an equation system)
  - Load balance (distribution of load on each core)
  - Hardware (latency/speed of network/memory access, cache coherence,...)



# Limit of parallelization - Scaling

- *Strong scaling*
  - Problem size constant
  - Speedup  $S(N_{cores})$  in time
  - Amdahl's law:

$$S(N_{cores}) = \frac{1}{(1-P)+C+P/N_{cores}}$$



# Limit of parallelization - Scaling

- *Strong scaling*

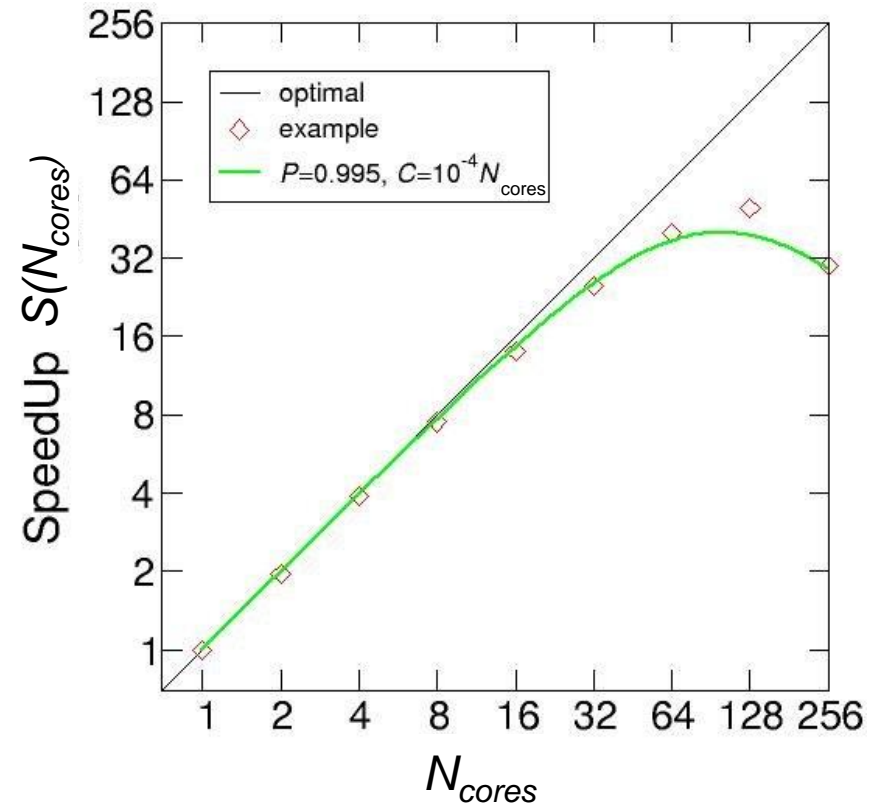
- Problem size constant
- Speedup  $S(N_{cores})$  in time
- Amdahl's law:

$$S(N_{cores}) = \frac{1}{(1-P) + C + P/N_{cores}}$$

- *Weak scaling*

- Constant time frame
- Speedup  $S(N_{cores})$  by increasing problem size/  
number of solved problems
- Gustavson's law:

