Parallel Processing

WS 2017/18

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 15, 2018

Parallel Processing

WS 2017/18

2 Parallel Programming with Shared Memory

Inhalt

- Basics
- Posix threads
- OpenMP basics
- Loop parallelization and dependeces
- OpenMP synchronization
- OpenMP details

Literature

- Wilkinson/Allen, Ch. 8.4, 8.5, Appendix C
- Hoffmann/Lienhart

Approaches to programming with threads

- Using (system) libraries
  - Examples: POSIX threads (☞ **2.2**), Intel Threading Building Blocks (TBB), C++ boost threads
- As part of a programming language
  - Examples: Java threads (☞ **BS_I**), C++11
- Using compiler directives (pragmas)
  - Examples: OpenMP (☞ **2.3**)

Notes for slide 147:

To support thread programming C++11 provides, among others, a thread class, mutexes and condition variables, and *futures*.

147-1

2.1 Basics

2.1.1 Synchronization

- Ensuring conditions on the possible sequences of events in threads
  - mutual exclusion
  - temporal order of actions in different threads
- Tools:
  - shared variables
  - semaphores / mutexes
  - monitors / condition variables
  - barriers

Parallel Processing

WS 2017/18

30.10.2017

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 15, 2018

2.1.1 Synchronization ...

Synchronization using shared variables

- Example: waiting for a result

Thread 1

```
// compute and
// store result
ready = true;
...
```

Thread 2

```
while (!ready); // wait
// read / process the result
...
```

- Extension: atomic *read-modify-write* operations of the CPU
  - e.g., *test-and-set, fetch-and-add*
- Potential drawback: *busy waiting*
  - but: in high performance computing we often have exactly one thread per CPU ⇒ performance advantage, since no system call

2.1.1 Synchronization ...

Semaphores

- Components: counter, queue of blocked threds
- **atomic** operations:
  - `P()` (also `acquire`, `wait` or `down`)
    - decrements the counter by 1
    - if counter < 0: block the thread
  - `V()` (also `release`, `signal` or `up`)
    - increments counter by 1
    - if counter ≤ 0: wake up one blocked thread
- **Binary semaphore**: can only assume the values 0 and 1
  - usually for mutual exclusion

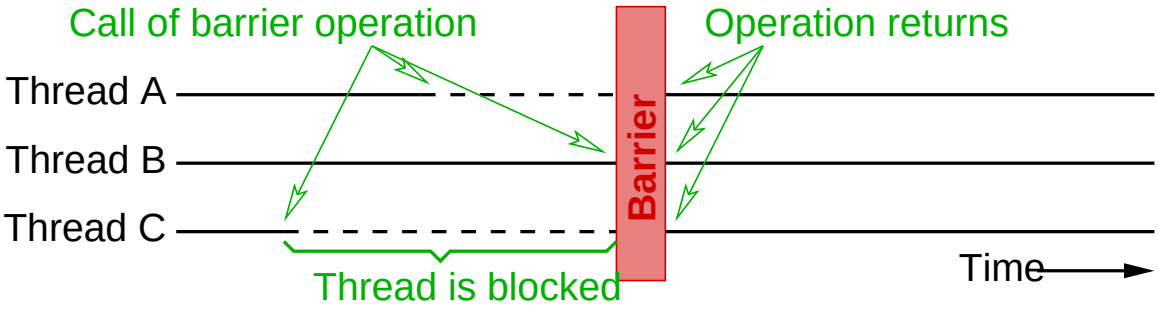
2.1.1 Synchronization ...

Monitors

- Module with data, procedures and initialization code
  - access to data only via the monitor procedures
  - (roughly corresponds to a class)
- All procedures are under mutual exclusion
- Further synchronization via **condition variables**
  - two operations:
    - `wait()`: blocks the calling thread
    - `signal()`: wakes up some blocked threads
      - variants: wake up only one thread / wake up all thread
  - no “memory”: `signal()` only wakes a thread, if it already has called `wait()` before

Barrier

- Synchronization of groups of processes or threads, respectively
- Semantics:
  - thread which reaches the barrier is blocked, until all other threads have reached the barrier, too



- Used to structure concurrent applications into synchronous phases

2.1.2 Synchronization errors

- Insufficiend synchronization: *race conditions*
  - result of the calculation is different (or false), depending on temporal interleaving of the threads
  - important: do not assume FIFO semantics of the queues in synchronization constructs!
- Deadlocks
  - a group of threads waits for conditions, which can only be fulfilled by the other threads in this group
- Starvation (unfairness)
  - a thread waiting for a condition can never execute, although the condition is fulfilled regularly

Example for *race conditions*

- Task: synchronize two threads, such that they print something alternatingly
- Wrong solution with semaphores:

```
Semaphore s1 = 1;
Semaphore s2 = 0;
```

Thread 1

```
while (true) {
   P(s1);
   print("1");
   V(s2);
   V(s1);
}
```

Thread 2

```
while (true) {
   P(s2);
   P(s1);
   print("2");
   V(s1);
}
```


2.1.3 *Lock-Free* and *Wait-Free* Data Structures

- Goal: Data structures (typically *collections*) without mutual exclusion
  - more performant, no danger of deadlocks
- *Lock-free*: under **any circumstances** at least one of the threads makes progress after a finite number of steps
  - in addition, *wait-free* also prevents starvation
- Typical approach:
  - use atomic *read-modify-write* instrcutions instead of locks
  - in case of conflict, i.e., when there is a simultaneous change by another thread, the affected operation is repeated

Example: appending to an array (at the end)

```
int fetch_and_add(int* addr, int val) {
   int tmp = *addr;
   *addr += val;
   return tmp;
}
```

Atomic!

```
Data buffer[N];    // Buffer array
int wrPos = 0;     // Position of next element to be inserted

void add_last(Data data) {
   int wrPosOld = fetch_and_add(&wrPos, 1);
   buffer[wrPosOld] = data;
}
```


Example: prepend to a linked list (at the beginning)

```
bool compare_and_swap(void* addr, void* exp, void* new) {
   if (*addr == *exp) {
      *addr = *new;
      return true;
   }
   return false;
}
```

Atomic!

```
Element* firstNode = NULL;        // Pointer to first element
void add_first(Element* node) {
   Element* tmp;
   do {
     tmp = firstNode;
     node->next = tmp;
   } while (!compare_and_swap(firstNode, tmp, node));
}
```

2.1.3 *Lock-Free* and *Wait-Free* Data Structures ...

- Problems
  - re-use of memory addresses can result in corrupt data structures
    - assumption in linked list: if `firstNode` is still unchanged, the list was not accessed concurrently
  - thus, we need special procedures for memory deallocation
- There is a number of libraries for C++ and also for Java
  - C++: e.g., boost.lockfree, libcds, Concurrency Kit, liblfds
  - Java: e.g., Amino Concurrent Building Blocks, Highly Scalable Java
- Compilers usually offer *read-modify-write* operations
  - e.g., `gcc/g++`: built-in functions `__sync_...()`

2.2 POSIX Threads (PThreads)

- IEEE 1003.1c: standard interface for programming with threads
  - implemented by a system library
  - (mostly) independent of operating system
- Programming model:
  - at program start: exactly one (master) thread
  - master thread creates other threads
    and should usualy wait for them to finish
  - process terminates when master thread terminates

Functions for thread management (incomplete)

- `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`
  - creates a new thread
  - input parameters:
    - `attr`: thread attributes
    - `start_routine`: function that should be executed by the thread
    - `arg`: parameter, which is passed to `start_routine`
  - Result:
    - `*thread`: thread *handle* (= reference)

Notes for slide 160:

In the simplest case, the parameter `attr` is set to `NULL`. It allows to set attributes for the newly created thread, e.g., the scheduling method to be used for the thread (e.g., round-robin or non-preemptive) and the priority of the thread. Most of these attributes are platform specific! See the PThreads documentation for the respective platform.

Functions for thread management ...

- `void pthread_exit(void *retval)`
  - calling thread will exit (with return value `retval`)
- `int pthread_cancel(pthread_t thread)`
  - terminates the specified thread
  - termination can only occur at specific places in the code
  - before termination, a *cleanup handler* is called, if defined
  - termination can be masked
- `int pthread_join(pthread_t th, void **thread_return)`
  - waits for the specified thread to terminate
  - returns the result value in `*thread_return`

Example: Hello world (☞ 02/helloThread.cpp)

```
#include <pthread.h>

void * SayHello(void *arg) {
  cout << "Hello World!\n";
  return NULL;
}
int main(int argc, char **argv) {
  pthread_t t;
  if (pthread_create(&t, NULL, SayHello, NULL) != 0) {
    /* Error! */
  }
  pthread_join(t, NULL);
  return 0;
}
```

Example: Summation of an array with multiple threads

(☞ 02/sum.cpp)

```
#include <pthread.h>

#define N 5
#define M 1000

/* This function is called by each thread */
void *sum_line(void *arg)
{
  int *line = (int *)arg;
  int i;
  long sum = 0;

  for (i=0; i<M; i++)
    sum += line[i];

  return (void *)sum;     /* return the sum, finished. */
}
```


```
/* Initialize the array */
void init_array(int array[N][M])
{
  ...
}

/* Main program */
int main(int argc, char **argv)
{
  int array[N][M];
  pthread_t threads[N];
  int i;
  void *result;
  long sum = 0;

  init_array(array);   /* initialize the array */
```

```
  /* Create a thread for each line, with a pointer to the line as parameter */
  /* Caution: error checks and thread attributes are missing! */
  for (i=0; i<N; i++) {
    pthread_create(&threads[i], NULL, sum_line, array[i]);
  }

  /* Wait for termination, sum up the results */
  for (i=0; i<N; i++) {
    pthread_join(threads[i], &result);
    sum += (long)result;
  }

  cout << "Sum: " << sum << "\n";
}
```

Compile and link the program

- `g++ -o sum sum.cpp -pthread`

Remarks on the example

- PThreads only allows to pass *one* argument
  - usually, a pointer to an argument data structure
    - pointer to a local variable is only allowed, if `pthread_join` is called within the same procedure!
- The return value of a thread is `void *`
  - ugly type conversion
  - with multiple return values: pointer to a data structure (allocated globally or dynamically)
- No synchronization is required
- `pthread_join` can only wait for a specific thread
  - inefficient, when threads have different execution times

Synchronization: mutex variables

- Behavior similar to binary semaphore
  - states: locked, unlocked; initial state: unlocked
- Declaration and initialization (global/static variable):
  `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Operations:
  - lock: `pthread_mutex_lock(&mutex)`
    - behavior for recursive locking is not specified
  - unlocked: `pthread_mutex_unlock(&mutex)`
    - when a thread terminates, its mutexes are **not** automatically unlocked!
  - try locking: `pthread_mutex_trylock(&mutex)`
    - does not block; returns an error code, if applicable

Notes for slide 167:

The general method for declaring and initializing a mutex is shown below. It also works when the mutex is stored in a local variable:

```
pthread_mutex_t mutex;                /* Declaration */
pthread_mutex_init(&mutex, NULL);     /* Initialization */
/* use the mutex ... */
pthread_mutex_destroy(&mutex);        /* De—allocation */
```

When initializing a mutex, additional attributes can be passed in the second argument. Among others, the behavior of the mutex with recursive locking can be specified:

- deadlock: thread waits forever, since the mutex is locked (by himself)
- error message
- lock counter: thread can lock the mutex again, and must unlock it again the same number of times

Many of these attributes are platform specific.

