Parallel Processing

WS 2020/21

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

Stand: October 22, 2020

Parallel Processing

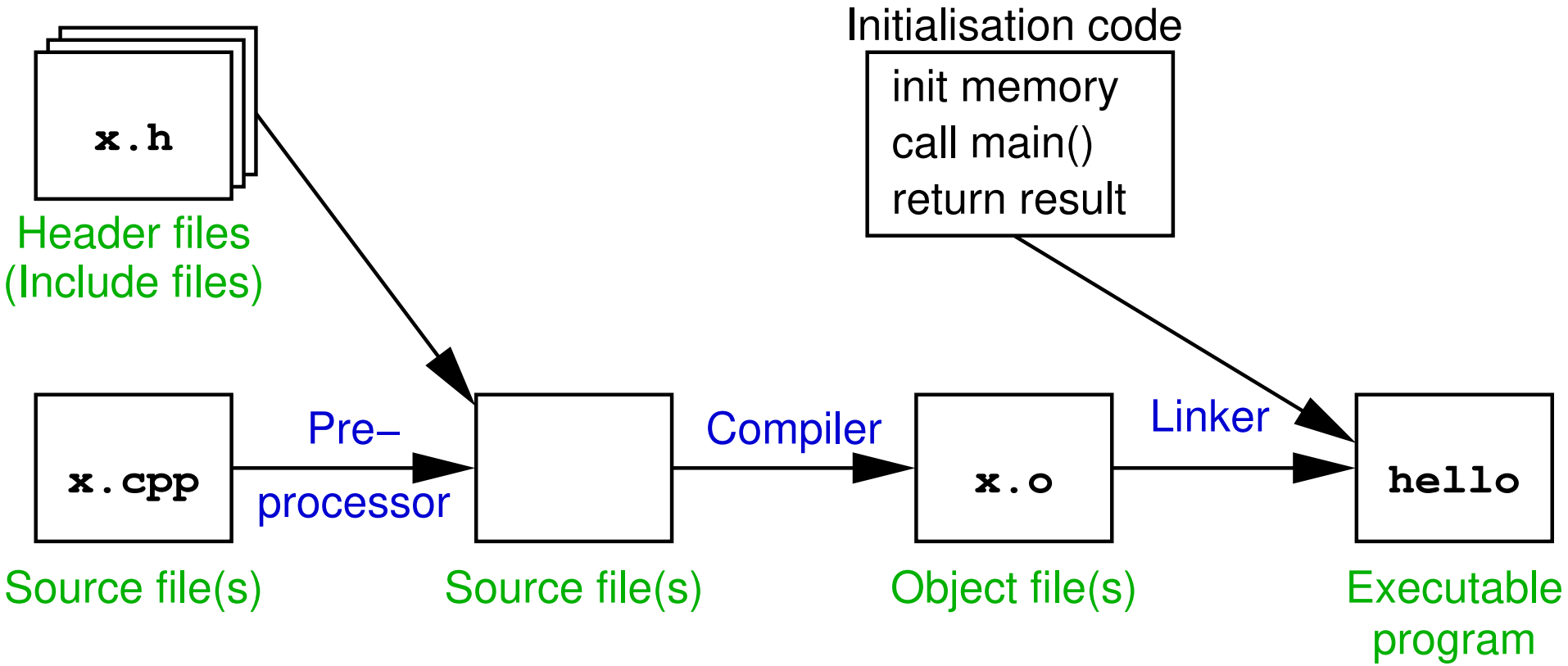
WS 2020/21

5 Appendix

5.1.1 Fundamentals of C++

- Commonalities between C++ and Java:
  - imperative programming language
  - syntax is mostly identical
- Differences between C++ and Java:
  - C++ is not purely object oriented
  - C++ programs are translated directly to machine code (no virtual machine)
- Usual file structure of C++ programs:
  - header files (`*.h`) contain declarations
    - types, classes, constants, ...
  - source files (`*.cpp`) contain implementations
    - methods, functions, global variables

Compilation of C++ programs



- Preprocessor: embedding of files, expansion of macros
- Linker: binds together object files and libraries

Compilation of C++ programs ...

- Invocation of the compiler in the lab room H-A 4111:
  - `g++ -Wall -o <output-file> <source-files>`
  - executes preprocessor, compiler and linker
  - `-Wall`: report all warnings
  - `-o <output-file>`: name of the executable file
- Additional options:
  - `-g`: enable source code debugging
  - `-O`: enable code optimization
  - `-l<library>`: link the given library
  - `-c`: do not execute the linker
    - later: `g++ -o <output-file> <object-files>`

An example: *Hello World*! (☞ 05/hello.cpp)

```
#include <iostream>
```
Preprocessor directive: inserts contents of file 'iostream' (e.g., declaration of cout)

```
using namespace std;
```
Use the namespace 'std'

```
void sayHello()
{
cout << "Hello World\n";
}
```
Function definition