$$S(N_{cores}) = (1 - P) + N_{cores} \cdot P$$



# PARALLEL PROGRAMMING MODELS

# Parallel programming models

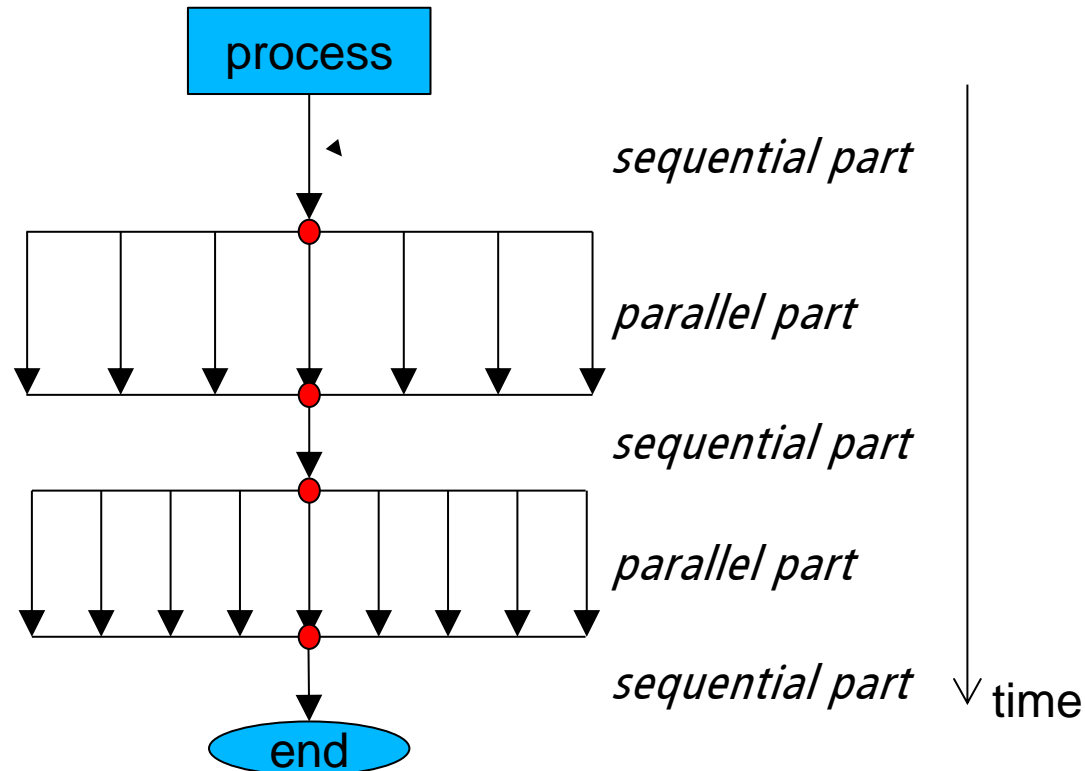
- Distributed memory
  - MPI (Message Passing Interface)
  - PVM (Parallel Virtual Machine)
- Distributed shared memory
  - PGAS (Partitioned Global Address Space)
- Shared memory
  - PThread (POSIX Threads)
  - OpenMP (Open Multi-Processing)
- Accelerator devices
  - Nvidia's CUDA (Compute Unified Device Architecture)
  - OpenCL (Open Computing Language)
  - OpenACC, OpenMP 4.x (new)

→ *most common: MPI, PThread and OpenMP*



# Shared memory models - PThreads and OpenMP

- One process with multiple threads



- Use same memory segment
  - variables are accessible from every thread
  - can lead to race-conditions (need synchronization)

# Shared memory models - PThreads and OpenMP

- OpenMP
  - C/C++:  
Pragma compiler statements
  - Fortran:  
Directives within comments
  - Minimal change of code
- PThreads
  - Library interface
  - Mutexes to avoid data collisions
  - Code has to be changed
- Note
  - Thread creation can be time consuming
  - Only usable on shared memory architectures!