Synchronization: condition variables

- Declaration and initialization (global/static variable):
  `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
- Operations:
  - wait: `pthread_cond_wait(&cond, &mutex)`
    - thread is blocked, `mutex` will be unlocked temporarily
    - signaling thread keeps the `mutex`, i.e., the condition may no longer be fulfilled when the routine returns!
    - typical use:

```
while (!condition_met)
  pthread_cond_wait(&cond, &mutex);
```

  - signal:
    - to a single thread: `pthread_cond_signal(&cond)`
    - to all threads: `pthread_cond_broadcast(&cond)`

Notes for slide 168:

The general method for declaring and initializing a condition variable is shown below. It also works when the condition variable is stored in a local variable:

```
pthread_cond_t cond;                /* Declaration */
pthread_cond_init(&cond, NULL);     /* Initialization */
/* ... use the condition variable */
pthread_cond_destroy(&cond);        /* De—allocation */
```

As with the mutex, the second argument can pass additional attributes to the initialization routine.

Example: simulating a monitor with PThreads

(☞ 02/monitor.cpp)

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;
volatile int ready = 0;

volatile int result;

void StoreResult(int arg)
{
   pthread_mutex_lock(&mutex);
   result = arg;   /* store result */
   ready = 1;
   pthread_cond_broadcast(&cond);
   pthread_mutex_unlock(&mutex);
}
```


Notes for slide 169:

The keyword `volatile` at the beginning of the declarations of global variables indicates the compiler that the value of this variable is “transient”, i.e. can change at any time (by a write access in another thread). This forbids certain optimizations of the compiler (especially “caching” the value in a register).

The pthread objects (mutex, condition variable) do no require the `volatile` keyword (since it is already contained in the respective type declarations, if need be).

Example: simulating a monitor with PThreads ...

```
int ReadResult()
{
   int tmp;
   pthread_mutex_lock(&mutex);
   while (ready != 1)
      pthread_cond_wait(&cond, &mutex);
   tmp = result;   /* read result */
   pthread_mutex_unlock(&mutex);
   return tmp;
}
```

- `while` is important, since the waiting thread unlocks the `mutex`
  - another thread could destroy the condition again before the waiting thread regains the `mutex`
    (although this cannot happen in this concrete example!)

Notes for slide 170:

Note that the PThread standard allows `pthread_cond_wait` to return even in cases where the condition has **not** been signalled. Thus, you always must use a `while` loop!

The PThread standard also defines a number of additional functions, among others:

`pthread_self`	returns own thread ID
`pthread_once`	executes a (initialization) function exactly once
`pthread_kill`	sends a signal to another thread within the same process – from 'outside' only a signal to the **process** (i.e., **some** thread) is possible
`pthread_sigmask`	sets the signal mask – each thread has its own signal mask
`pthread_cond_timedwait`	like `...wait`, but with maximum waiting time

Background

- Thread libraries (for FORTRAN and C) are often too complex (and partially system dependent) for application programmers
  - wish: more abstract, portable constructs
- OpenMP is an inofficial standard
  - since 1997 by the OpenMP forum (`www.openmp.org`)
- API for parallel programming with shared memory using FORTRAN / C / C++
  - **source code directives**
  - library routines
  - environment variables
- Besides parallel processing with threads, OpenMP also supports SIMD extensions and external accelerators (since version 4.0)

Parallel Processing

WS 2017/18

06.11.2017

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 15, 2018

Parallelization using directives

- The programmer must specify
  - which code regions should be executed in parallel
  - where a synchronization is necessary
- This specification is done using **directives** (**pragmas**)
  - special control statements for the compiler
  - unknown directives are ignored by the compiler
- Thus, a program with OpenMP directives can be compiled
  - with an OpenMP compiler, resulting in a parallel program
  - with a standard compiler, resulting in a sequential program

Parallelization using directives ...

- Goal of parallelizing with OpenMP:
  - distribute the execution of sequential program code to several threads, without changing the code
  - identical source code for sequential and parallel version
- Three main classes of directives:
  - directives for creating threads (`parallel`, parallel region)
  - within a parallel region: directives to distribute the work to the individual threads
    - data parallelism: distribution of loop iterations (`for`)
    - task parallelism: parallel code regions (`sections`) and explicit tasks (`task`)
  - directives for synchronization

Parallelization using directives: discussion

- Compromise between
  - completely manual parallelization (as, e.g., with MPI)
  - automatic parallelization by the compiler
- Compiler takes over the organization of the parallel tasks
  - thread creation, distribution of tasks, ...
- Programmer takes over the necessary dependence analysis
  - which code regions can be executed in parallel?
  - enables detailed control over parallelism
  - but: programmer is responsible for correctness

2.3.1 The `parallel` directive

(Animated slide)

An example (☞ `02/firstprog.cpp`)

Program

```
main() {
   cout << "Serial\n";
   #pragma omp parallel
   {
      cout << "Parallel\n";
   }
   cout << "Serial\n";
}
```

Compilation

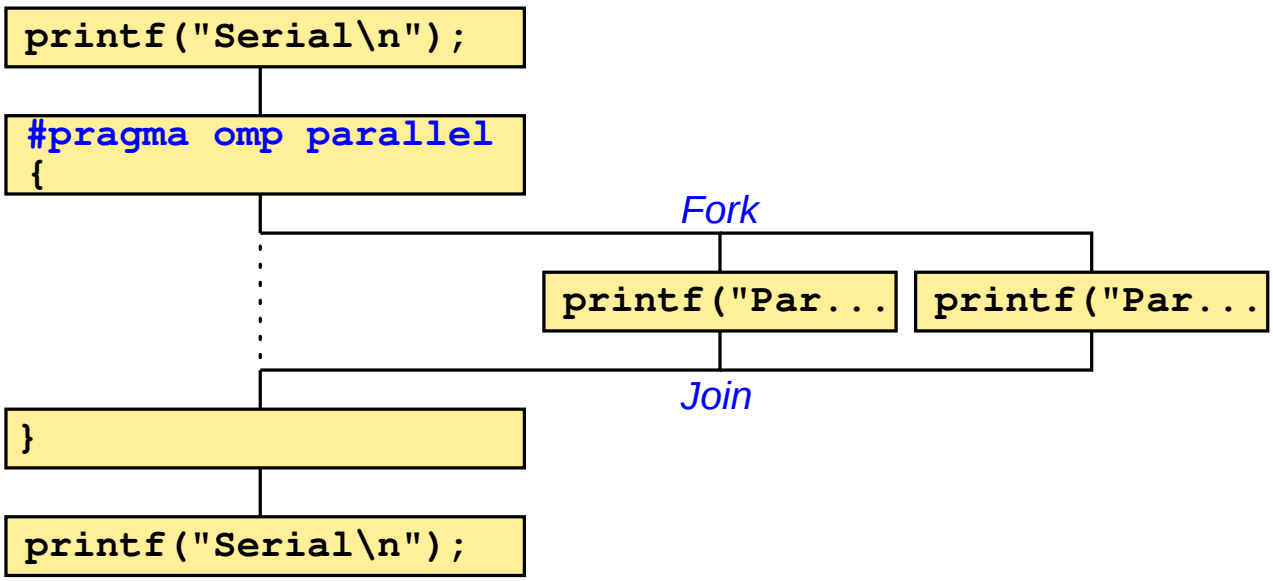
```
g++ -fopenmp -o tst
    firstprog.cpp
```

Execution

```
% export OMP_NUM_THREADS=2
% ./tst
Serial
Parallel
Parallel
Serial
```

```
% export OMP_NUM_THREADS=3
% ./tst
Serial
Parallel
Parallel
Parallel
Serial
```

Execution model: fork/join



Execution model: fork/join ...

- Program starts with exactly one *master* thread
- When a parallel region (`#pragma omp parallel`) is reached, additional threads will be created (fork)
  - environment variable `OMP_NUM_THREADS` specifies the total number of threads in the **team**
- The parallel region is executed by all threads in the team
  - at first redundantly, but additional OpenMP directives allow a partitioning of tasks
- At the end of the parallel region:
  - all threads terminate, except the master thread
  - master thread waits, until all other threads have terminated (join)

2.3.1 The `parallel` directive ...

Syntax of directives (in C / C++)

- `#pragma omp <directive> [ <clause_list> ]`
  - `<clause_list>`: List of options for the directive
- Directive only affects the immediately following statement or the immediately following block, respectively
  - **static extent** (***statischer Bereich***) of the directive

```
#pragma omp parallel
cout << "Hello\n";      // parallel
cout << "Hi there\n"; // sequential again
```

- **dynamic extent** (***dynamischer Bereich***) of a directive
  - also includes the functions being called in the static extent (which thus are also executed in parallel)

2.3.1 The `parallel` directive ...

Shared and private variables

- For variables in a parallel region there are two alternatives
  - the variables is shared by all threads (*shared variable*)
    - all threads access the same variable
      - usually, some synchronization is required!
  - each thread has its own private instance (*private variable*)
    - can be initialized with the value in the master thread
    - value is dropped at the end of the parallel region
- For variables, which are declared **within** the dynamic extent of a `parallel` directive, the following holds:
  - local variables are private
  - `static` variables and heap variables (`new`) are shared

Shared and private variables ...

- For variables, which have been declared **before entering** a parallel region, the behavior can be specified by an option of the `parallel` directive:
  - `private ( <variable_list> )`
    - private variable, without initialization
  - `firstprivate ( <variable_list> )`
    - private variable
    - initialized with the value in the master thread
  - `shared ( <variable_list> )`
    - shared variable
    - `shared` is the default for all variables

Notes for slide 180:

`private` and `firstprivate` are also possible with arrays. In this case, each thread gets its own private array (i.e., in this case an array variable is not regarded as a pointer, in contrast to the usual behavior in C/C++). When using `firstprivate`, the entire array of the master thread is copied.

Global and static variables can be defined as private variables by a separate directive `#pragma omp threadprivate( <variable_list> )`. An initialization when entering a parallel region can be achieved by using the `copyin` option.

Shared and private variables: an example (☞ `02/private.cpp`)

Each thread has a (non−initialized) copy of i

Each thread has an initialized copy of j

```
int i = 0, j = 1, k = 2;
#pragma omp parallel private(i) firstprivate(j)
{
   int h = random() % 100;
   cout << "P: i=" << i << ", j=" << j
        << ", k=" << k << ", h=" << h << "\n";
   i++; j++; k++;
}
cout << "S: i=" << i << ", j=" << j
     << ", k=" << k << "\n";
```

h ist privat

Accesses to k usually should be synchronized!

Output (with 2 threads):

