
Parallel Processing

Winter Term 2025/26

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

Stand: November 24, 2025

Parallel Processing

Winter Term 2025/26

3 Parallel Programming with Shared Memory



Contents

- ➔ OpenMP basics
- ➔ Loop parallelization and dependences
- ➔ OpenMP synchronization
- ➔ Exercise: The Jacobi and Gauss/Seidel Methods
- ➔ Task parallelism with OpenMP
- ➔ Advanced OpenMP features
- ➔ 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



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**)



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)



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



Compiling and executing OpenMP programs

- ➔ Compilation with gcc (g++)
 - ➔ typical call: `g++ -fopenmp myProg.cpp -o myProg`
- ➔ 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



An example (👉 03/firstprog.cpp)

Program

```
main() {  
    cout << "Serial\n";  
  
    {  
        cout << "Parallel\n";  
    }  
    cout << "Serial\n";  
}
```

3.1.1 The parallel directive



An example (👉 03/firstprog.cpp)

Program

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

Compilation

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

3.1.1 The parallel directive



An example (👉 03/firstprog.cpp)

Program

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

Compilation

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

Execution

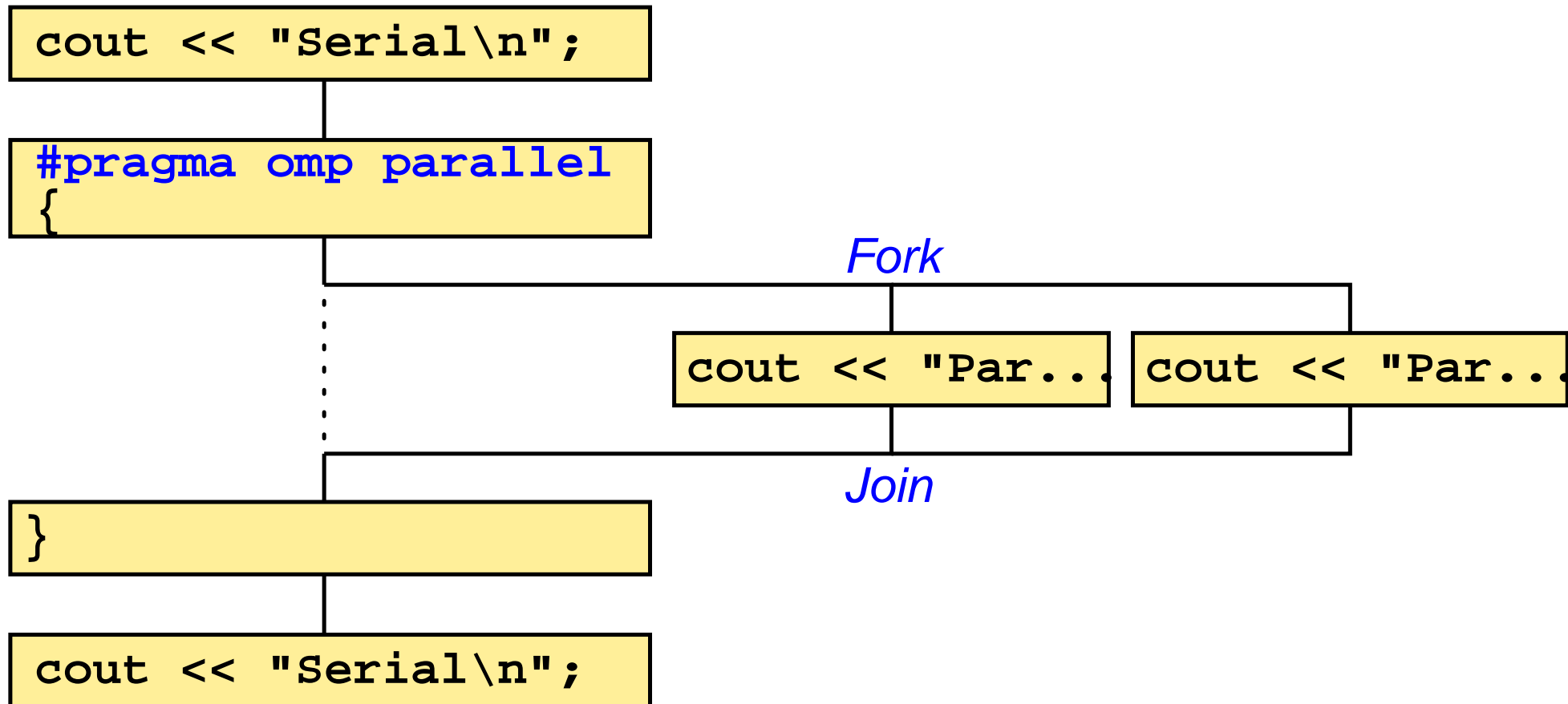
```
% export OMP_NUM_THREADS=2  
% ./firstprog  
Serial  
Parallel  
Parallel  
Serial
```

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

3.1.1 The parallel directive ...



Execution model: fork/join





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)

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)



Shared and private variables

- ➔ For variables in a parallel region there are two alternatives
 - ➔ the variable 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

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 parallel private(i) firstprivate(j)
{
    int h = random() % 100;
    cout << "P: i=" << i << ", j=" << j
          << ", k=" << k << ", h=" << h << "\n";
    i++; j++; k++;
}
```

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
```



- ➔ 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



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

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

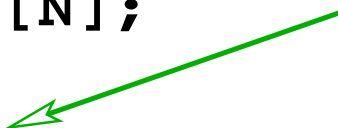
3.2.1 The `for` directive: parallel loops ...



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, ...



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 compromise between load balancing and locality (cache usage)



Shared and private variables in loops

- ➔ The `parallel for` directive can be supplemented with the options `private`, `shared` and `firstprivate` (see slide 208 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
```

Parallel Processing

Winter Term 2025/26

10.11.2025

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

Stand: November 24, 2025

3.2.2 Parallelization of Loops



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];
```

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

3.2.2 Parallelization of Loops



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 dependences 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:

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



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:

```
for(i=0;i<N;i++)  
    a[i] = a2[i+1] + b[i];
```



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:

```
for(i=0;i<N;i++)
    a2[i+1] = a[i+1];

for(i=0;i<N;i++)
    a[i] = a2[i+1] + b[i];
```



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=0;i<N;i++)
    a2[i+1] = a[i+1];
  #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 (☞ **3.3**):

```
#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 (☞ **3.6.2**), e.g.:

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

3.2.3 Simple Examples



Matrix addition

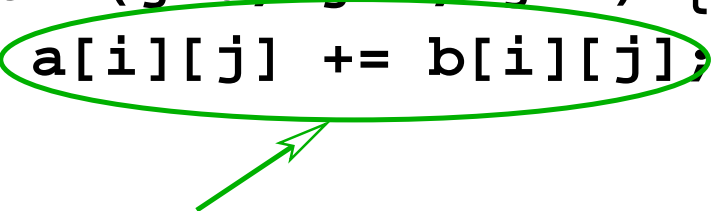
```
double a[N][N];
double b[N][N];
int i,j;

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

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];
    }
}
```



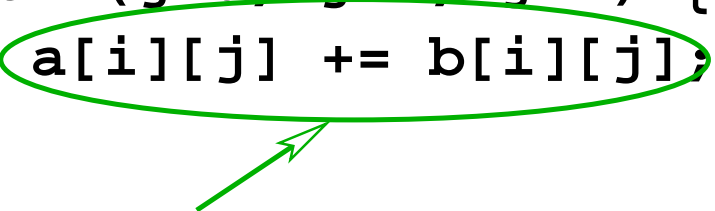
No dependences in 'j' loop:

- 'b' is read only
- Elements of 'a' are always read in the same 'j' iteration, in which they are written

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];
  }
}
```

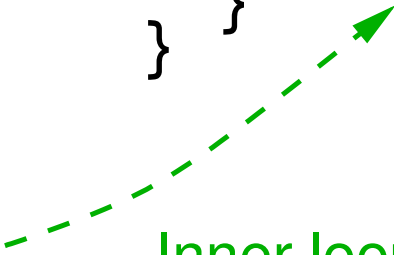


No dependences in 'j' loop:

- 'b' is read only
- Elements of 'a' are always read in the same 'j' iteration, in which they 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];
  }
}
```

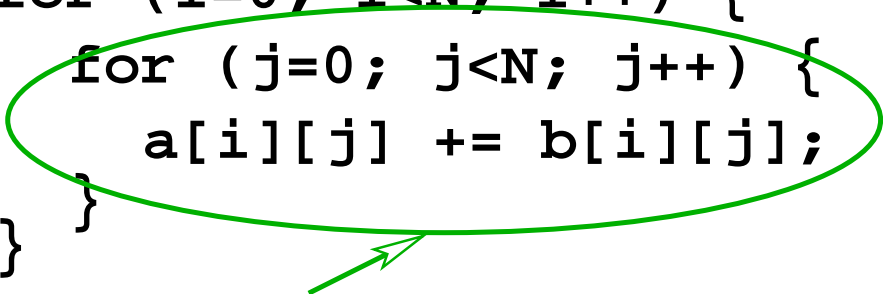


Inner loop can be executed in parallel

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];
  }
}
```



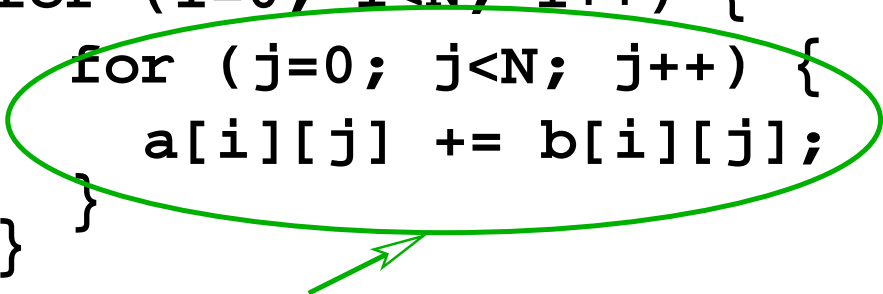
No dependences in 'i' loop:

- 'b' is read only
- Elements of 'a' are always read in the same 'i' iteration, in which they are written

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];
  }
}
```

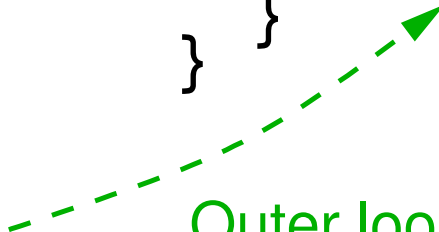


No dependences in 'i' loop:

- '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!



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 dependence 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



Handling 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];
```



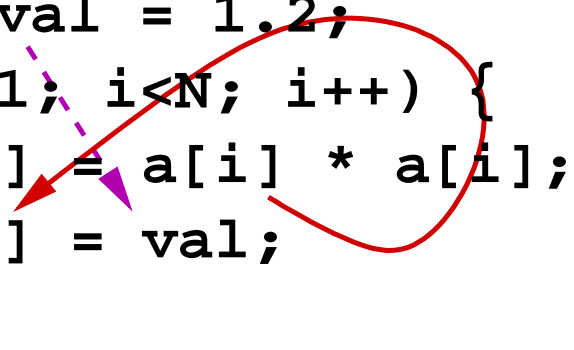
Handling dependences

```
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];
```

Anti depend. between iterations

Handling 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];
```

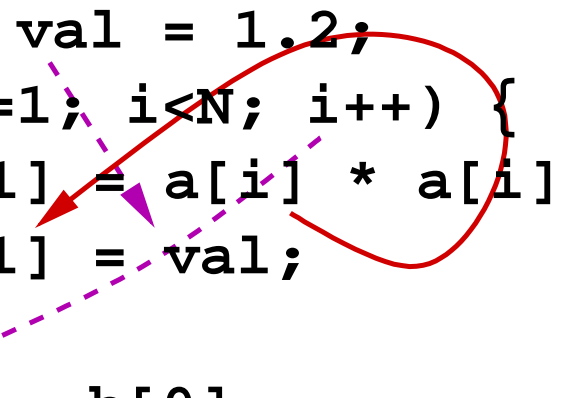


Anti depend. between iterations

True dependece between
loop and environment

Handling 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];
```

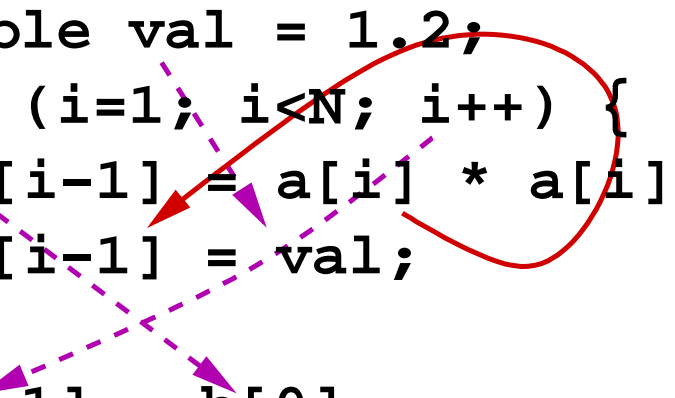


Anti depend. between iterations

True dependece between
loop and environment

Handling 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];
```



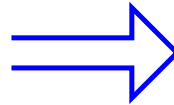
Anti depend. between iterations

True dependece between
loop and environment



Handling 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], 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



Renaming + barrier

True dependece between loop and environment



lastprivate(i) + barriers



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;  
}
```

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

```
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];  
  }  
}
```

3.2.4 Dependence Analysis in Loops



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;  
}
```

$S1 \xrightarrow{\delta_{(=)}^t} S2$

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;  
}
```

```
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];  
  }  
}
```

3.2.4 Dependence Analysis in Loops



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;  
}
```

$S1 \xrightarrow{\delta_{(=)}^t} S2$

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;  
}
```

S1 in earlier iteration than S2

$S1 \xrightarrow{\delta_{(<)}^t} S2$

Loop carried dependence

```
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];  
  }  
}
```

3.2.4 Dependence Analysis in Loops



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;  
}
```

$S1 \xrightarrow{\delta_{(=)}^t} S2$

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 dependence

S1 in earlier iteration than S2

$S1 \xrightarrow{\delta_{(\leq)}^t} S2$

```
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];  
  }  
}
```

$S1 \xrightarrow{\delta_{(=,=)}^a} S2$

3.2.4 Dependence Analysis in Loops



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;  
}
```

$S1 \xrightarrow{\delta_{(=)}^t} S2$

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 dependence

S1 in earlier iteration than S2

$S1 \xrightarrow{\delta_{(<)}^t} S2$

```
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

$S1 \xrightarrow{\delta_{(<, <)}^t} S2$

$S1 \xrightarrow{\delta_{(=, =)}^a} S2$



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:

$$0 \leq i_1 < 10$$

$$0 \leq i_2 < 10$$

$$0 \leq j_1 \leq i_1$$

$$0 \leq j_2 \leq i_2$$

$$10 i_1 + j_1 = 20 i_2 + j_2 - 1$$

➔ Dependence analysis always is a conservative approximation!

➔ unknown loop bounds, non-linear index expressions, pointers (aliasing), ...



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 $(\dots, =, \dots)$ or $(\dots, \neq, \dots, *, \dots)$ [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 $(\dots, <, >, \dots)$

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)
```

```
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)
```

*Loop
interchange*

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 $S1 \xrightarrow{\delta_{(<) }^t} S2$, which prevents parallelization of the loop without synchronization

→ However, since we do *not* have any dependence $S2 \xrightarrow{\delta_{(<) }} S1$, loop splitting is permitted, which results in:

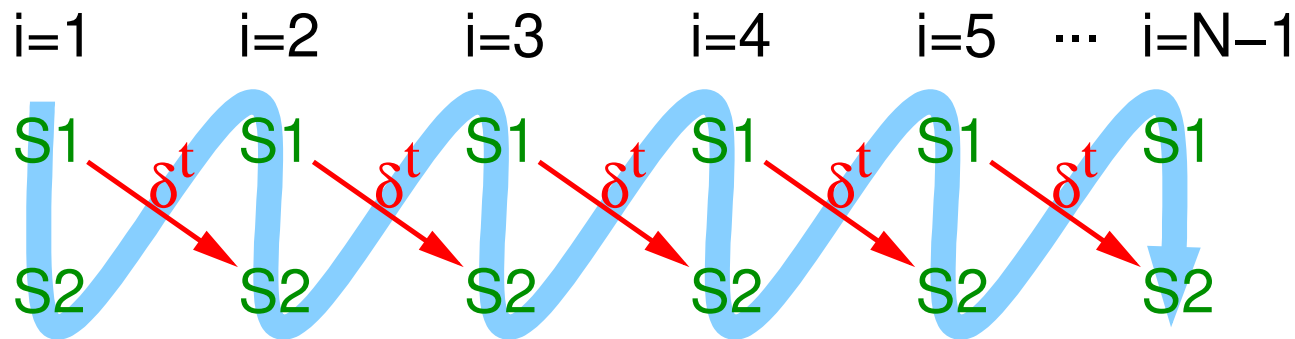
```
for (i=1; i<N-1; i++)  
    a[i] = (c[i-1] + c[i+1])/2; // S1  
for (i=1; i<N-1; i++)  
    b[i] = a[i-1];             // S2
```

3.2.4 Dependence Analysis in Loops ...

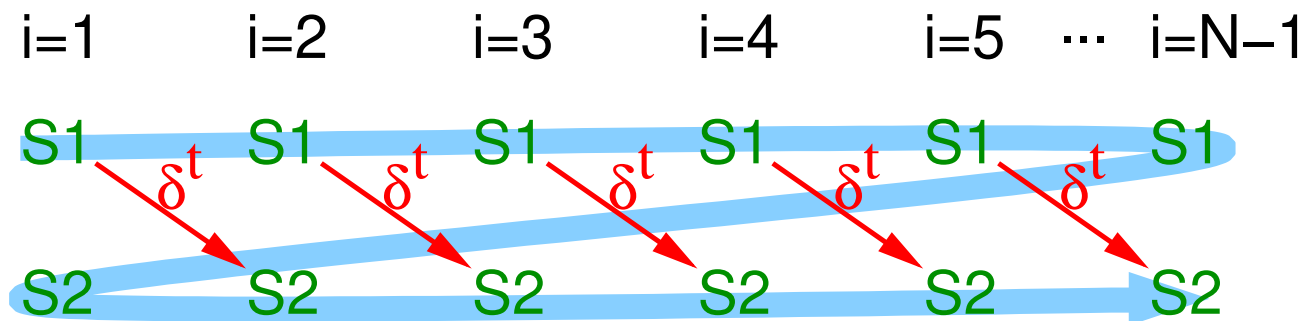


Example: loop splitting ...

➔ Execution of the original loop:



➔ Execution of the transformed loop:



3.3 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 \Rightarrow nondeterministic errors!



