Introduction to High Performance Computing

OpenMP

Literature

Many good books:

- S. Hoffmann, R. Lienhart: OpenMP, Springer 2008. (German, easy to read)
- B. Chapman et al., Using OpenMP, MIT Press 2008 (excellent)
- Specification: Internet

Programming Model

- Shared Memory programming model
- Based on directives for C/C++/Fortran
- Concept of parallel threads
- Single-program multiple data (SPMD) model

A **Process** is a running program with ist own ressource and process identification.

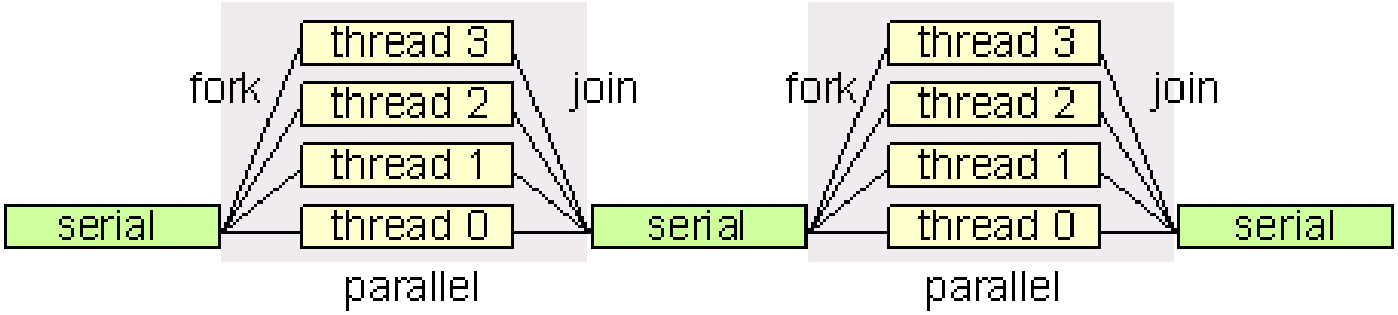
Definition: A **Task** is an independent process.

Definition:
A **Thread** ist able to independently execute a stream of instructions but it is a part of a Task. Threads form tasks and share runtime ressources of a task, like data segments and file descriptors.

Specification: http://www.openmp.org

Program Execution

Fork-join execution model:



A serial task may fork several times into a parallel region and the parallel threads may join again into a single task.

- Master Thread
- Forking into a team of threads
- Based on POSIX standard
- So called lightweight threads

Simple Example

```
#ifdef _OPENMP
#include <omp.h>
#endif
main(int argc,char **argv) {
 int inode=1, nnode=1;
#ifdef _OPENMP
 nnode=omp_get_num_procs();
 omp_set_num_threads(nnode);
#endif
 printf("The machine has %d CPUs! We use all of them\n",
        nnode);
#pragma omp parallel private(inode)
{
#ifdef _OPENMP
   inode=omp_get_thread_num();
#endif
   printf("hallo world from %d\n",inode);
 }}
```

Set with compile option – allows portability

Header – available with Compiler

Runtime functions – get number of available cores and sets number of used cores

Starts a parallel region with its memory management

All standard functions are thread safe. No ordering.

thread safeness

Functions have to be thread safe.
Results must be independent of
- the number of threads used
- the order of thread execution

Carefulness with
- global variables (best praxis: read only)
- shared pointers (best praxis: read only or work on different locations)

All standard functions in the **math library**, …. are **thread safe**.