```
P: i=1023456, j=1, k=2, h=86
P: i=-123059, j=1, k=3, h=83
S: i=0, j=1, k=4
```


2.3.2 Library routines

- OpenMP also defines some library routines, e.g.:
  - `int omp_get_num_threads()`: returns the number of threads
  - `int omp_get_thread_num()`: returns the thread number
    - between 0 (master thread) and `omp_get_num_threads()-1`
  - `int omp_get_num_procs()`: number of processors (cores)
  - `void omp_set_num_threads(int nthreads)`
    - defines the number of threads (maximum is `OMP_NUM_THREADS`)
  - `double omp_get_wtime()`: wall clock time in seconds
    - for runtime measurements
  - in addition: functions for mutex locks
- When using the library routines, the code can, however, no longer be compiled without OpenMP ...

Example using library routines (☞ 02/threads.cpp)

```
#include <omp.h>
int me;
omp_set_num_threads(2);              // use only 2 threads
#pragma omp parallel private(me)
{
    me = omp_get_thread_num();       // own thread number (0 or 1)
    cout << "Thread " << me << "\n";
    if (me == 0)                     // threads execute different code!
       cout << "Here is the master thread\n";
    else
       cout << "Here is the other thread\n";
}
```

- In order to use the library routines, the header file `omp.h` must be included

2.4 Loop parallelization

Motivation

- Implementation of data parallelism
  - threads perform identical computations on different parts of the data
- Two possible approaches:
  - primarily look at the data and distribute them
    - distribution of computations follows from that
    - e.g., with HPF or MPI
  - primarily look at the computations and distribute them
    - computations virtually always take place in loops (⇒ loop parallelization)
    - no explicit distribution of data
    - for programming models with shared memory

2.4.1 The `for` directive: parallel loops

```
#pragma omp for [<clause_list>]
for(...) ...
```

- Must only be used within the dynamic extent of a `parallel` directive
- Execution of loop iterations will be distributed to all threads
  - loop variables automatically is private
- Only allowed for “simple” loops
  - no `break` or `return`, integer loop variable, ...
- No synchronization at the beginning of the loop
- Barrier synchronization at the end of the loop
  - unless the option `nowait` is specified

Notes for slide 185:

Besides the option `nowait`, the following additional options can be specified in the `<clause_list>` of a `for` directive:

- `private`, `firstprivate`, `lastprivate`, `shared`: see slides 180 and 190
- `schedule`: see slide 187
- `ordered`: see slide 225
- `reduction`: see slide 227
- `collapse(<num>)`: this option tells the compiler that the next `<num>` (perfectly) nested loops should be collapsed into a single loop, whose iterations will then be distributed.

Example: vector addition

```
double a[N], b[N], c[N];
int i;
#pragma omp parallel for
for (i=0; i<N; i++) {
   a[i] = b[i] + c[i];
}
```

Short form for

```
#pragma omp parallel
{
   #pragma omp for
   ...
}
```

- Each thread processes a part of the vector
  - data partitioning, data parallel model
- Question: exactly how will the iterations be distributed to the threads?
  - can be specified using the `schedule` option
  - default: with n threads, thread 1 gets the first n-th of the iterations, thread 2 the second n-th, ...

Scheduling of loop iterations

- Option `schedule(` <class> [, <size>] `)`
- Scheduling classes:
  - `static`: blocks of given size (optional) are distributed to the threads in a round-robin fashion, before the loop is executed
  - `dynamic`: iterations are distributed in blocks of given size, execution follows the work pool model
    - better load balancing, if iterations need a different amount of time for processing
  - `guided`: like `dynamic`, but block size is decreasing exponentially (smallest block size can be specified)
    - better load balancing as compared to equal sized blocks
  - `auto`: determined by the compiler / run time system
  - `runtime`: specification via environment variable

Scheduling example(☞ 02/loops.cpp)

```
int i, j;
double x;

#pragma omp parallel for private(i,j,x) schedule(runtime)
for (i=0; i<40000; i++) {
    x = 1.2;
    for (j=0; j<i; j++) {          // triangular loop
        x = sqrt(x) * sin(x*x);
    }
}
```

- Scheduling can be specified at runtime, e.g.:
  - `export OMP_SCHEDULE="static,10"`
- Useful for optimization experiments

Scheduling example: results

- Runtime with 4 threads on the lab computers:

OMP_SCHEDULE	"static"	"static,1"	"dynamic"	"guided"
Time	3.1 s	1.9 s	1.8 s	1.8 s

- Load imbalance when using `"static"`
  - thread 1: i=0..9999, thread 4: i=30000..39999
- `"static,1"` and `"dynamic"` use a block size of 1
  - each thread executes every 4th iteration of the `i` loop
  - can be very inefficient due to caches (*false sharing*, ☞ **4.1**)
    - remedy: use larger block size (e.g.: `"dynamic,100"`)
- `"guided"` often is a good compromize between load balancing and locality (cache usage)

Shared and private variables in loops

- The `for` directive can be supplemented with the options `private`, `shared` and `firstprivate` (see slide 179 ff.)
- In addition, there is an option `lastprivate`
  - private variable
  - after the loop, the master thread has the value of the last iteration
- Example:

```
int i = 0;
#pragma omp parallel for lastprivate(i)
for (i=0; i<100; i++) {
    ...
}
printf("i=%d\\n",i);     // prints the value 100
```


2.4.2 Parallelization of Loops

(Animated slide)

When can a loop be parallelized?

```
for(i=1;i<N;i++)
   a[i] = a[i]
        + b[i-1];
```

No dependence

```
for(i=1;i<N;i++)
   a[i] = a[i-1]
        + b[i];
```

True dependence

```
for(i=0;i<N;i++)
   a[i] = a[i+1]
        + b[i];
```

Anti dependence

- Optimal: independent loops (**forall** loop)
  - loop iterations can be executed concurrently without any synchronization
  - there must not be any dependeces between statements in **different** loop iterations
  - (equivalent: the statements in different iterations must fulfill the **Bernstein conditions**)

Handling of data dependences in loops

- Anti and output dependences:
  - can always be removed, e.g., by consistent renaming of variables
  - in the previous example:

```
#pragma omp parallel
{
   #pragma omp for
   for(i=1;i<=N;i++)
      a2[i] = a[i];
   #pragma omp for
   for(i=0;i<N;i++)
      a[i] = a2[i+1] + b[i];
}
```

    - the barrier at the end of the first loop is necessary!

Handling of data dependences in loops ...

- True dependence:
  - introduce proper synchronization between the threads
    - e.g., using the `ordered` directive (☞ **2.6**):

```
#pragma omp parallel for ordered
for (i=1;i<N;i++) {
    // long computation of b[i]
    #pragma omp ordered
    a[i] = a[i-1] + b[i];
}
```

    - disadvantage: degree of parallelism often is largely reduced
  - sometimes, a vectorization (SIMD) is possible (☞ **2.8.2**), e.g.:

```
#pragma omp simd safelen(4)
for (i=4;i<N;i++)
   a[i] = a[i-4] + b[i];
```

2.4.3 Simple Examples

(Animated slide)

Matrix addition

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

No dependences in 'j' loop:
- 'b' is read only
- Elements of 'a' are always read in the same 'j' iteration, in which thay are written

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
#pragma omp parallel for
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Inner loop can be executed in parallel

2.4.3 Simple Examples ...

(Animated slide)

Matrix addition

```
double a[N][N];
double b[N][N];
int i,j;

for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

No dependences in 'i' loop:
- 'b' is read only
- Elements of 'a' are always read in the same 'i' iteration, in which they are written

```
double a[N][N];
double b[N][N];
int i,j;

#pragma omp parallel for
                private(j)
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    a[i][j] += b[i][j];
  }
}
```

Outer loop can be executed in parallel

Advantage: less overhead!

Parallel Processing

WS 2017/18

13.11.2017

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 15, 2018

2.4.3 Simple Examples ...

Matrix multiplication

```
double a[N][N], b[N][N], c[N][N];
int i,j,k;
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    c[i][j] = 0;
    for (k=0; k<N; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
  }
}
```

True dependece in the 'k' loop

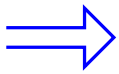
No dependences in the 'i' and 'j' loops

- Both the `i` and the `j` loop can be executed in parallel
- Usually, the outer loop is parallelized, since the overhead is lower

(Animated slide)

Removing dependences

```
double a[N], b[N];
int i;
double val = 1.2;
for (i=1; i<N; i++) {
  b[i-1] = a[i] * a[i];
  a[i-1] = val;
}
a[i-1] = b[0];
```



```
double a[N], b[N];
int i;
double val = 1.2;
#pragma omp parallel
{
  #pragma omp for
  for (i=1; i<N; i++)
    b[i-1] = a[i] * a[i];
  #pragma omp for
          lastprivate(i)
  for (i=1; i<N; i++)
    a[i-1] = val;
}
a[i-1] = b[0];
```

Anti depend. between iterations ⟶ Loop splitting + barrier

True dependece between loop and environment ⟶ lastprivate(i) + barriers

2.4.4 Dependence Analysis in Loops

(Animated slide)

Direction vectors

- Is there a dependence within a single iteration or between different iterations?

```
for (i=0; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i] * 5;
}
```

S1 $\delta^t_{(=)}$ S2

Direction vector:
S1 and S2 in same iteration

```
for (i=1; i<N; i++) {
S1:  a[i] = b[i] + c[i];
S2:  d[i] = a[i-1] * 5;
}
```

Loop carried dependece

S1 in earlier iteration than S2

S1 $\delta^t_{(<)}$ S2

```
for (i=1; i<N; i++) {
  for (j=1; j<N; j++) {
S1:  a[i][j] = b[i][j] + 2;
S2:  b[i][j] = a[i-1][j-1] - b[i][j];
  }
}
```

S1 in earlier iteration of 'I' and 'J' loop than S2

S1 $\delta^t_{(<,<)}$ S2

S1 $\delta^a_{(=,=)}$ S2

Formal computation of dependences

- Basis: Look for integer solutions of systems of (in-)equations
- Example:

```
for (i=0; i<10; i++ {
  for (j=0; j<i; j++) {
    a[i*10+j] = ...;
    ... = a[i*20+j-1];
  }
}
```

Equation system:

$$
\begin{aligned}
0 &\le i_1 < 10 \\
0 &\le i_2 < 10 \\
0 &\le j_1 < i_1 \\
0 &\le j_2 < i_2 \\
10\,i_1 + j_1 &= 20\,i_2 + j_2 - 1
\end{aligned}
$$

- Dependence analysis always is a conservative approximation!
  - unknown loop bounds, non-linear index expressions, pointers (aliasing), ...

