Parallel Processing

Winter Term 2024/25

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

Stand: January 13, 2025

Parallel Processing

Winter Term 2024/25

3 Parallel Programming with Shared Memory

3 Parallel Programming with Shared Memory ...

Contents

- OpenMP basics
- Loop parallelization and dependeces
- Exercise: The Jacobi and Gauss/Seidel Methods
- OpenMP synchronization
- Task parallelism with OpenMP
- Tutorial: tools for OpenMP
- Exercise: A solver for the Sokoban game
- Excursion: *Lock-Free* and *Wait-Free* Data Structures

Literature

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

3 Parallel Programming with Shared Memory ...

Approaches to programming with threads

- Using (system) libraries
  - Examples: POSIX threads, Intel Threading Building Blocks (TBB)
- As part of a programming language
  - Examples: Java threads (☞ **BS_I**), C++ threads (☞ **1.3**)
- Using compiler directives (pragmas)
  - Examples: OpenMP (☞ **3.1**)

3.1 OpenMP Basics

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)

3.1 OpenMP Basics ...

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

3.1 OpenMP Basics ...

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

3.1 OpenMP Basics ...

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

3.1 OpenMP Basics ...

Compiling and executing OpenMP programs

- Compilation with gcc (g++)
  - typical call: `g++ -fopenmp myProg.cpp -o myProg`
  - OpenMP 4.0 is supported since gcc 4.9
- Execution: identical to a sequential program
  - e.g.: `./myProg`
  - (maximum) number of threads can be specified in environment variable `OMP_NUM_THREADS`
    - e.g.: `export OMP_NUM_THREADS=4`
    - specification holds for all programs started in the same shell
  - also possible: temporary (re-)definition of `OMP_NUM_THREADS`
    - e.g.: `OMP_NUM_THREADS=2 ./myProg`

3.1.1 The `parallel` directive

(Animated slide)

An example (☞ 03/firstprog.cpp)

Program

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

Compilation

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

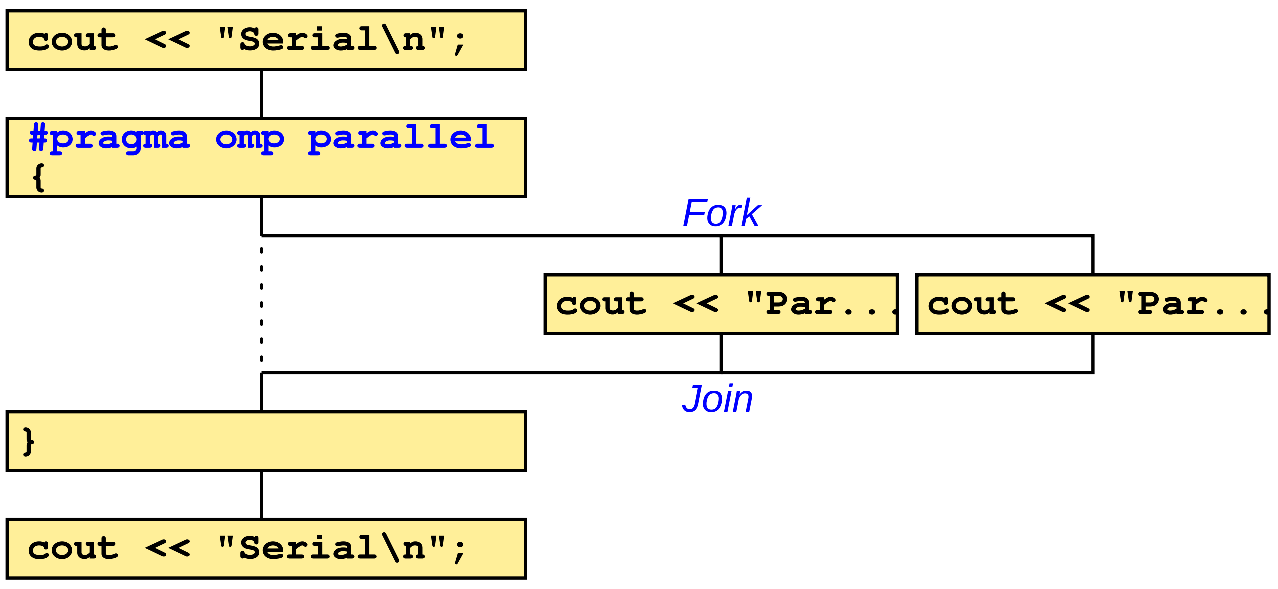
Execution

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

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

3.1.1 The `parallel` directive ...

Execution model: fork/join



3.1.1 The `parallel` directive ...

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)

3.1.1 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)

3.1.1 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

3.1.1 The `parallel` directive ...

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 208:

`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.

3.1.1 The `parallel` directive ...

Shared and private variables: an example (☞ `03/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 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 is private

Accesses to k usually should be synchronized!

Output (with 2 threads)**:**

```
P: i=1028465, j=1, k=2, h=86
P: i=-128755, j=1, k=3, h=83
S: i=0, j=1, k=4
```

3.1.2 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 ...

3.1.2 Library routines ...

Example using library routines (☞ 03/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

3.2 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 ($\Rightarrow$ loop parallelization)
    - no explicit distribution of data
    - for programming models with shared memory

3.2.1 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 variable 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 213:

- The option nowait is not accepted in a #pragma omp parallel for (as at the end of a parallel region, there always is a global synchronisation)
- 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 208 and 218 (These options are only accepted in a #pragma omp parallel for, not in a #pragma omp for inside a parallel region)
  - schedule: see slide 215
  - ordered: see slide 251
  - reduction: see slide 249
  - 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.

3.2.1 The `for` directive: parallel loops ...

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, ...

3.2.1 The `for` directive: parallel loops ...

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

3.2.1 The `for` directive: parallel loops ...

Scheduling example(☞ `03/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

3.2.1 The `for` directive: parallel loops ...

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*, ☞ **5.1**)
    - remedy: use larger block size (e.g.: `"dynamic,100"`)
- `"guided"` often is a good compromize between load balancing and locality (cache usage)

3.2.1 The `for` directive: parallel loops ...

Shared and private variables in loops

- The `parallel for` directive can be supplemented with the options `private`, `shared` and `firstprivate` (see slide 207 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++) {
    ...
}
std::cout << "i=" << i << std::endl;   // prints the value 100
```

3.2.2 Parallelization of Loops

(Animated slide)

When can a loop be parallelized?

```for(i=1;i<N;i++) a[i] = a[i] + b[i-1];```	```for(i=1;i<N;i++) a[i] = a[i-1] + b[i];```	```for(i=0;i<N;i++) a[i] = a[i+1] + b[i];```
No dependence	True dependence	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**)

3.2.2 Parallelization of Loops ...

(Animated slide)

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!

3.2.2 Parallelization of Loops ...

Handling of data dependences in loops ...

- True dependence:
  - introduce proper synchronization between the threads
    - e.g., using the `ordered` directive (☞ **3.4**):

```
#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 (☞ **??**), e.g.:

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

3.2.3 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

3.2.3 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!

3.2.3 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

3.2.3 Simple Examples ...

(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];
```

$\Longrightarrow$

```
double a[N], b[N], a2[N];
int i;
double val = 1.2;
#pragma omp parallel
{
  #pragma omp for
  for (i=1; i<N; i++)
    a2[i] = a[i];
  #pragma omp for
                lastprivate(i)
  for (i=1; i<N; i++)
    b[i-1] = a2[i] * a2[i];
    a[i-1] = val;
}
a[i-1] = b[0];
```

Anti depend. between iterations $\longrightarrow$ Renaming + barrier

True dependece between loop and environment $\longrightarrow$ lastprivate(i) + barriers

3.2.4 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

3.2.4 Dependence Analysis in Loops ...

Formal computation of dependences

- Basis: Look for an integer solution of a system 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:

$$0 \le i_1 < 10$$
$$0 \le i_2 < 10$$
$$0 \le j_1 < i_1$$
$$0 \le j_2 < i_2$$
$$10\, i_1 + j_1 = 20\, i_2 + j_2 - 1$$

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

3.2.4 Dependence Analysis in Loops ...

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 $(\ldots, <, >, \ldots)$

Notes for slide 228:

Here is an example with a dependence vector $(>, *)$, which means that the inner loop (i.e. the j-loop) can be parallelized:

```
for (i=1; i<N; i++) {
  #pragma omp parallel for
  for (j=1; j<N; j++) {
    a[i][j] = b[j] + c[j];  // S1
    d[j] = a[i+1][j-1] + 5; // S2
  }
}
```

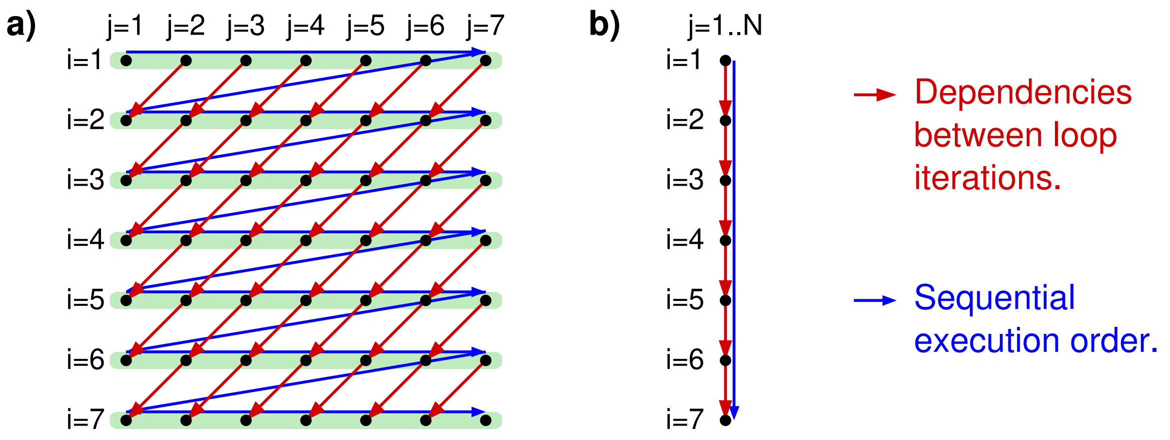
There is an anti-dependence from S2 to S1 (consider e.g. a[3][3]: it is read in iteration i=2, j=4 and is written later in iteration i=3, j=3.