Output of a text

```
int main()
{
sayHello();
return 0;
}
```
Main program

Return from main program: 0 = OK, 1,2,...,255: error

- Compilation: `g++ -Wall -o hello hello.cpp`
- Start: `./hello`

Syntax

- Identical to Java are among others:
  - declaration of variables and parameters
  - method calls
  - control statements (`if`, `while`, `for`, `case`, `return`, ...)
  - simple data types (`short`, `int`, `double`, `char`, `void`, ...)
    - deviations: `bool` instead of `boolean`; `char` has a size of 1 Byte
  - virtually all operators (`+`, `*`, `%`, `<<`, `==`, `?:`, ...)
- Very similar to Java are:
  - arrays
  - class declarations

Arrays

- Declaration of arrays
  - only with fixed size, e.g.:

```
int ary1[10];                  // int array with 10 elements
double ary2[100][200];         // 100 * 200 array
int ary3[] = { 1, 2 };         // int array with 2 elements
```

  - for parameters: size can be omitted for **first** dimension

```
int funct(int ary1[], double ary2[][200]) { ... }
```

- Arrays can also be realized via pointers (see later)
  - then also dynamic allocation is possible
- Access to array elements
  - like in Java, e.g.: `a[i][j] = b[i] * c[i+1][j];`
  - but: **no** checking of array bounds!!!

Classes and objects

- Declaration of classes (typically in `.h` file):

```
class Example {
 private:              // private attributes/methods
   int attr1;                       // attribute
   void pmeth(double d);            // method
 public:               // public attributes/methods
   Example();                       // default constructor
   Example(int i);                  // constructor
   Example(Example &from);          // copy constructor
   ~Example();                      // destructor
   int meth();                      // method
   int attr2;                       // attribute
   static int sattr;                // class attribute
};
```

Classes and objects ...

- Definition of class attributes and methods (`*.cpp` file):

```
int Example::sattr = 123; // class attribute

Example::Example(int i) { // constructor
  this->attr1 = i;
}
int Example::meth() {     // method
  return attr1;
}
```

  - specification of class name with attributes and methods
    - separator `::` instead of .
  - `this` is a pointer (☞ **5.1.3**), thus `this->attr1`
  - alternatively, method bodies can also be specified in the class definition itself

Classes and objects ...

- Declaration of objects:

```
{
   Example ex1;        // initialisation using default constructor
   Example ex2(10);    // constructor with argument
   ...
} // now the destructor for ex1, ex2 is called
```

- Access to attributes, invocation of methods

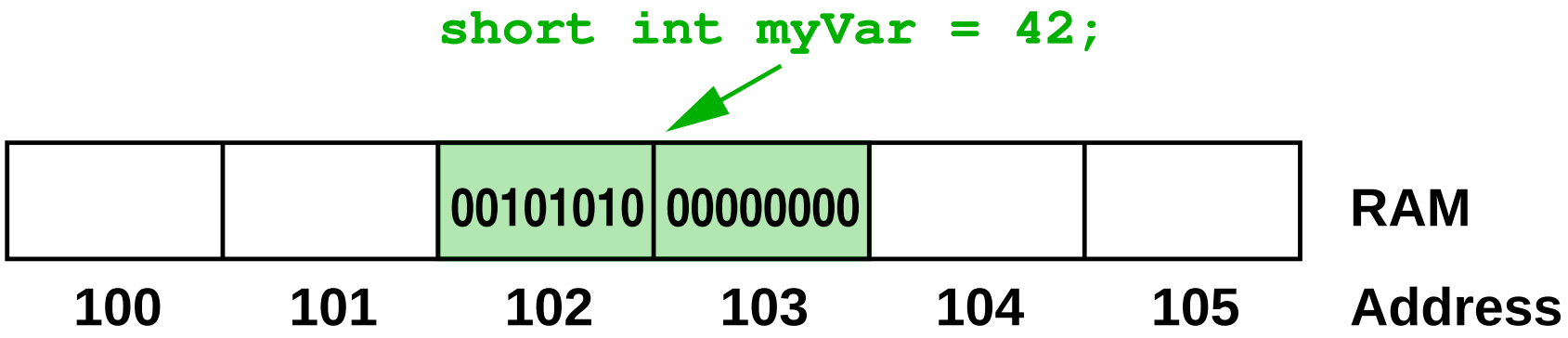
```
ex1.attr2 = ex2.meth();
j = Example::sattr;          // class attribute
```

- Assignment / copying of objects

```
ex1 = ex2;                   // object is copied!
Example ex3(ex2);  // initialisation using copy constructor
```

Variables in memory

- Reminder: variables are stored in main mamory

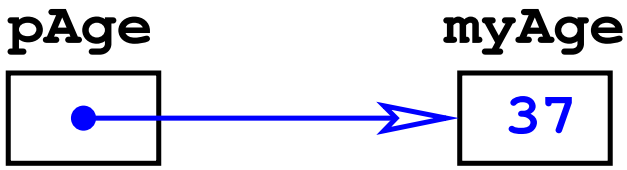


  - a variable gives a name and a type to a memory block
    - here: `myVar` occupies 2 bytes (`short int`) starting with address 102
- A **pointer** is a memory address, together with a type
  - the type specifies, how the memory block is interpreted

Declaration and use of pointers

- Example:

```
int myAge = 25;      // an int variable
int *pAge;           // a pointer to int values
pAge = &myAge;       // pAge now points to myAge
*pAge = 37;          // myAge now has the value 37
```



- The **address operator** `&` determines the adress of a variable
- The access to `*pAge` is called **dereferencing** `pAge`
- Pointers (nearly) always have a type
  - e.g. `int *`, `Example *`, `char **`, ...