Usage: applicability of code transformations

- Permissibility of code transformations depends on the (possibly) present data dependences
- E.g.: parallel execution of a loop is possible, if
  - this loop does not carry any data dependence
  - i.e., all direction vectors have the form $(\ldots, =, \ldots)$ or $(\ldots, \neq, \ldots, *, \ldots)$ [red: considered loop]
- E.g.: *loop interchange* is permitted, if
  - loops are perfectly nested
  - loop bounds of the inner loop are independent of the outer loop
  - no dependences with direction vector $(\ldots, <, >, \ldots)$

Example: block algorithm for matrix multiplication

```
DO I = 1,N
  DO J = 1,N
    DO K = 1,N
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
```

Strip mining

```
DO I = 1,N
      ↓
DO IT = 1,N,IS
DO I = IT, MIN(N,IT+IS-1)
```

```
DO IT = 1,N,IS
DO I = IT, MIN(N,IT+IS-1)
  DO JT = 1,N,JS
  DO J = JT, MIN(N,JT+JS-1)
    DO KT = 1,N,KS
    DO K = KT, MIN(N,KT+KS-1)
      A(I,J)=A(I,J)+B(I,K)*C(K,J)
```

Loop interchange

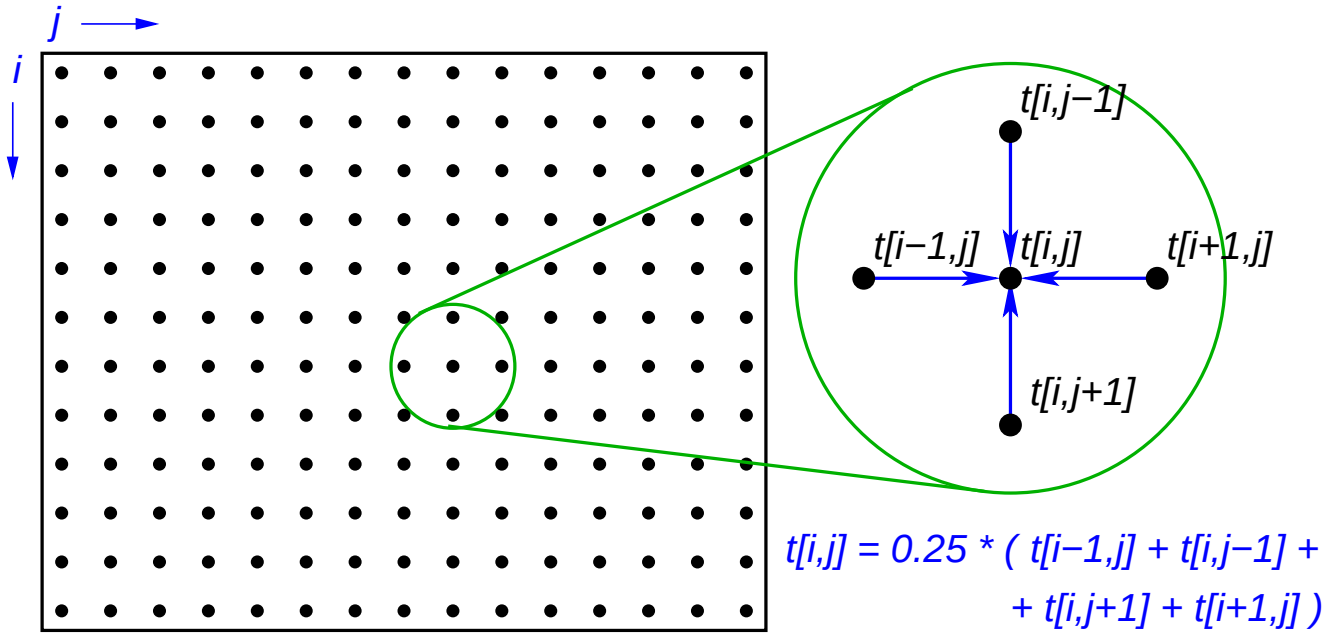
```
DO IT = 1,N,IS
DO JT = 1,N,JS
DO KT = 1,N,KS
  DO I = IT, MIN(N,IT+IS-1)
  DO J = JT, MIN(N,JT+JS-1)
  DO K = KT, MIN(N,KT+KS-1)
    A(I,J)=A(I,J)+B(I,K)*C(K,J)
```


2.5 Example: The Jacobi and Gauss/Seidel Methods

Numerical solution of the equations for thermal conduction

- Concrete problem: thin metal plate
  - given: temperature profile of the boundary
  - gesucht: temperature profile of the interior
- Approach:
  - discretization: consider the temperature only at equidistant grid points
    - 2D array of temperature values
  - iterative solution: compute ever more exact approximations
    - new approximations for the temperature of a grid point: mean value of the temperatures of the neighboring points

Numerical solution of the equations for thermal conduction ...



Metal plate

Variants of the method

- **Jacobi iteration**
  - to compute the new values, only the values of the last iteration are used
  - computation uses two matrices
- **Gauss/Seidel relaxation**
  - to compute the new values, also some values of the current iteration are used:
    - $t[i-1,j]$ and $t[i,j-1]$
  - computation uses only one matrix
  - usually faster convergence as compared to Jacobi

Variants of the method ...

Jacobi

```
do {
  for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
      b[i][j] = 0.25 *
        (a[i-1][j] + ...);
    }
  }
  for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
      a[i][j] = b[i][j];
    }
  }
} until (converged);
```

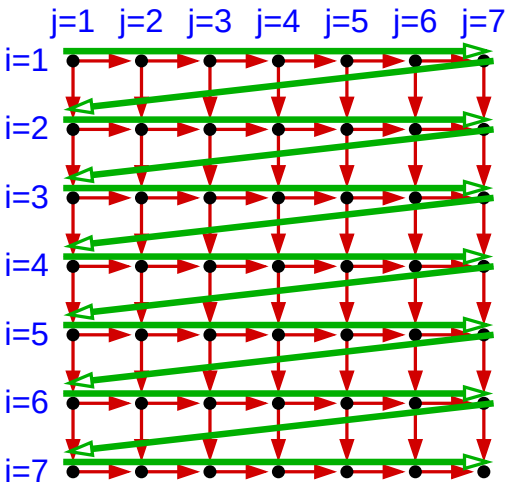
Gauss/Seidel

```
do {
  for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
      a[i][j] = 0.25 *
        (a[i-1][j] + ...);
    }
  }
} until (converged);
```


(Animated slide)

Dependences in Jacobi and Gauss/Seidel

- Jacobi: only between the two i loops
- Gauss/Seidel: iterations of the i, j loop depend on each other



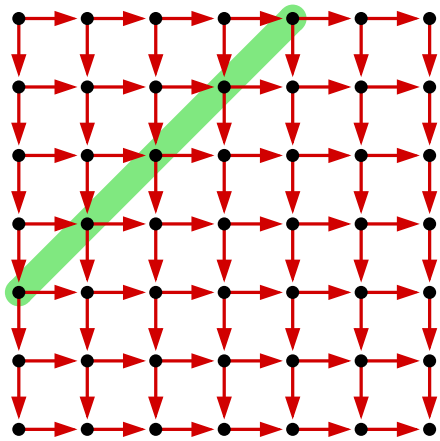
Sequential execution order

The figure shows the loop iterations, not the matrix elements!

(Animated slide)

Parallelisation of the Gauss/Seidel method

- Restructure the i, j loop, such that the iteration space is traversed diagonally
  - no dependences between the iterations of the inner loop
  - problem: varying degree of parallelism



Loop restructuring in the Gauss/Seidel method

- Row-wise traversal of the matrix:

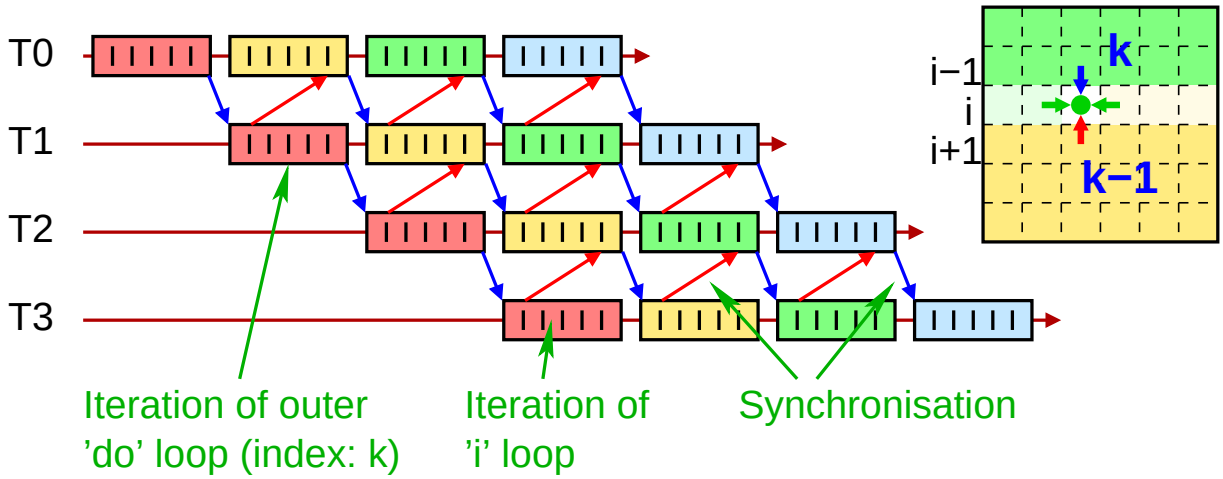
```
for (i=1; i<n-1; i++) {
   for (j=1; j<n-1; j++) {
      a[i][j] = ...;
```

- Diagonal traversal of the matrix (☞ `02/diagonal.cpp`):

```
for (ij=1; ij<2*n-4; ij++) {
   int ja = (ij <= n-2) ? 1 : ij-(n-3);
   int je = (ij <= n-2) ? ij : n-2;
   for (j=ja; j<=je; j++) {
      i = ij-j+1;
      a[i][j] = ...;
```

Alternative parallelization of the Gauss/Seidel method

- Requirement: number of iterations is known in advance
  - (or: we are allowed to execute a few more iterations after convergence)
- Then we can use a pipeline-style parallelization



Results

- Speedup using `g++ -O` on bslab10 in H-A 4111 (`eps`=0.001):

	Jacobi					Gauss/Seidel (diagonal)				
Thr.	500	700	1000	2000	4000	500	700	1000	2000	4000
1	0.9	0.9	0.9	0.9	0.9	1.8	2.0	1.6	1.6	1.3
2	1.8	1.5	1.4	1.4	1.4	3.5	3.7	2.1	2.6	2.6
3	2.6	2.0	1.6	1.6	1.6	4.0	4.4	2.5	2.7	3.1
4	3.3	2.3	1.7	1.6	1.6	4.1	4.8	3.0	3.0	3.5

- Slight performance loss due to compilation with OpenMP
- Diagonal traversal in Gauss/Seidel improves performance
- High speedup with Gauss/Seidel at a matrix size of 700
  - data size: $\sim$ 8MB, cache size: 4MB per dual core CPU