thread safeness

```
#include <stdio.h>
#include <omp.h>

int flag;
```

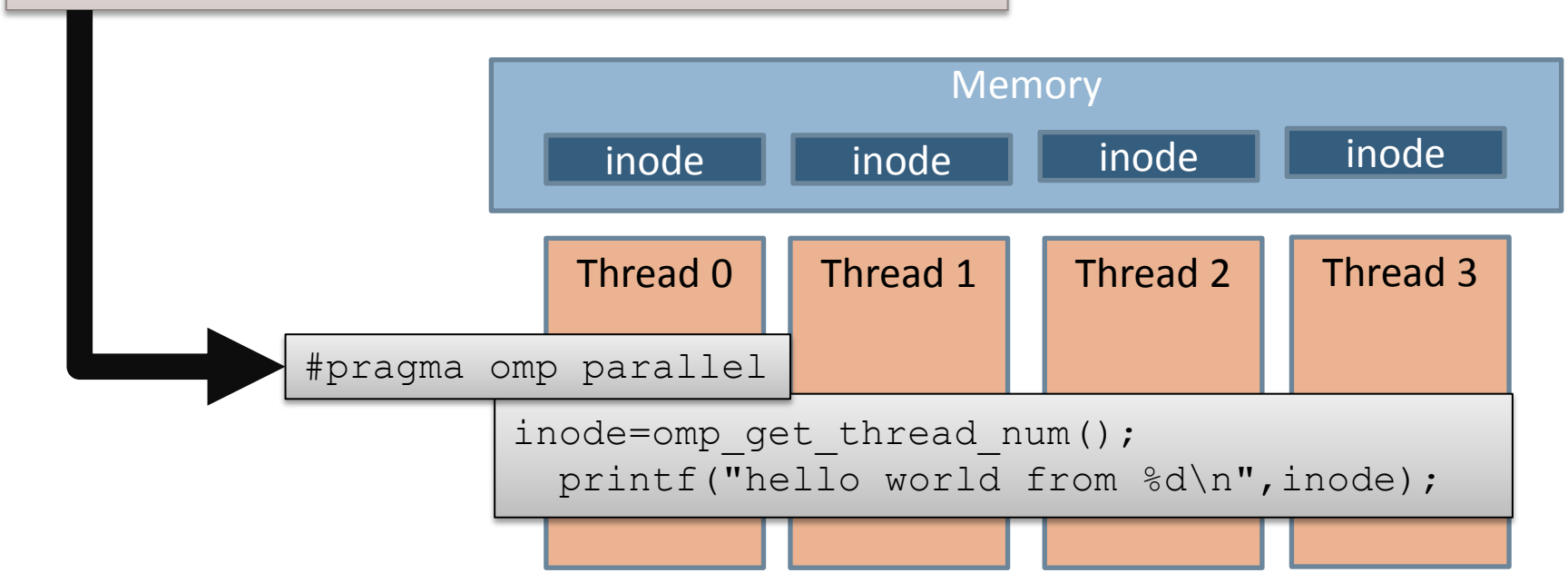
```
f1(int *a, int b) {
me=omp_get_thread_num();
a[me]=b;
}
```

```
f2(int *b,int c) {
if(flag==0) *b=c;
flag++;
}
```

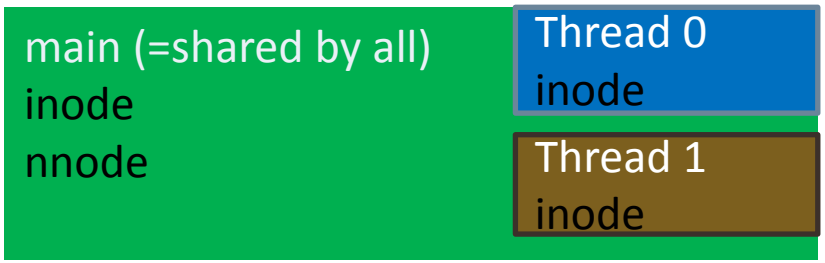
```
main()
 int a[32],b=4,c=57;
 flag=0;
#pragma omp parallel
{
 f1(a,b);
 f2(&b,c); % only first proc should set b=c
 printf("%d\n",b);
 print_array(a);  % array should read 0 1 2 3 4 . ….
}
```

Memory management

```
#pragma omp parallel private(inode)
{
  inode=omp_get_thread_num();
  printf("hello world from %d\n",inode);
}
```



Memory management



Multiple versions of variable **inode**:

- one version for the task
- own versions for each thread

No **copy**(!)

inode (thread 0) knows nothing about value of inode in task

Memory management

Address space of master thread = address space of task

- all data allocated in this address space is available to all threads
- life span of data = life span of task

Each thread owns its own private stack

- all automatic generated variables are on this private stack
- life span of data = life span of threads (typically)
- some data may be set to outlive a simple life cycle of its thread

Types of memory locations

```
int outside=4;
function1 (int, *int);
function2 (int, *int);
main(){
 int inside=1;
 function1(inside,&inside);
 function2(inside,&inside);
}
function1(int i, int *j) {
 *j=outside;
 outside=i;
}
```