Passing parameters *by reference*

- Pointers allow to pass parameters *by reference*
- Instead of a value, a **pointer** to the values is passed:

```
void byReference(Example *e, int *result) {
    *result = e->attr2;
}
int main() {
    Example obj(15);                // obj is more efficiently
    int res;                        //    passed by reference
    byReference(&obj, &res);        // res is a result parameter
    ...
```

  - short notation: `e->attr2` means `(*e).attr2`

`void` pointers and type conversion

- C++ also allows the use of generic pointers
  - just a memory addess without type information
  - declared type is `void *` (pointer to `void`)
- Dereferencing only possible after a type conversion
  - caution: no type safety / type check!
- Often used for generic parameters of functions:

```
void bsp(int type, void *arg) {
    if (type == 1) {
        double d = *(double *)arg;   // arg must first be converted
                                     //  to double *
    } else {
        int i = *(int *)arg;         // int argument
```

Arrays and pointers

- C++ does not distinguish between one-dimensional arrays and pointers (with the exception of the declaration)
- Consequences:
  - array variables can be used like (constant) pointers
  - pointer variables can be indexed

```
int a[3] = { 1, 2, 3 };
int b = *a;                 // equivalent to: b = a[0]
int c = *(a+1);             // equivalent to: c = a[1]
int *p = a;                 // equivalent to: int *p = &a[0]
int d = p[2];               // d = a[2]
```

Arrays and pointers ...

- Consequences ...:
  - arrays as parameters are always passed *by reference*!

```
void swap(int a[], int i, int j) {
    int h = a[i];    // swap a[i] and a[j]
    a[i] = a[j];
    a[j] = h;
}
int main() {
    int ary[] = { 1, 2, 3, 4 };
    swap(ary, 1, 3);
    // now:  ary[1] = 4,  ary[3] = 2;
}
```

Dynamic memory allocation

- Allocation of objects and arrays like in Java

```
Example *p = new Example(10);
int *a = new int[10];              // a is not initialised!
int *b = new int[10]();            // b is initialised (with 0)
```

  - allocation of multi-dimensional arrays does not work in this way

- Important: C++ does not have a garbage collection
  - thus explicit deallocation is necessary:

```
delete p;     // single object
delete[] a;   // array
```

  - caution: do not deallocate memory multiple times!

Function pointers

- Pointers can also point to functions:

```
void myFunct(int arg) { ... }
void test1() {
    void (*ptr)(int) = myFunct;  // function pointer + init.
    (*ptr)(10);                  // function call via pointer
```

- Thus, functions can, e.g., be passed as parameters to other functions:

```
void callIt(void (*f)(int)) {
    (*f)(123);          // calling the passed function
}
void test2() {
   callIt(myFunct);  // function as reference parameter
```

5.1.4 Strings and Output

- Like Java, C++ has a string class (`string`)
  - sometimes also the type `char *` is used
- For console output, the objects `cout` and `cerr` are used
- Both exist in the name space (packet) `std`
  - for using them without name prefix:

```
using namespace std; // corresponds to 'import std.*;' in Java
```

- Example for an output:

```
double x = 3.14;
cout << "Pi ist approximately " << x << "\n";
```

- Special formatting functions for the output of numbers, e.g.:

```
cout << setw(8) << fixed << setprecision(4) << x << "\n";
```

  - output with a field length of 8 and exacly 4 decimal places

5.1.5 Further specifics of C++

- **Global** variables
  - are declared outside any function or method
  - live during the complete program execution
  - are accessible by all functions
- Global variables and functions can be used only **after** the declaration
  - thus, for functions we have **function prototypes**

```
int funcB(int n);          // function prototype
int funcA() {              // function definition
    return funcB(10);
}
int funcB(int n) {         // function definition
    return n * n;
}
```

5.1.5 Further specifics of C++ ...

- Keyword `static` used with the declaration of gloabal variables or functions

```
static int number;
static void output(char *str) { ... }
```

  - causes the variable/function to be usable only in the local source file

- Keyword `const` used with the declaration of variables or parameters

```
const double PI = 3.14159265;
void print(const char *str) { ... }
```

  - causes the variables to be read-only
  - roughly corresponds to `final` in Java
  - (note: this description is extremely simplified!)