However, this dependence is not carried by the j-loop, but by the i-loop: If we consider a fixed iteration of the i-loop, e.g., i=2, then the j-loop never reads and writes the same element of a. E.g., it writes a[2][4] in iteration j=4, but reads a[3][4] in iteration j=5.

On the other hand, in iteration, e.g., i=2, the body of the i-loop reads the elements a[3][0..N-1], and later in iteration i=3, it writes the elements a[3][1..N], so we have a loop carried (anti-)dependence in the i-loop.

The dependencies can be visualized in a diagram showing the iteration space of the loops, where each loop iteration is shown as a dot. Figure a) shows that although there are dependencies, the iterations of the `j`-loop can be carried out concurrently (indicated by the green bars in the background), as there is no dependene between the iterations.

Note that when looking at the outer `i`-loop, we have to consider its complete body as one statement (i.e., we have to look at the union of all iterations of the inner `j`-loop), so we end up with the picture in figure b). We immediately see that this is a sequential loop.


a)
j=1 j=2 j=3 j=4 j=5 j=6 j=7
i=1 i=2 i=3 i=4 i=5 i=6 i=7
b)
j=1..N
i=1 i=2 i=3 i=4 i=5 i=6 i=7
Dependencies between loop iterations.
Sequential execution order.

(Actually, figure a) shows that we also could execute the `j`-loop in parallel, **if** we interchange the loops, such that the `j`-loop becomes the u outer loop and takes care about carrying the dependencies.)

3.2.4 Dependence Analysis in Loops ...

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)
```

3.2.4 Dependence Analysis in Loops ...

Example: loop splitting

- Consider the following loop:

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

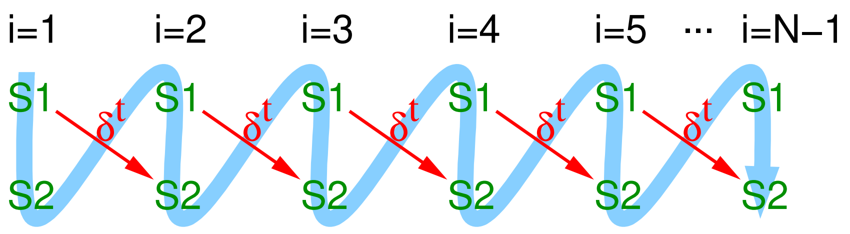
- We have $S1\ \delta^t_{(<)}\ S2$, which prevents parallelization of the loop without synchronization
- However, since we do *not* have any dependence $S2\ \delta_{(<)}\ S1$, loop splitting is permitted, which results in:

```
for (i=1; i<N-1; i++)
   a[i] = (c[i-1] + c[i+1])/2; // S1
for (i=1; i<N-1; i++)
   b[i] = a[i-1];              // S2
```

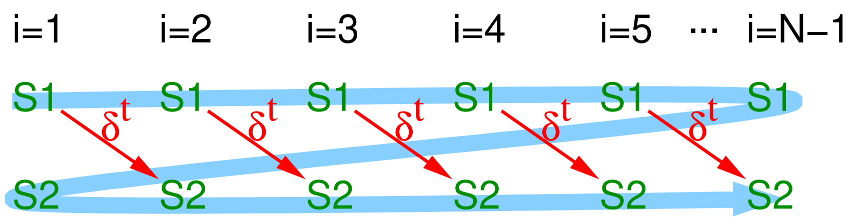
3.2.4 Dependence Analysis in Loops ...

Example: loop splitting ...

- Execution of the original loop:



- Execution of the transformed loop:



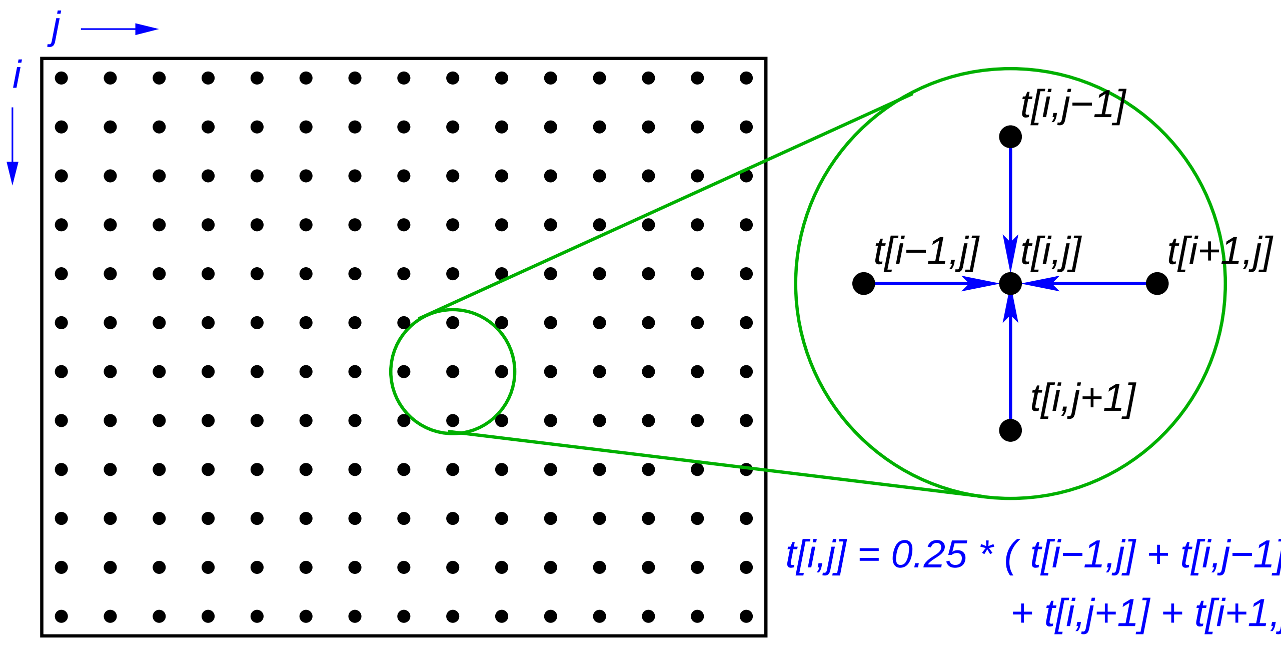
3.3 Exercise: The Jacobi and Gauss/Seidel Methods

Numerical solution of the equations for thermal conduction

- Concrete problem: thin metal plate
  - given: temperature profile of the boundary
  - wanted: temperature profile of the interior (at equilibrium)
- 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

3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

Numerical solution of the equations for thermal conduction ...



*t[i,j] = 0.25 * (t[i−1,j] + t[i,j−1] +*
+ t[i,j+1] + t[i+1,j])

Metal plate

3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

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

Parallel Processing

Winter Term 2024/25

18.11.2024

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

Stand: January 13, 2025

3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

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

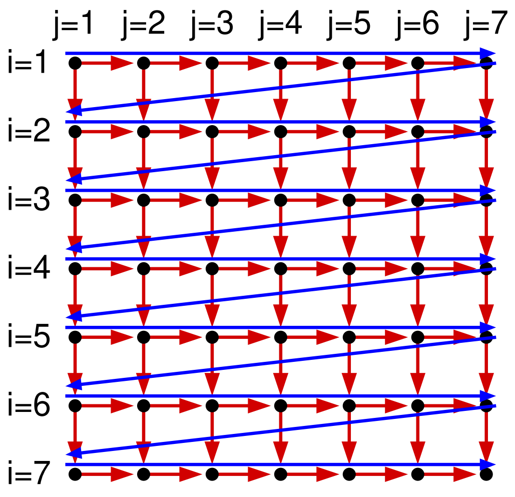
```
do {
  for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
      a[i][j] = 0.25 *
        (a[i-1][j] + ...);
    }
  }
} until (converged);
```

3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

(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

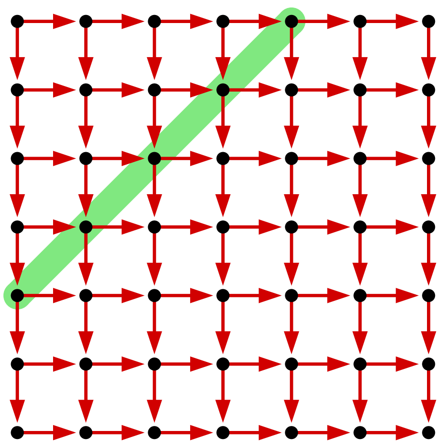
The figure shows the loop iterations, not the matrix elements!

3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

(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



3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

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 (☞ `03/diagonal.cpp`):

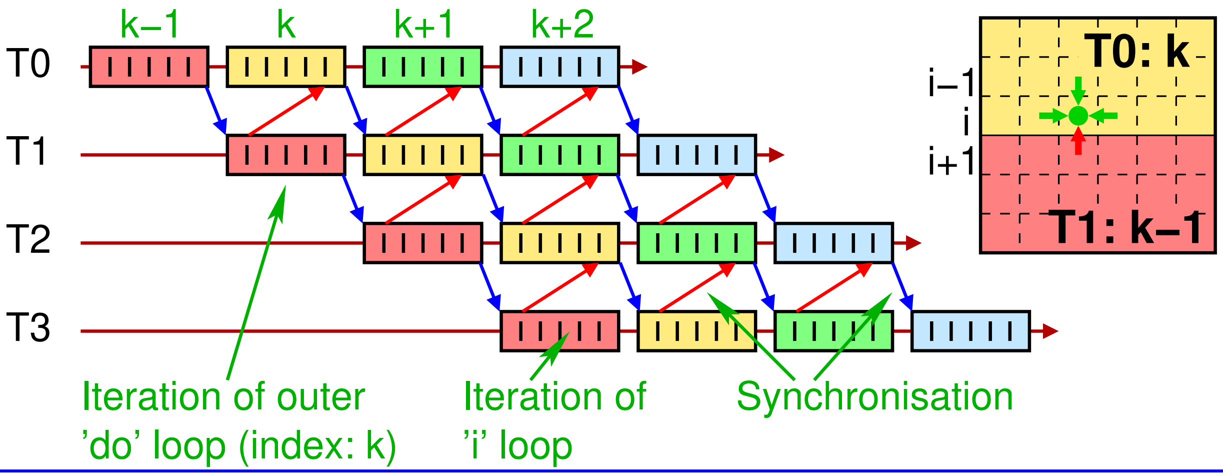
```
for (ij=1; ij<2*n-4; ij++) {
   int ja = (ij <= n-2) ? 1 : ij-(n-3);
   int je = (ij <= n-2) ? ij : n-2;
   for (j=ja; j<=je; j++) {
      i = ij-j+1;
      a[i][j] = ...;
```

3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

(Animated slide)

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
  - synchronisation via `ordered` (☞ **3.4.4**)



3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

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

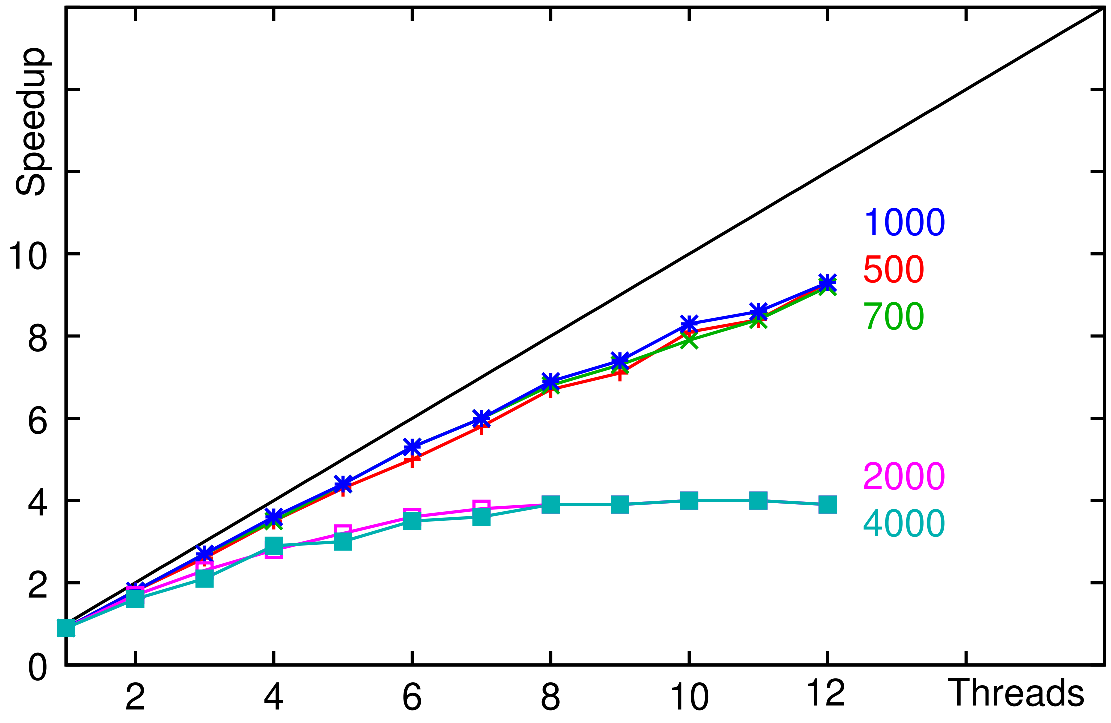
Notes for slide 240:

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

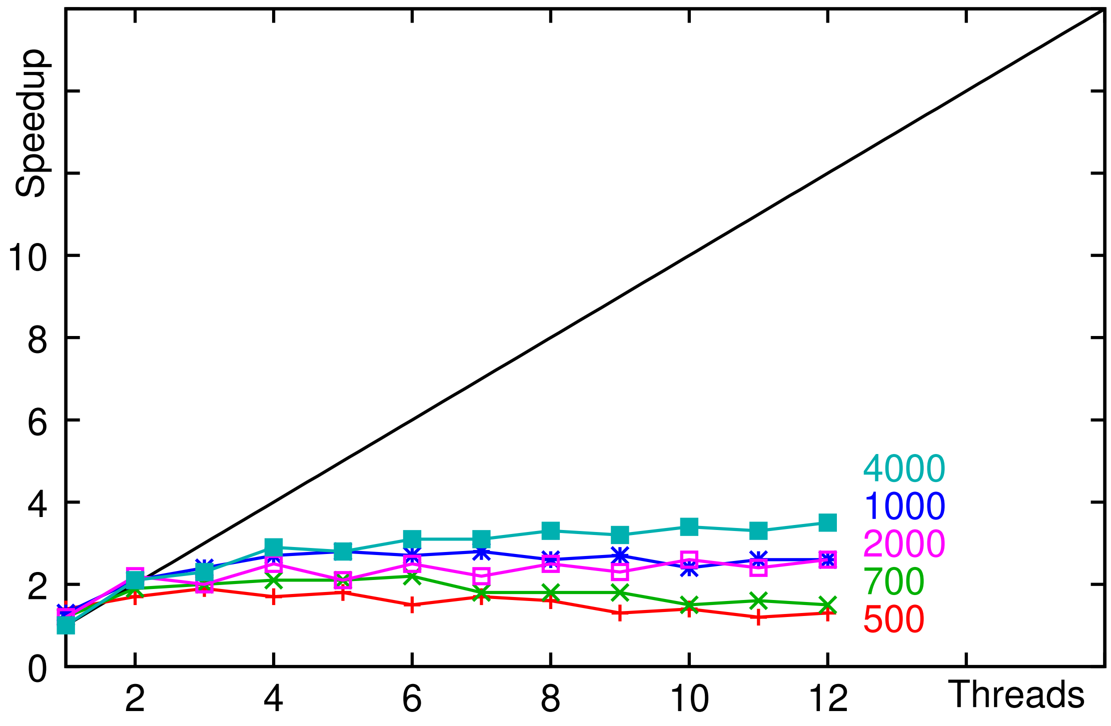
	Diagonal traversal					Pipelined parallelization				
Thr.	500	700	1000	2000	4000	500	700	1000	2000	4000
1	1.8	2.0	1.6	1.6	1.3	1.0	1.0	1.0	1.0	1.0
2	3.5	3.7	2.1	2.6	2.6	1.9	1.9	1.9	1.9	1.9
3	4.0	4.4	2.5	2.7	3.1	2.7	2.7	2.7	2.6	2.7
4	4.1	4.8	3.0	3.0	3.5	2.4	3.3	3.5	3.2	3.3

3.3 Exercise: The Jacobi and Gauss/Seidel Methods ...

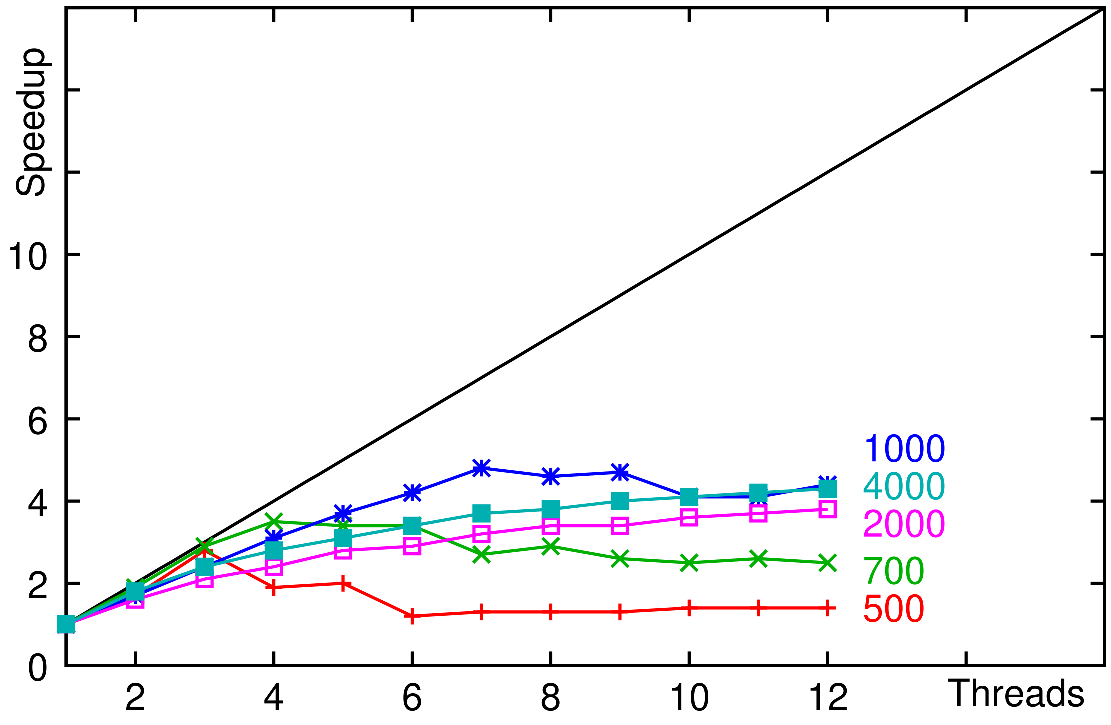
Speedup on the HorUS cluster: Jacobi



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



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



3.4 OpenMP Synchronization

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