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



3.3.1 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

3.3.2 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 instructions
- ➔ 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`)



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)

3.3.3 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

3.3.3 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 ← No barrier at the end
    for (int i=0; i<N; i++) { of the loop
        lsum += a[i];
    }
    # pragma omp atomic
    sum += lsum; ← Add local partial sum
}
printf("sum=%d\n", sum);
```

3.3.4 Execution in program order

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

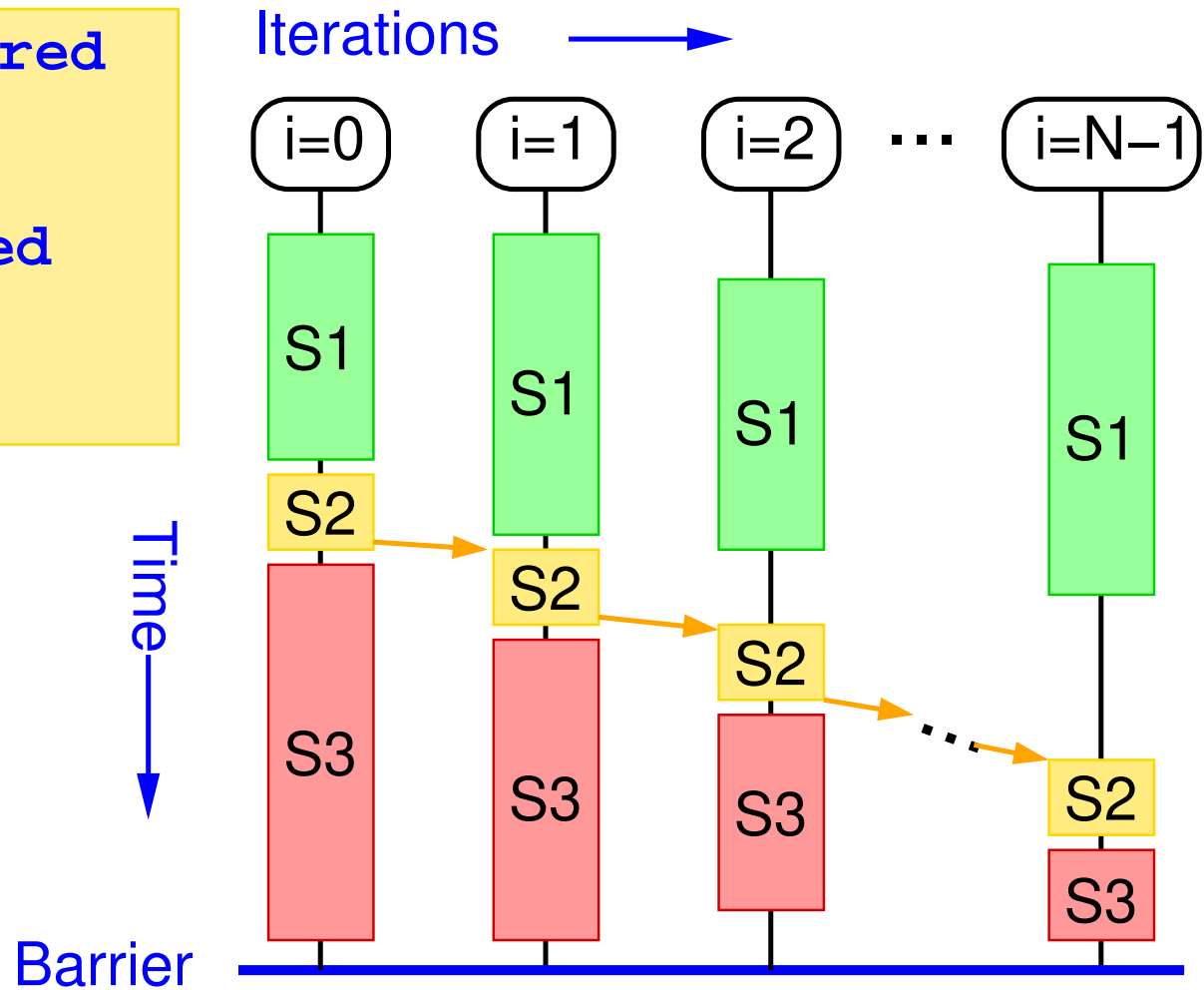
- ➔ The ordered directive is only allowed in the dynamic extent of a for directive with option ordered
 - ➔ recommendation: use option `schedule(static,1)`
 - ➔ or `schedule(static,n)` with small n
- ➔ The threads will execute the instances of the statement / block exactly in the same order as in the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

3.3.4 Execution in program order ...



Execution with ordered

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





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)`

3.3.5 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`

3.3.5 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";
    }
}
```

Parallel Processing

Winter Term 2025/26

17.11.2025

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

Stand: November 24, 2025

3.3.5 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";
}
}
```



Example ...

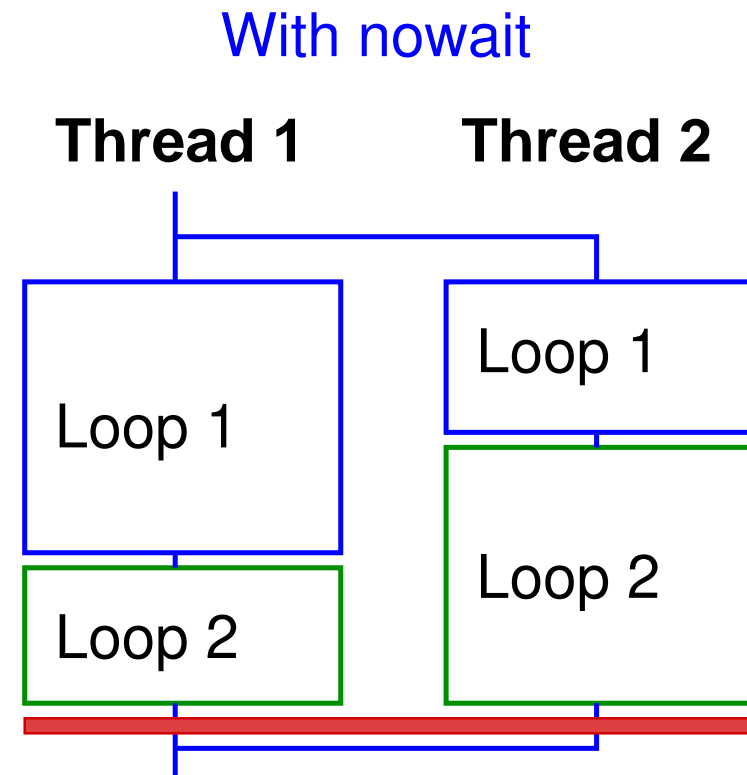
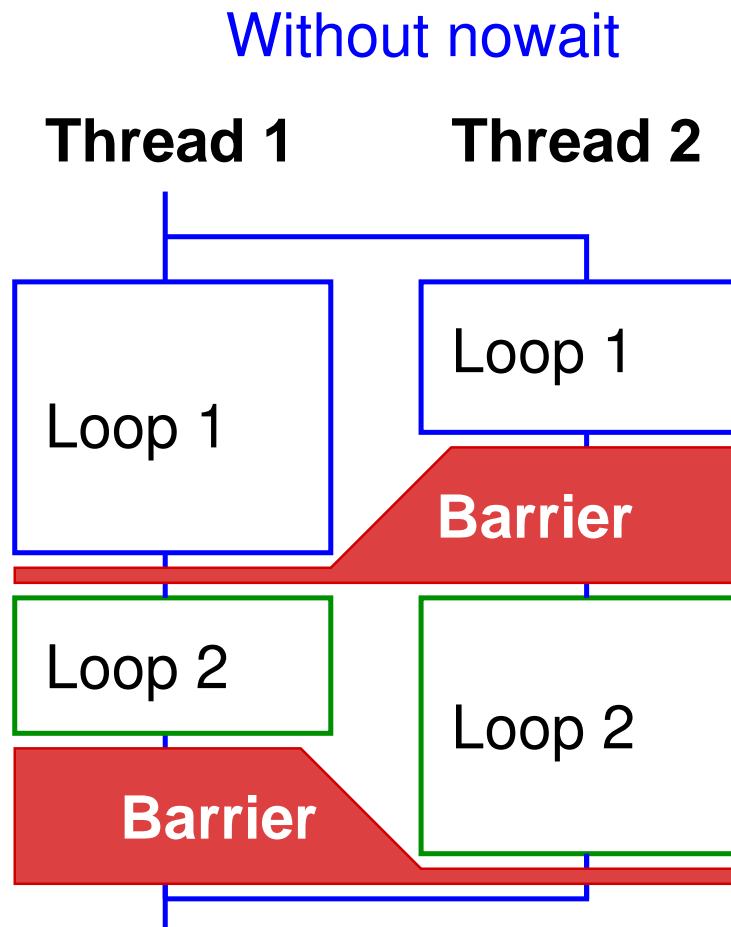
- ➔ The first loop processes the lower triangle of the matrix a , the second loop processes the upper triangle
 - ➔ load imbalance between the threads
 - ➔ barrier at the end of the loop results in waiting time
- ➔ But: the second loop does not depend on the first one
 - ➔ i.e., the computation can be 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

3.3.5 Barrier ...



Example ...

➔ Executions of the program:



3.3.6 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



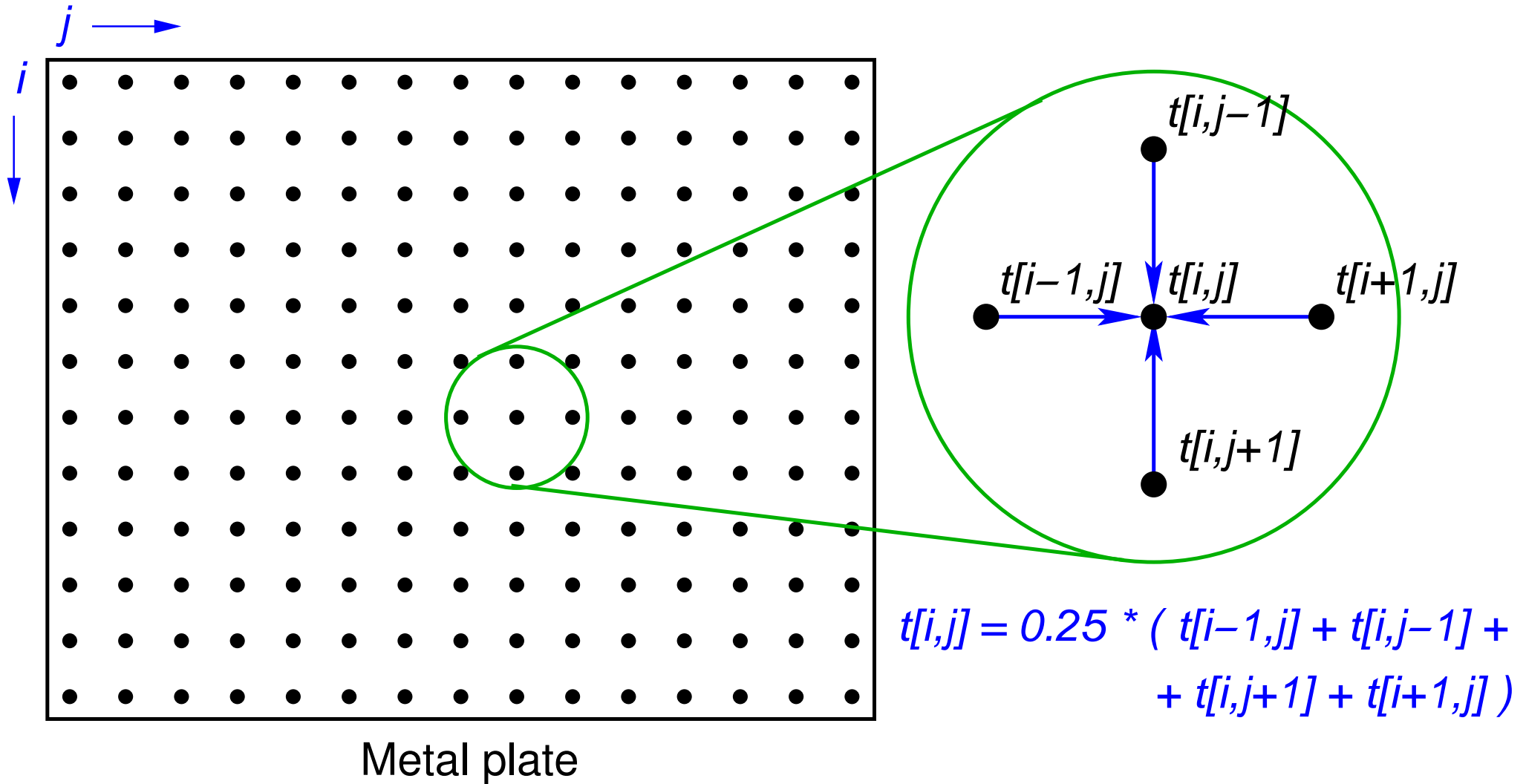
Numerical solution of the equations for thermal conduction

- ➔ 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

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



Numerical solution of the equations for thermal conduction ...





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] + ...);
    }
  }
  copy b to a;
} until (converged);
```

Gauss/Seidel

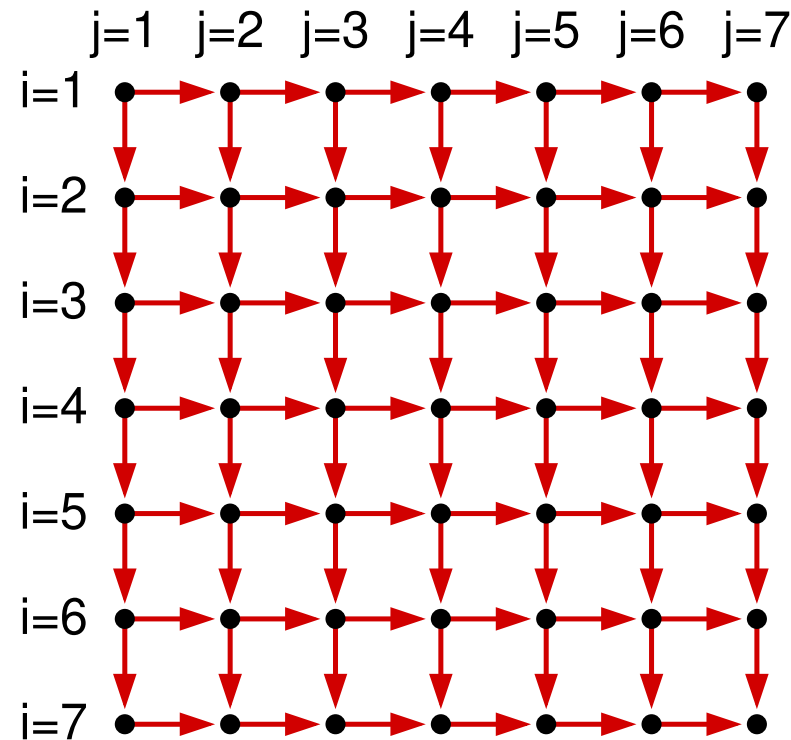
```
do {
  for (i=1;i<N-1;i++) {
    for (j=1;j<N-1;j++) {
      a[i][j] = 0.25 *
        (a[i-1][j] + ...);
    }
  }
} until (converged);
```

- ➔ For Jacobi: if *a* and *b* are pointers, they just can be swapped instead of copying *b* to *a*
 - ➔ at the very end, one copy may be needed to store the result in the correct array



Dependencies in Jacobi and Gauss/Seidel

- ➔ Jacobi: only between the i loop and the copy / pointer swap
- ➔ Gauss/Seidel: iterations of the i, j loop depend on each other



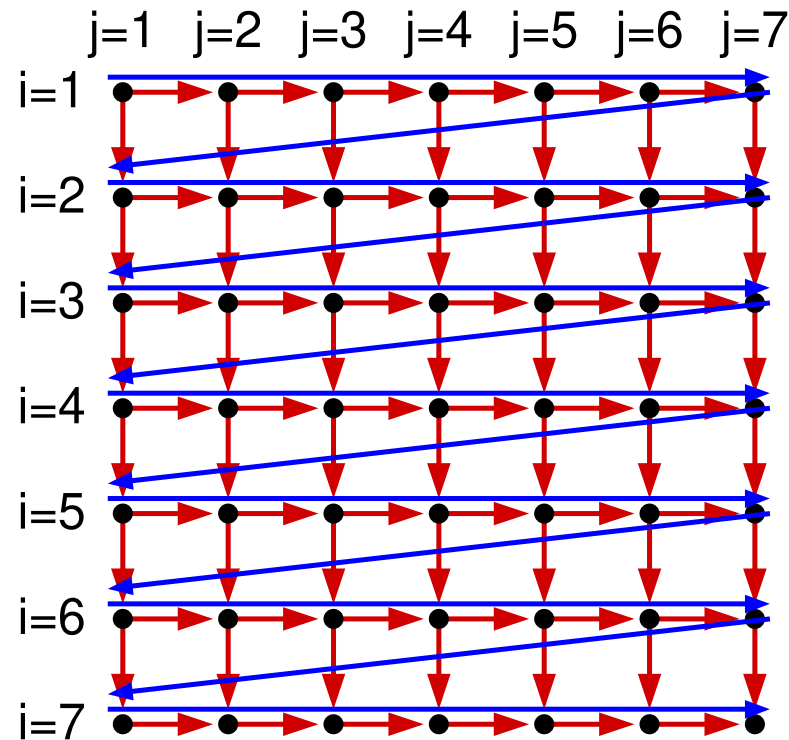
Sequential
execution
order

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



Dependencies in Jacobi and Gauss/Seidel

- ➔ Jacobi: only between the i loop and the copy / pointer swap
- ➔ Gauss/Seidel: iterations of the i, j loop depend on each other



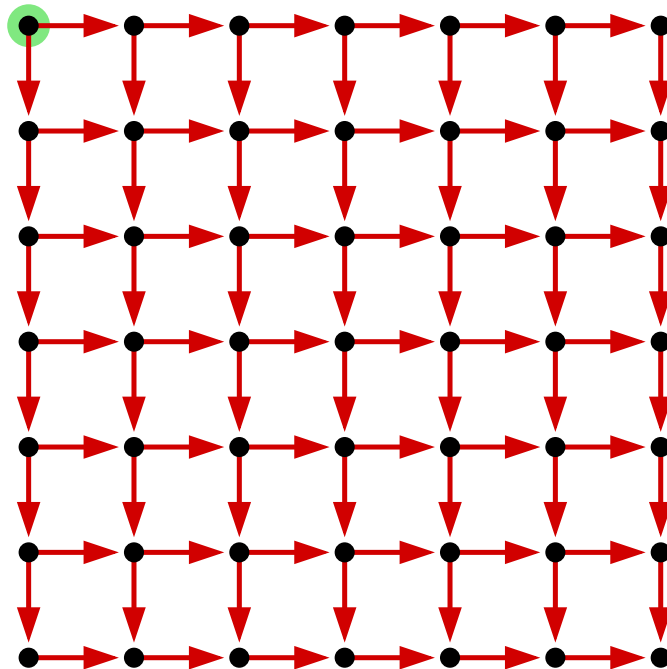
Sequential
execution
order

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



Parallelisation of the Gauss/Seidel method

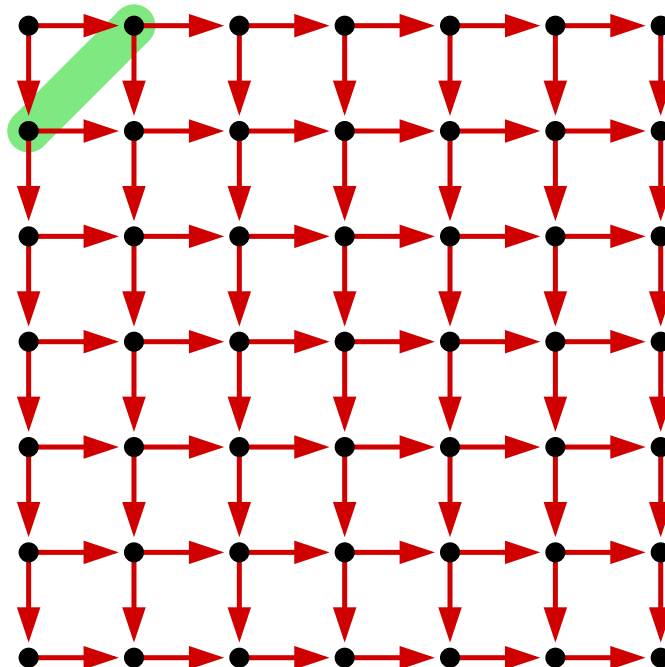
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