# OPENMP

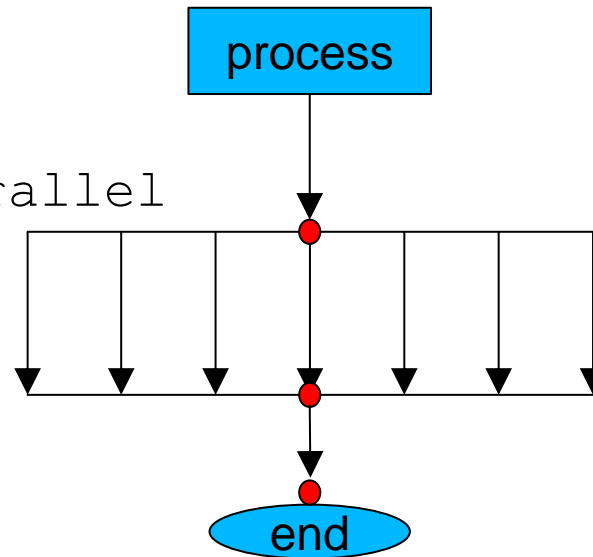
# OpenMP code structure

C/C++

Fortran

```
#pragma omp parallel  
{  
  code block  
}
```

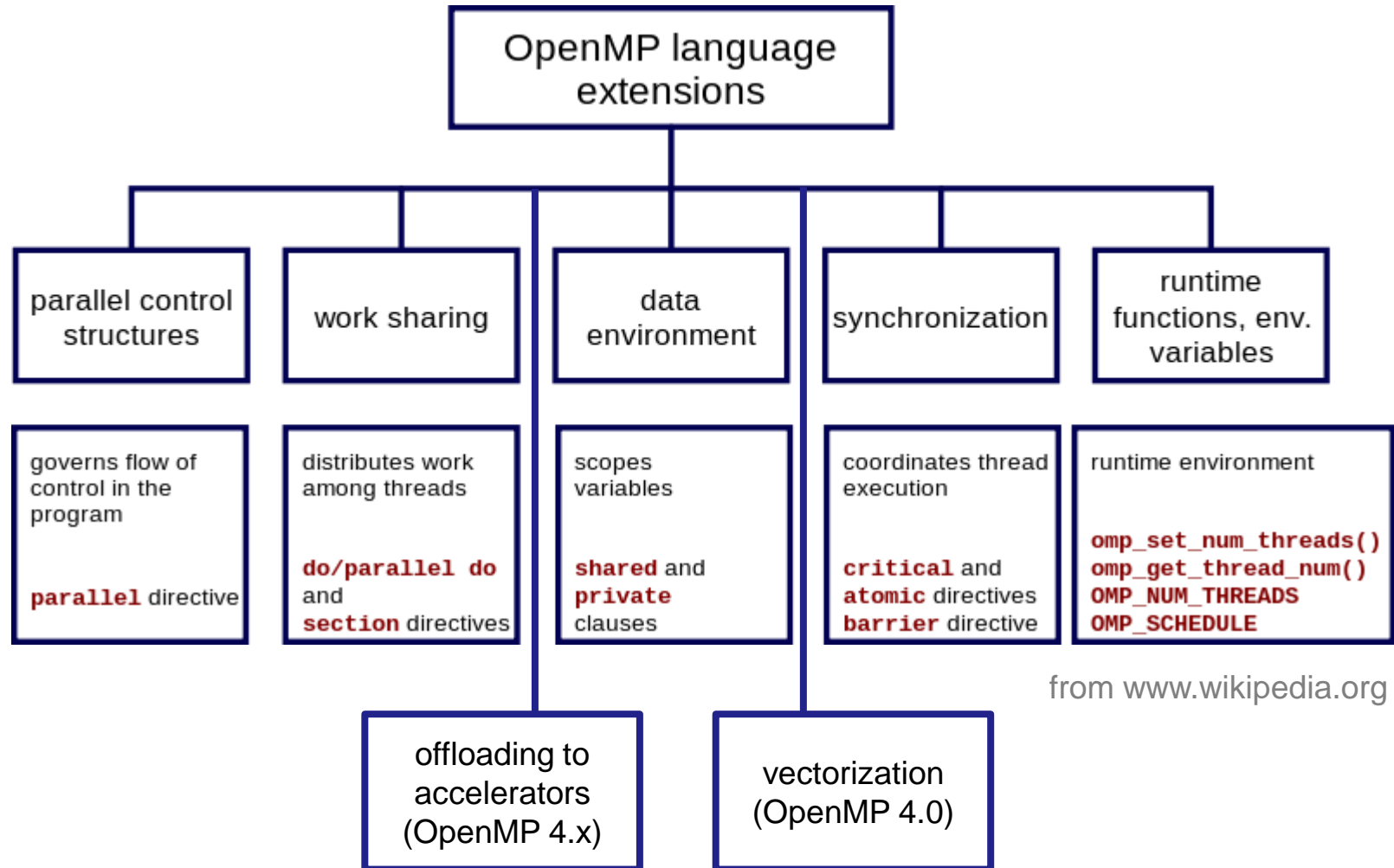
```
!$OMP PARALLEL  
  
  code block  
  
!$OMP END PARALLEL
```



## Extension

```
#pragma omp [directive] [clause] [clause] ...  
!$OMP PARALLEL [directive] [clause] ...  
C$OMP PARALLEL [directive] [clause] ...
```

# OpenMP overview



from [www.wikipedia.org](http://www.wikipedia.org)

# OpenMP runtime library

- additional routines by
  - C/C++: `#include <omp.h>`
  - Fortran: `use omp_lib` or `!$ INCLUDE 'omp_lib.h'`
- features
  - setting and querying the number of threads  
(`omp_set_num_threads()` and `omp_get_num_threads()`)
  - querying the thread ID (`omp_get_thread_num()`)
  - querying if in a parallel region (`omp_in_parallel()`)
  - wall clock timers (`omp_get_wtime()`, `omp_get_wtick()`)
  - ...
- environment variables enable control of runtime
  - setting the number of threads , e.g.  
`export OMP_NUM_THREADS=12`
  - setting the maximal number of threads, e.g.  
`export OMP_THREAD_LIMIT=24`