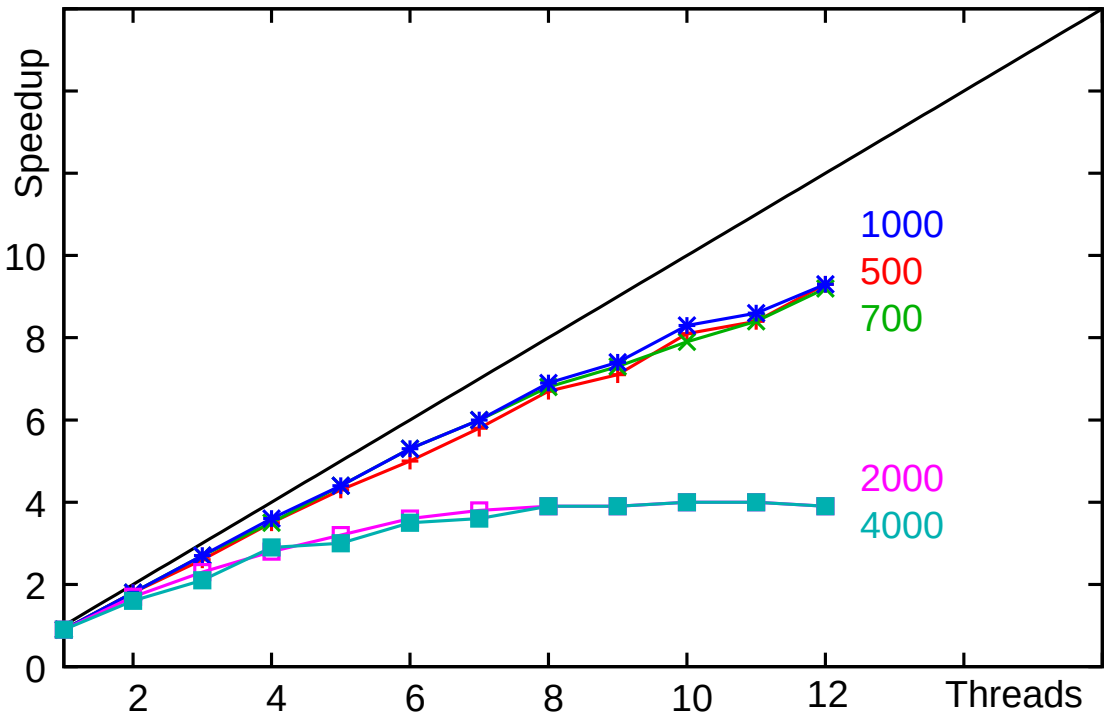
Notes for slide 210:

Results of the pipelined parallelization of the Gauss/Seidel method (g++ -O, bslab10, eps=0.001):

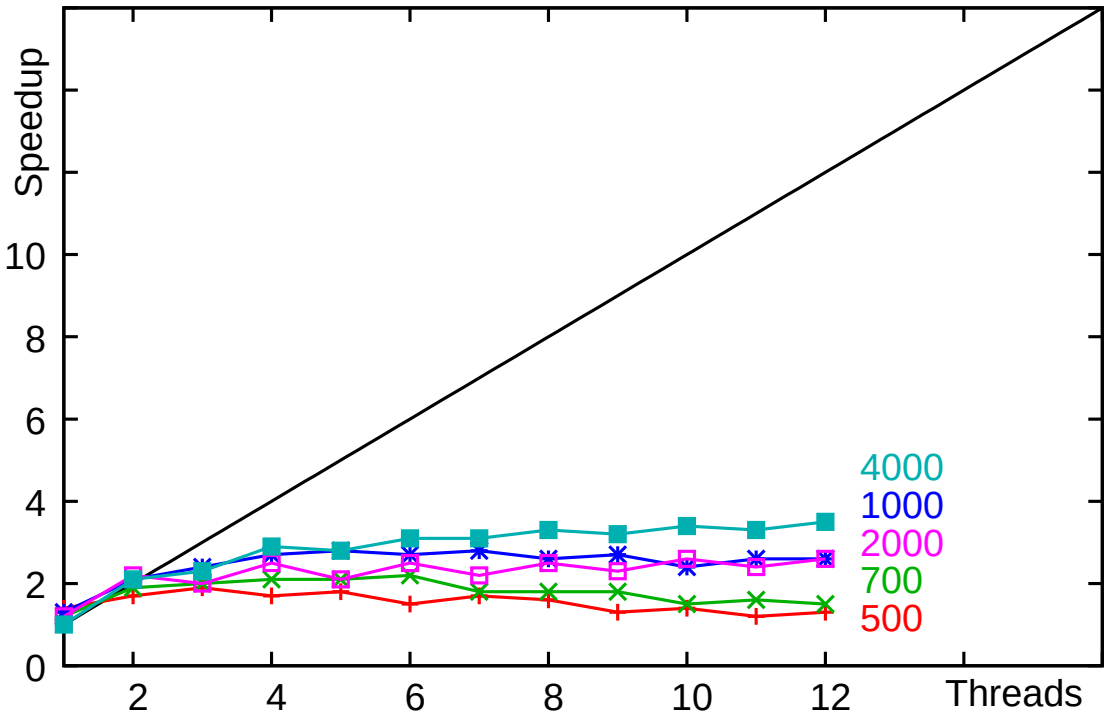
	Diagonal traversal					Pipelined parallelization				
Thr.	500	700	1000	2000	4000	500	700	1000	2000	4000
1	1.8	2.0	1.6	1.6	1.3	1.0	1.0	1.0	1.0	1.0
2	3.5	3.7	2.1	2.6	2.6	1.9	1.9	1.9	1.9	1.9
3	4.0	4.4	2.5	2.7	3.1	2.7	2.7	2.7	2.6	2.7
4	4.1	4.8	3.0	3.0	3.5	2.4	3.3	3.5	3.2	3.3

2.5 Example: The Jacobi and Gauss/Seidel Methods ...

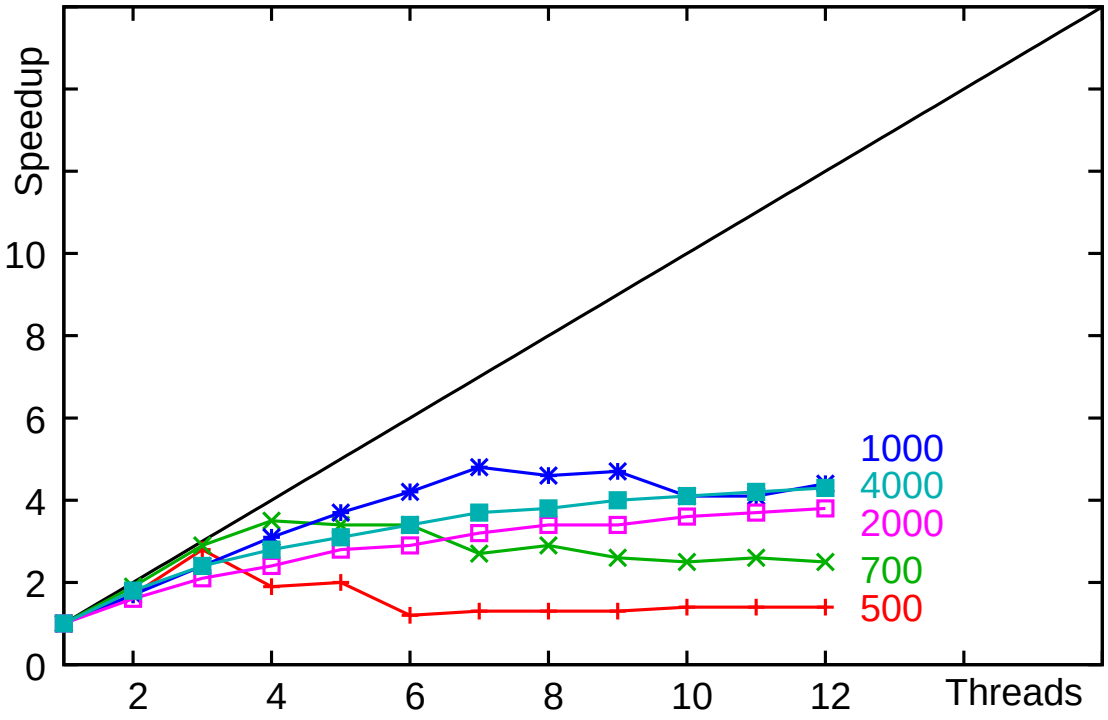
Speedup on the HorUS cluster: Jacobi



Speedup on the HorUS cluster: Gauss/Seidel (diagonal)



Speedup on the HorUS cluster: Gauss/Seidel (pipeline)



- When using OpenMP, the programmer bears full responsibility for the correct synchronization of the threads!
- A motivating example:

```
int i, j = 0;

#pragma omp parallel for private(i)
for (i=1; i<N; i++) {
   if (a[i] > a[j])
      j = i;
}
```

  - when the OpenMP directive is added, does this code fragment still compute the index of the largest element in `j`?
  - the memory accesses of the threads can be interleaved in an arbitrary order ⇒ nondeterministic errors!

Synchronization in OpenMP

- Higher-level, easy to use constructs
- Implementation using directives:
  - `barrier`: barrier
  - `single` and `master`: execution by a single thread
  - `critical`: critical section
  - `atomic`: atomic operations
  - `ordered`: execution in program order
  - `taskwait` and `taskgroup`: wait for tasks (☞ **2.7.2**)
  - `flush`: make the memory consistent
    - memory barrier (☞ **1.4.2**)
    - implicitly executed with the other synchronization directives

Parallel Processing

WS 2017/18

27.11.2017

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 15, 2018

2.6 OpenMP Synchronization ...

Barrier

```
#pragma omp barrier
```

- Synchronizes all threads
  - each thread waits, until all other threads have reached the barrier
- Implicit barrier at the end of `for`, `sections`, and `single` directives
  - can be removed by specifying the option `nowait`

Barrier: example (☞ 02/barrier.cpp)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 10000

float a[N][N];

main() {
  int i, j;

#pragma omp parallel
  {
    int thread = omp_get_thread_num();
    cout << "Thread " << thread << ": start loop 1\n";
#pragma omp for private(i,j)   // add nowait, as the case may be
    for (i=0; i<N; i++) {
```

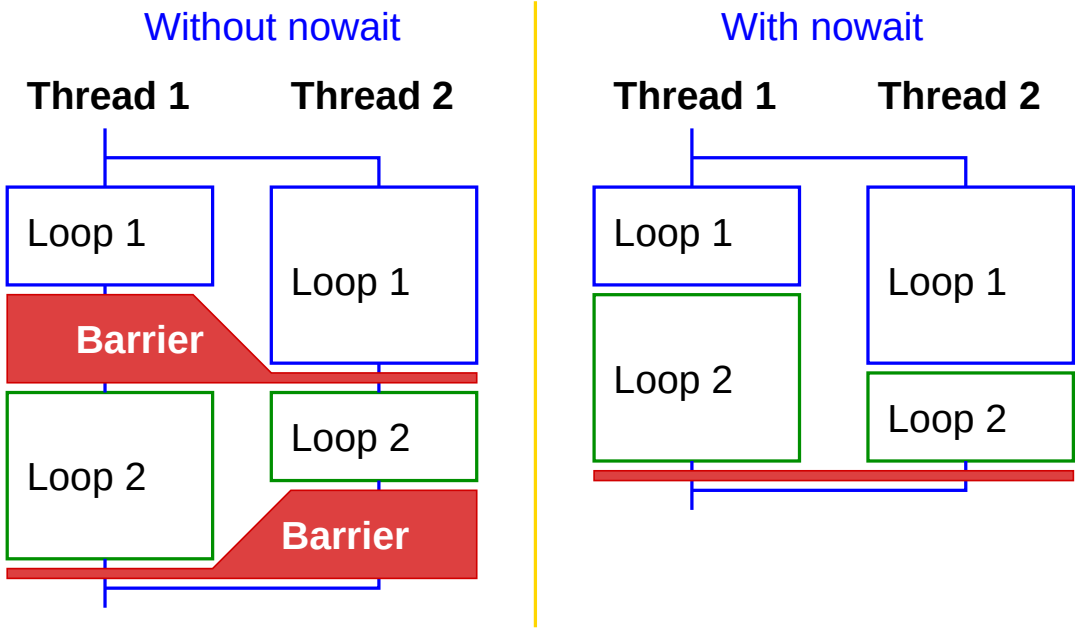