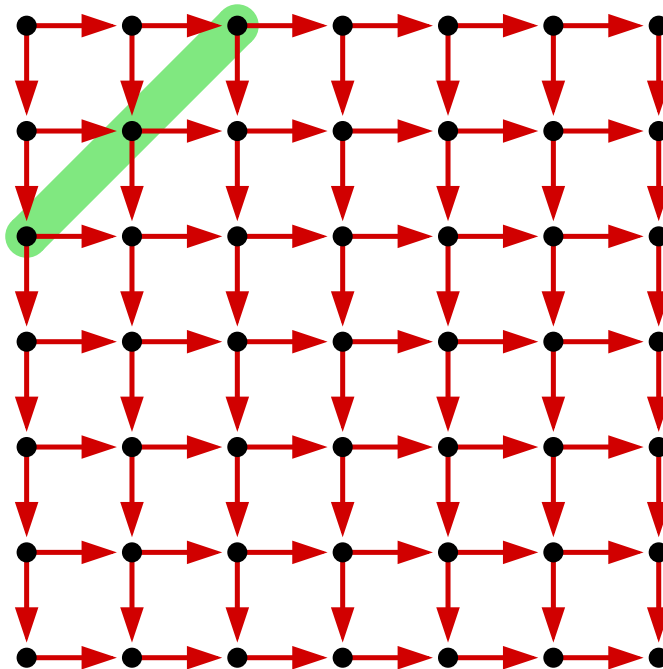
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

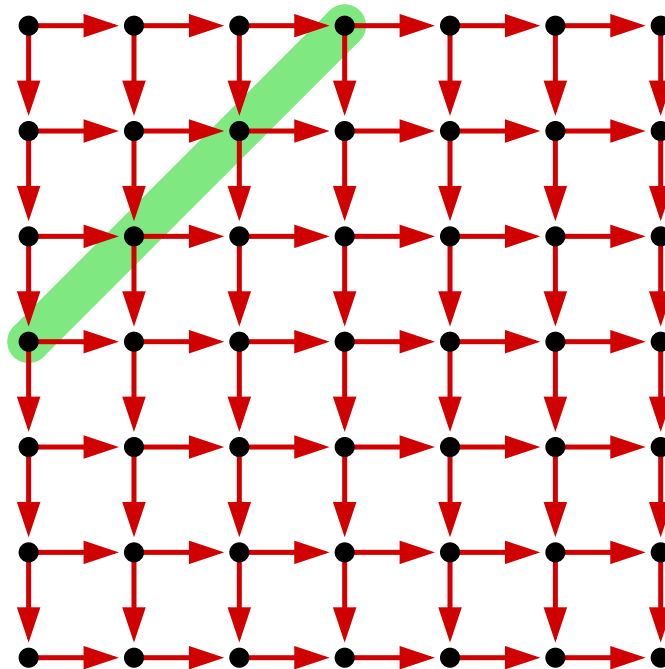
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

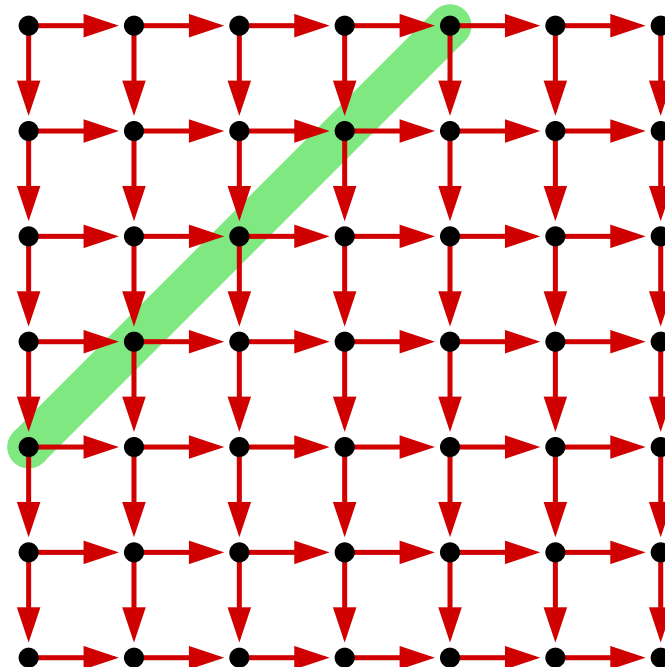
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

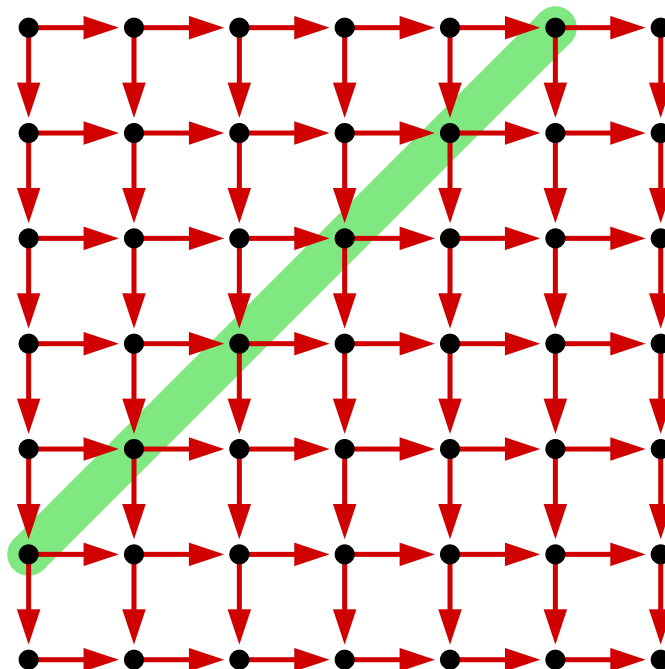
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

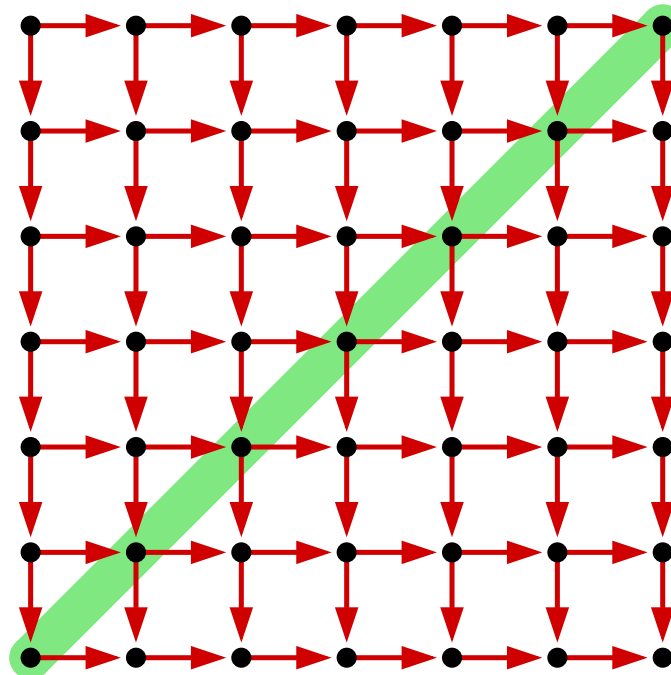
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

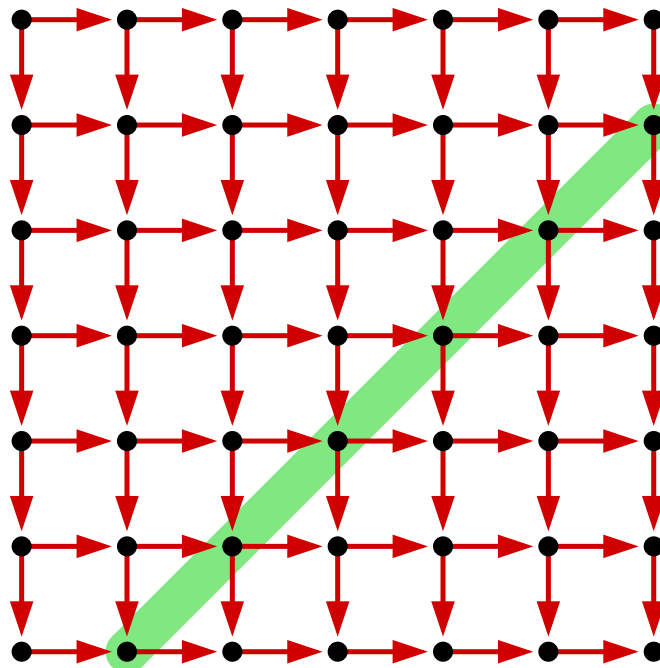
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

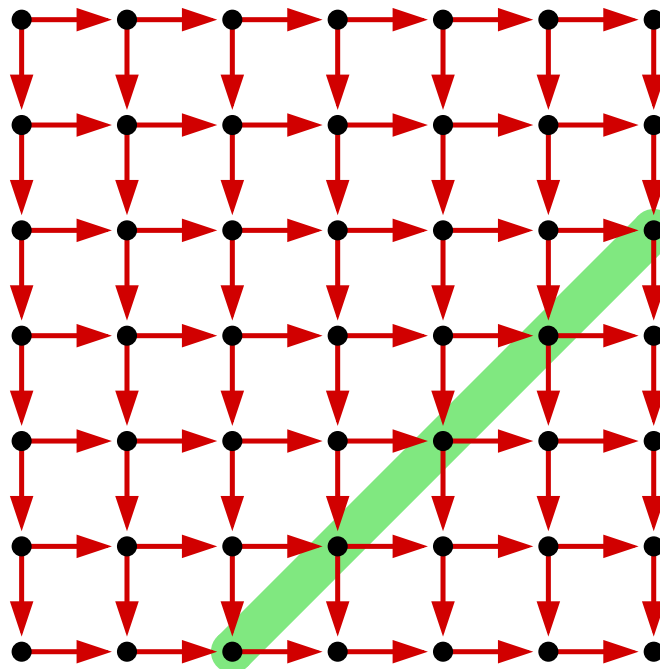
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

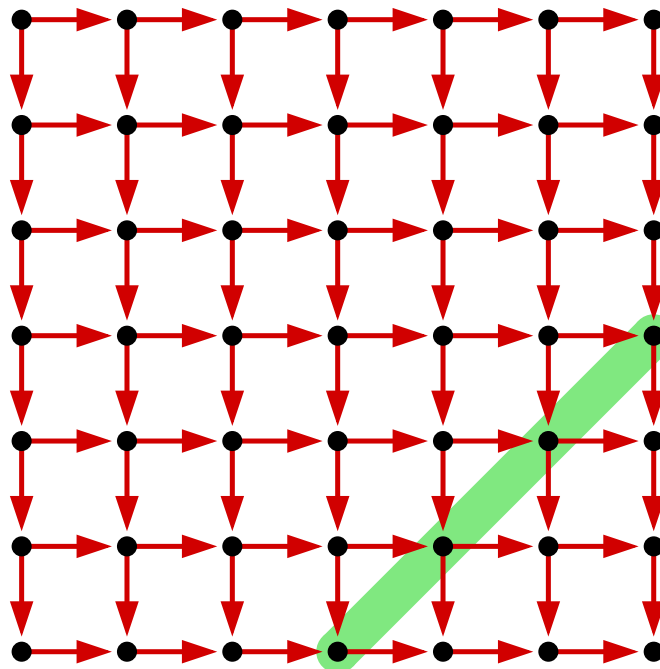
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

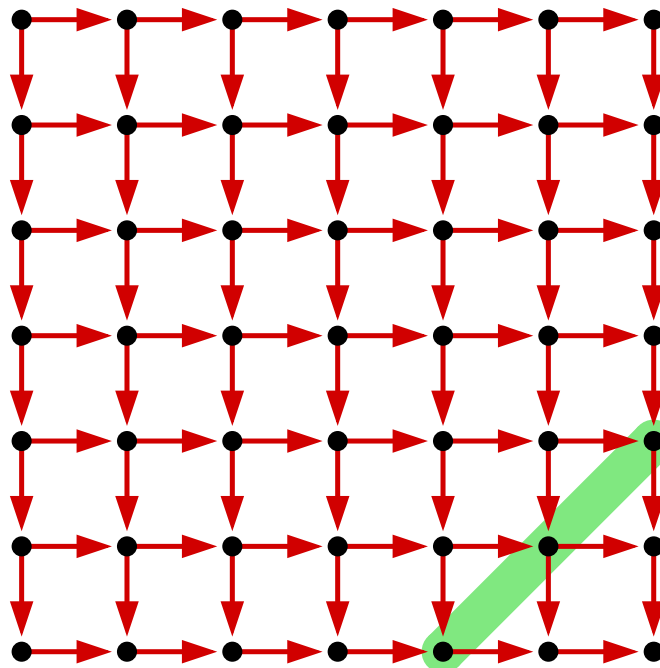
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

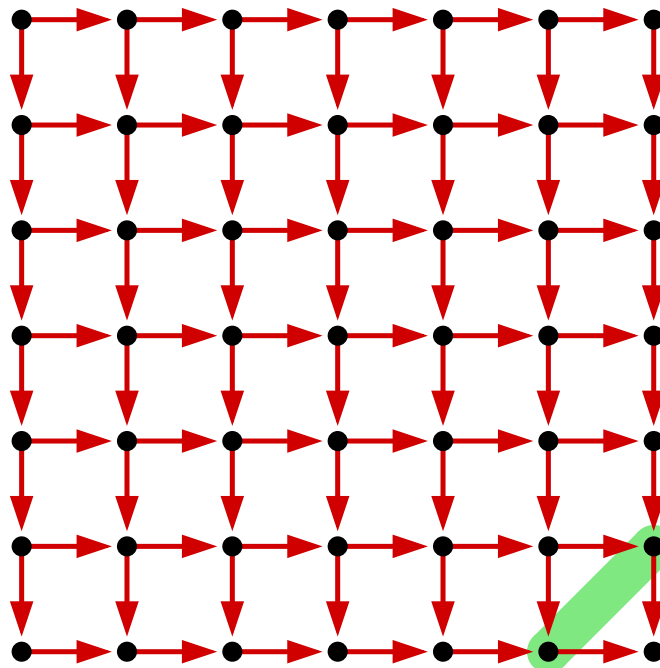
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

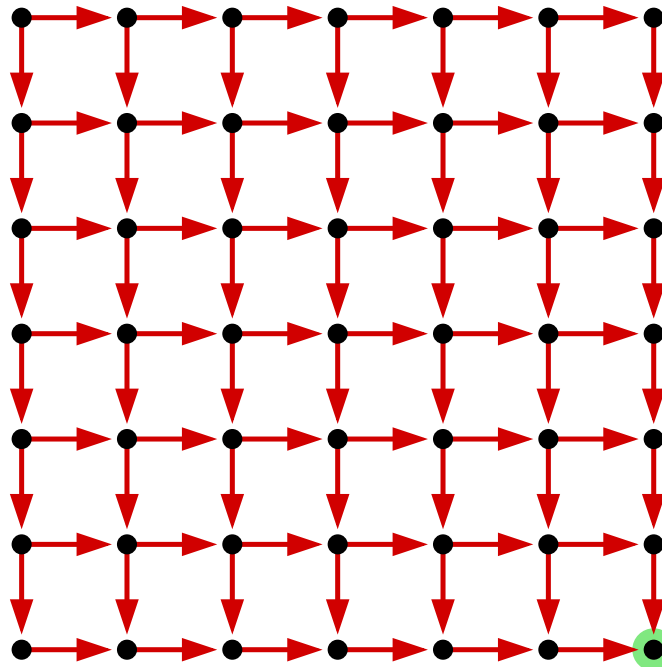
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Parallelisation of the Gauss/Seidel method

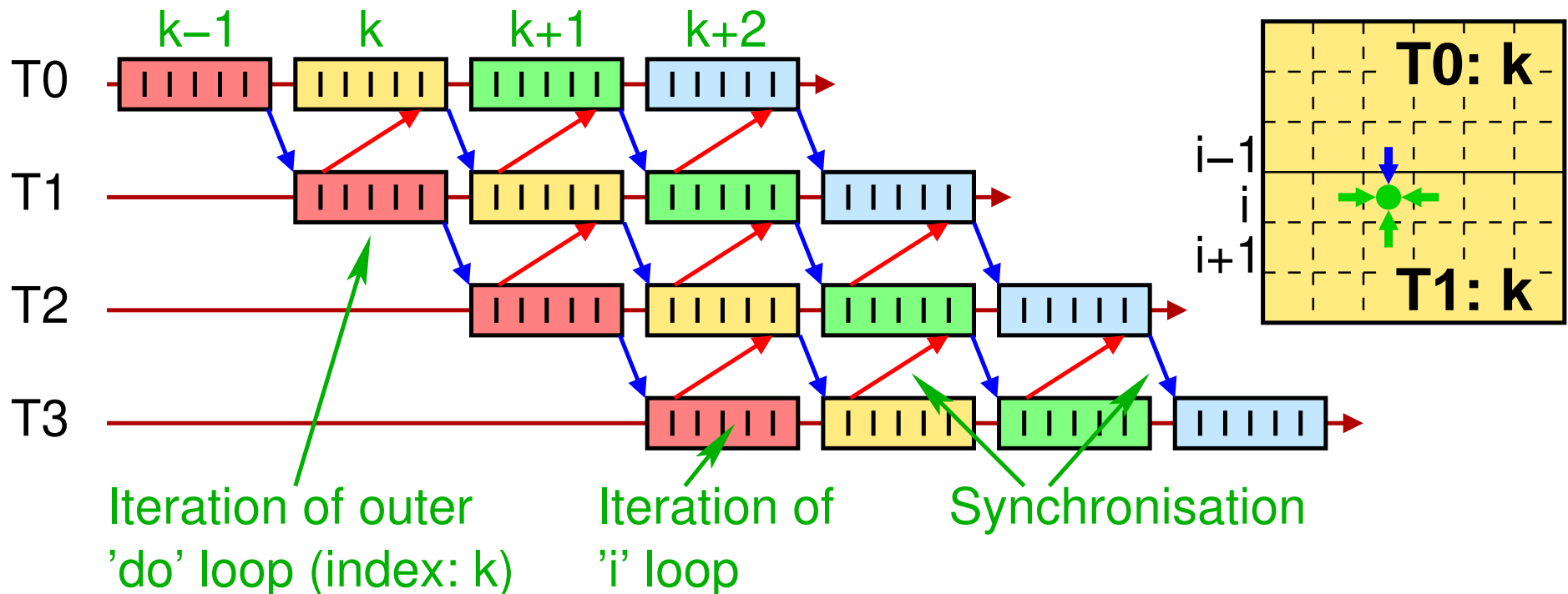
- ➔ Restructure the i, j loop, such that the iteration space is traversed diagonally
- ➔ no dependences between the iterations of the inner loop
- ➔ problem: varying degree of parallelism