5.1.5 Further specifics of C++ ...

- Passing command line arguments:

```
int main(int argc, char **argv) {
    if (argc > 1)
        cout << "Argument 1: " << argv[1] << "\n";
}
```

Example invocation:

```
bslab1% ./myprog -p arg2
Argument 1:  -p
```

- `argc` is the number of arguments (incl. program name)
- `argv` is an array (of length `argc`) of strings (`char *`)
- in the example:

```
argv[0] = "./myprog"
argv[1] = "-p"
argv[2] = "arg2"
```

- important: check the index against `argc`

Overview

- There are several (standard) libraries for C/C++, which always come with one or more header files, e.g.:

Header file	Library (g++ option)	Description	contains, e.g.
iostream string stdlib.h sys/time.h		input/output C++ strings standard funct. time functions	cout, cerr string exit() gettimeofday()
math.h	-lm	math functions	sin(), cos(), fabs()
pthread.h	-lpthread	threads	pthread_create()
mpi.h	-lmpich	MPI	MPI_Init()

Functions of the preprocessor:

- Embedding of header file

```
#include <stdio.h>   // searches only in system directories
#include "myhdr.h"   // also searches in current directory
```

- Macro expansion

```
#define BUFSIZE   100            // Constant
#define VERYBAD   i + 1;         // Extremely bad style !!
#define GOOD      (BUFSIZE+1)    // Parenthesis are important!
...
  int i = BUFSIZE;     // becomes int i = 100;
  int a = 2*VERYBAD    // becomes int a = 2*i + 1;
  int b = 2*GOOD;      // becomes int a = 2*(100+1);
```

Functions of the preprocessor: ...

- Conditional compliation (e.g., for debugging output)

```
int main() {
#ifdef DEBUG
    cout << "Program has started\n";
#endif
    ...
}
```

  - output statement normally will not be compiled
  - to activate it:
    - either `#define DEBUG` at the beginning of the program
    - or compile with `g++ -DDEBUG ...`

5.2.1 Compilation and Execution

- Compilation: use gcc (`g++`)
  - typical call: `g++ -fopenmp myProg.cpp -o myProg`
  - OpenMP 3.0 since gcc 4.4, OpenMP 4.0 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`

5.2.2 Debugging

- There are only few debuggers that fully support OpenMP
  - e.g., Totalview
  - requires tight cooperation between compiler and debugger
- On the PCs in the lab room H-A 4111:
  - `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”
- On the HorUS cluster: `totalview`
  - graphical debugger
  - supports both OpenMP and MPI

5.2.2 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 ...`

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`

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`

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
```

5.2.2 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
```

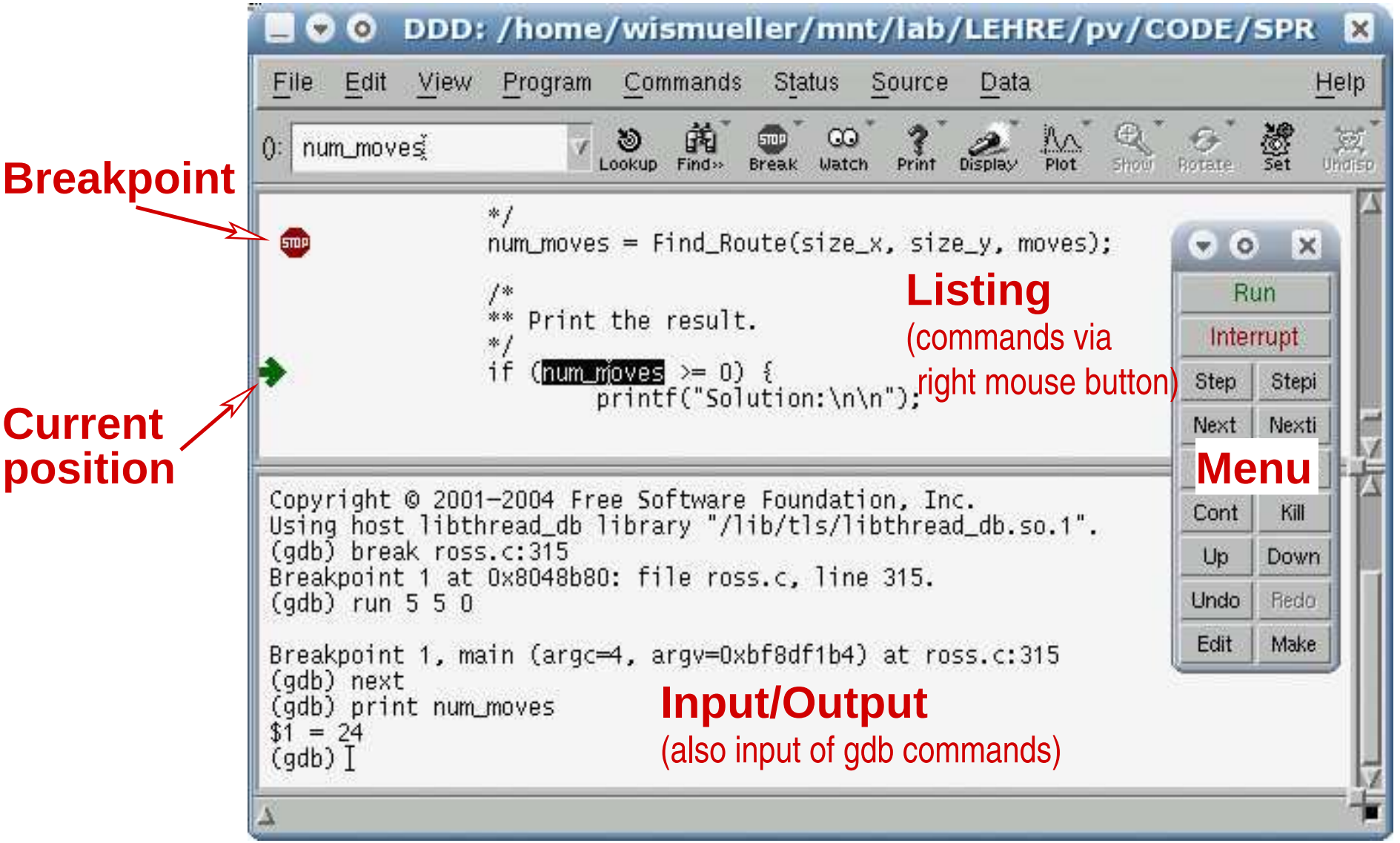
5.2.2 Debugging ...

```
(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>
```