# OpenMP “hello world”

C:

```
#include <omp.h>

int main(int argc, char *argv[])
{
#pragma omp parallel
{
    printf(„Hello World! (%d)\n“, omp_get_thread_num());
}
return 0
}
```

Fortran:

```
PROGRAM HELLOWORLD
!$ INCLUDE 'omp_lib.h'
!$OMP PARALLEL
    PRINT *, 'Hello World!', OMP_GET_THREAD_NUM()
!$OMP END PARALLEL
END
```

# OpenMP “hello world”

```
flow01> make
gcc -fopenmp      -c helloWorld.c
gcc -fopenmp     helloWorld.o  -o helloWorld
flow01> ./helloWorld
Hello World! (1)
Hello World! (2)
Hello World! (3)
Hello World! (0)
flow01> ./helloWorld
Hello World! (0)
Hello World! (3)
Hello World! (1)
Hello World! (2)
```

→ Threads run in arbitrary order



# OpenMP “hello world” 2

## What happens here?

```
int main(int argc, char *argv[])
{
    int threadID, nThreads;

    #pragma omp parallel
    {
        threadID = omp_get_thread_num();
        printf("Hello World (%d)!\n", threadID);
        #pragma omp barrier
        if ( threadID == 0 ) {
            nThreads = omp_get_num_threads();
            printf("Using %d threads\n",nThreads);
        }
    }

    return 0;
}
```

# OpenMP “hello world” 2

```
flow01> ./helloWorld
```

```
Hello World (0)!
```

```
Hello World (3)!
```

```
Hello World (1)!
```

```
Hello World (2)!
```

```
flow01> ./helloWorld
```

```
Hello World (1)!
```

```
Hello World (2)!
```

```
Hello World (3)!
```

```
Hello World (0)!
```

```
Using 4 threads
```

```
Using 4 threads
```

```
Using 4 threads
```

```
Using 4 threads
```

# OpenMP “hello world” 2

## Problem: Race condition

```
int main(int argc, char *argv[])
{
    int threadID, nThreads;

    #pragma omp parallel
    {
        threadID = omp_get_thread_num();
        printf("Hello World (%d)!\n", threadID);
        #pragma omp barrier
        if ( threadID == 0 ) {
            nThreads = omp_get_num_threads();
            printf("Using %d threads\n",nThreads);
        }
    }

    return 0;
}
```

→ variable `threadID` is shared with all threads

# OpenMP “hello world” 2

## Correct way:

```
int main(int argc, char *argv[])
{
    int threadID, nThreads;

    #pragma omp parallel private(threadID)
    {
        threadID = omp_get_thread_num();
        printf("Hello World (%d)!\n", threadID);
        #pragma omp barrier
        if ( threadID == 0 ) {
            nThreads = omp_get_num_threads();
            printf("Using %d threads\n",nThreads);
        }
    }

    return 0;
}
```

→ **variable** threadID is now private for each thread

# OpenMP “hello world” 2

```
flow01> ./helloWorld
```

```
Hello World (0)!
```

```
Hello World (3)!
```

```
Hello World (1)!
```

```
Using 4 threads
```

```
Hello World (2)!
```

```
flow01> ./helloWorld
```

```
Hello World (1)!
```

```
Hello World (2)!
```

```
Hello World (3)!
```

```
Hello World (0)!
```

```
Using 4 threads
```

# Clauses for parallel regions

- `private(variable list)`
  - definition of private variables in parallel region
  - variables are not initialized
  - variables have same name but different values in different threads
- `shared(variable list)`
  - define variables which are shared over threads
- `default(type)`
  - default for variables (shared or private)
- `reduction(operator:list)`
  - reduce values in a safe way after joining threads
- `firstprivate(variable list)`
  - private variable will be initialized by the initial value
- `threadprivate(variable list)`
  - variables are replicated, each thread has its own copy

# WORK SHARING DIRECTIVES

# Work sharing directives

- usable in parallel regions
- directives specify how to work in threads
- no synchronization (barrier) at entry
- directives
  - `for` or `do` : (Fortran/C) split loops into parallel tasks
  - `sections / section` : definition of tasks for one thread
  - `single` : code block will be executed only in one thread (synchronization at the end)
  - `master` : code block will be executed only in master thread (no synchronization at the end)
  - `workshare` : Fortran directive for statements like `FORALL`, `WHERE` and array/scalar assignment
  - `task` : definition of a specific task