Alternative parallelization of the Gauss/Seidel method

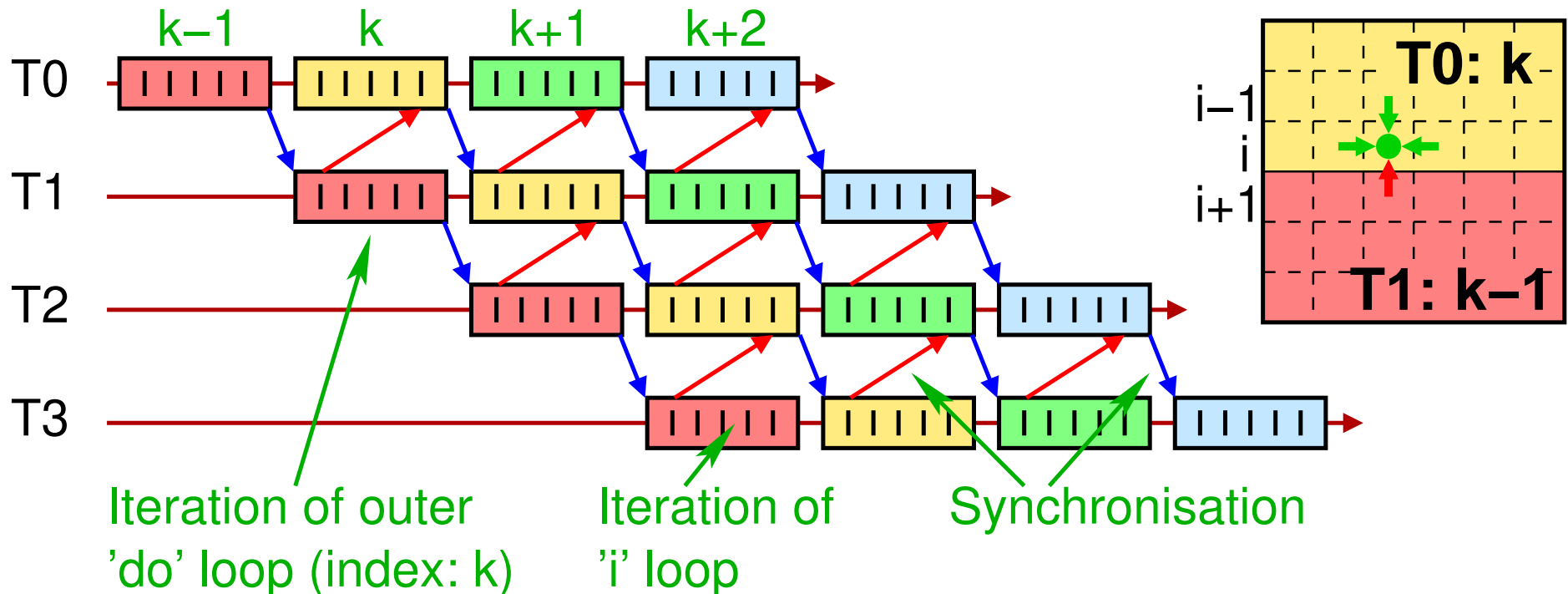
- ➔ Requirement: number of iterations is known in advance
- ➔ (or: we are allowed to execute a few more iterations after convergence)
- ➔ Then we can use a pipeline-style parallelization
- ➔ synchronisation via ordered (👉 3.3.4)





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 (👉 3.3.4)





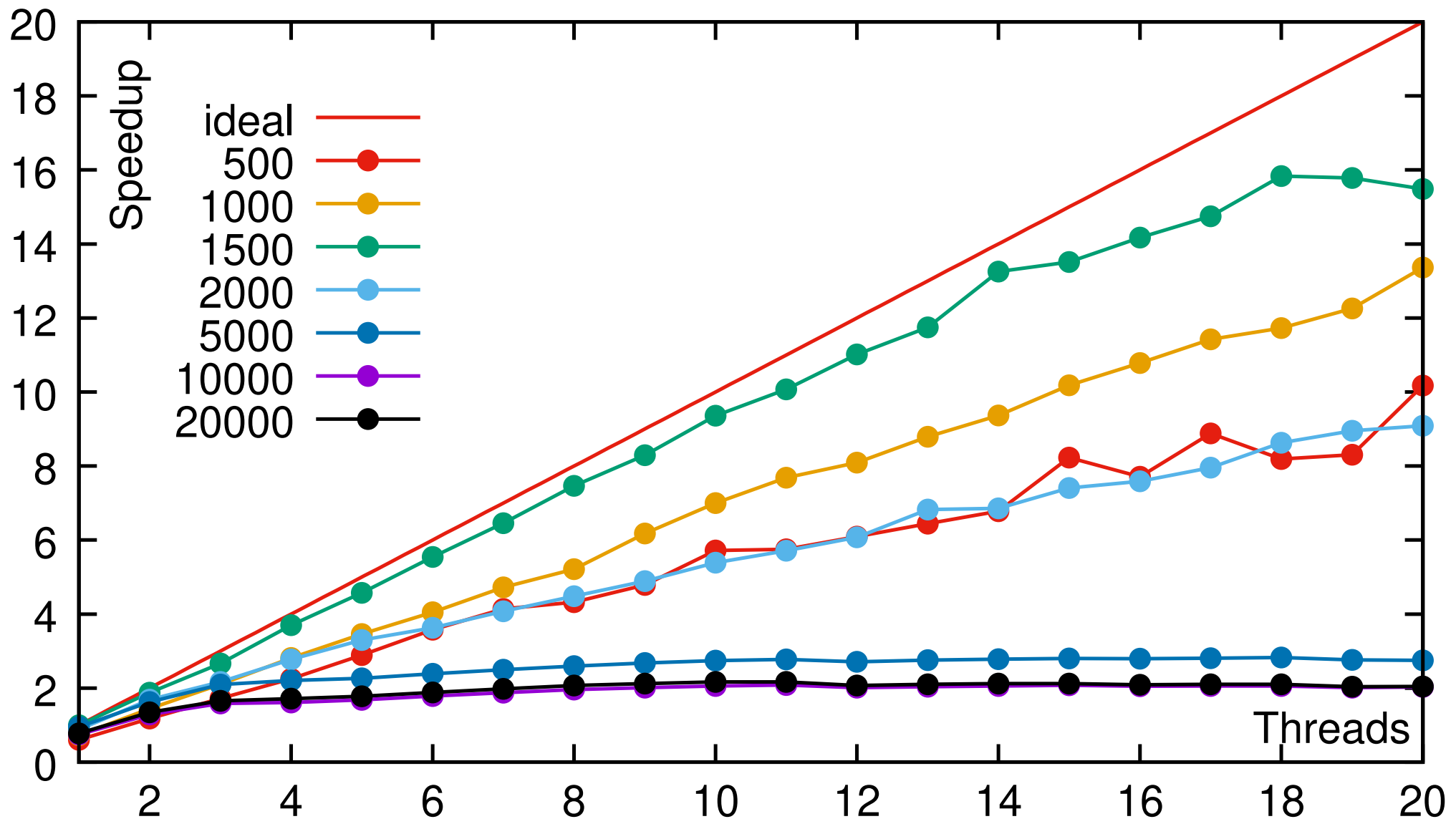
Results

- ➔ Platform: PCs in the lab H-A 4111
 - ➔ Intel Core Ultra 7, 20 Cores (8 * 5.2 GHz; 12 * 4.6 GHz)
 - ➔ compilers: g++ 13.3 and nvc++ 25.5
- ➔ No performance loss due to compilation with OpenMP
- ➔ Jacobi: extremely good speedup with matrix size of 1500
 - ➔ data size: ~ 18MB, L2 cache size: 3MB per performance core
- ➔ Diagonal traversal in Gauss/Seidel
 - ➔ improves performance for small matrices
 - ➔ shows extremely poor parallel performance
- ➔ Pipelined Gauss/Seidel
 - ➔ good speedup, but performance is worse than Jacobi

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



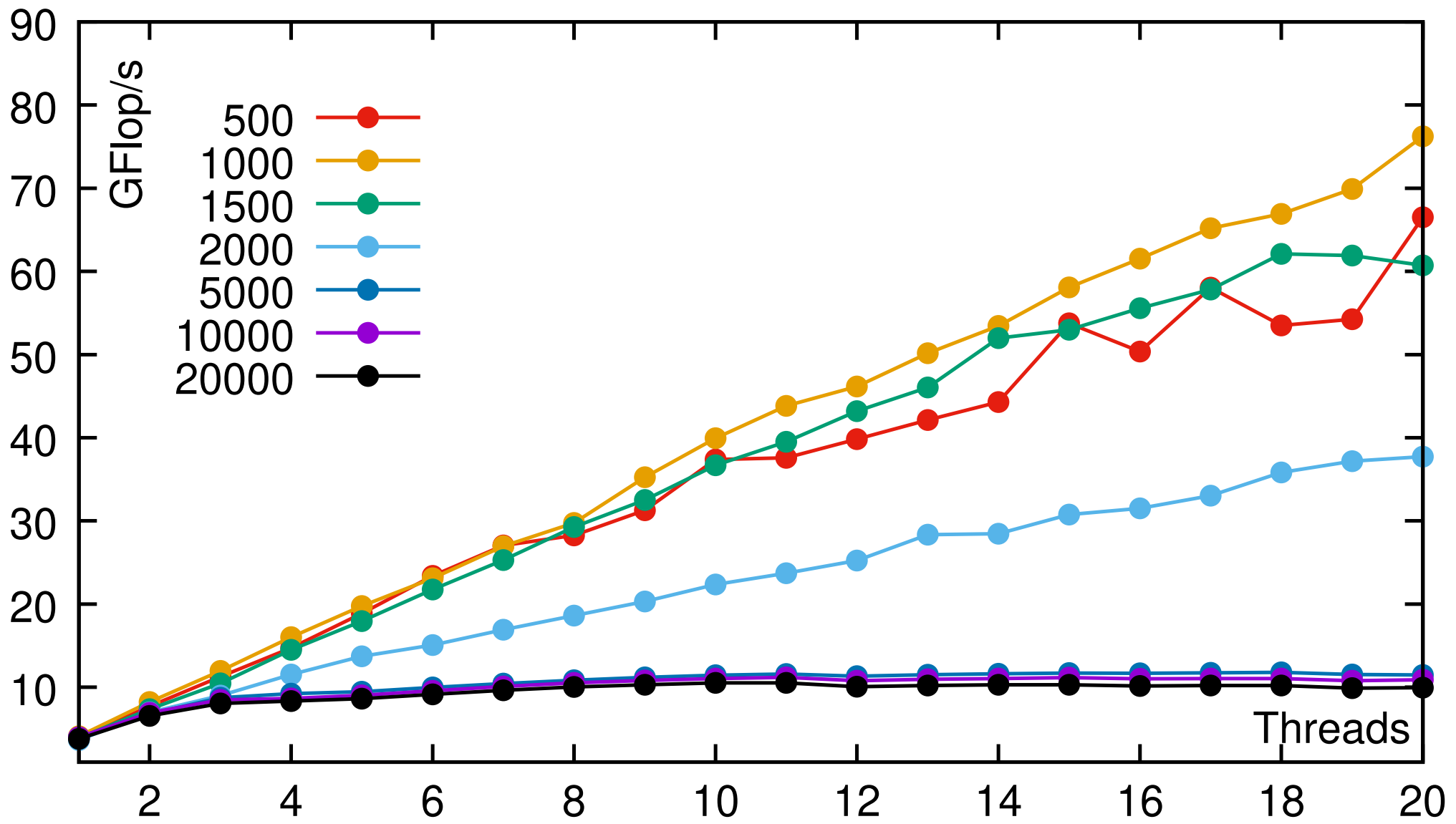
Jacobi: Speedup (g++)



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



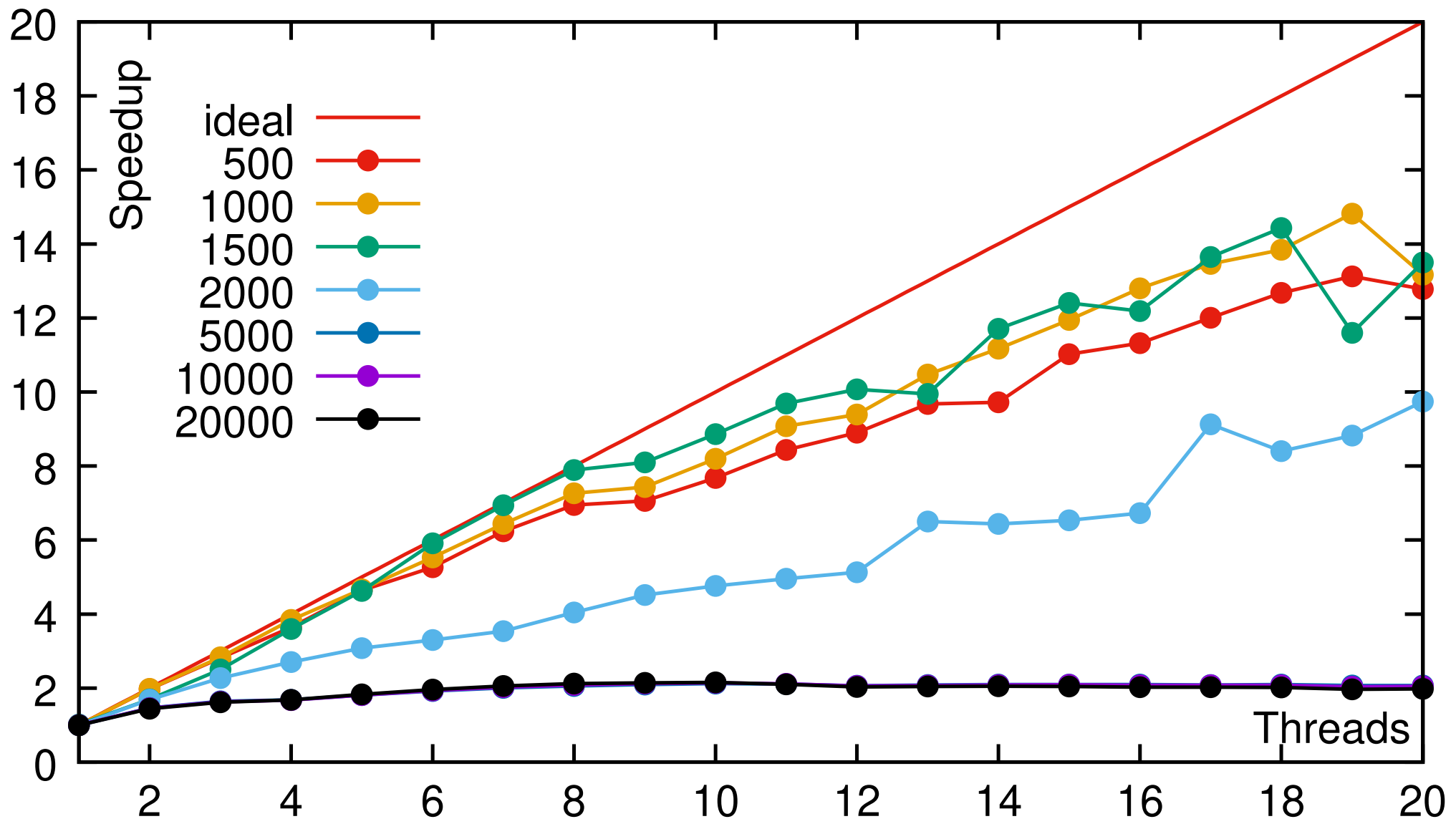
Jacobi: Performance (g++)



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



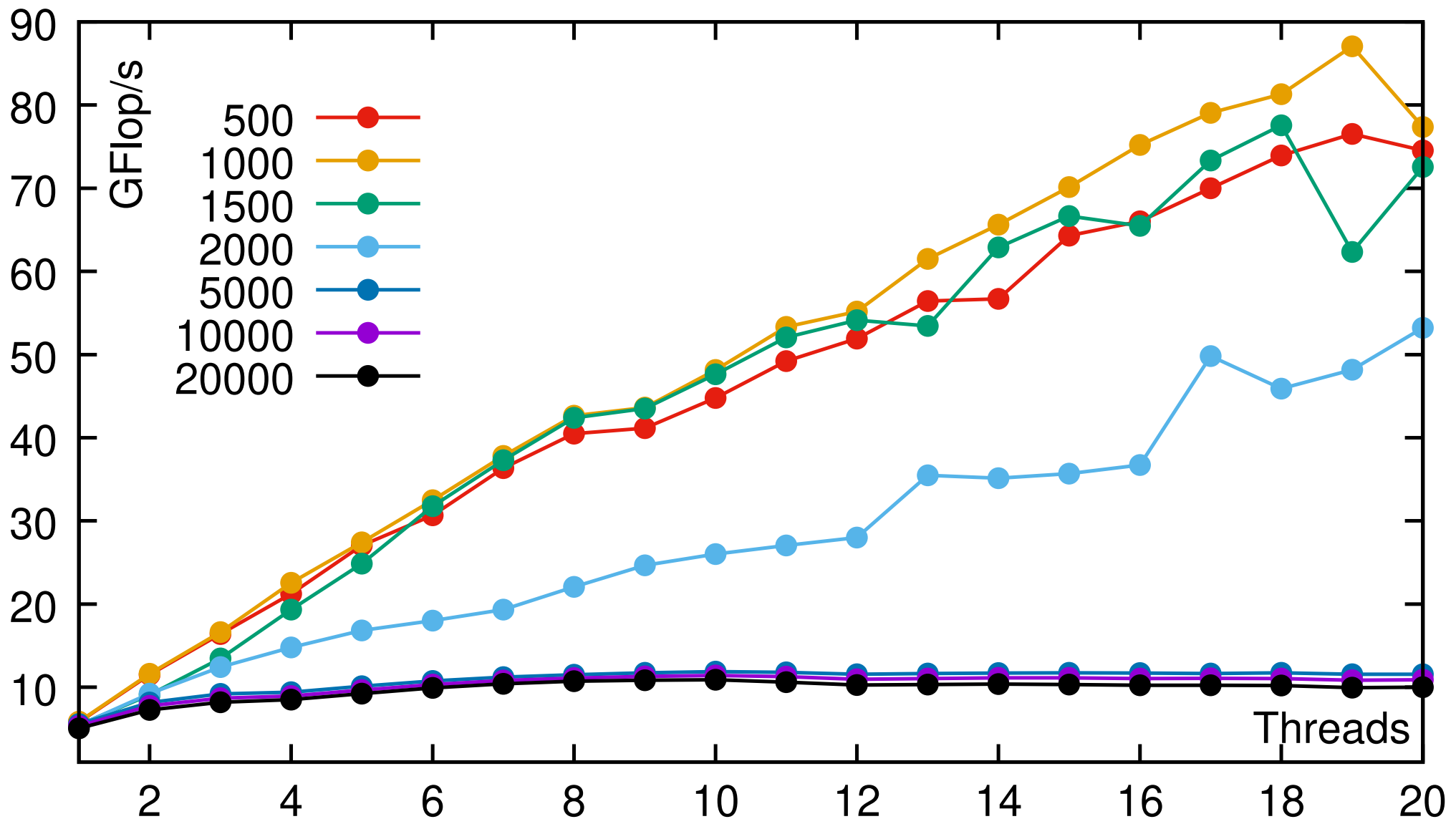
Jacobi: Speedup (nvc++)