```
      for (j=0; j<i; j++) {
        a[i][j] = sqrt(i) * sin(j*j);
      }
    }

    cout << "Thread " << thread << ": start loop 2\n";
#pragma omp for private(i,j)
    for (i=0; i<N; i++) {
      for (j=i; j<N; j++) {
        a[i][j] = sqrt(i) * cos(j*j);
      }
    }
    cout << "Thread " << thread << ": end loop 2\n";
  }
}
```

Barriere: example ...

- The first loop processes the upper triangle of the matrix `a`, the second loop processers the lower triangle
  - load imbalance between the threads
  - barrier at the end of the loop results in waiting time
- But: the second loop does not depend on the first one
  - i.e., the computation can be started, before the first loop has been executed completely
  - the barrier at the end of the first loop can be removed
    - option `nowait`
  - run time with 2 threads only 4.8 s instead of 7.2 s

Barriere: example ...

- Executions of the program:



Execution using a single thread

```
#pragma omp single
    Statement / Block
```

```
#pragma omp master
    Statement / Block
```

- Block is only executed by a single thread
- No synchronization at the beginning of the directive
- `single` directive:
  - first arriving thread will execute the block
  - barrier synchronization at the end (unless: `nowait`)
- `master` directive:
  - master thread will execute the block
  - no synchronization at the end

Notes for slide 221:

Strictly speaking, the `single` directive is no Synchronization, but a directive for work distribution. It distributes the work in such a way, that a single thread does all the work.

Critical sections

```
#pragma omp critical [(<name>)]
    Statement / Block
```

- Statement / block is executed under mutual exclusion
- In order to distinguish different critical sections, they can be assigned a name

Atomic operations

```
#pragma omp atomic [read|write|update|capture][seq_cst]
    Statement / Block
```

- Statement or block (only with `capture`) will be executed atomically
  - usually by compiling it into special machine instrcutions
- Considerably more efficient than critical section
- The option defines the type of the atomic operation:
  - `read` / `write`: atomic read / write
  - `update` (default): atomic update of a variable
  - `capture`: atomic update of a variable, while storing the old or the new value, respectively
- Option `seq_cst`: enforce memory consistency (`flush`)

Notes for slide 223:

Read and write operations are atomic, only if they can be implemented using a single machine instruction. With larger data types it may happen that more than one machine word must be read or written, respectively, which requires several memory accesses. In these cases, `atomic read` and `atomic write` can be used to enforce an atomic read or atomic write, respectively.

2.6 OpenMP Synchronization ...

Atomic operations: examples

- Atomic adding:

```
#pragma omp atomic update
x += a[i] * a[j];
```

  - the right hand side will **not** be evaluated atomically!

- Atomic *fetch-and-add*:

```
#pragma omp atomic capture
{ old = counter; counter += size; }
```

- Instead of +, all other binary operators are possible, too
- Currently, an atomic *compare-and-swap* can not (yet) be implemented with OpenMP
  - use builtin functions of the compiler, if necessary

Notes for slide 224:

When using the `atomic` directive the statement must have one of the following forms:

- With the `read` option: `v = x;`
- With the `write` option: `x = <expr>;`
- With the `update` option (or without option): `x++;` `++x;` `x--;` `++x;`
  `x <binop>= <expr>;` `x = x <binop> <expr>;` `x = <expr> <binop> x;`
- With the `capture` option: `v = x++;` `v = ++x;` `v = x--;` `v = ++x;`
  `v = x <binop>= <expr>;` `v = x = x <binop> <expr>;`
  `v = x = <expr> <binop> x;`

Here, `x` and `v` are *Lvalue*s (for example, a variable) of a scalar type, `<binop>` is one of the binary operators `+`, `*`, `-`, `/`, `&`, `^`, `|`, << or >> (not overloaded!), `expr` is a scalar expression.

Note that `expr` is **not** evaluated atomically!

The `capture` option can also be used with a block, which has one of the following forms:

```
{ v = x; x <binop>= <expr>; }      { x <binop>= <expr>; v = x; }
{ v = x; x = x <binop> <expr>; }   { v = x; x = <expr> <binop> x; }
{ x = x <binop> <expr>; v = x; }   { x = <expr> <binop> x; v = x; }
{ v = x; x = <expr>; }
{ v = x; x++; }                    { v = x; ++x; }
{ ++x; v = x; }                    { x++; v = x; }
{ v = x; x--; }                    { v = x; --x; }
{ --x; v = x; }                    { x--; v = x; }
```

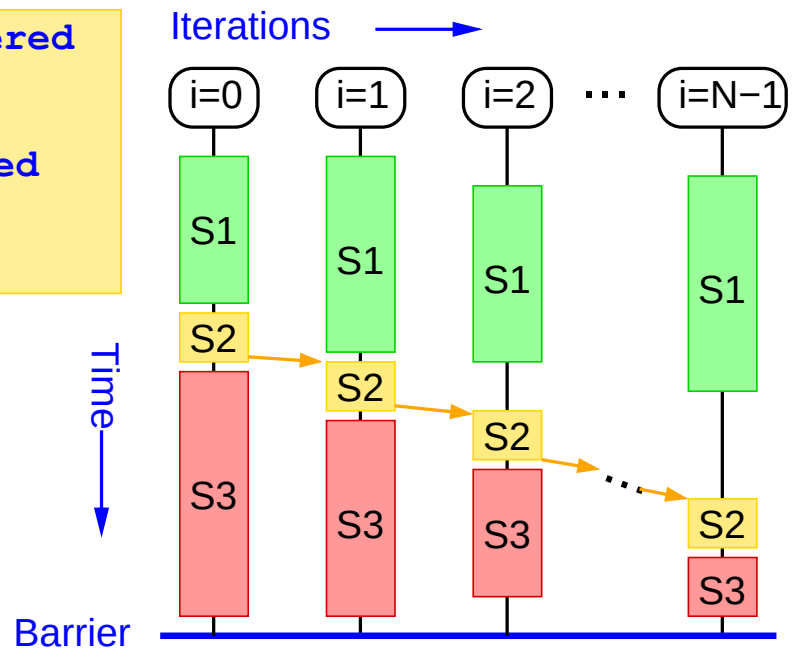
Execution in program order

```
#pragma omp for ordered
for(...) {
   ...
   #pragma omp ordered
   Statement / Block
   ...
}
```

- The `ordered` directive is only allowed in the dynamic extent of a `for` directive with option `ordered`
  - recommendation: use option `schedule(static,1)`
- The threads will execute the instances of the statement / block exacly in the same order as in the sequential program

Execution with `ordered`

```
#pragma omp for ordered
for(i=0; i<N; i++) {
   S1;
   #pragma omp ordered
      S2;
   S3;
}
```



Reduction operations

- Often loops aggregate values, e.g.:

```
int a[N];
int i, sum = 0;
#pragma omp parallel for reduction(+: sum)
for (i=0; i<N; i++){
   sum += a[i];
}
printf("sum=%d\n",sum);
```

At the end of the loop, 'sum' contains the sum of all elements

- `reduction` saves us a critical section
  - each thread first computes its partial sum in a private variable
  - after the loop ends, the total sum is computed
- Instead of `+` is is also possible to use other operators:
  `-` `*` `&` `|` `^` `&&` `||` `min` `max`
  - in addition, user defined operators are possible

(Animated slide)

Example: reduction without `reduction` option

```
int a[N];
int i, sum = 0;
int lsum = 0;  // local partial sum
#pragma omp parallel firstprivate(lsum) private(i)
{
#  pragma omp for nowait
   for (i=0; i<N; i++) {
      lsum += a[i];
   }
#  pragma omp atomic
   sum += lsum;
}
printf("sum=%d\n",sum);
```

'lsum' is initialized with 0

No barrier at the end of the loop

Add local partial sum to the global sum

2.7.1 The `sections` Directive: Parallel Code Regions

```
#pragma omp sections [<clause_list>]
{
   #pragma omp section
   Statement / Block
   #pragma omp section
   Statement / Block
   ...
}
```

- Each section will be executed exactly once by one (arbitrary) thread
- At the end of the `sections` directive, a barrier synchronization is performed
  - unless the option `nowait` is specified

2.7.1 The `sections` directive ...

Example: independent code parts

```
double a[N], b[N];
int i;
#pragma omp parallel sections private(i)
{
    #pragma omp section
    for (i=0; i<N; i++)
        a[i] = 100;
    #pragma omp section
    for (i=0; i<N; i++)
        b[i] = 200;
}
```

Important!!

- The two loops can be executed concurrently to each other
- Task partitioning

Example: influence of nowait (☞ 02/sections.cpp)

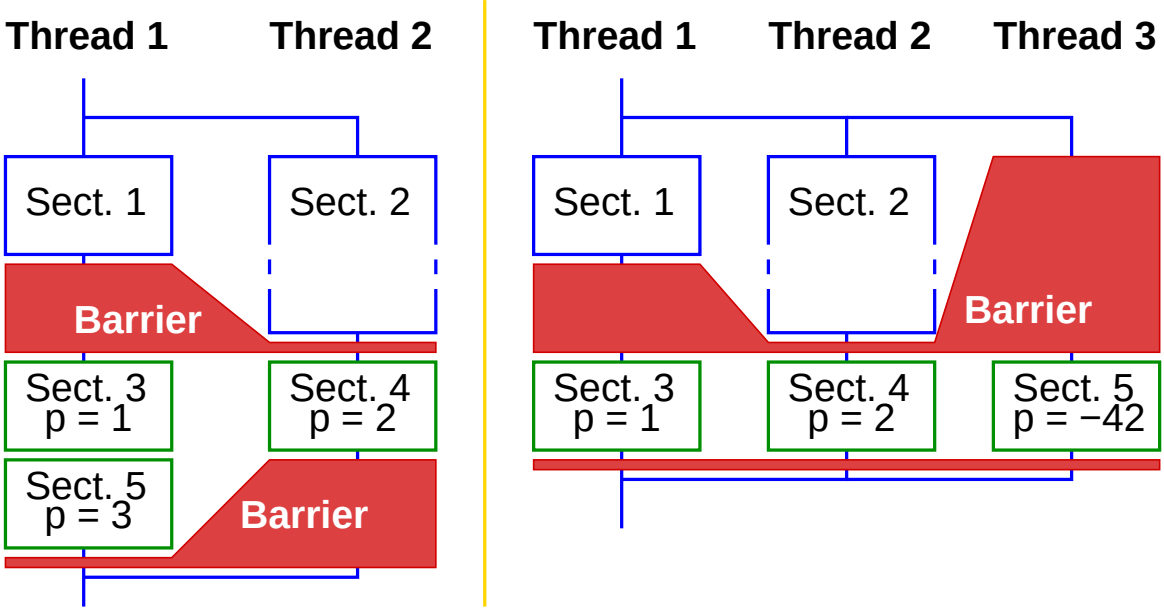
```
main() {
  int p;
#pragma omp parallel private(p)
  {
    int thread = omp_get_thread_num();
#pragma omp sections // ggf. nowait
    {
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 1 start\n";
        usleep(200000);
        cout << "Thread " << thread << ", Section 1 end\n";
        p = 1;
      }
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 2 start\n";
```