# WORK SHARING DIRECTIVES FOR/DO

# For/Do directive

```
int main(int argc, char *argv[])
{
    int i;
    const N=1000000;
    double x[N];

    #pragma omp parallel for
    for(i=0; i<N; i++)
    {
        x[i] = 1./(i+1.);
    }

    return 0;
}
```

- loop executed in parallel
- per default the range  $[0, N[$  is divided in  $N_{threads}$  parts, e.g.  $N_{threads}=2$ :  
$$i_{thread\ 0}=0, \dots, N/2-1, \quad i_{thread\ 1}=N/2, \dots, N-1$$

# Clauses

- `lastprivate(variable list)`
  - variables get the last value of parallelized loop
- `collapse(n)`
  - collapse `n` nested loops to one loop (to get a better load balance)
- `ordered`
  - loop will run in the same order as in a serial code
- `schedule(type, [chunk size])`
  - specify how the loop will split up (optionally the chunk size)
    - `static` – fixed chunks, like in demo before
    - `dynamic` – waiting threads will get the next task
    - `guided` – decrease the chunk size to get a better load balance
    - `auto` – compiler decide best scheduling
- `nowait`
  - no implicit barrier at the end

# Scheduling

## Example on 3 threads

```
#pragma omp for
for(i=0; i<N; i++)
{
    ....
}
```

## Distribution of $i$ on thread IDs 0,1,2

$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
static	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	2
dynamic	0	1	2	0	2	1	0	2	1	2	0	1	2	0	1	2
guided	0	0	0	1	1	1	2	2	2	0	0	1	1	2	2	0
auto	<i>decided at runtime or from compiler</i>															

# For/Do directive – 2<sup>nd</sup> example

```
int main(int argc, char *argv[])
{
    int i;
    const int N=1000000;
    double x[N];
    double nrm = 0.;

    #pragma omp parallel reduction(+:nrm)
    {
        /* init variables */
        #pragma omp for
        for(i=0; i<N; i++)
        {
            x[i] = 1./(i+1.);
        }

        /* calculate the norm */
        #pragma omp for
        for(i=0; i<N; i++)
        {
            nrm += x[i]*x[i];
        }
    }
    printf("Nrm = %g\n", sqrt(nrm));
    ...
}
```

# Synchronization clauses

- `critical`
  - the code block will be executed by only one thread per time
- `atomic`
  - memory updates within the block will be made atomically (if hardware supports atomic operations)
- `ordered`
  - structured block (e.g. loops) will run in the same order as in a serial code
- `barrier`
  - each thread waits until all threads of a team reach this point
- `nowait`
  - no implicit barrier at the end of a work share

# For/Do directive – 2<sup>nd</sup> example alternative

```
...
double nrm = 0.;

#pragma omp parallel shared(nrm)
{
    double private_nrm = 0.;
    /* init variables */
    #pragma omp for
    for(i=0; i<N; i++)
    {
        x[i] = 1./(i+1.);
    }

    /* calculate the norm */
    #pragma omp for
    for(i=0; i<N; i++)
    {
        private_nrm += x[i]*x[i];
    }
    #pragma omp critical
    nrm += private_nrm;
}
printf("Nrm = %g\n", sqrt(nrm));
...
```

# WORK SHARING DIRECTIVES SECTION



# Sections directive

sections / section : definition of tasks for one thread

```
...
#pragma omp parallel sections private(i)
{
    {
        printf("Task %d has to do init x\n", omp_get_thread_num());
        for(i=0; i<N; i++)
            x[i] = 0.;
    }
    #pragma omp section
    {
        printf("Task %d has to do init y\n", omp_get_thread_num());
        for(i=0; i<N; i++)
            y[i] = 0.;
    }
    #pragma omp section
    {
        printf("Task %d has to do init z\n", omp_get_thread_num());
        for(i=0; i<N; i++)
            z[i] = 0.;
    }
}
```

# OpenMP “hello world”

Output:

```
flow01> ./section
Task 7 has to do init x
Task 3 has to do init y
Task 1 has to do init z
flow01> export OMP_NUM_THREADS=2
flow02> ./section
Task 1 has to do init y
Task 0 has to do init x
Task 0 has to do init z
```