```
int j = 0;

#pragma omp parallel for
for (int 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!

3.4 OpenMP Synchronization ...

Synchronization in OpenMP

- Higher-level, easy to use constructs
- Implementation using directives:
  - `critical`: critical section
  - `atomic`: atomic operations
  - `ordered`: execution in program order
  - `barrier`: barrier
  - `single` and `master`: execution by a single thread
  - `taskwait` and `taskgroup`: wait for tasks (☞ **3.5.2**)
  - `flush`: make the memory consistent
    - memory barrier (☞ **2.4.2**)
    - implicitly executed with the other synchronization directives

3.4 OpenMP Synchronization ...

3.4.1 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

3.4 OpenMP Synchronization ...

3.4.2 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 247:

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.

3.4.2 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
- With OpenMP 4, an atomic *compare-and-swap* can not yet be implemented
  - use builtin functions of the compiler, if necessary
  - (OpenMP 5.1 introduces a `compare` clause)

Notes for slide 248:

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; }
```

3.4 OpenMP Synchronization ...

3.4.3 Reduction operations

- Often loops aggregate values, e.g.:

```
int a[N];
int sum = 0;
#pragma omp parallel for reduction(+: sum)
for (int 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

3.4.3 Reduction operations ...

- In the example, the `reduction` option transforms the loop like this:

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

No barrier at the end of the loop

Add local partial sum to the global sum

3.4 OpenMP Synchronization ...

3.4.4 Execution in program order

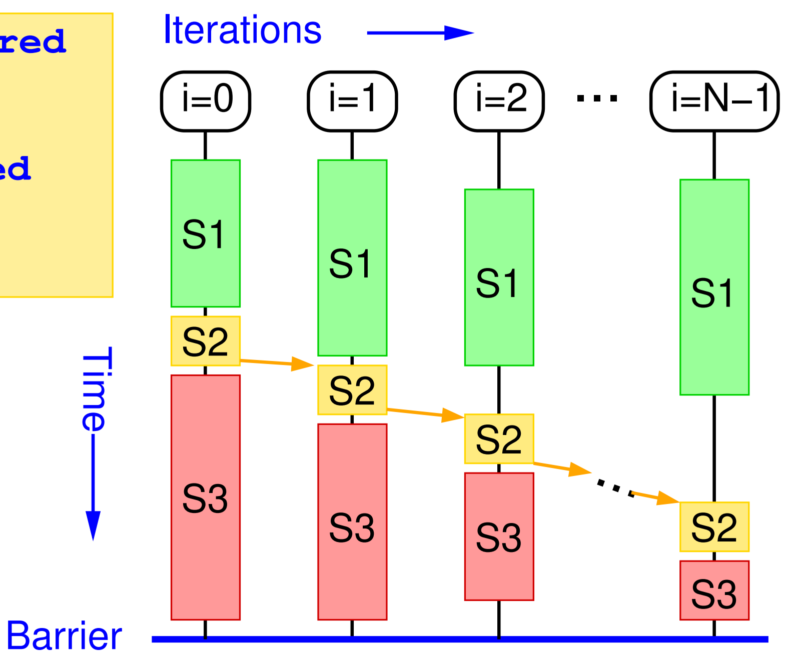
```
#pragma omp for ordered
for(...) {
    ...
    #pragma omp ordered
    Statement / Block
    ...
}
```

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

3.4.4 Execution in program order ...

Execution with `ordered`

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



3.4.4 Execution in program order ...

Execution with `ordered` ...

- Since OpenMP 4.5: `ordered` also allows to explicitly specify dependencies that must be met
- Example:

```
#pragma omp parallel for ordered(1)
for (int i=3; i<100; i++) {
  #pragma omp ordered depend(source)
  a[i] = ...;
  #pragma omp ordered depend(sink: i-3)
  ... = a[i-3];
}
```

- Argument of `ordered`: number of nested loops to be considered
  - allows to specify dependencies in nested loops
  - e.g.: `...(sink: i-1,j)`

Notes for slide 253:

Example for a nested loop with dependencies:

```
#pragma omp parallel for ordered(2)
for (int i=1; i<100; i++) {
  for (int j=1; j<100; j++) {
    #pragma omp ordered depend(source)
    a[i][j] = ...;
    #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
    ... = a[i-1][j] + a[i][j-1];
  }
}
```

In an analogous way, the `ordered` directive allows to parallelize the Gauss/Seidel-method in a pipeline style (☞ **page 239**).

3.4 OpenMP Synchronization ...

3.4.5 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`

3.4.5 Barrier ...

Example (☞ 03/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";
```

3.4.5 Barrier ...

```
#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";
  }
}
```

3.4.5 Barrier ...

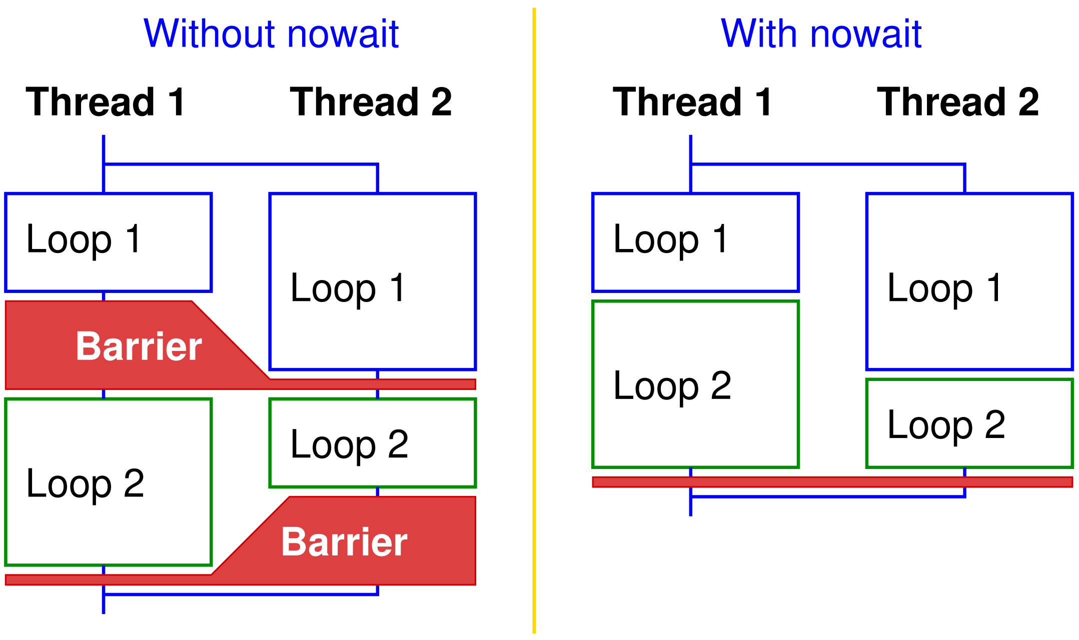
Example ...

- The first loop processes the lower triangle of the matrix `a`, the second loop processes the upper 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

3.4.5 Barrier ...

Example ...

- Executions of the program:



3.4 OpenMP Synchronization ...

3.4.6 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 259:

Strictly speaking, the `single` directive is no Synchronization, but a directive for work distribution. It distributes the work in such a way, that the block below the directive is executed by the first thread arriving at the directive. Thus, the directive can be used to implement task parallelism, e.g.:

```
#pragma omp parallel
{
  #pragma omp single nowait
  firstTask();
  #pragma omp single nowait
  secondTask();
}
```

Parallel Processing

Winter Term 2024/25

25.11.2024

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

Stand: January 13, 2025

3.5 Task Parallelism with OpenMP

3.5.1 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 thread
  - scheduling is implementation-defined (`gcc`: dynamic)
- At the end of the `sections` directive, a barrier synchronization is performed
  - unless the option `nowait` is specified

3.5.1 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

3.5.1 The sections directive ...

Example: scheduling / influence of nowait (☞ 03/sections.cpp)

```
void task(int no, int delay) {
  int thread = omp_get_thread_num();
  #pragma omp critical
  cout << "Thread " << thread << ", Section " << no << " start\n";
  usleep(delay);
  #pragma omp critical
  cout << "Thread " << thread << ", Section " << no << " end\n";
}

main() {
  #pragma omp parallel
  {
    #pragma omp sections // ggf. nowait
    {
      #pragma omp section
      task(1, 200000);
```

3.5.1 The sections directive ...

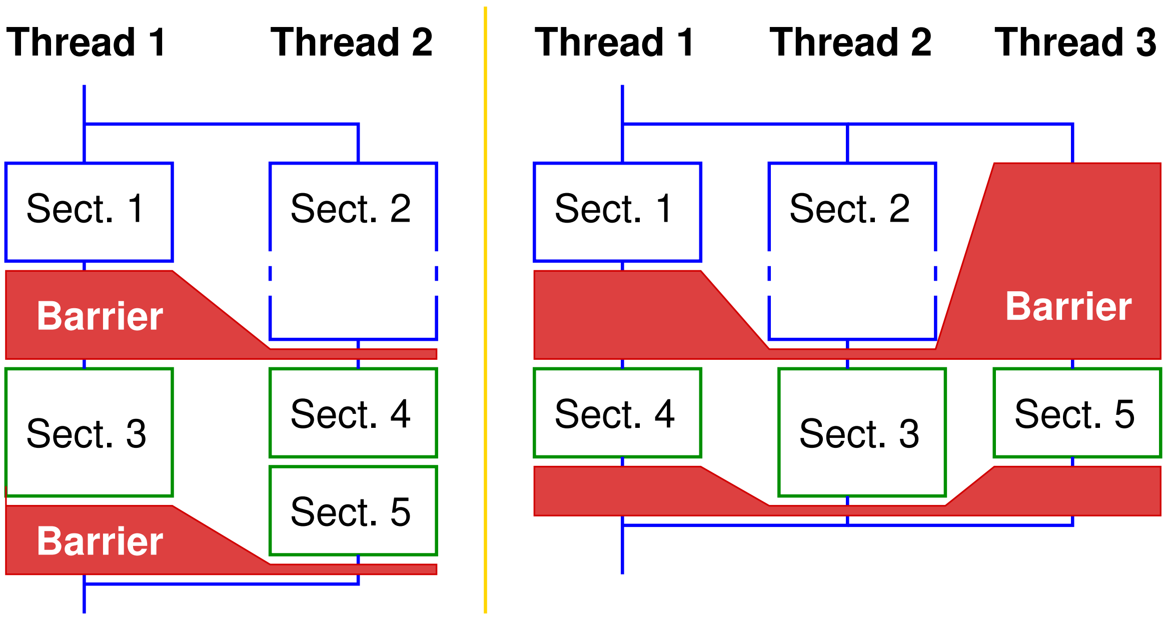
Example: scheduling / influence of nowait ...

```
    #pragma omp section
      task(2, 1000000);
    }
    #pragma omp sections
    {
      #pragma omp section
      task(3, 300000);
      #pragma omp section
      task(4, 200000);
      #pragma omp section
      task(5, 200000);
    }
  }
}
```

3.5.1 The `sections` directive ...

Example: scheduling / influence of `nowait` ...

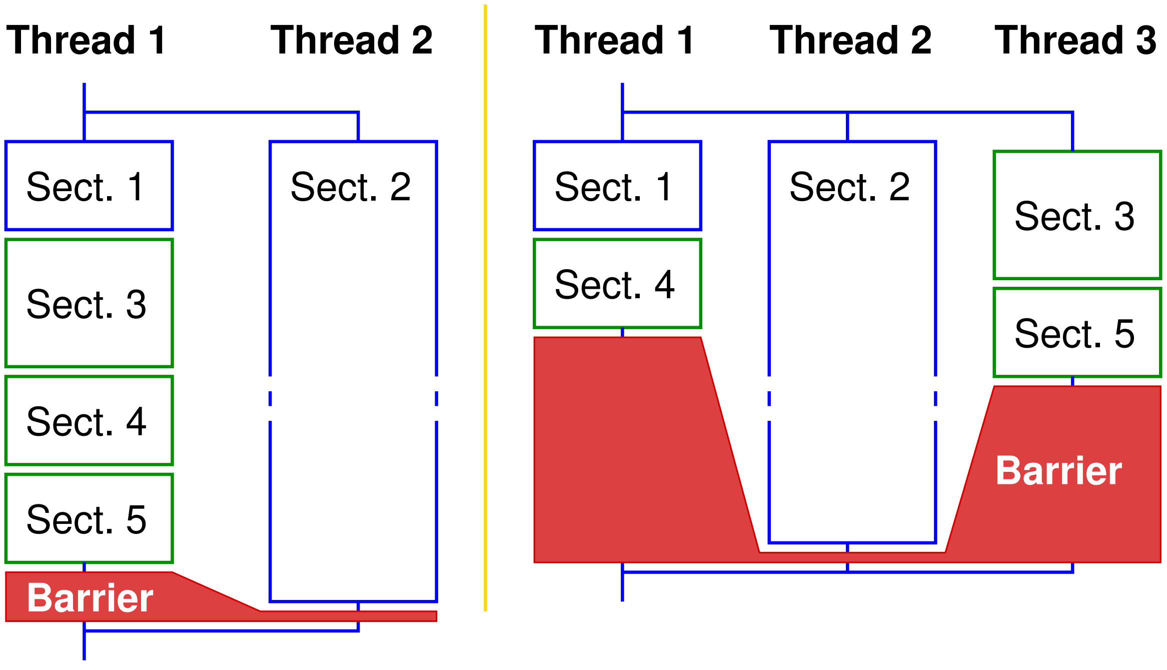
➥ Executions of the program **without** `nowait` option:



3.5.1 The `sections` directive ...

Example: scheduling / influence of `nowait` ...

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



3.5 Task Parallelism with OpenMP ...

3.5.2 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
  - the default for local variables is `firstprivate`, i.e., local variables declared outside but used inside the block are the task's input arguments
- Option `if` allows to determine, when an explicit task should be created

3.5.2 The `task` Directive ...

Example: parallel quicksort (☞ `03/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 267:

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 copied 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
```

3.5.2 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 tasks, which have been created within the block by the current task or one of its subtasks
  - available since OpenMP 4.0
  - caution: older compilers ignore this directive!

3.5.2 The `task` Directive ...

Example: parallel quicksort (☞ `03/qsort.cpp`)

- Imagine the following change when calling quicksort:

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

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

3.5.2 The `task` Directive ...

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 270:

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.