- What memory types?
- What life span have these types?
- What values do **outside** and **inside** have?

```
FILE function2.c
function2(int i, int *j) {
 extern int outside;
 *j=outside;
 outside=i;
}
```

Types of memory locations

```
int outside=4;
function1 (int, *int);
function2 (int, *int);
main(){
 int inside=1;
#pragma omp parallel
{
 function1(inside,&inside);
 function2(inside,&inside);
}
}
function1(int i, int *j) {
 *j=outside;
 outside=i;
}
```

What values do **outside** and **inside** have?

```
FILE function2.c
function2(int i, int *j) {
 extern int outside;
 *j=outside;
 outside=i;
}
```

Memory management

```
…
#pragma omp parallel private(inode)shared(problem)
{
#ifdef _OPENMP
   inode=omp_get_thread_num();
#endif
 if(inode > 0) problem=array[inode];
 else problem=0;
 func3(&problem);
}
```

What value for variable `problem`?

Memory management

Two different data-sharing attributes:

- shared (this is the default for all variables)
- private (threads have their own copy)

Writing to a shared variable **without synchronization**

- not atomic
- race condition

Reading a shared variable (written by another thread) without memory fence

- not atomic
- race condition
- **OPENMP violates cache coherency**

Cache coherency

Only one valid copy of a memory location does exist – irrespective of its location in main memory or one of the caches.

OpenMP **flush** operation enforces consistency.

Memory management

Changing the life span of data

```
#pragma omp threadprivate(list)
```

- applies to global variables only
- changes life span of thread data to life span of task
- data is private
- not consistent with dynamic thread management
- data is initialized (if coded)
- changes to global data of master thread do not apply

Clauses to other directives specify

- copy of data (when entering: **copyin, firstprivate**)
- copy of data (leaving: **lastprivate**)
- broadcast of data (**copyprivate**)

parallel construct

```
#pragma omp parallel [clause …]
```

Possible clauses:

copyin(list)	global variables are initialized (threadprivate)
firstprivate(list)	initialization
if(scalar_expression)	will run in parallel if true
num_threads(int)	number of parallel threads
private(list)	variables are treated as private
reduction(op:list)	declares reduction variable and operation
shared(list)	variables are treated as shared

parallel construct

Number of threads created:

1. if clause
2. num_threads clause in parallel construct
3. omp_set_num_threads library function
4. environment variable OMP_NUM_THREADS

export OMP_NUM_THREADS=4

greatest flexibility:

```
$ OMP_NUM_THREADS=4 ./omp_job
$ OMP_NUM_THREADS=6 ./omp_job
```

work sharing

parallel construct forks into a team of thread – no work distribution !

```
#pragma omp for [clause …]
```

Possible clauses:

firstprivate(list)	initialization
lastprivate(list)	value of last iteration is copied to master thread
nowait	no barrier at end
ordered	together with ordered directives to generate a sequence
private(list)	variables are treated as private
reduction(op:list)	declares reduction variable and operation
schedule(kind[,size])	various options to guide subdivision of work

work sharing

#pragma omp for nowait

Per default all threads finish the for-loop at the same time (synchronization).
This synchronization is removed with the **nowait** clause.

```
#pragma omp for nowait
   for(i=0;i<n;i++) a[i]*=s;
#pragma omp for
  for(i=0;i<m;i++) b[i]+=x[i];
```

Loops are independent of each other. Second loop may start while first one is not finished completely.

As the number of thread increases – synchronization gets more and more expensive.

work sharing

#pragma omp for reduction(operator:list)

```
#pragma omp for reduction(+:s)
  for(i=0,s=0.;i<n;i++) s+=x[i];
```