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



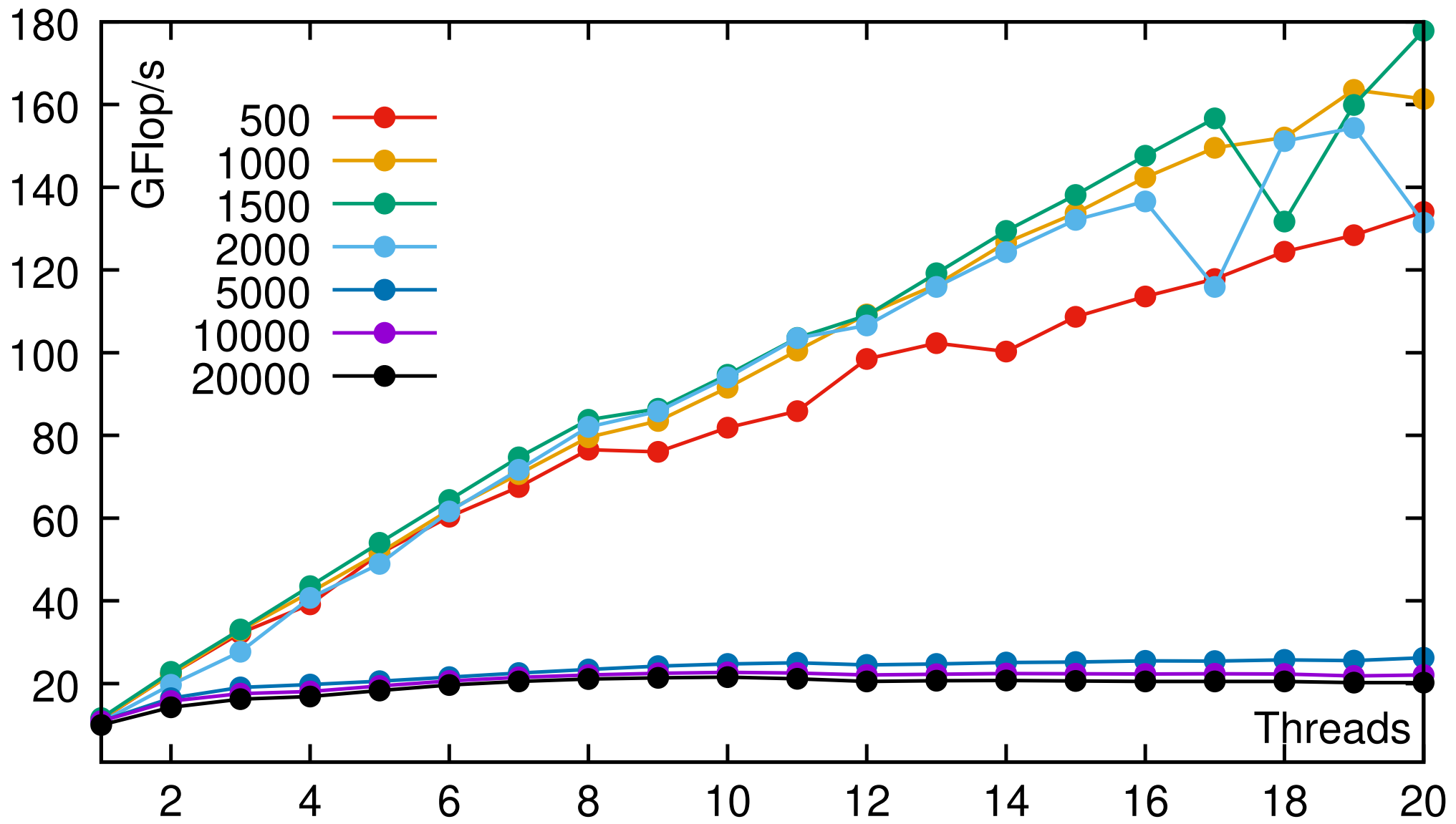
Jacobi: Performance (nvc++)



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



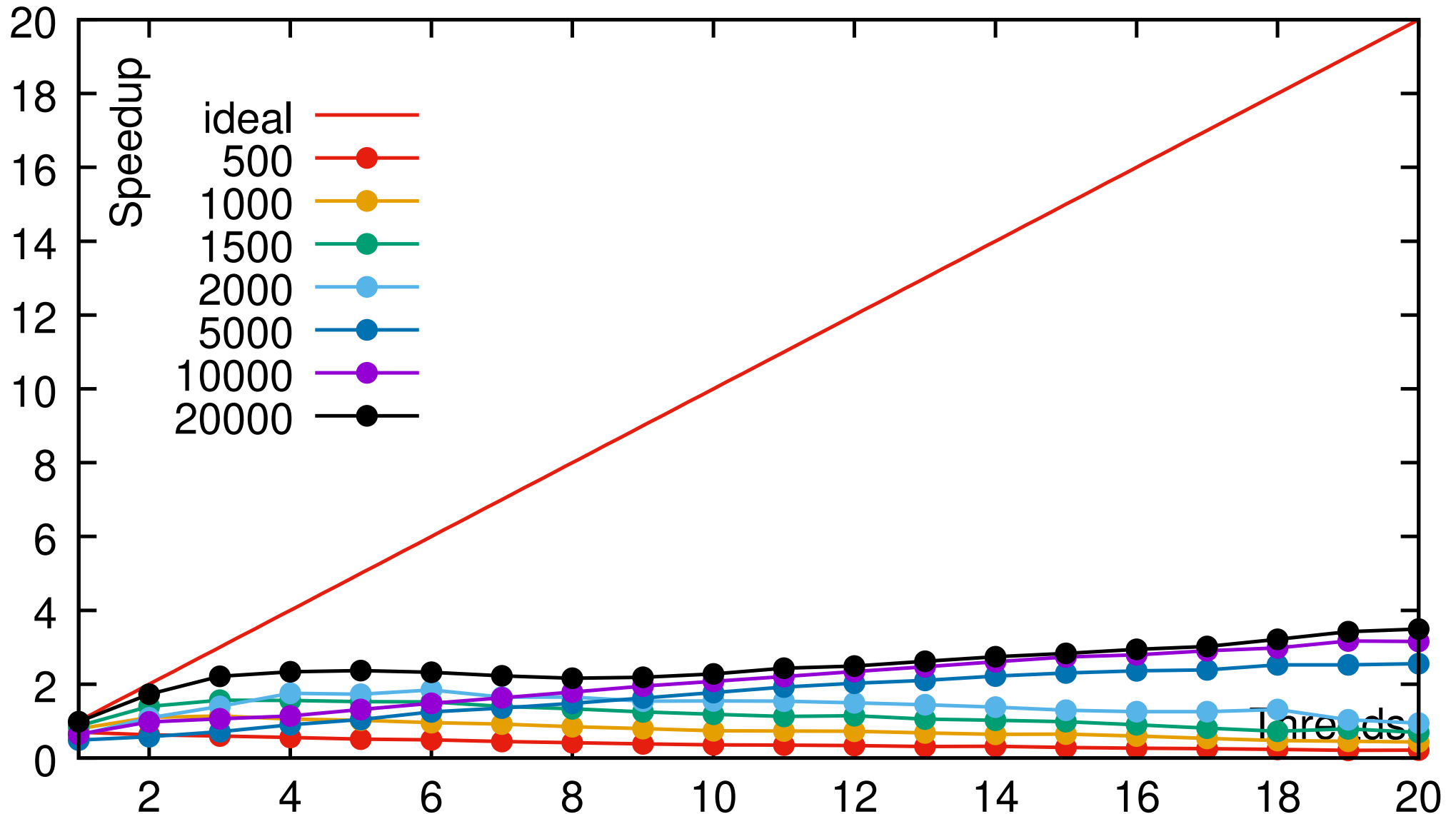
Jacobi: Performance (nvc++ with 32-Bit floating point)



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



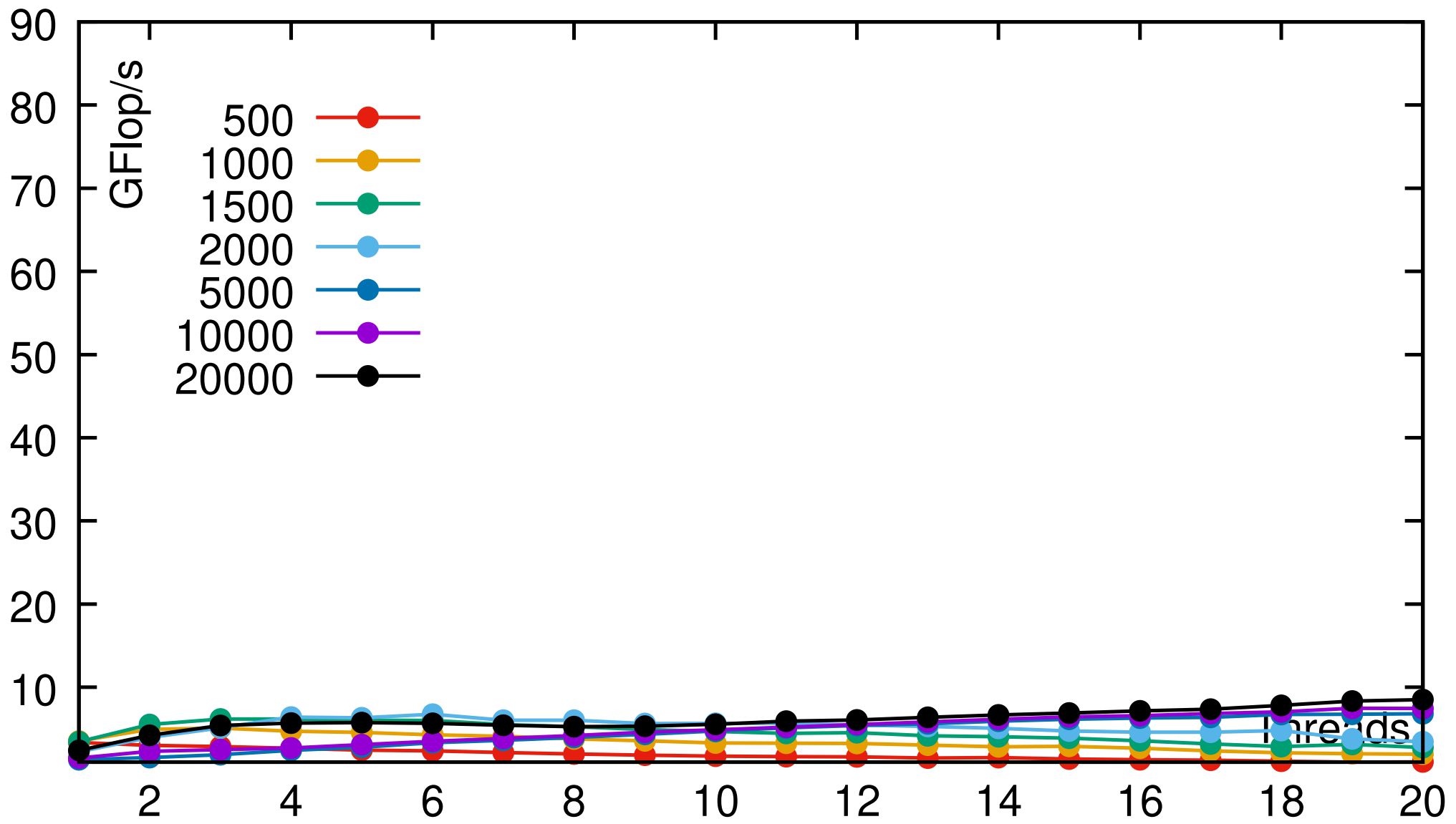
Gauss/Seidel: Speedup (diagonal, g++)



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



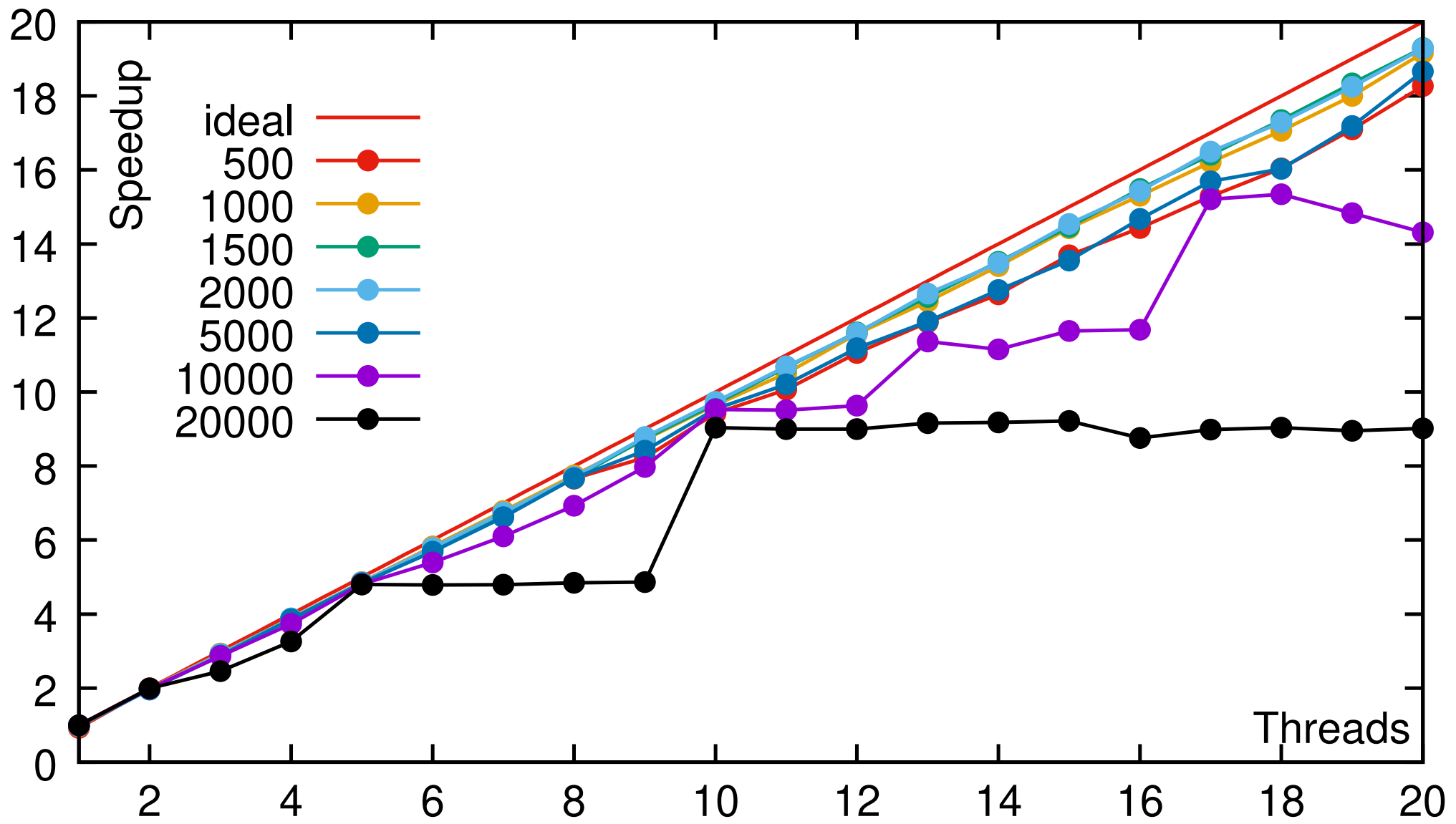
Gauss/Seidel: Performance (diagonal, g++)



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



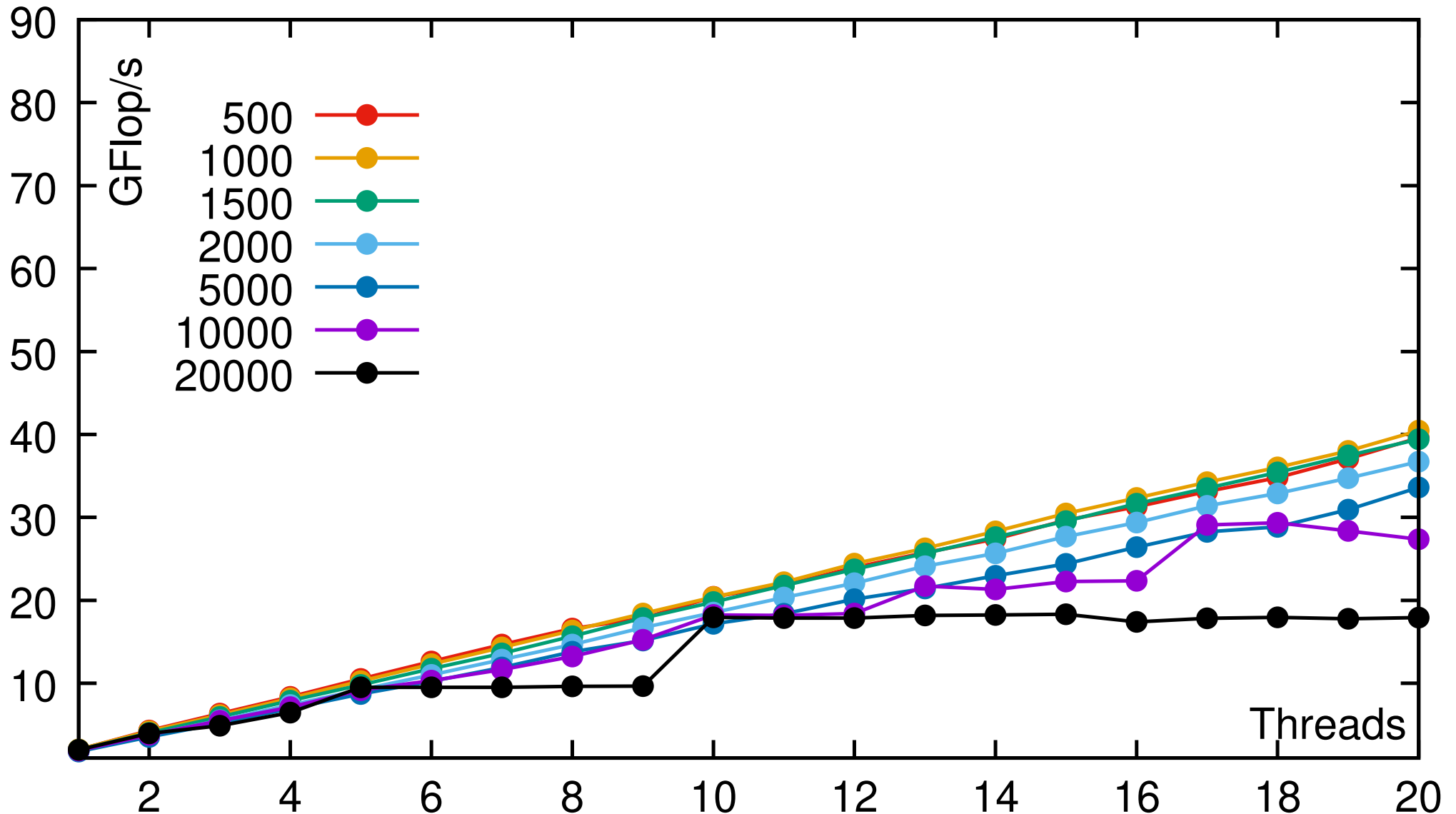
Gauss/Seidel: Speedup (pipeline, g++)



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



Gauss/Seidel: Performance (pipeline, g++)



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



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



3.5.2 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/qsrt.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
```



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 silently ignore this directive!



Example: parallel quicksort (👉 03/qsor t . 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

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, possible load imbalance



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);
    }
    checkSorted(array, n);
}
```

← wait for all tasks created in the block

➔ advantage: only wait at one single place

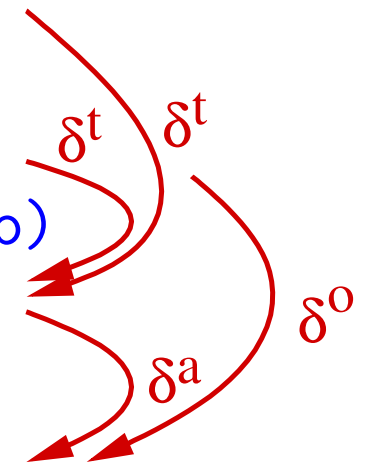
➔ disadvantage: semantics of `quicksort()` must be very well documented

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

3.6 Advanced OpenMP Features



3.6.1 Thread Affinity

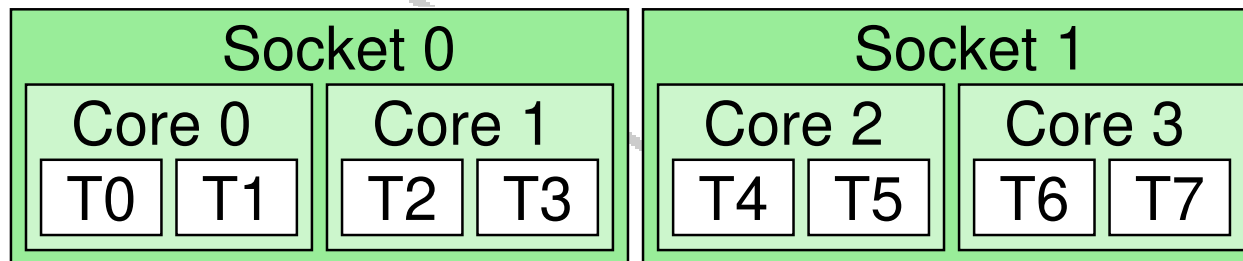
- ➔ Goal: control where threads are executed
 - ➔ i.e., by which HW threads on which core on which CPU
- ➔ Important (among others) because of the architecture of today's multicore CPUs
 - ➔ HW threads share the functional units of a core
 - ➔ cores share the L2 caches and the memory interface
- ➔ Concept 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