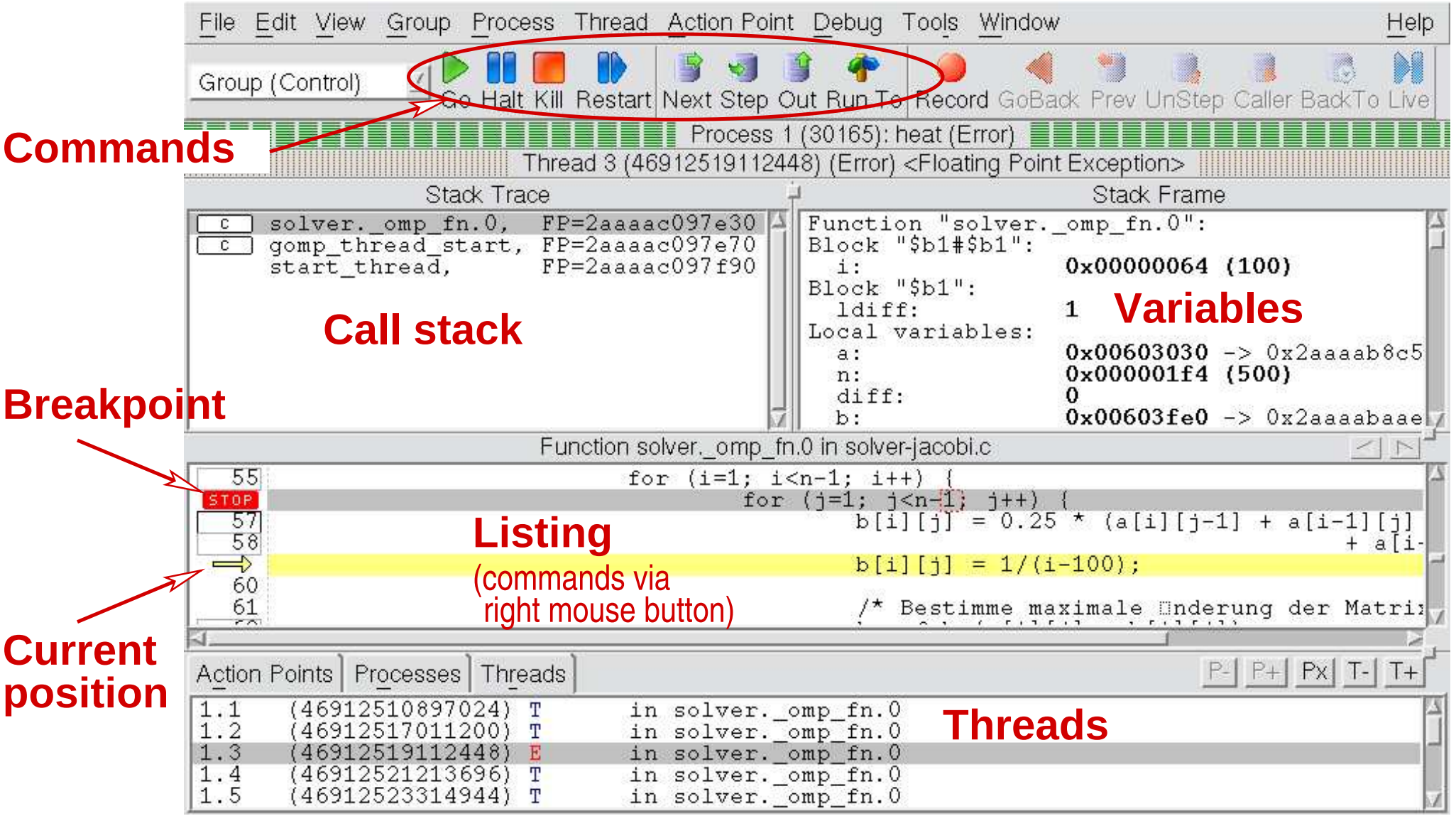
Sample session with gdb (OpenMP)

```
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
```