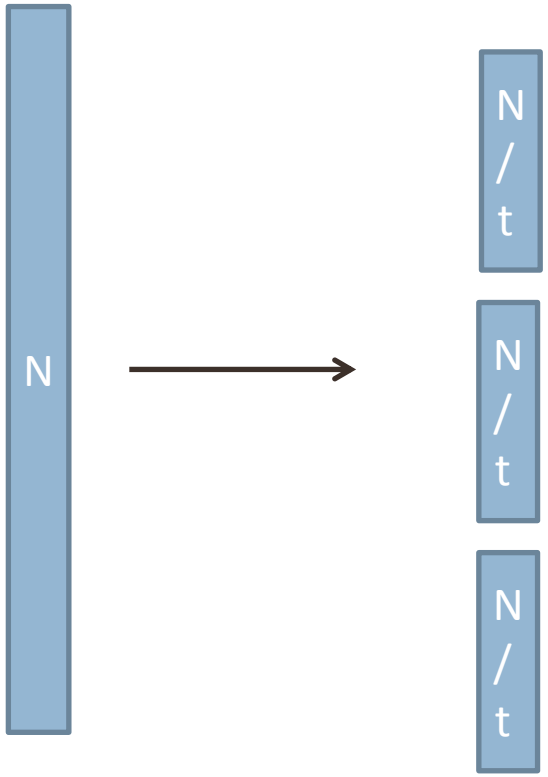
OpenMP organizes

- formation of partial sums into private variables `s_privat`
- addition (reduction) of these partial sums into a single (shared) variable s

work sharing

```
#pragma omp for schedule(static)
```

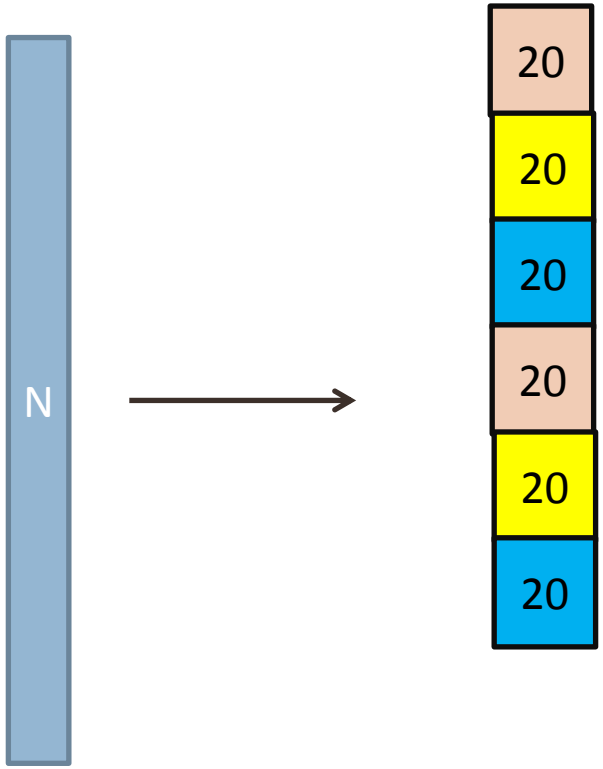
default behaviour:



loop is subdivided into almost equal blocks

work sharing

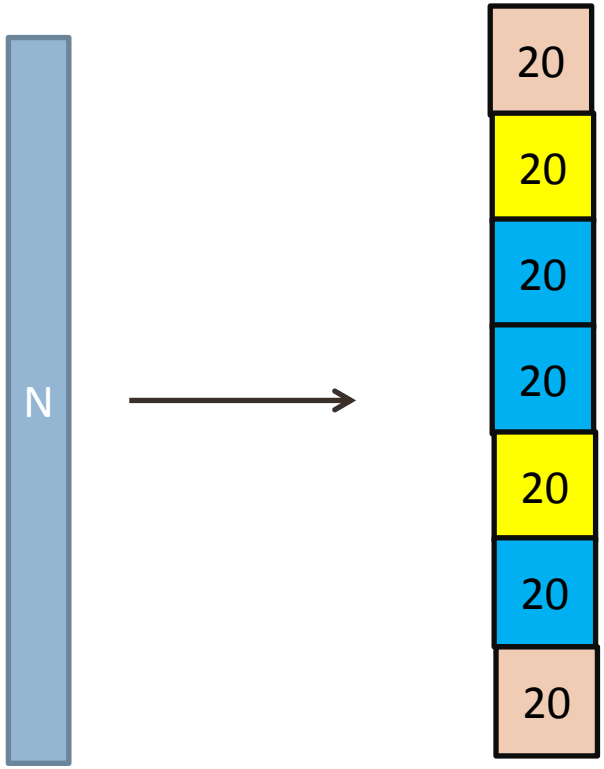
```
#pragma omp for schedule(static,20)
```