3.5.2 The `task` Directive ...

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);
}
```

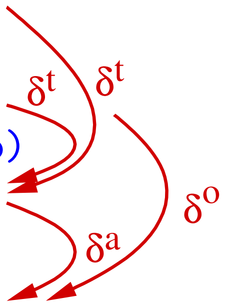
  - advantage: only wait at one single place
  - disadvantage: semantics of `quicksort()` must be very well documented

3.5.2 The task Directive ...

Dependences between tasks (☞ 03/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 272:

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.

3.6 Tutorial: Tools for OpenMP

3.6.1 Debugging

- There are only few debuggers that fully support OpenMP
  - e.g., Totalview
  - requires tight cooperation between compiler and debugger
- On Linux PCs:
  - `gdb` and `ddd` allow halfway reasonable debugging
    - they support multiple threads
  - `gdb`: textual debugger (standard LINUX debugger)
  - `ddd`: graphical front end for `gdb`
    - more comfortable, but more "heavy-weight"

3.6.1 Debugging ...

- Prerequisite: compilation with debugging information
  - sequential: `g++ -g -o myProg myProg.cpp`
  - with OpenMP: `g++ -g -fopenmp ...`
- Limited(!) debugging is also possible in combination with optimization
  - however, the debugger may show unexpected behavior
  - if possible: switch off the optimization
    - `g++ -g -O0 ...`

3.6.1 Debugging ...

Important functions of a debugger (Examples for `gdb`):

- Start the programm: `run arg1 arg2`
- Set breakpoints on code lines: `break file.cpp:35`
- Set breakpoints on functions: `break myFunc`
- Show the procedure call stack: `where`
- Navigate in the procedure call stack: `up` bzw. `down`
- Show the contents of variables: `print i`
- Change the contents of variables: `set variable i=i*15`
- Continue the program (after a breakpoint): `continue`
- Single-step execution: `step` bzw. `next`

3.6.1 Debugging ...

Important functions of a debugger (Examples for `gdb`): ...

- Show all threads: `info threads`
- Select a thread: `thread 2`
  - subsequent commands typically only affect the selected thread
- Source code listing: `list`
- Help: `help`
- Exit the debugger: `quit`

- All commands can also be abbreviated in `gdb`

3.6.1 Debugging ...

Sample session with gdb (sequential)

```
bsclk01> g++ -g -O0 -o ross ross.cpp   ← Option -g for debugging
bsclk01> gdb ./ross
GNU gdb 6.6
Copyright 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public ...
(gdb) b main   ← Set breakpoint on function main
Breakpoint 1 at 0x400d00: file ross.cpp, line 289.
(gdb) run 5 5 0   ← Start program with given arguments
Starting program: /home/wismueller/LEHRE/pv/ross 5 5 0
Breakpoint 1, main (argc=4, argv=0x7fff0a131488) at ross.cpp:289
289     if (argc != 4) {
(gdb) list   ← Listing around the current line
284
285     /*
286     ** Get and check the command line arguments
```

3.6.1 Debugging ...

```
287     */
288
289     if (argc != 4) {
290         cerr << "Usage: ross <size_x> <size_y> ...
291         cerr << "       <size_x> <size_y>: size...
292         cerr << "       <all>: 0 = compute one ...
293         cerr << "              1 = compute all ...
(gdb) b 315   ← Set breakpoint on line 315
Breakpoint 2 at 0x400e59: file ross.cpp, line 315.
(gdb) c   ← Continue the program
Continuing.
Breakpoint 2, main (argc=4, argv=0x7fff0a131488) at ross.cpp:315
315         num_moves = Find_Route(size_x, size_y, moves);
(gdb) n   ← Execute next source line (here: 315)
320         if (num_moves >= 0) {
(gdb) p num_moves   ← Print contents of num_moves
$1 = 24
```

3.6.1 Debugging ...

```
(gdb) where    ← Where is the program currently stopped?
#0  main (argc=4, argv=0x7fff0a131488) at ross.cpp:320
(gdb) c    ← Continue program
Continuing.
Solution:
  ...
Program exited normally.
(gdb) q    ← exit gdb
bsclk01>
```

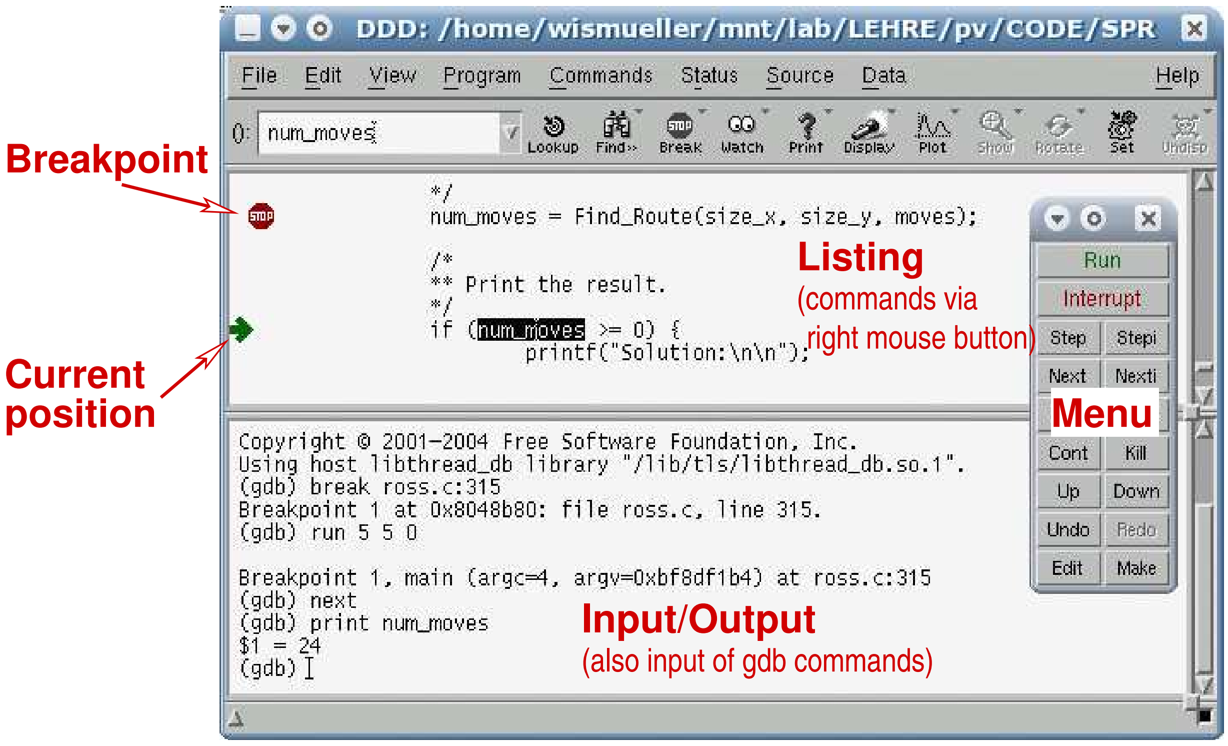
3.6.1 Debugging ...

Sample session with gdb (OpenMP)

```
bslab03> g++ -fopenmp -O0 -g -o heat heat.cpp solver-jacobi.cpp
bslab03> gdb ./heat
GNU gdb (GDB) SUSE (7.5.1-2.1.1)
...
(gdb) run 500
...
Program received signal SIGFPE, Arithmetic exception.
0x0000000000401711 in solver._omp_fn.0 () at solver-jacobi.cpp:58
58                                b[i][j] = i/(i-100);
(gdb) info threads
  Id   Target Id          Frame
  4    Thread ... (LWP 6429) ... in ... at solver-jacobi.cpp:59
  3    Thread ... (LWP 6428) ... in ... at solver-jacobi.cpp:59
  2    Thread ... (LWP 6427) ... in ... at solver-jacobi.cpp:63
* 1    Thread ... (LWP 6423) ... in ... at solver-jacobi.cpp:58
(gdb) q
```

3.6.1 Debugging ...

Sample session with `ddd`


DDD: /home/wismueller/mnt/lab/LEHRE/pv/CODE/SPR
File Edit View Program Commands Status Source Data Help
(): num_moves
Lookup Find» Break Watch Print Display Plot Show Rotate Set Undisp
Breakpoint
Current position
Listing
(commands via right mouse button)
Run Interrupt Step Stepi Next Nexti Cont Kill Up Down Undo Redo Edit Make
Menu
Input/Output
(also input of gdb commands)

```
*/
num_moves = Find_Route(size_x, size_y, moves);

/*
** Print the result.
*/
if (num_moves >= 0) {
        printf("Solution:\n\n");
```

```
Copyright © 2001-2004 Free Software Foundation, Inc.
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) break ross.c:315
Breakpoint 1 at 0x8048b80: file ross.c, line 315.
(gdb) run 5 5 0