Sample session with ddd



Sample session with `totalview`



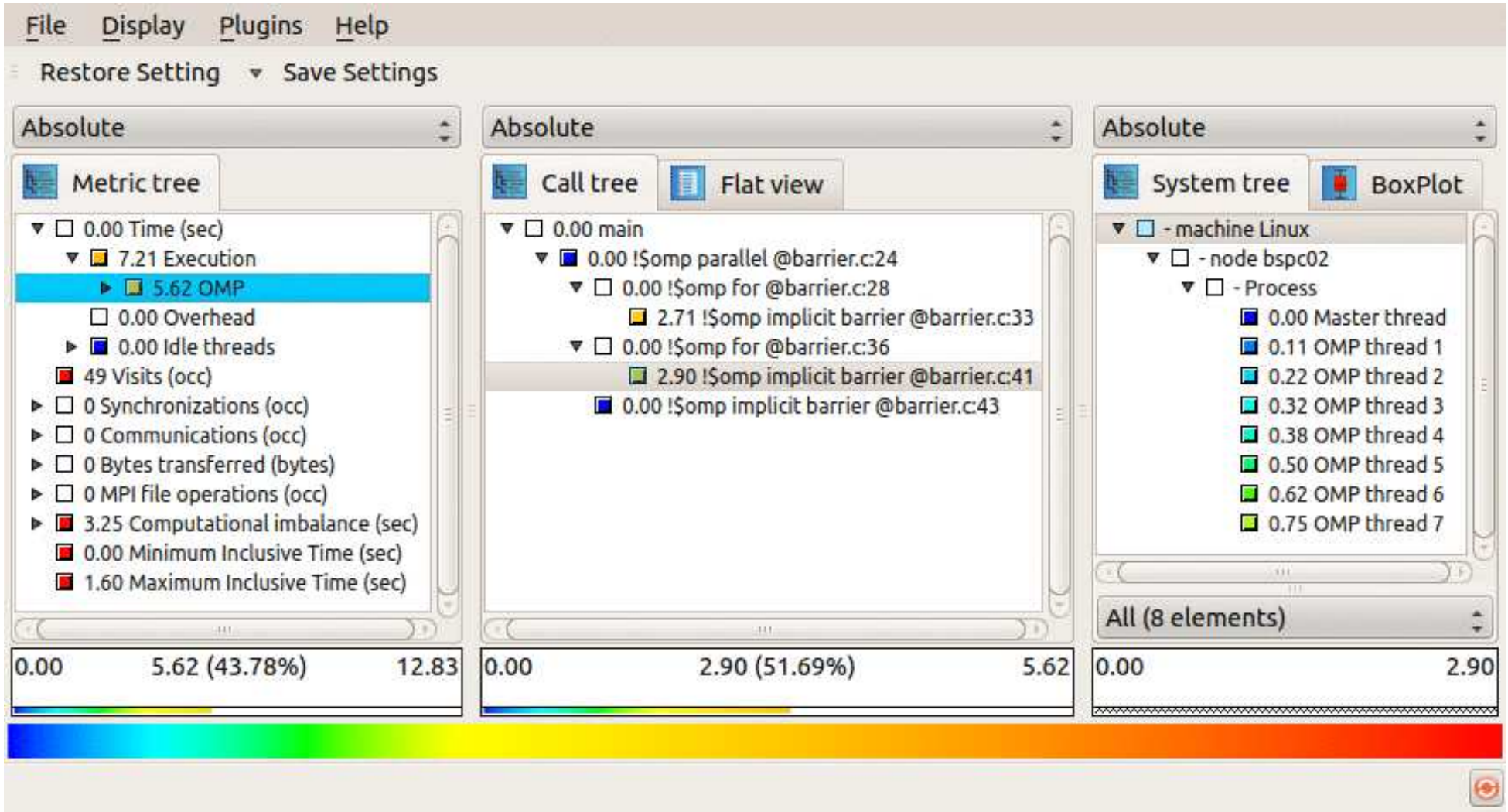
5.2.3 Performance Analysis

- 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 1.9.6
- In the H-A 4111: Scalasca