- loop is subdivided in chunks of size 20
- threads work on chunks ordered

work sharing

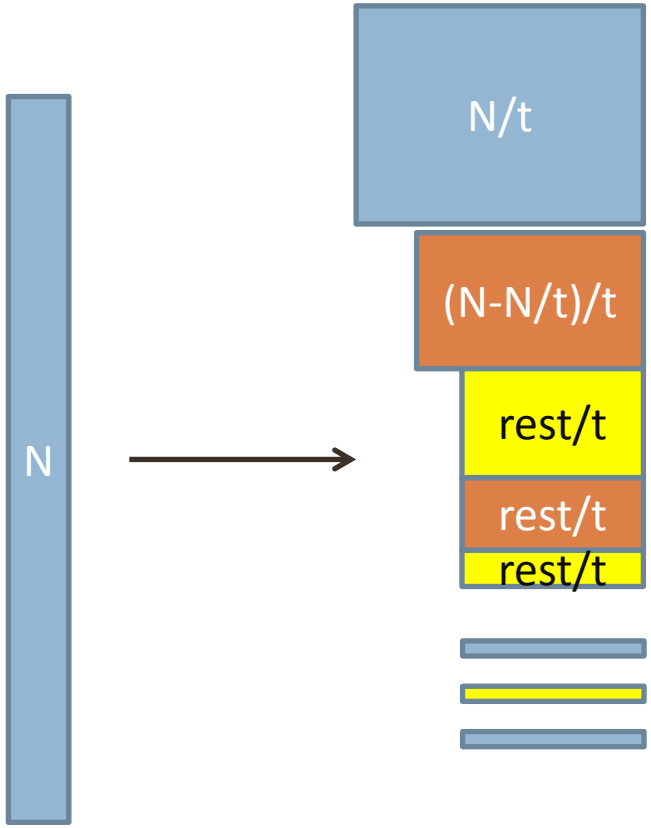
```
#pragma omp for schedule(dynamic,20)
```



- loop is subdivided in chunks of size 20
- threads work on chunks as they finish their work
- default chunk size is 1

work sharing

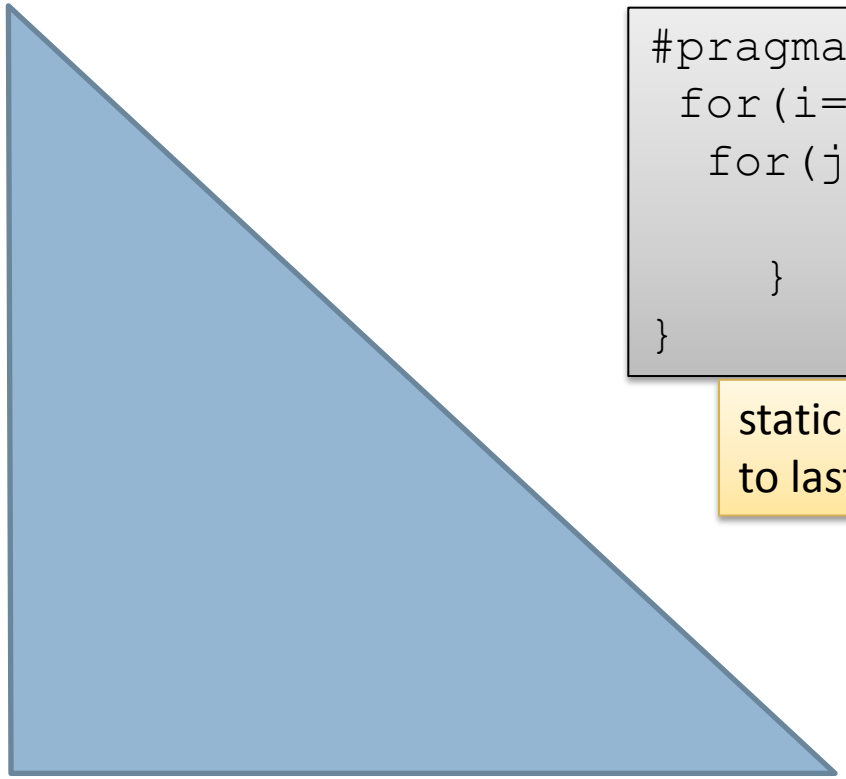
```
#pragma omp for schedule(guided[,1])
```



The rest of the work still to be done is divided by the number of threads. Thus the pieces become smaller and smaller.

work sharing

Why do we need different scheduling strategies?



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

static schedule leaves largest pieces to last thread.

Einführung in das Hochleistungsrechnen

Introduction to High Performance Computing

VIELEN DANK
THANK YOU