Breakpoint 1, main (argc=4, argv=0xbf8df1b4) at ross.c:315
(gdb) next
(gdb) print num_moves
$1 = 24
(gdb)
```

3.6 Tutorial: Tools for OpenMP ...

3.6.2 Performance Analysis

- Typically: **instrumentation** of the generated executable code during/after the compilation
  - insertion of code at important places in the program
    - in order monitor relevant events
    - e.g., at the beginning/end of parallel regions, barriers, ...
  - during the execution, the events will be
    - individually logged in a **trace file** (***Spurdatei***)
    - or already summarized into a **profile**
  - Evaluation is done after the program terminates
  - c.f. Section 2.8.6
- Example: Scalasca
  - see `https://www.scalasca.org/scalasca/software`

Notes for slide 282:

If you want to use Scalasca, there are two possibilities:

- You can download an appliance for Oracle VirtualBox, which includes Linux, g++ compilers, OpenMP, MPI, Scalasca and Visual Studio Code with g++ plugins (see `https://moodle.uni-siegen.de/mod/url/view.php?id=884597`).
- You can use the script, which is provided on the course's web page (see `https://www.bs.informatik.uni-siegen.de/web/wismueller/vl/pv/build-scalasca.sh`) to download and build Scalasca on a Linux computer.

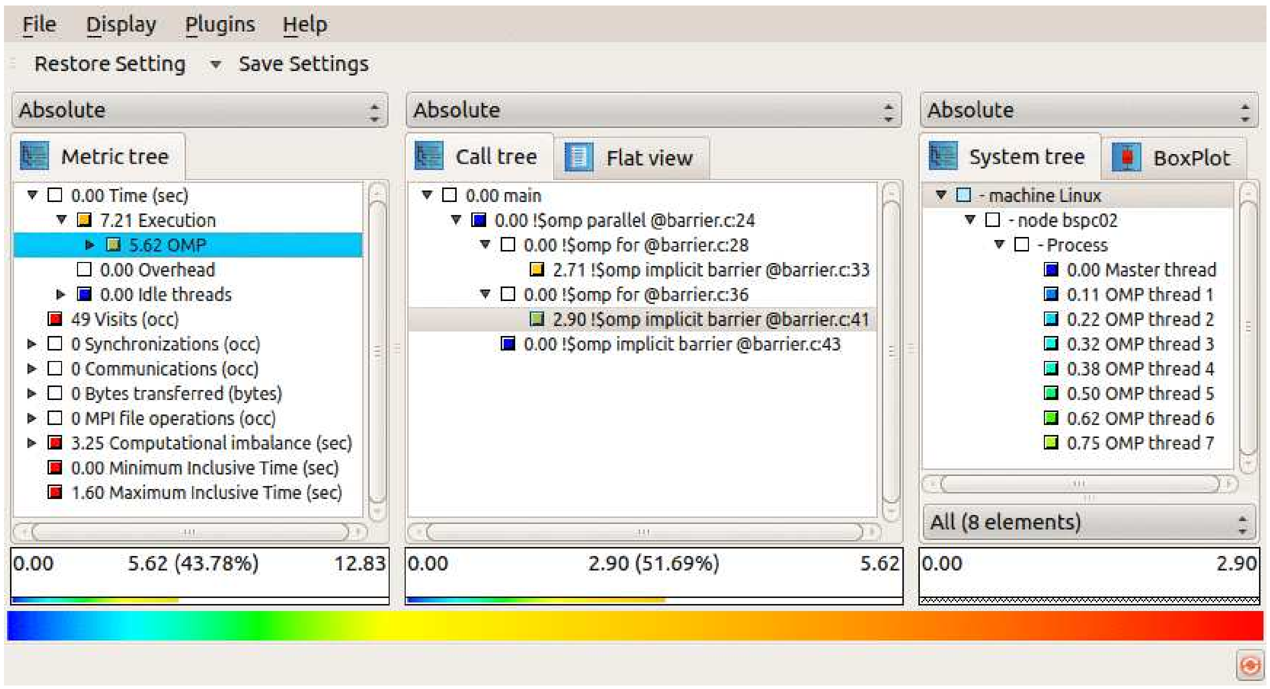
3.6.2 Performance Analysis ...

Performance analysis using Scalasca

- Compile the program:
  - `scalasca -instrument g++ -fopenmp ...  barrier.cpp`
- Execute the program:
  - `scalasca -analyze ./barrrier`
  - stores data in a directory `scorep_barrier_0x0_sum`
    - `0x0` indicates the number of threads (0 = default)
    - directory must not yet exist; remove it, if necessary
- Interactive analysis of the recorded data:
  - `scalasca -examine scorep_barrier_0x0_sum`

3.6.2 Performance Analysis ...

Performance analysis using Scalasca: Example from slide 255

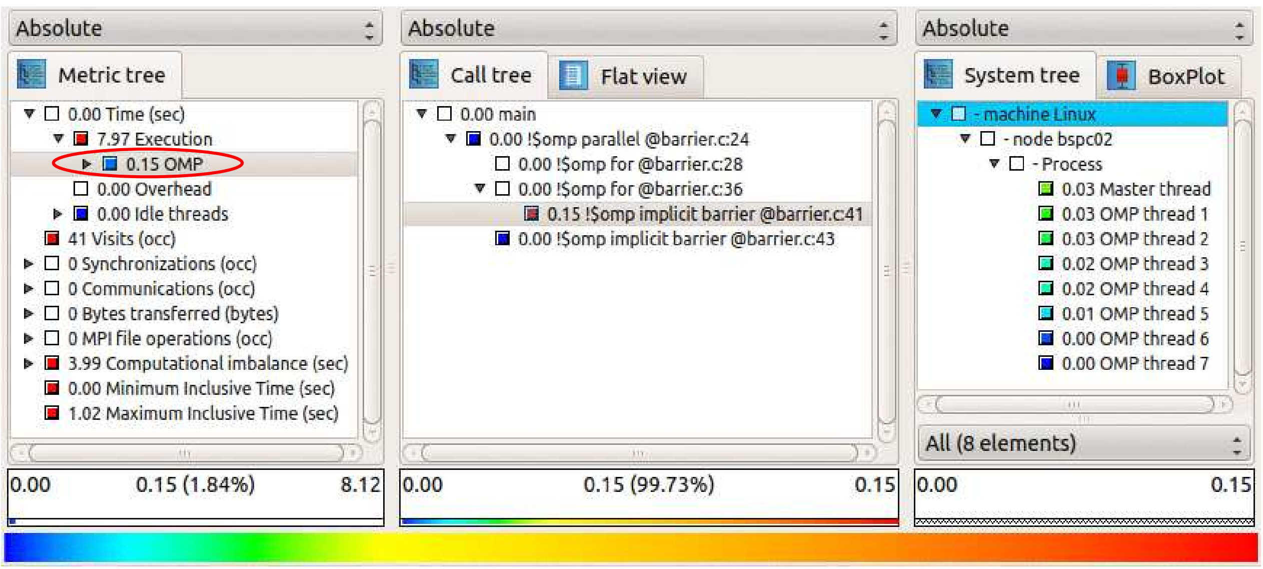


```
File   Display   Plugins   Help
Restore Setting ▾  Save Settings

Absolute                          Absolute                                   Absolute
Metric tree                       Call tree | Flat view                      System tree | BoxPlot
▼ □ 0.00 Time (sec)               ▼ □ 0.00 main                              ▼ □ - machine Linux
  ▼ 7.21 Execution                  ▼ 0.00 !$omp parallel @barrier.c:24        ▼ □ - node bspc02
    ▶ 5.62 OMP                        ▼ □ 0.00 !$omp for @barrier.c:28           ▼ □ - Process
    □ 0.00 Overhead                       2.71 !$omp implicit barrier @barrier.c:33   0.00 Master thread
  ▶ 0.00 Idle threads                 ▼ □ 0.00 !$omp for @barrier.c:36                0.11 OMP thread 1
  49 Visits (occ)                         2.90 !$omp implicit barrier @barrier.c:41   0.22 OMP thread 2
▶ □ 0 Synchronizations (occ)          0.00 !$omp implicit barrier @barrier.c:43       0.32 OMP thread 3
▶ □ 0 Communications (occ)                                                             0.38 OMP thread 4
▶ □ 0 Bytes transferred (bytes)                                                        0.50 OMP thread 5
▶ □ 0 MPI file operations (occ)                                                        0.62 OMP thread 6
▶ 3.25 Computational imbalance (sec)                                                   0.75 OMP thread 7
  0.00 Minimum Inclusive Time (sec)                                          All (8 elements)
  1.60 Maximum Inclusive Time (sec)

0.00   5.62 (43.78%)   12.83      0.00   2.90 (51.69%)   5.62      0.00              2.90
```

3.6.2 Performance Analysis ...

Performance analysis using Scalasca: Example from slide 255 ...

- ➥ In the example, the waiting time at barriers in the first loop can be reduced drastically by using the option `nowait`:



```
Absolute                          Absolute                                   Absolute
Metric tree                       Call tree | Flat view                      System tree | BoxPlot
▼ □ 0.00 Time (sec)               ▼ □ 0.00 main                              ▼ □ - machine Linux
  ▼ 7.97 Execution                  ▼ 0.00 !$omp parallel @barrier.c:24        ▼ □ - node bspc02
    ▶ 0.15 OMP                        □ 0.00 !$omp for @barrier.c:28             ▼ □ - Process
    □ 0.00 Overhead                 ▼ □ 0.00 !$omp for @barrier.c:36                0.03 Master thread
  ▶ 0.00 Idle threads                     0.15 !$omp implicit barrier @barrier.c:41   0.03 OMP thread 1
  41 Visits (occ)                     0.00 !$omp implicit barrier @barrier.c:43       0.03 OMP thread 2
▶ □ 0 Synchronizations (occ)                                                           0.02 OMP thread 3
▶ □ 0 Communications (occ)                                                             0.02 OMP thread 4
▶ □ 0 Bytes transferred (bytes)                                                        0.01 OMP thread 5
▶ □ 0 MPI file operations (occ)                                                        0.00 OMP thread 6
▶ 3.99 Computational imbalance (sec)                                                   0.00 OMP thread 7
  0.00 Minimum Inclusive Time (sec)                                          All (8 elements)
  1.02 Maximum Inclusive Time (sec)