→ each section will assigned to a thread

# FEATURES OF OPENMP 4.0

# SIMD directive

## example

```
...  
int i;  
double x = 0, A[n];  
...  
for (i=0; i<n; i++)  
{  
    x += A[i];  
}
```

`simd` can help the compiler to use SIMD instructions for vectorization

```
...  
int i;  
double x = 0, A[n];  
...  
#pragma omp simd reduction(+:x)  
for (i=0; i<n; i++)  
{  
    x += A[i];  
}
```

# COMPILING

# OpenMP Compilation

To use/activate OpenMP extensions you need additional compiler flags:

- GNU compiler: `-fopenmp`
- Intel compiler: `-openmp`
- PGI compiler: `-mp`
- Open64 compiler: `-mp` **or** `-openmp`

# PITFALLS

# Implied flush

- **consistent view on data in memory**
  - data from cache will be flushed implicitly only at some points:
    - end of `do/for/sections/single/workshare`
    - at begin and end of `parallel/critical/ordered` blocks
    - `barrier`
  - on other points the shared variables may have temporarily different values
  - if you need a consistent view use `flush(varlist)` directive
  - `nowait` directives



# Race conditions, Dead locks and I/O

- **race conditions**
  - two or more threads access the same variable and at least one thread modify the variable and no synchronization is applied
  - lead to unpredictable results
  - use *hellgrind* or *valgrind* or the *Intel Inspector* for detecting race conditions
- **deadlocks**
  - two or more threads blocked and wait on each other  
→ program run (wait) forever
- **I/O**
  - I/O within parallel regions from multiple threads can be unpredictable

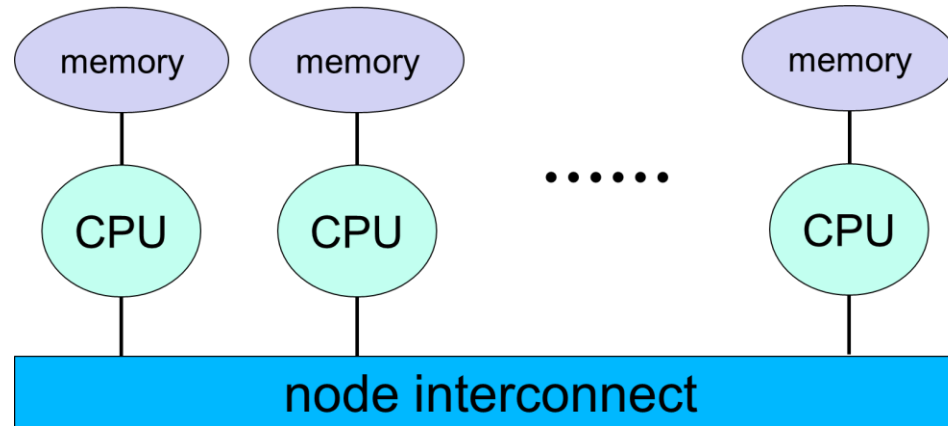
# Overhead

## Overhead

- synchronization/barriers/serial regions  
→ avoid unnecessary synchronization  
(e.g. barriers, critical sections)
- fork and join of parallel regions  
→ try to build large parallel blocks
- so called *false-sharing*
  - two threads modify same memory block which is loaded in 2 different caches due to cache lines
  - e.g. by modification of an vector with dynamic scheduling

# Optimization

- for multi-socket systems think about ccNUMA



→ variables attached to the socket where the first access happens (array allocated typically in 4Kb blocks, blocks can be assigned to memory at different sockets)

- pin threads to cores/CPU, e.g. by `numactl`

# Summary

- usable by standardized compiler directives
- using threads within parallel regions
- simplify step by step parallelization
- can increase performance significantly
- disadvantages
  - limited to shared memory machines
  - race-conditions sometimes not easy to detect and to understand

Thanks a lot for your attention!

For further information please visit the HPC Wiki

<http://wiki.hpcuser.uni-oldenburg.de>

# Exercises

- Login to FLOW/HERO

```
ssh -XY abcd1234@flow.hpc.uni-oldenburg.de
```

```
ssh -XY abcd1234@hero.hpc.uni-oldenburg.de
```

- Try to parallelize optimized sample code from yesterday
- Write submission script to test the program
- How does the code scale ( $t_{wall\ clock}$  vs.  $N_{threads}$ )?
  - for vector size of  $N=10000$
  - for vector size of  $N=30000$
  - for different compilers (GNU, Intel,...)