Performance analysis using Scalasca

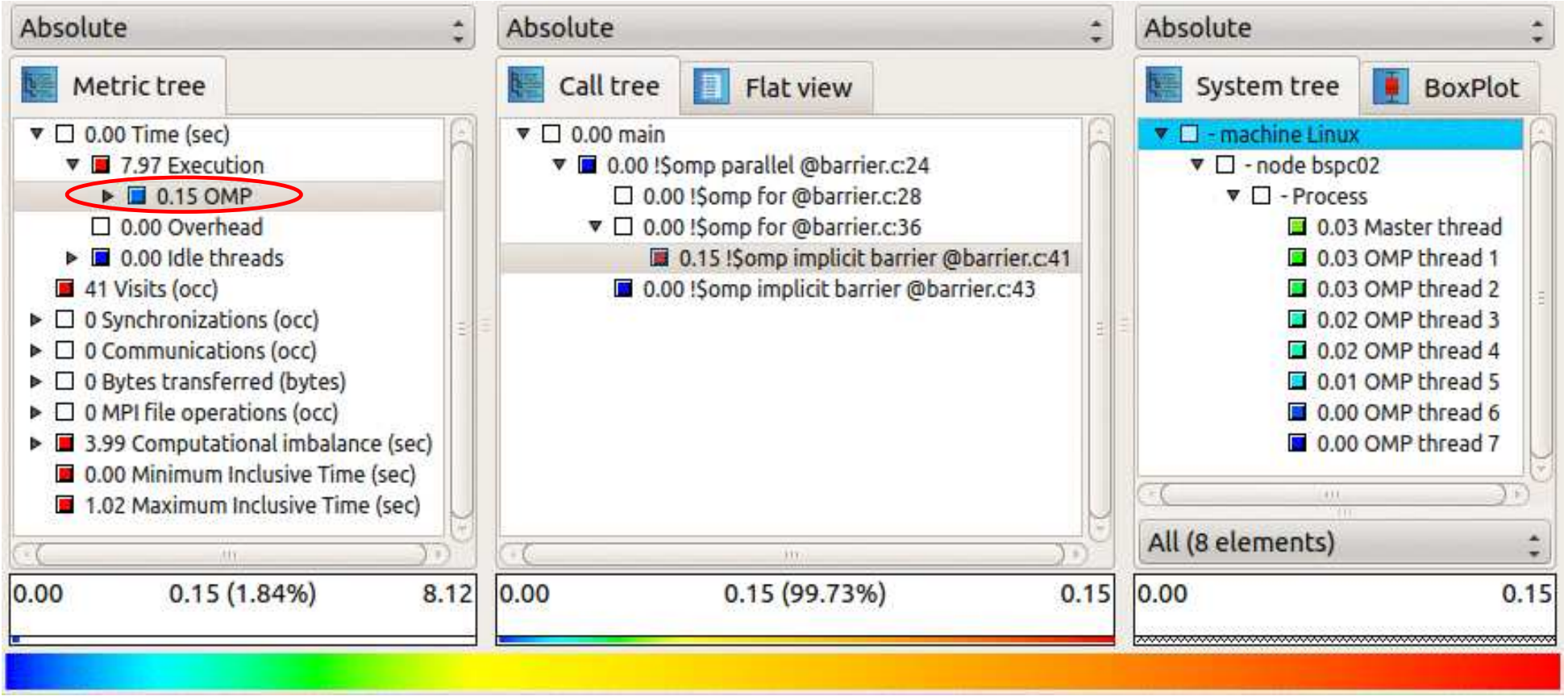
- Set the paths (in H-A 4111; HorUS: see 5.2.4)
  - `export PATH=/opt/dist/scorep-1.4.1/bin:\`
    `/opt/dist/scalasca-2.2.1/bin:$PATH`
- Compile the program:
  - `scalasca -instrument g++ -fopenmp ...  barrier.cpp`
- Execute the program:
  - `scalasca -analyze ./barrrier`
  - stores data in a directory `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`

Performance analysis using Scalasca: Example from slide 217



Performance analysis using Scalasca: Example from slide 217 ...