0.00   0.15 (1.84%)   8.12      0.00   0.15 (99.73%)   0.15      0.00              0.15
```

Notes for slide 285:

When interpreting the times indicated by Scalasca, the following must be observed:

- The metric displayed for an entry (here: time) always excludes the visible sub-entries. When, e.g., the item "7.97 Execution" in the *Metric tree* shown in the screen dump is folded (i.e., no longer visible), Scalasca displays "8.12 Execution" (0.15s execution time for OMP + 7.97s for the remaining execution).

  In the example, you can see that the `nowait` option has made the time for OpenMP (synchronization) significantly smaller (0.15s instead of 5.62s), but the pure execution time has slightly increased (from 7.21s to 7.97s), possibly because of competition for the memory.

- The time that Scalasca displays is the **summed execution time of all threads**, including waiting times. In the example, the program actually terminated after 1.3s.

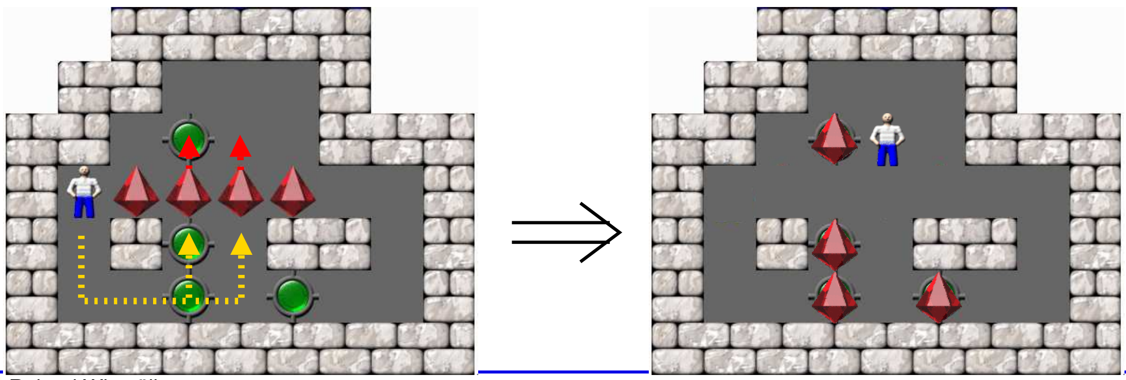
- Scalasca still shows a load imbalance (*Computational imbalance*), since, e.g., thread 7 still calculates much more in the first loop than thread 1. Scalasca is not able to recognize that this imbalance exactly cancels the corresponding imbalance in the second loop.

3.7 Exercise: A Solver for the Sokoban Game

(Animated slide)

Background

- Sokoban: japanese for "warehouse keeper"
- Computer game, developed in 1982 by Hiroyuki Imabayashi
- Goal: player must push all objects (boxes) to the target positions (storage locations)
  - boxes can only be pushed, not pulled
  - only one box can be pushed at a time

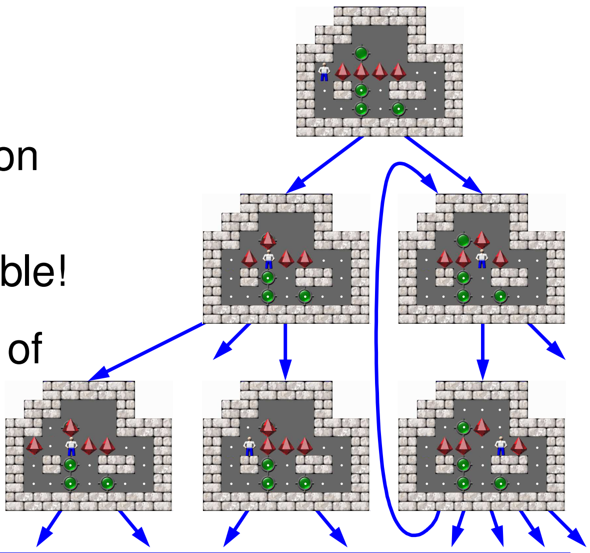


3.7 Exercise: A Solver for the Sokoban Game ...

(Animated slide)

How to find the sequence of moves?

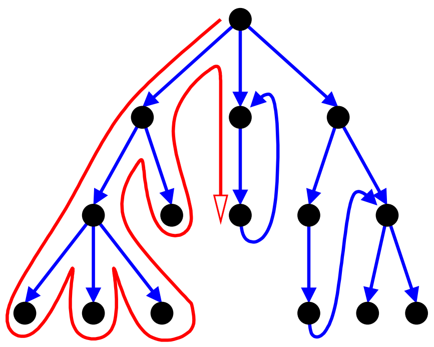
- ➥ Configuration: state of the play field
  - ➥ positions of the boxes
  - ➥ position of the player (connected component)
- ➥ Each configuration has a set of successor configurations
- ➥ Configurations with successor relation build a directed graph
  - ➥ not a tree, since cycles are possible!
- ➥ Wanted: shortest path from the root of the graph to the goal configuration
  - ➥ i.e., smallest number of box pushes



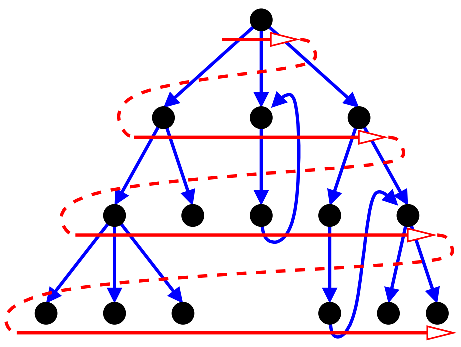
3.7 Exercise: A Solver for the Sokoban Game ...

How to find the sequence of moves? ...

- ➥ Two alternatives:
  - ➥ depth first search



  - ➥ breadth first search



  - ➥ problems (depth first search):
    - ➥ cycles
    - ➥ handling paths with different lengths
  - ➥ problems (breadth first search):
    - ➥ reconstruction of the path to a node
    - ➥ memory requirements

Parallel Processing

Winter Term 2024/25

02.12.2024

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

Stand: January 13, 2025

3.7 Exercise: A Solver for the Sokoban Game ...

***Backtracking* algorithm for depth first search:**

```
DepthFirstSearch(conf):    // conf = current configuration
    append conf to the soultion path
    if conf is a solution configuration:
        found the solution path
        return
    if depth is larger than the depth of the best solution so far:
        remove the last element from the solution path
        return      // cancel the search in this branch
    for all possible successor configurations c of conf:
        if c has not yet been visited at a smaller or equal depth:
            remember the new depth of c
            DepthFirstSearch(c)      // recursion
    remove the last element from the solution path
    return      // backtrack
```

3.7 Exercise: A Solver for the Sokoban Game ...

Algorithm for breadth first search:

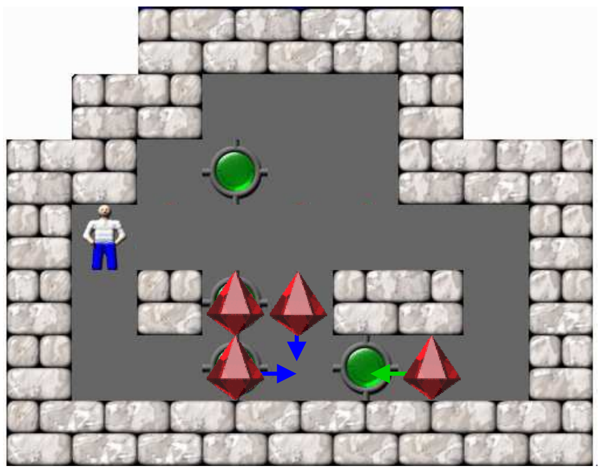
```
BreadthFirstSearch(conf):    // conf = start configuration
  add conf to the queue at depth 0
  depth = 1;
  while the queue at depth depth-1 is not empty:
    for all configurations conf in this queue:
      for all possible successor configurations c of conf:
        if configuration c has not been visited yet:
          add the configuration c with predecessor conf to the
            set of visited configurations and to the queue for
            depth depth
          if c is a solution configuration:
            determine the solution path to c
            return      // found a solution
    depth = depth+1
  return      // no solution
```

3.7 Exercise: A Solver for the Sokoban Game ...

(Animated slide)

Example for the *backtracking* algorithm

Configuration with possible moves



← Possible move

← Chosen move

3.8 Excursion: *Lock-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* instructions instead of locks
  - in case of conflict, i.e., when there is a simultaneous change by another thread, the affected operation is repeated

3.8 Excursion: *Lock-Free* Data Structures ...

Example: appending to an array (at the end)

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

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;
}
```

3.8 Excursion: *Lock-Free* Data Structures ...

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

```
bool compare_and_swap(void **addr, void *exp, void *newVal) {
   if (*addr == exp) {
      *addr = newVal;
      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));
}
```

3.8 Excursion: *Lock-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.:
  - C++ type: `std::atomic<T>`
  - `gcc/g++`: built-in functions `__sync_...()` or `__atomic_...()`