```
        usleep(1000000);
        cout << "Thread " << thread << ", Section 2 end\n";
        p = 2;
      }
    } // End omp sections
#pragma omp sections
    {
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 3 start, p = "
             << p << "\n";
        usleep(200000);
        cout << "Thread " << thread << ", Section 3 end\n";
        p = 3;
      }
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 4 start, p = "
```

```
                << p << "\n";
        usleep(200000);
        cout << "Thread " << thread << ", Section 4 end\n";
        p = 4;
      }
#pragma omp section
      {
        cout << "Thread " << thread << ", Section 5 start, p = "
             << p << "\n";
        usleep(200000);
        cout << "Thread " << thread << ", Section 5 end\n";
        p = 5;
      }
    } // End omp sections
  } // End omp parallel
}
```

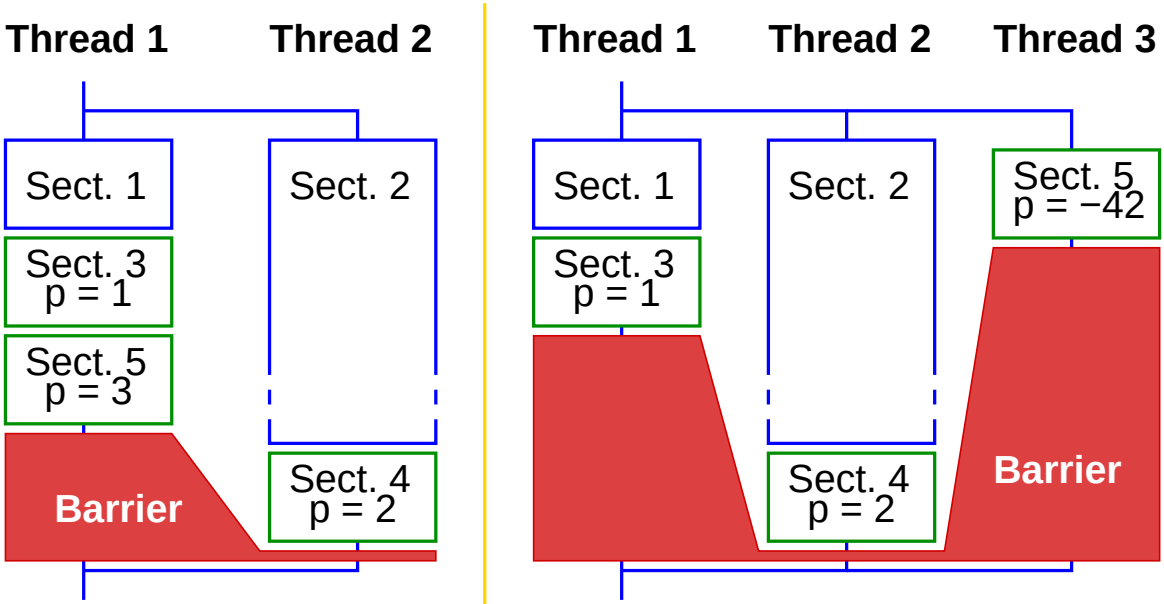

Example: influence of `nowait` ...

➥ Executions of the program **without** `nowait` option:



Example: influence of `nowait` ...

- Executions of the program **with** `nowait` option:



2.7.2 The `task` Directive: Explicit Tasks

```
#pragma omp task[<clause_list>]
Statement/Block
```

- Creates an explicit task from the statement / the block
- Tasks will be executed by the available threads (*work pool* model)
- Options `private`, `firstprivate`, `shared` determine, which variables belong to the data environment of the task
- Option `if` allows to determine, when an explicit task should be created

Example: parallel quicksort (☞ 02/qsort.cpp)

```
void quicksort(int *a, int lo, int hi) {
  ...
  // Variables are 'firstprivate' by default
  #pragma omp task if (j-lo > 10000)
  quicksort(a, lo, j);
  quicksort(a, i, hi);
}

int main() {
  ...
  #pragma omp parallel
  #pragma omp single nowait     // Execution by a single thread
  quicksort(array, 0, n-1);
  // Before the parallel region ends, we wait for the termination of all threads
```


Notes for slide 237:

In the `task` construct, global and static variables, as well es objects allocated on the heap are `shared` by default. For global and static variables, this can be changed using the `threadprivate` directive. Otherwise, all other variables used in the affected code block are `firstprivate` by default, i.e., their value is copies when the task is created. However, the `shared` attribute is inherited from the lexically enclosing constructs. For example:

```
int glob;
void example() {
  int a, b;
  #pragma omp parallel shared(b) private(a)
  {
    int c;
    #pragma omp task
    {
      int d;
      // glob: shared
      // a: firstprivate
      // b: shared
      // c: firstprivate
      // d: private
```

2.7.2 The `task` Directive ...

Task synchronization

```
#pragma omp taskwait
```

```
#pragma omp taskgroup
{
    Block
}
```

- `taskwait`: waits for the completion of all direct subtasks of the current task
- `taskgroup`: at the end of the block, the program waits for all subtasks (direct and indirect ones), which the current task has been created within the block
  - available since OpenMP 4.0
  - caution: older compilers ignore this directive!

2.7.2 The `task` Directive ...

Example: parallel quicksort (☞ `02/qsort.cpp`)

- Changes when calling quicksort:

```
#pragma omp parallel
{
   #pragma omp single nowait // Execution by exactly one thread
   quicksort(array, 0, n-1);
   checkSorted(array, n);
}
```

- Problem:
  - `quicksort()` starts new tasks
  - tasks are not yet finished, when `quicksort()` returns

Example: parallel quicksort ...

- Solution 1:

```
void quicksort(int *a, int lo, int hi) {
  ...
  #pragma omp task if (j-lo > 10000)
  quicksort(a, lo, j);
  quicksort(a, i, hi);
  #pragma omp taskwait    ← wait for the created task
}
```

  - advantage: subtask finishes, before `quicksort()` returns
    - necessary, when there are computations after the recursive call
  - disadvantage: relatively high overhead

Notes for slide 240:

In this example, an additional overhead is created by always waiting for the subtasks after the recursive calls, even if none were generated (because `j-lo <= 10000`). For the `taskwait` directive, there is no `if` option, so you might need to include a conditional statement here.

Example: parallel quicksort ...

- Solution 2:

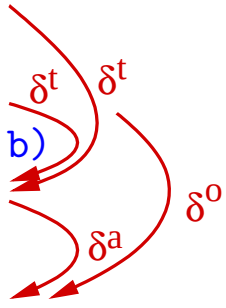
```
#pragma omp parallel
{
   #pragma omp taskgroup
   {
      #pragma omp single nowait  // Execution by exactly one thread
      quicksort(array, 0, n-1);
   }                    ← wait for all tasks created in the block
   checkSorted(array, n);
}
```

  - advantage: only wait at one single place
  - disadvantage: semantics of `quicksort()` must be very well documented

Dependences between tasks (☞ `02/tasks.cpp`)

- Option `depend` allows to specify dependences between tasks
  - you must specify the affected variables (or array sections, if applicable) and the direction of data flow
- Beispiel:

```
#pragma omp task shared(a) depend(out:  a)
  a = computeA();
#pragma omp task shared(b) depend(out:  b)
  b = computeB();
#pragma omp task shared(a,b,c) depend(in:  a,b)
  c = computeCfromAandB(a, b);
#pragma omp task shared(b) depend(out:  b)
  b = computeBagain();
```



  - the variables `a`, `b`, and `c` must be `shared` in this case, since they contain the result of the computation of a task

Notes for slide 242:

In the `depend` option, a dependency type is defined, which specifies the direction of the data flow. Possible values are `in`, `out`, and `inout`.

- With `in`, the generated task will depend on all previously created “sibling” tasks that specify at least one of the listed variables in a `depend` option of type `out` or `inout`.
- With `out` and `inout`, the generated task will depend on all previously created “sibling” tasks that specify at least one of the listed variables in a `depend` option of type `in`, `out`, or `inout`.

Array sections can be specified using the notation:

```
<name> [ [<lower-bound>] : [<length>] ]
```

A missing lower bound is assumed to be 0, a missing length as the array length minus lower bound.

2.8 OpenMP Details

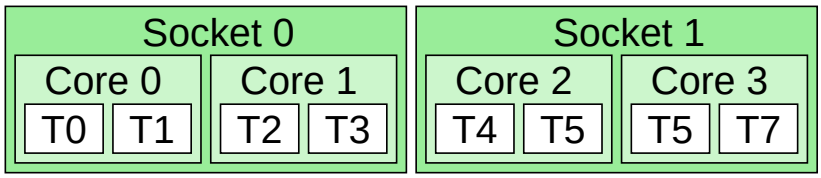
2.8.1 Thread Affinity

- Goal: control where threads are executed
  - i.e., by which HW threads on which core on which CPU
- Important (among others) because of the architecture of today’s multicore CPUs
  - HW threads share the functional units of a core
  - cores share the L2 caches and the memory interface
- Concept of OpenMP:
  - introduction of **places**
    - place: set of hardware execution environments
    - e.g., hardware thread, core, processor (socket)
  - options control the distribution from threads to places

(Animated slide)

Environment variable OMP_PLACES

- Defines the places of a computer
- e.g., nodes with 2 dual-core CPUs with 2 HW threads each:



- To consider each core as a place, e.g.:

```
OMP_PLACES = "cores"
OMP_PLACES = "{0,1},{2,3},{4,5},{6,7}"
OMP_PLACES = "{0,1}:4:2"
```

- To consider each socket as a place, e.g.:

```
OMP_PLACES = "sockets"
OMP_PLACES = "sockets(2)"
OMP_PLACES = "{0:4},{4:4}"
```


Notes for slide 244:

The list of places in OMP_PLACES can be defined by a symbolic name (threads, cores, sockets or an implementation-dependent name) or by explicit listing of places.

A single place is defined as a set of hardware execution environments (on a standard CPU, these are typically the hardware threads), where a length and an optional step size can be specified for the simplified definition of intervals (separated by ":").

Mapping from threads to places

- Option `proc_bind( spread | close | master )` of the `parallel` directive
  - `spread`: threads will be evenly distributed to the available places; the list of places will be partitioned
    - avoids resource conflicts between threads
  - `close`: threads are allocated as close to the master thread as possible
    - e.g., to make optimal use ot the shared cache
  - `master`: threads will be allocated on the same place as the master thread
    - closest possible locality to master thread
- Usually combined with nested parallel regions

Notes for slide 245:

In the example of slide 244, if the master thread is executed by hardware thread T0, `OMP_PLACES = threads` is specified, and a parallel region with 4 threads is created, the following happens:

- with `proc_bind(spread)`: The threads are placed on T0, T2, T4 and T6. The thread on T0 is given `{0},{1}` as its new place list, the thread on T2 receives `{2},{3}`, etc.
- with `proc_bind(close)`: The threads are placed on T0, T1, T2 and T3. The place list remains unchanged.
- with `proc_bind(master)`: The threads are all placed on T0. The place list remains unchanged.

Parallel Processing

WS 2017/18

11.12.2017

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 15, 2018

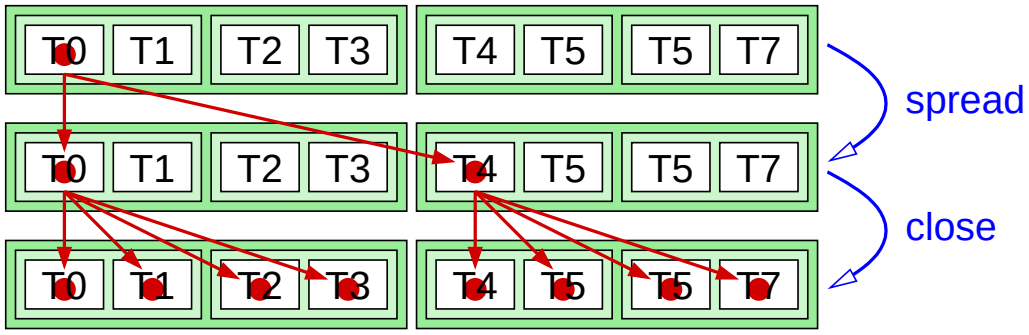
2.8.1 Thread Affinity ...

Example: nested parallel regions

```
double f1(double x)
{
  #pragma omp parallel for proc_bind(close)
  for (i=0; i<N; i++) {
     ...
}
...
  #pragma omp parallel proc_bind(spread)
  #pragma omp single
  {
    #pragma omp task shared(a)
    a = f1(x);
    #pragma omp task shared(b)
    b = f1(y);
  }
  ...
```

Example: nested parallel regions ...

- Allow nested parallelism: `export OMP_NESTED=true`
- Define the number of threads for each nesting level:
  - `export OMP_NUM_THREADS=2,4`
- Define the places: `export OMP_PLACES=cores`
- Allow the binding of threads to places:
  - `export OMP_PROC_BIND=true`
- Result:



Notes for slide 247:

The number of threads, their binding to places, and some other parameters can usually be specified in three different ways in OpenMP:

1. by using an option of a directive (e.g., `num_threads`),
2. by calling an OpenMP library routine (e.g., `omp_set_num_threads`),
3. by setting an environment variable (e.g., `OMP_NUM_THREADS`)

Here, the option of the directive has the highest priority, the environment variable the lowest one.

In the example, you also could directly specify the (maximum) number of threads in the `parallel` directives, using the option `num_threads`.

Vice versa, you could omit the `proc_bind` options and specify the binding via the environment variable `OMP_PROC_BIND`:

```
export OMP_PROC_BIND="spread,close"
```

2.8.2 SIMD Vectorization

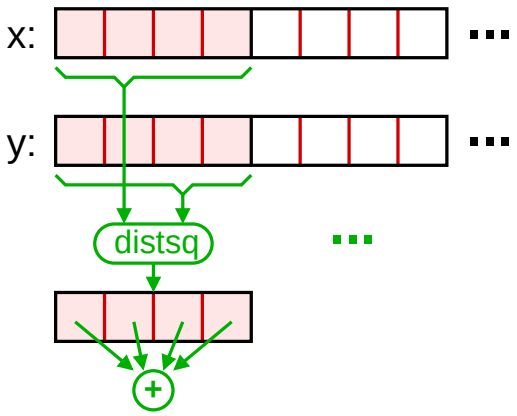
```
#pragma omp simd [<clause_list>]
for(...) ...
```

- Restructuring of a loop in order to use the SIMD vector registers
  - e.g., Intel SSE: 4 `float` operations in parallel
- Loop will be executed by a sinlge thread
  - combination with `for` is possible
- Options (among others): `private`, `lastprivate`, `reduction`
- Option `safelen`: maximum vector length
  - i.e., distance (in iterations) of data dependences, e.g.:

```
for (i=0; i<N; i++)
    a[i] = b[i] + a[i-4];   // safelen = 4
```


Example

```
#pragma omp declare simd
float distsq(float x, float y) {
  return (x - y) * (x - y);
}
...
  #pragma omp simd reduction(+:s)
  for (i=0; i<N; i++) {
    s += distsq(x[i], y[i]);
  }
```



- The directive `declare simd` generates a version of the function with vector registers as arguments and result
- For larger `N` the following may be useful:

```
#pragma omp parallel for simd reduction(+:s)
```

2.8.3 Using External Accelerators

- Model in OpenMP 4.0:
  - one host (multi processor with shared memory) with several identical accelerators (**targets**)
  - execution of a code block can be moved to a target using a directive
  - host and target can have a shared memory, but this is not required
    - data transport must be executed using directives
- In order to support the execution model of GPUs:
  - introduction of thread teams
  - threads of the same team are executed by one streaming multiprocessor (in SIMD manner)

The `target` directive

```
#pragma omp target [data] [<clause_list>]
```

- Transfers execution and data to a target
  - data will be copied between CPU memory and target memory
    - mapping and direction of transfer are specified using the `map` option
  - the subsequent code block will be executed on the target
    - except when only a data environment is created using `target data`
- Host waits until the computation on the target is finished
  - however, the `target` directive is also possible within an asynchronous task

The map option

- Maps variable v_H in the host data environment to the corresponding variable v_T in the target data environment
  - either the data is copied between v_H and v_T or v_H and v_T are identical
- Syntax: `map( alloc | to | from | tofrom : <list> )`
  - `<list>`: list of the original variables
    - array sections are allowed, too, ☞ **2.7.2**
  - `alloc`: just allocate memory for v_T
  - `to`: allocate v_T, copy v_H to v_T at the beginning
  - `from`: allocate v_T, copy v_T to v_H at the end
  - `tofrom`: default value, `to` and `from`

Notes for slide 252:

When declaring global variables, it is possible to specify that they should also be created on the target, using the directive `declare target`. With the same directive, you can also declare functions that must be callable on the target. These functions are then compiled appropriately for the host and the target.

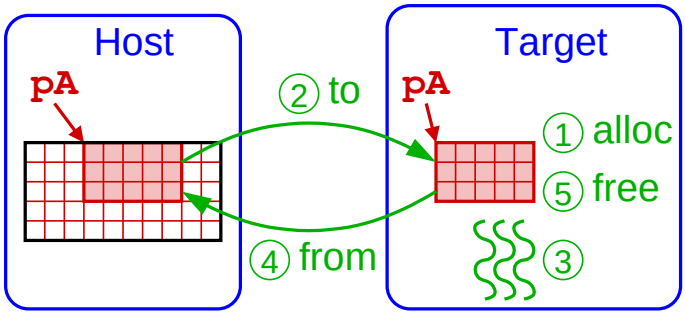
Example:

```
#define N 65536
#pragma omp declare target
float a[N], b[N];
float myFunction(float f1, float f2) {
  ...
}
#pragma omp end declare target
```

Target data environment

- Course of actions for the `target` construct:

```
#pragma omp target \
        map(tofrom: pA)
{① ②
   ... ③
}④ ⑤
```



- `target data` environments are used to optimize memory transfers
  - several code blocks can be executed on the target, without having to transfer the data several times
  - if needed, the directive `target update` allows a data transfer within the `target data` environment

Example

```
#pragma omp target data map(alloc:tmp[:N]) \
        map(to:in[:N]) map(from:res)
{
  #pragma omp target
  #pragma omp parallel for
  for (i=0; i<N; i++)
    tmp[i] = compute_1(in[i]);

  modify_input_array(in);

  #pragma omp target update to(in[:N])
  #pragma omp target
  #pragma omp parallel for reduction(+:res)
  for (i=0; i<N; i++)
    res += compute_2(in[i], tmp[i])
}
```

Host

Target

Host

Target

Host

Thread teams

- Allow a parallelization at two levels, e.g., on GPUs
- Create a set of thread teams:

```
#pragma omp teams [<clause_list>]
Statement/Block
```

  - statement/block is executed by the master thread of each team
  - teams can **not** synchronize
- Distribution of a loop to the master threads of the teams:

```
#pragma omp distribute [<clause_list>]
for(...) ...
```

- Parallelization within a team, e.g., using `parallel for`

Example: SAXPY

- On the host:

```
void saxpy(float *y, float *x, float a, int n) {
  #pragma omp parallel for
  for (int i = 0; i < n; i++)
    y[i] = a*x[i] + y[i];
}
```

- On the GPU (naive):

```
void saxpy(float *y, float *x, float a, int n) {
  #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
  {
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
      y[i] = a*x[i] + y[i];
  }
}
```

Example: SAXPY ...

➥ On the GPU (optimized): each team processes a block

```
void saxpy(float *y, float *x, float a, int n) {
  int nBlk = numBlocks(n);   // Number of blocks
  int nThr = numThreads(n); // Number of threads
  #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
  #pragma omp teams num_teams(nBlk) thread_limit(nThr)
  {
    #pragma omp distribute
    for (int i = 0; i < n; i += n/nBlk) {
      #pragma omp parallel for
      for (int j = i; j < i + n/nBlk; j++)
        y[j] = a*x[j] + y[j];
      }
    }
  }
}
```


Notes for slide 257:

The option `num_teams` determines the number of thread teams that should be created, the option `thread_limit` determines the number of threads in each team. Both numbers should be chosen according the the available accelerator and the problem size.

Example: SAXPY ...

- On the GPU (optimized, shorter):

```
void saxpy(float *y, float *x, float a, int n) {
  int nBlk = numBlocks(n);   // Number of blocks
  int nThr = numThreads(n); // Number of threads
  #pragma omp target map(to:x[0:n], n, a) map(y[0:n])
  #pragma omp teams distribute parallel for \
          num_teams(nBlk) thread_limit(nThr)
  for (int i = 0; i < n; i++)
    y[i] = a*x[i] + y[i];
}
```

- Iterations are first distributed to the streaming multiprocessors in blocks; there they are again distributed to the individual threads