3.6.1 Thread Affinity ...



Environment variable `OMP_PLACES`

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

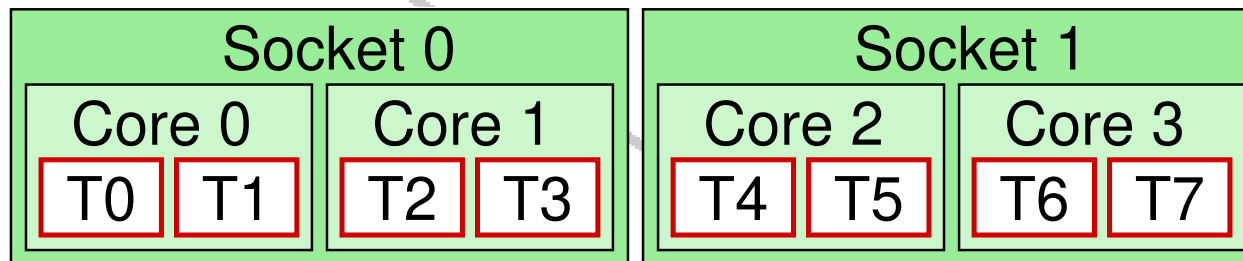


3.6.1 Thread Affinity ...



Environment variable `OMP_PLACES`

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



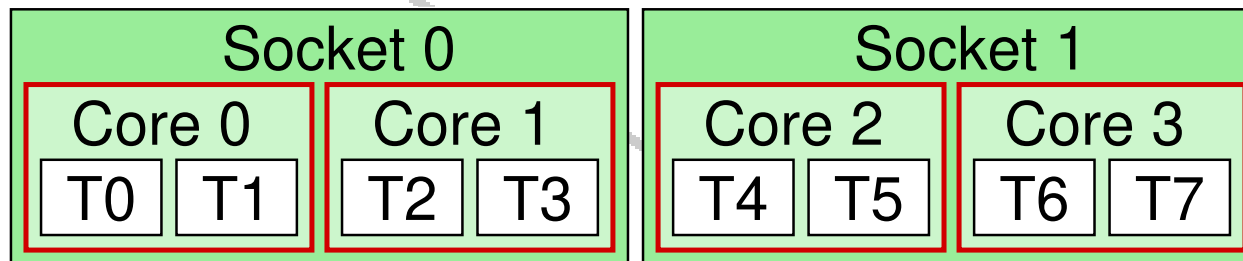
- ➔ To consider each hardware thread as a place, e.g.:
`OMP_PLACES = "{0},{1},{2},{3},{4},{5},{6},{7}"`
`OMP_PLACES = "threads"`

3.6.1 Thread Affinity ...



Environment variable `OMP_PLACES`

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



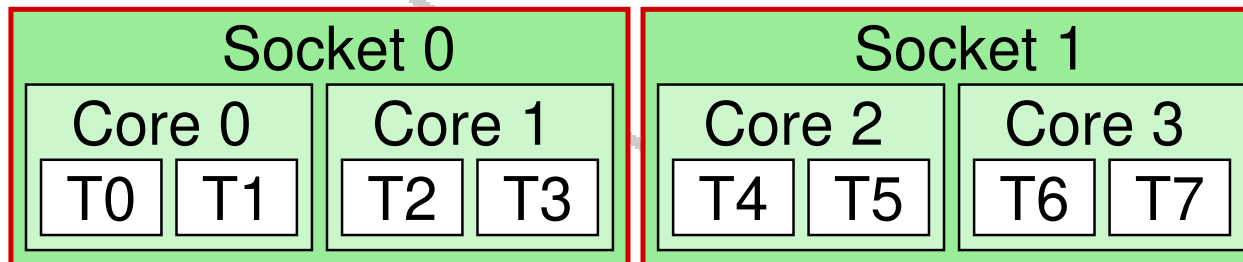
- ➔ To consider each hardware thread as a place, e.g.:
`OMP_PLACES = "{0},{1},{2},{3},{4},{5},{6},{7}"`
`OMP_PLACES = "threads"`
- ➔ To consider each core as a place, e.g.:
`OMP_PLACES = "{0,1},{2,3},{4,5},{6,7}"`
`OMP_PLACES = "cores"`

3.6.1 Thread Affinity ...



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 hardware thread as a place, e.g.:
`OMP_PLACES = "{0},{1},{2},{3},{4},{5},{6},{7}"`
`OMP_PLACES = "threads"`
- ➔ To consider each core as a place, e.g.:
`OMP_PLACES = "{0,1},{2,3},{4,5},{6,7}"`
`OMP_PLACES = "cores"`
- ➔ To consider each socket as a place, e.g.:
`OMP_PLACES = "sockets"`



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 of 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

3.6.1 Thread Affinity ...



Example: nested parallel regions

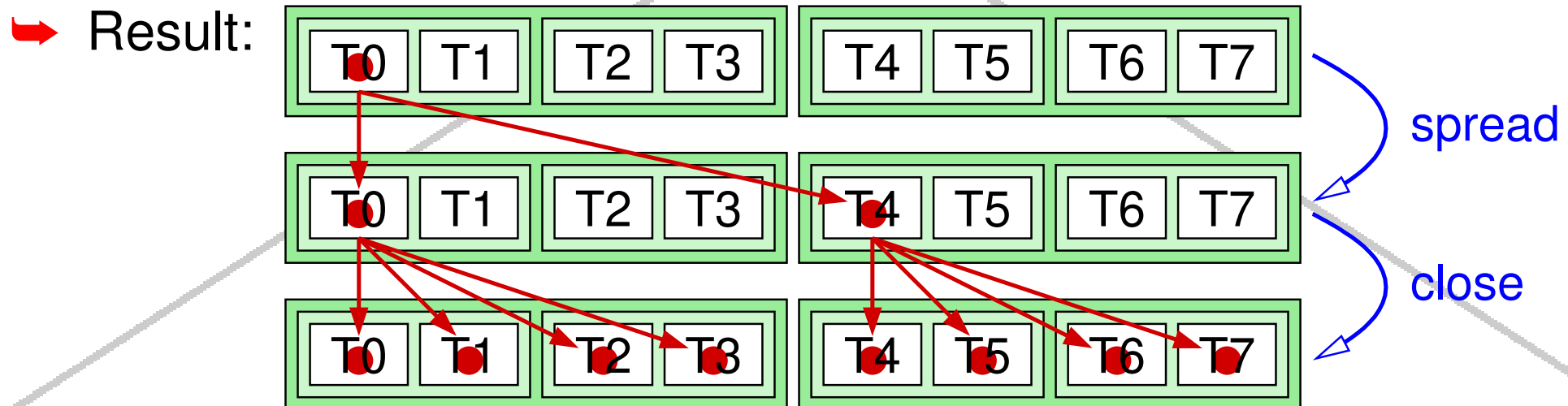
```
double f1(double x)
{
    #pragma omp parallel for proc_bind(close)
    for (i=0; i<N; i++) {
        ...
    }
    ...
    #pragma omp parallel proc_bind(spread)
    #pragma omp single
    {
        #pragma omp task shared(a)
        a = f1(x);
        #pragma omp task shared(b)
        b = f1(y);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

3.6.1 Thread Affinity ...



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`





3.6.2 SIMD Vectorization

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

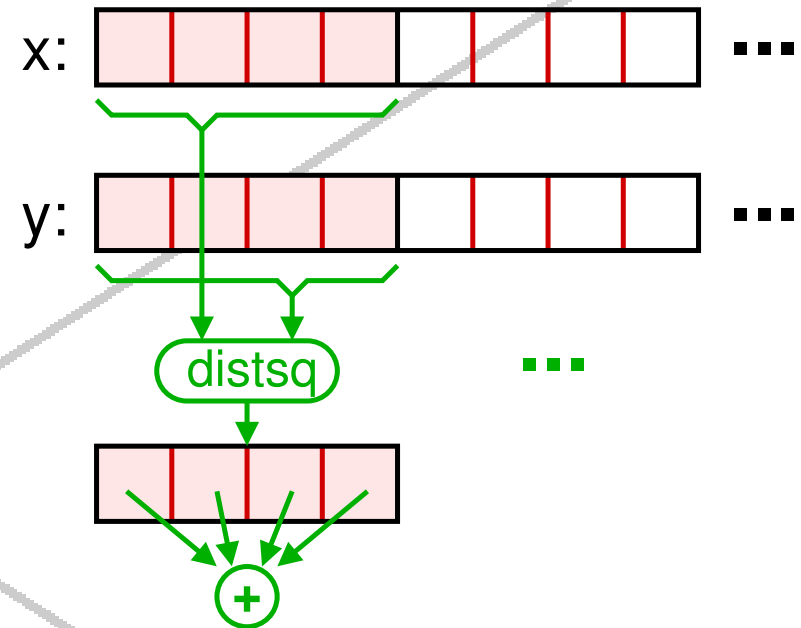
- ➔ Restructuring of a loop in order to use the SIMD vector registers
 - ➔ e.g., Intel SSE: 4 float operations in parallel
- ➔ Loop will be executed by a single thread
 - ➔ combination with `for` is possible
- ➔ Options (among others): `private`, `lastprivate`, `reduction`
- ➔ Option `safelen`: maximum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



3.6.3 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

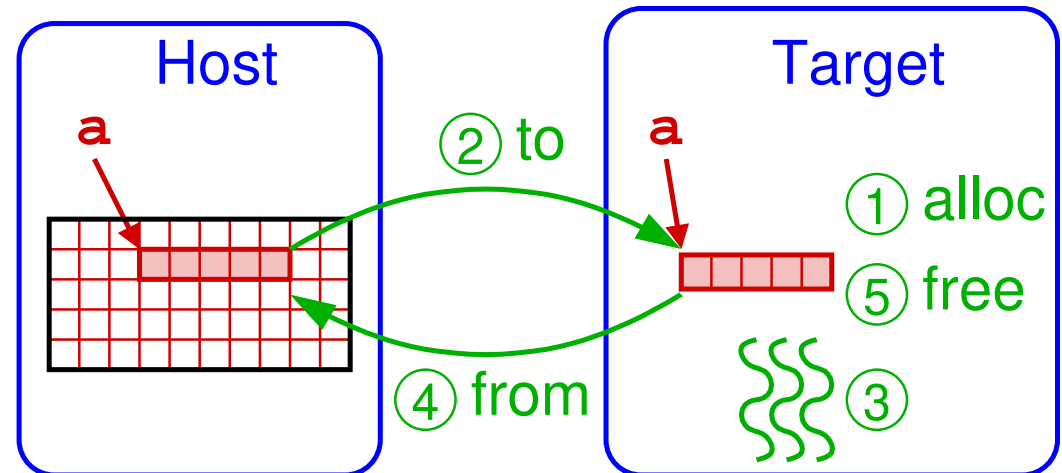
- ➔ Maps variable v_H in the host data environment to the corresponding variable v_T in the target data environment
 - ➔ either the data is copied between v_H and v_T or v_H and v_T are identical
- ➔ Syntax: `map(alloc | to | from | tofrom : <list>)`
 - ➔ `<list>`: list of the original variables
 - ➔ array sections are allowed, too, [3.5.2](#)
 - ➔ `alloc`: just allocate memory for v_T
 - ➔ no copy to / from v_H , however, v_H must be allocated
 - ➔ `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`

Target data environment

➔ Course of actions for the target construct:

```
#pragma omp target \  
    map(tofrom: a)
```

```
{ 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

Parallel Processing

Winter Term 2025/26

24.11.2025

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

Stand: November 24, 2025



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`

3.6.3 Using External Accelerators ...



Example: SAXPY (single precision $a \cdot \vec{x} + \vec{y}$)

➔ On the host:

```
void saxpy(float a, float *x, float *y, 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 a, float *x, float *y, 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];  
}
```

➔ uses just one streaming multiprocessor (SE)



Example: SAXPY (single precision $a \cdot \vec{x} + \vec{y}$) ...

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

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

➔ iterations are distributed to the streaming multiprocessors in blocks, where they are distributed to individual threads

➔ Important clauses:

➔ `num_teams(n)`: number of thread teams

➔ `num_threads(n)`: number of threads per team

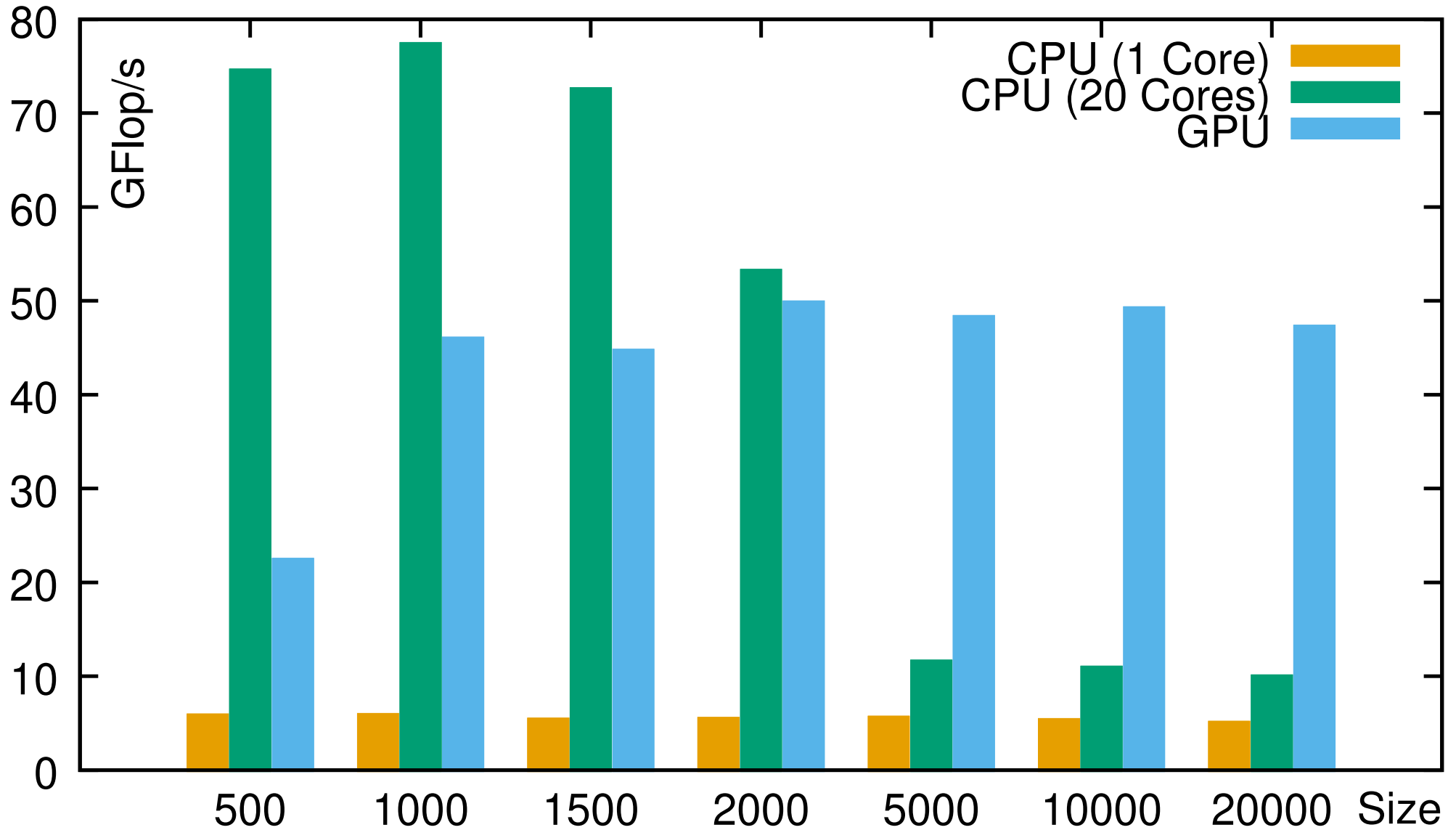
➔ `collapse(n)`: distribute loop nest with *n* loops

➔ `reduction(op : var)` is possible across teams

3.6.3 Using External Accelerators ...



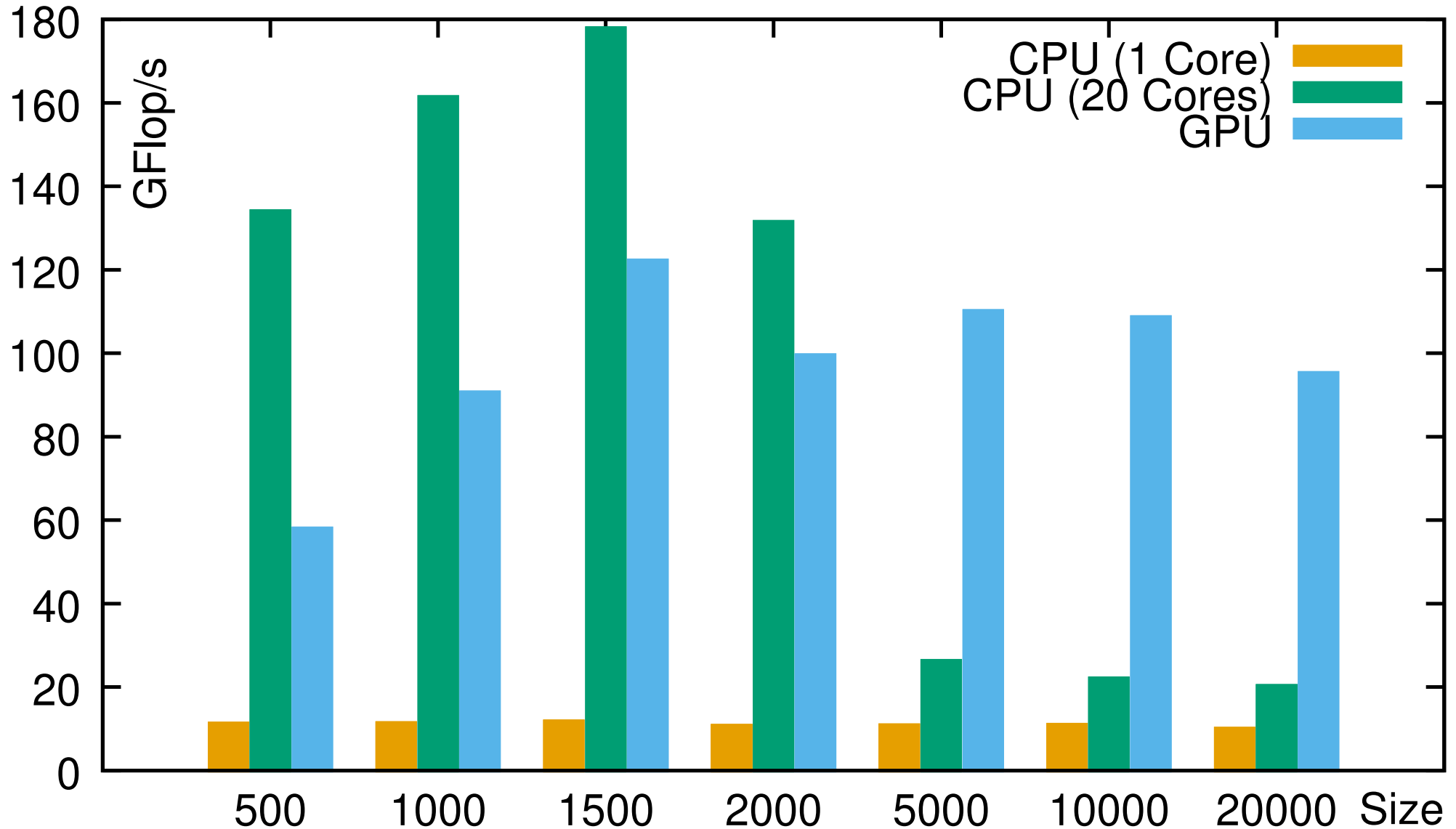
Jacobi: Performance Comparison (nvc)



3.6.3 Using External Accelerators ...

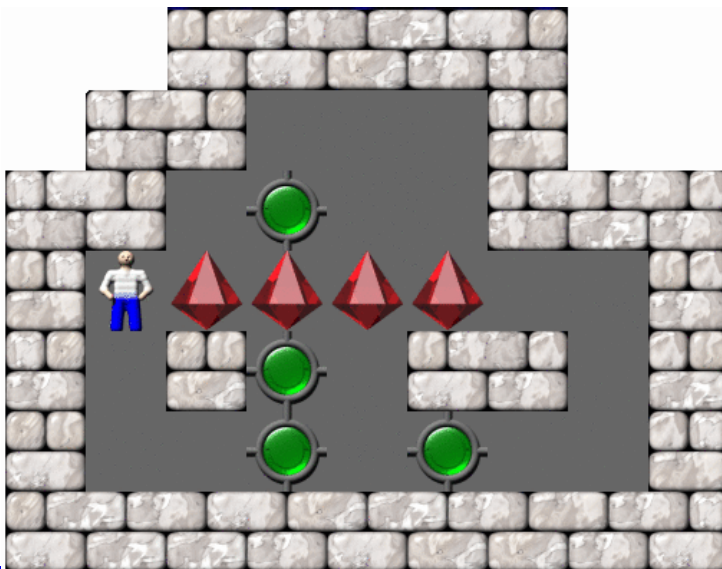


Jacobi: Performance Comparison (nvc with 32 Bit floating point)



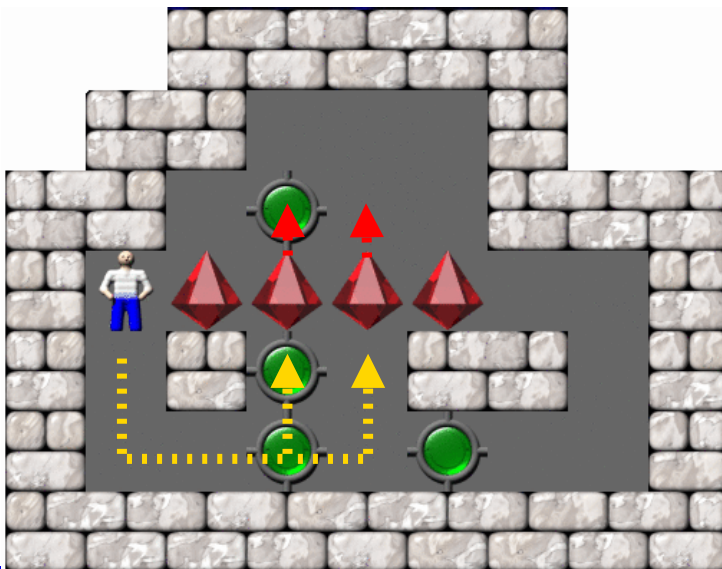
Background

- ➔ 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



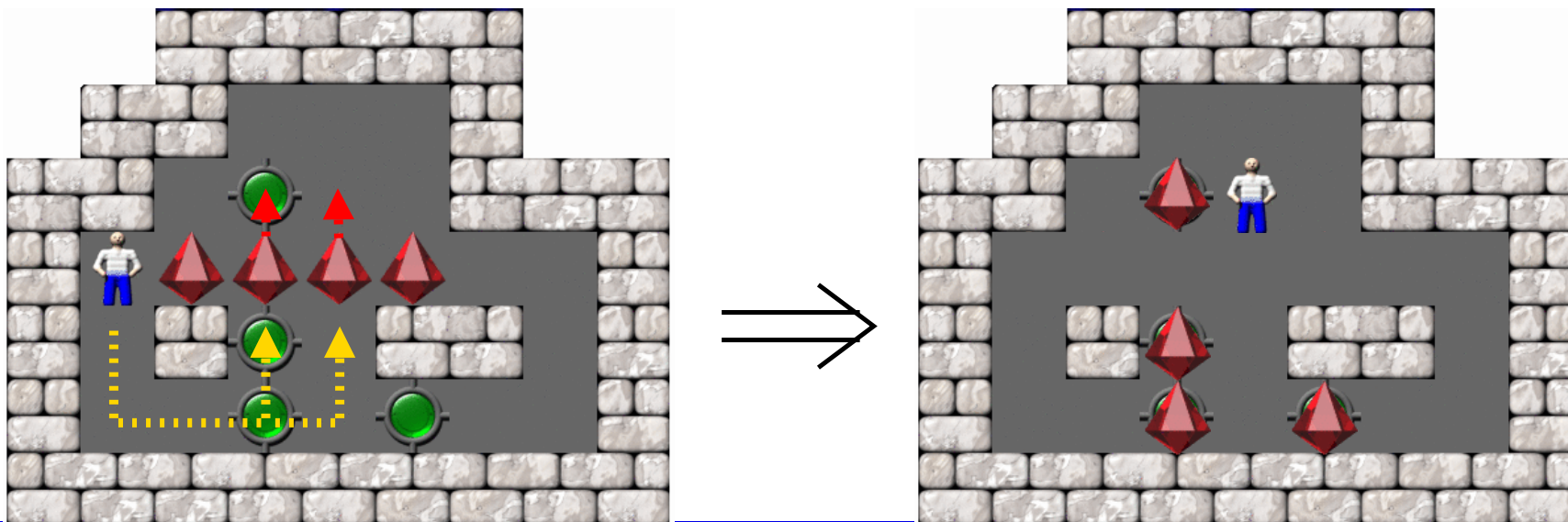
Background

- ➔ 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



Background

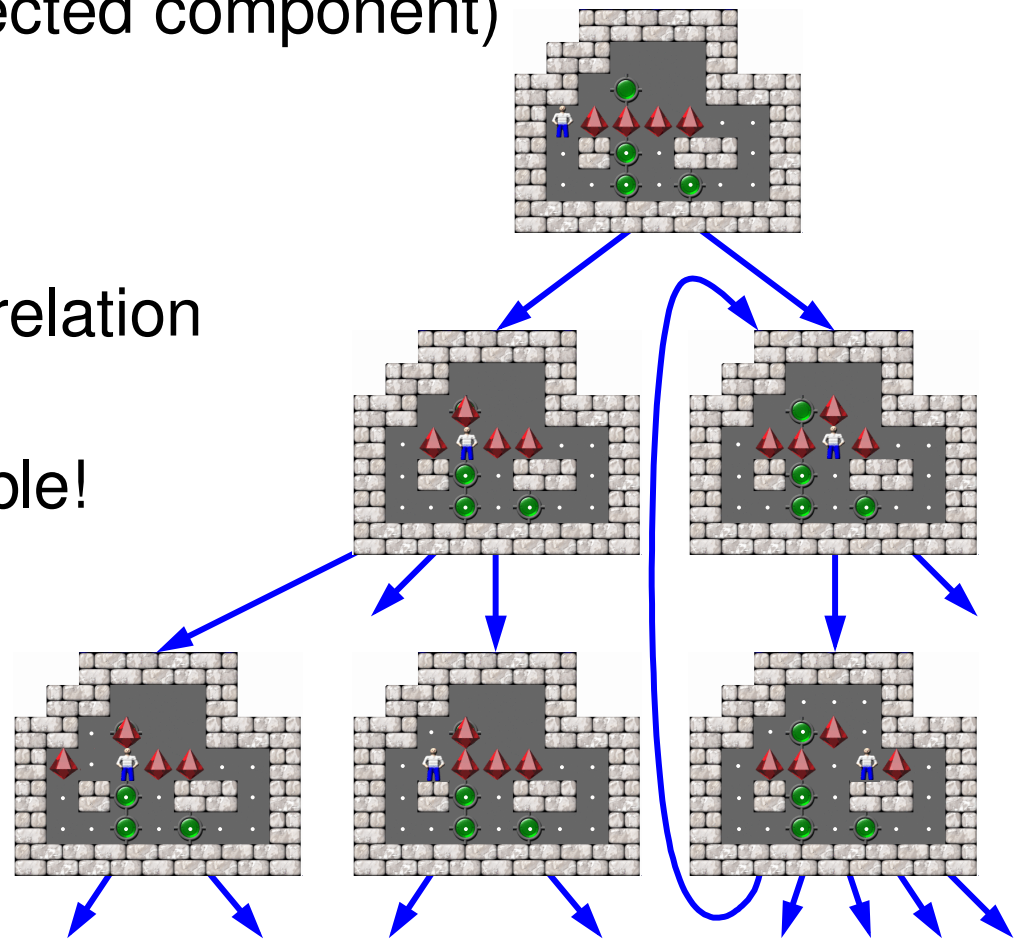
- ➔ Sokoban: japanese for “warehouse keeper”
- ➔ Computer game, developed in 1982 by Hiroyuki Imabayashi
- ➔ Goal: player must push all objects (boxes) to the target positions (storage locations)
 - ➔ boxes can only be pushed, not pulled
 - ➔ only one box can be pushed at a time





How to find the sequence of moves?

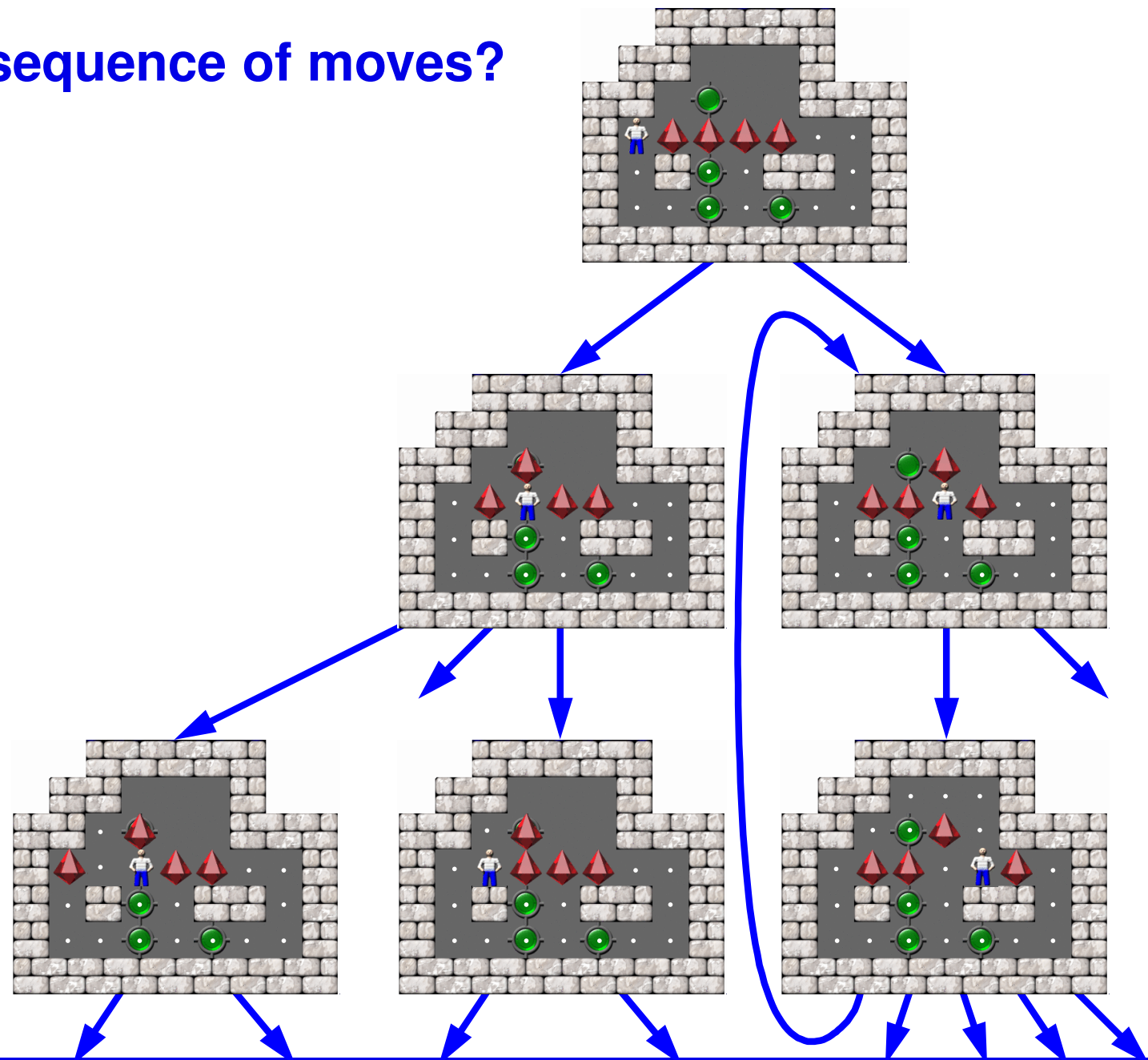
- ➔ Configuration: state of the play field
 - ➔ positions of the boxes
 - ➔ position of the player (connected component)
- ➔ Each configuration has a set of successor configurations
- ➔ Configurations with successor relation build a directed graph
 - ➔ no tree, as cycles are possible!
- ➔ Wanted: shortest 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?

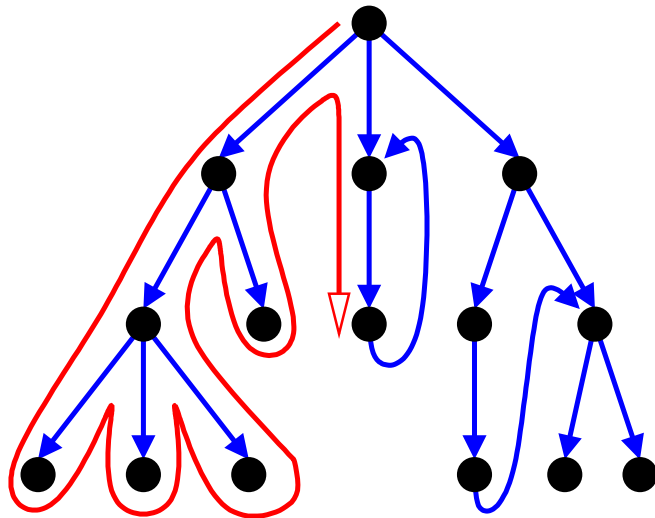




How to find the sequence of moves? ...

➔ Two alternatives:

➔ depth first search

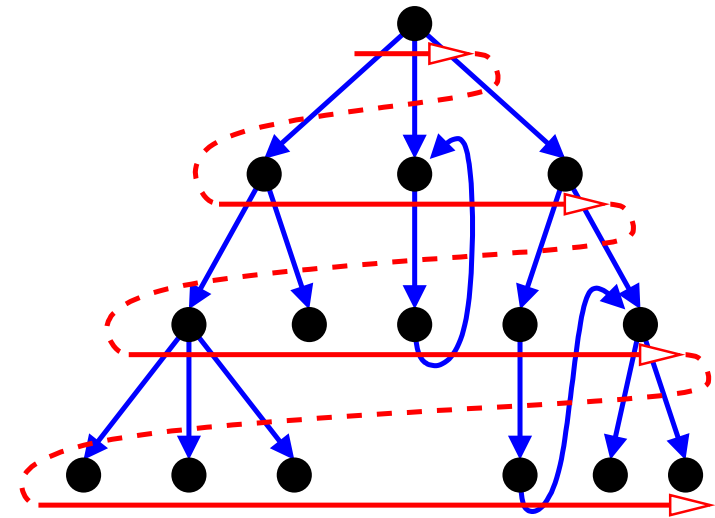


➔ problems:

➔ cycles

➔ handling paths with different lengths

➔ breadth first search



➔ problems:

➔ reconstruction of the path to a node

➔ memory requirements



Backtracking algorithm for depth first search:

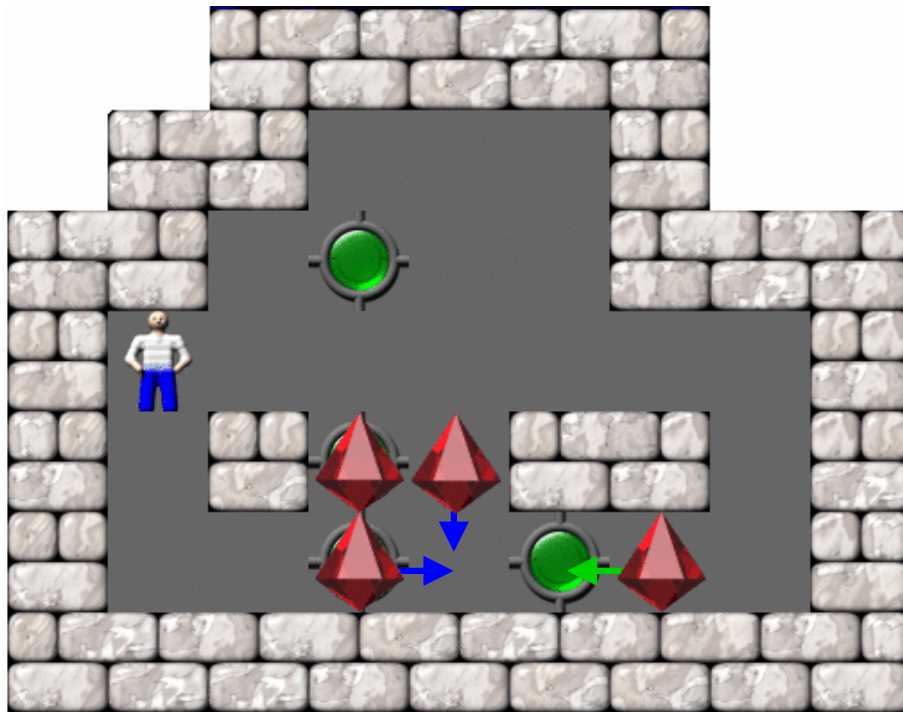
```
DepthFirstSearch(conf): // conf = current configuration
  append conf to the solution path
  if conf is a solution configuration:
    found the solution path
    return
  if current depth  $\geq$  depth of the best solution so far:
    remove the last element from the solution path
    return // 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 ...

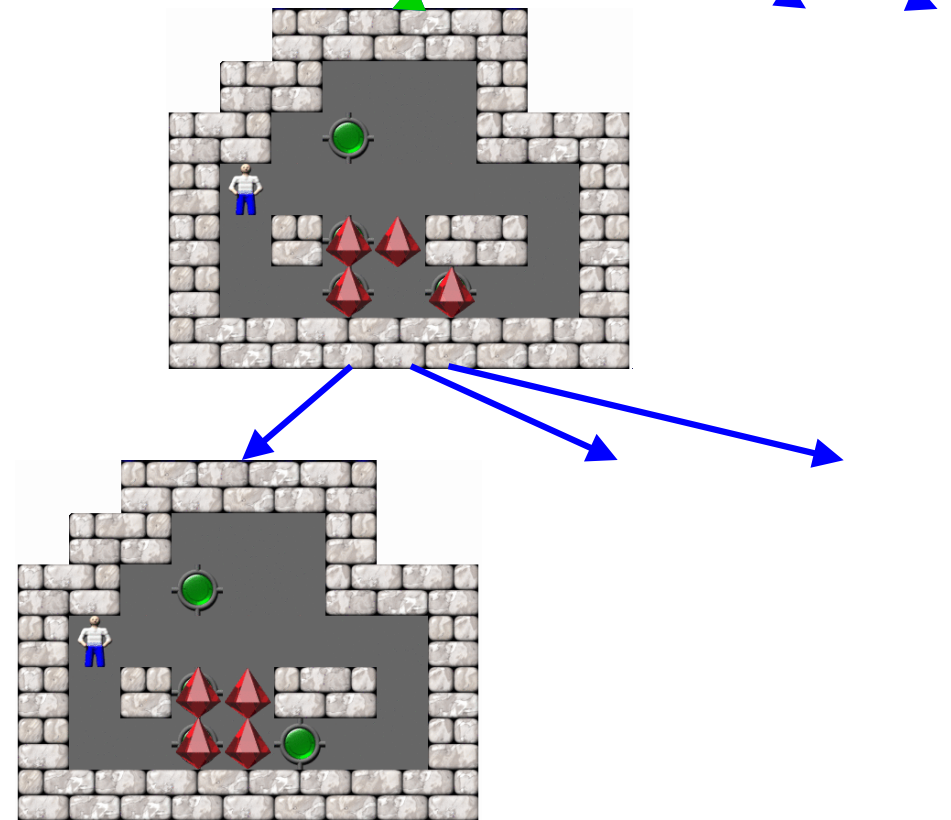
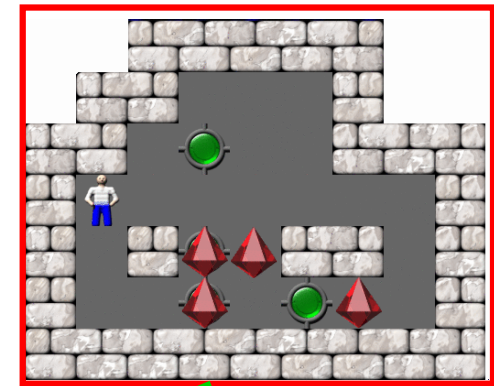


Example

Configuration with possible moves



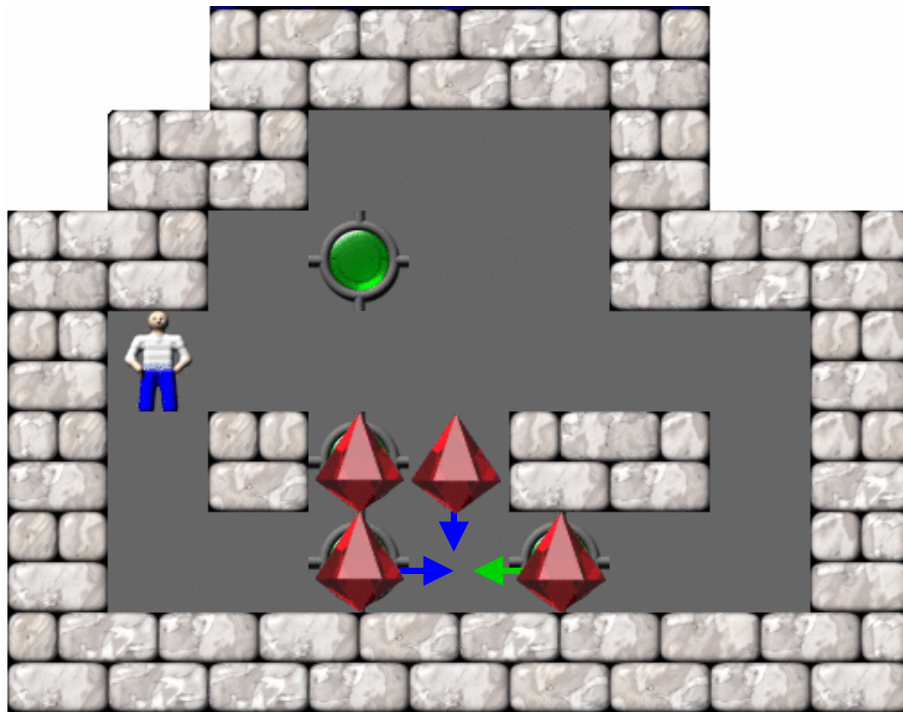
- ← Possible move
- ← Chosen move



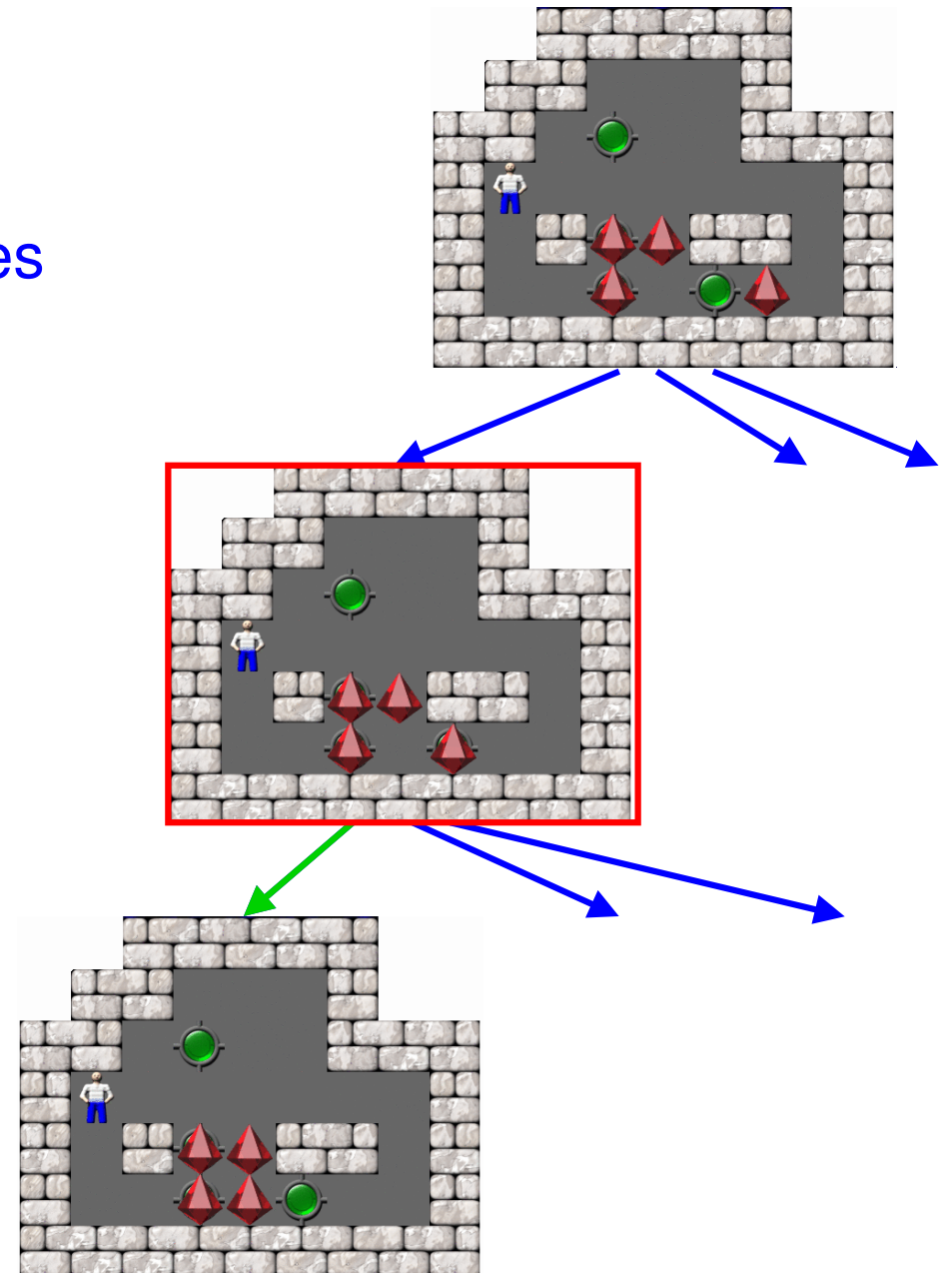


Example

Move has been executed
New configuration with possible moves



- ← Possible move
- ← Chosen move

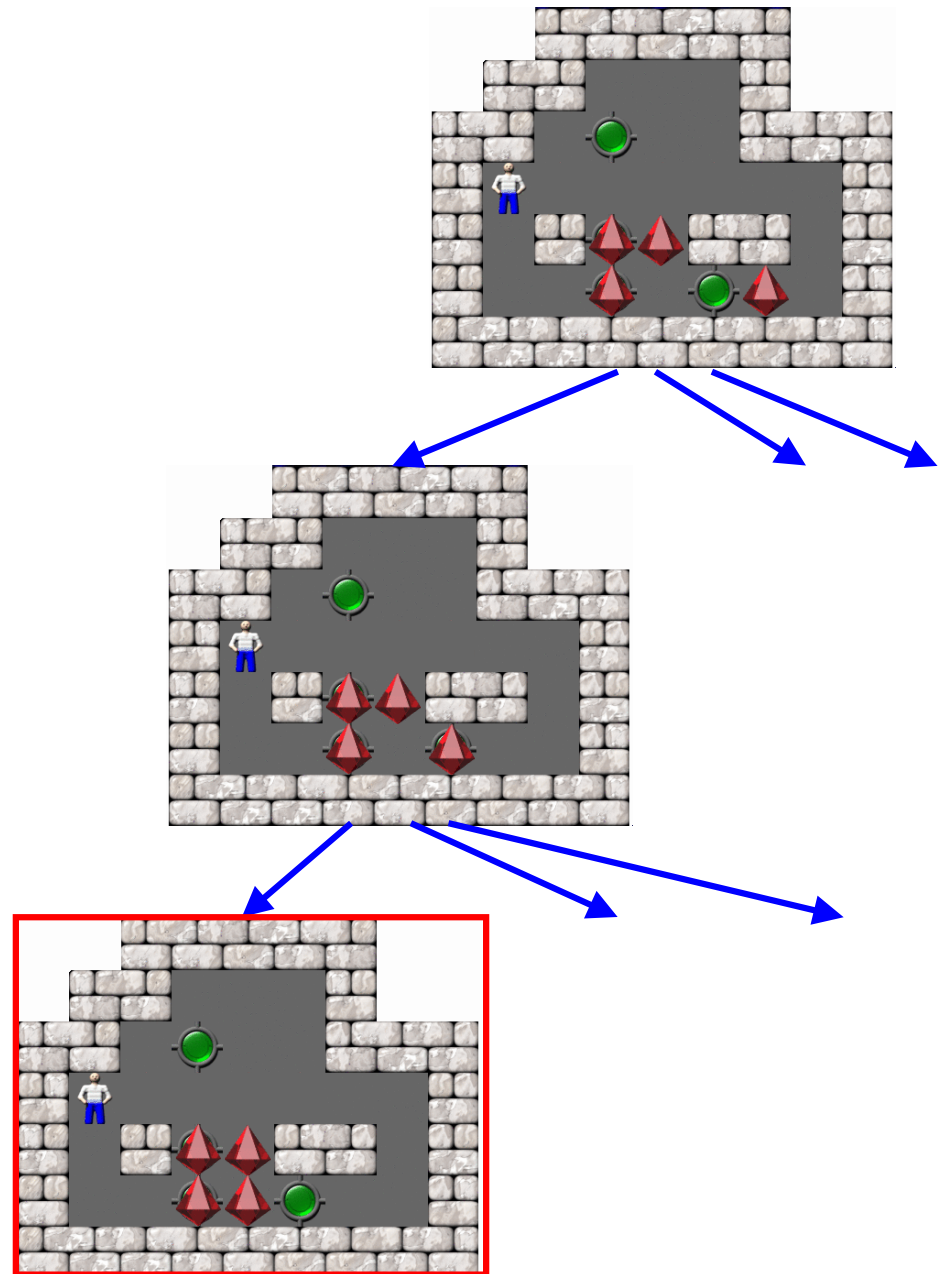
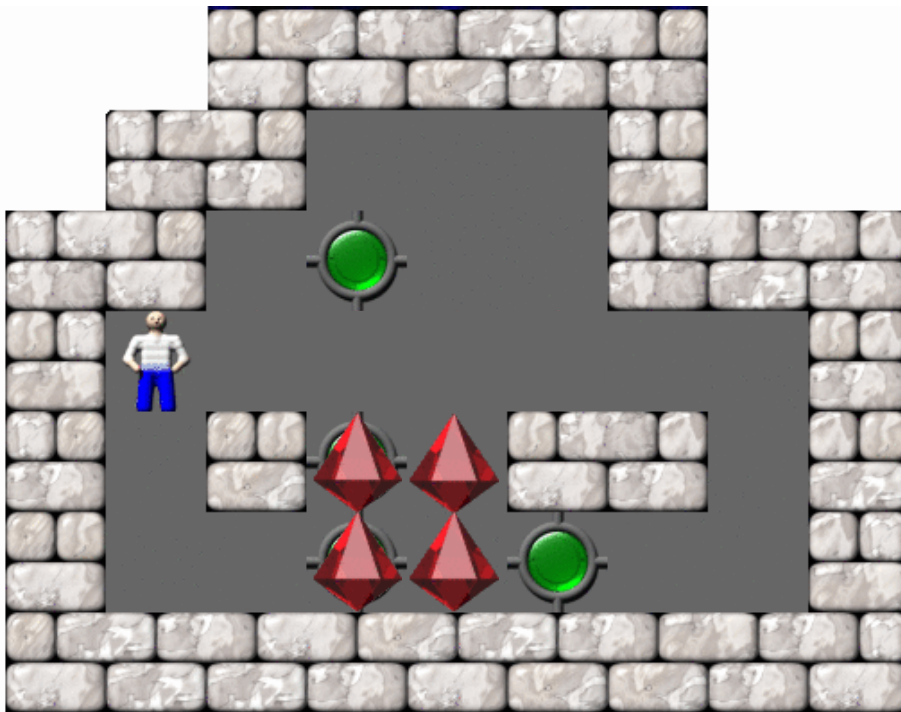


3.7 Exercise: A Solver for the Sokoban Game ...



Example

Move has been executed
No further move is possible



- ← Possible move
- ← Chosen move

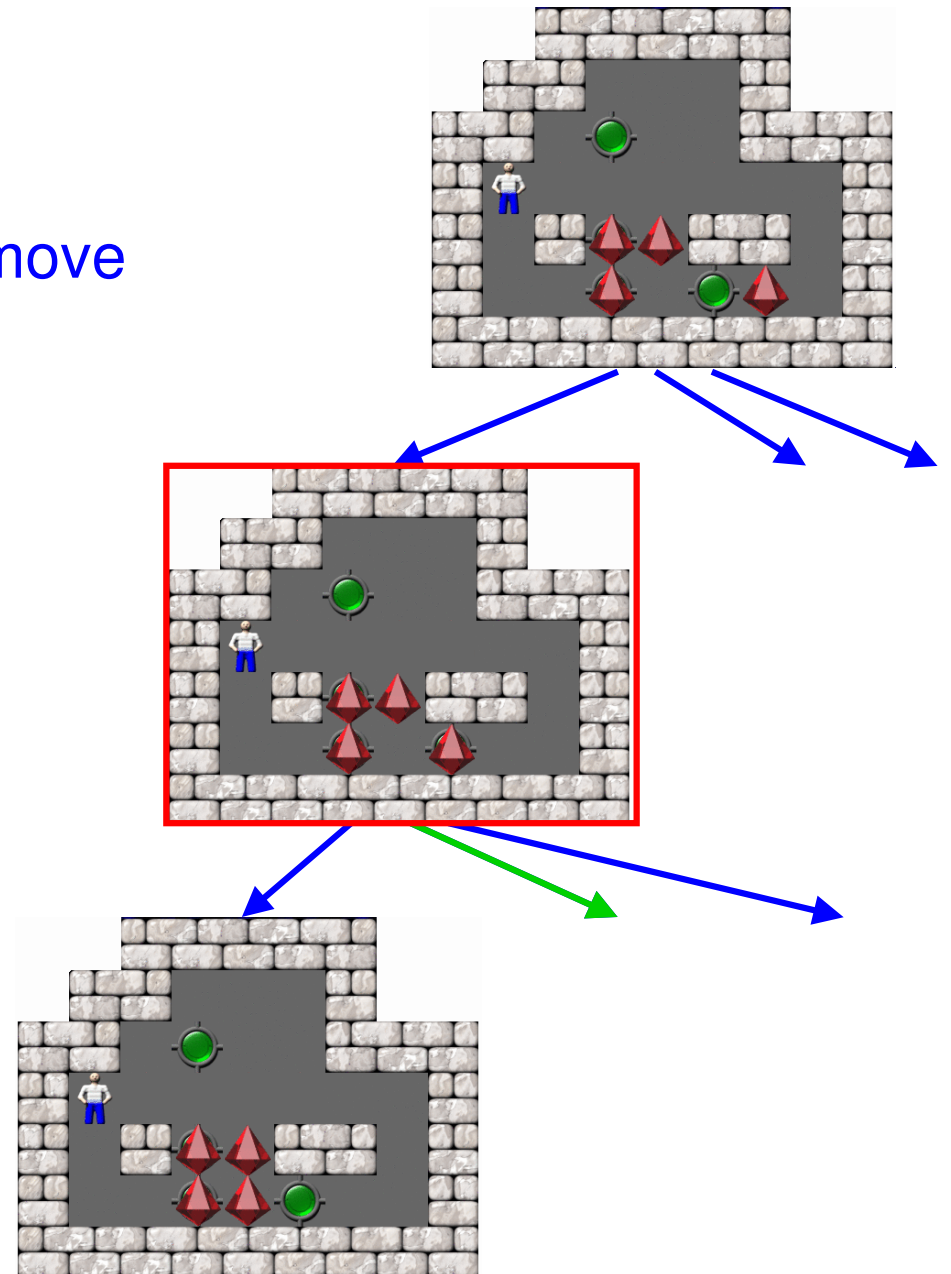
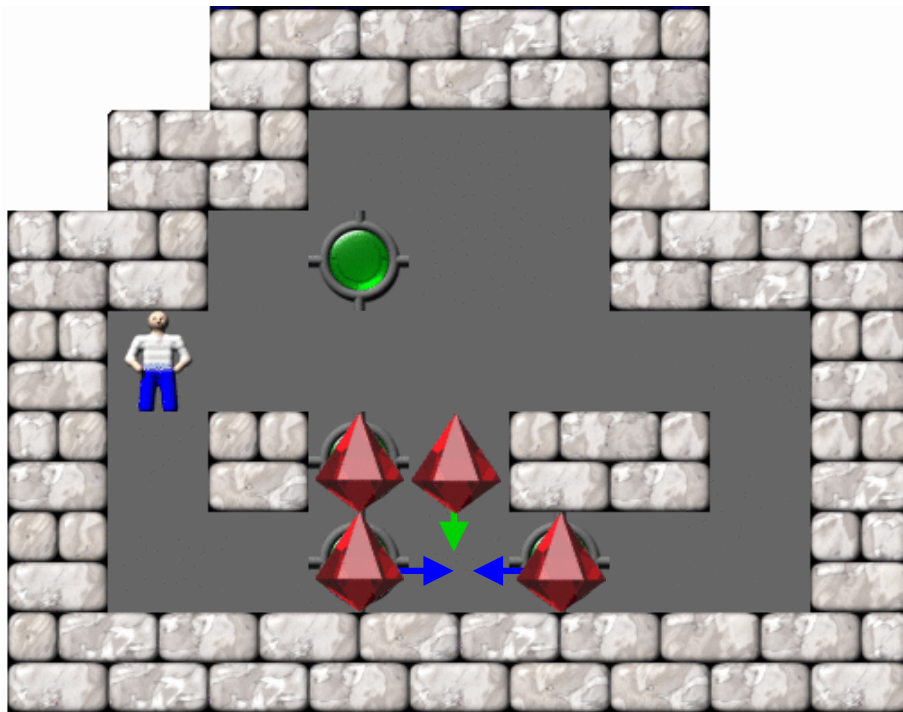
3.7 Exercise: A Solver for the Sokoban Game ...



Example

Backtrack

Back to previous configuration, next move



- ← Possible move
- ← Chosen move



Algorithm for breadth first search:

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



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



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 *val) {  
    if (*addr == exp) {  
        *addr = val;  
        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));  
}
```



- ➔ Problem: 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`, `liblfd`s
 - ➔ 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_...()`
 - ➔ OpenMP: `#pragma omp atomic`