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



Architecture of the HorUS cluster

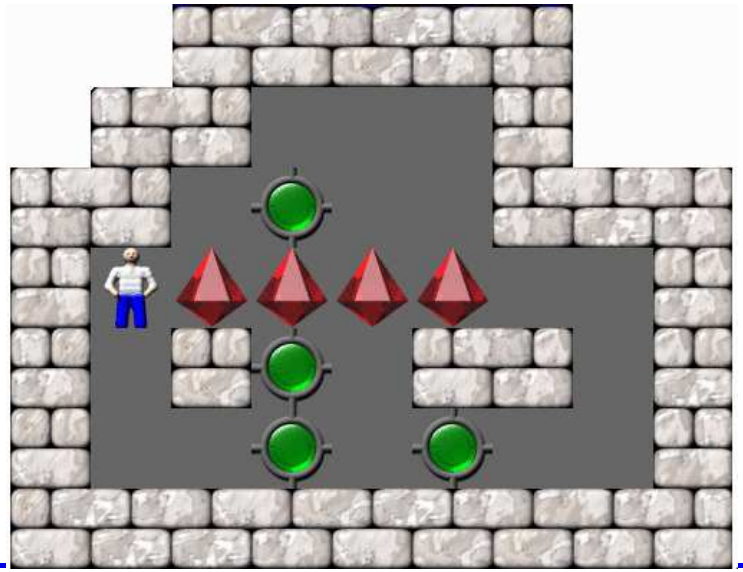
- 34 Dell PowerEdge C6100 systems, each with 4 nodes
- 136 compute nodes
  - CPU: 2 x Intel Xeon X5650, 2.66 GHz, 6 cores per CPU, 12 MB cache
  - main memory: 48 GB (4GB per core, 1333 MHz DRR3)
  - hard disk: 500 GB SATA (7,2k RPM, 3,5 Zoll)
- In total: 272 CPUs, 1632 cores, 6,4 TB RAM, 40 TB disk
- Parallel file system: 33 TB, 2.5 GB/s
- Infiniband network (40 GBit/s)
- Peak performance: 17 TFlop/s

Access

- Via SSH: `ssh -X` *g-account* `@hpc.zimt.uni-siegen.de`
- In the lab room H-A 4111:
  - forwarding of SSH connections by the lab gateway
  - `ssh -X -p 22222` *g-account* `@bslabgate1.lab.bvs`
  - ideally, use the file `$HOME/.ssh/config`:
    - ```
      Host horus
           user g-account
           hostname bslabgate1.lab.bvs
           ForwardX11 yes
           HostKeyAlias horus
           port 22222
      ```
    - then simply: `ssh horus`
      ```

Set-up the SSH in H-A 4111

- Create an SSH key:
  - `ssh-keygen -b 2048 -t rsa` (oder `-b 4096`)
  - when asked `"Enter file ..."` just press `Return`
  - choose a secure passphrase for your private key!
- Append the public key to the list of authorized keys:
  - `cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys`
- With that, you can also log into other lab computers without permanently having to type the password

Set-up the environment on HorUS

- Copy your (public!) SSH key to HorUS
  - `ssh-copy-id -i ~/.ssh/id_rsa.pub horus`
  - Important: take care that you type your password correctly!
- Define the modules/paths you need on the HorUS cluster:

```
module load DefaultModules
module load PrgEnv/gcc-openmpi/6.3.0-1.10.3
module load gcc/6.3.0
# Path for Scalasca:
export PATH=$PATH:/home/gl930/Scalasca/bin
```

  - the best is to append these commands to `~/.bash_profile`

Using the HorUS cluster in practice

- Mount the file system of HorUS onto your lab computer
  - create a directory for the mount point: `mkdir ~/mnt`
  - mount the HorUS file system: `sshfs horus: ~/mnt`
  - unmount: `fusermount -u ~/mnt`
- Starting programs on HorUS
  - HorUS uses the batch queueing system SLURM
    - see https://computing.llnl.gov/linux/slurm
  - starting an OpenMP program, e.g.:
    - ```
      export OMP_NUM_THREADS=8
      salloc --exclusive --partition short \
             $HOME/GAUSS/heat 500
      ```
      ```

Background

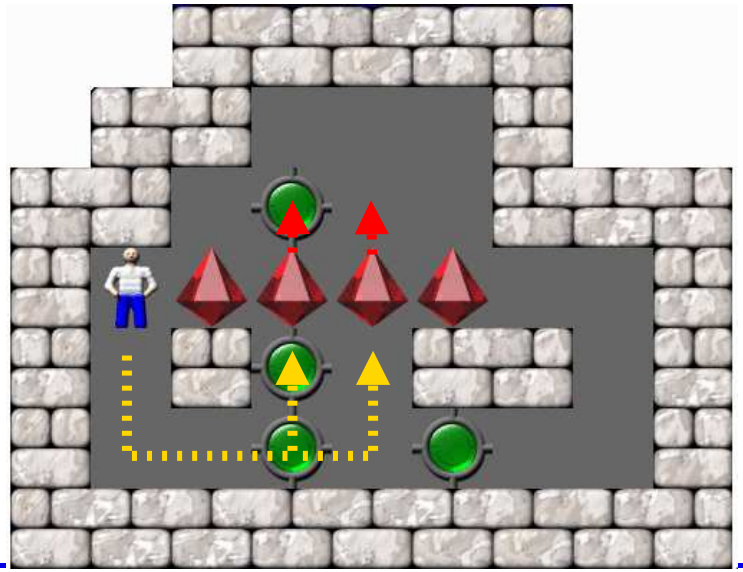
- Sokoban: japanese for “warehouse keeper”
- Computer game, developed in 1982 by Hiroyuki Imabayashi
- Goal: player must push all objects (boxes) to the target positions (storage locations)
  - boxes can only be pushed, not pulled
  - only one box can be pushed at a time



5.3 Exercise: Sokoban Solver

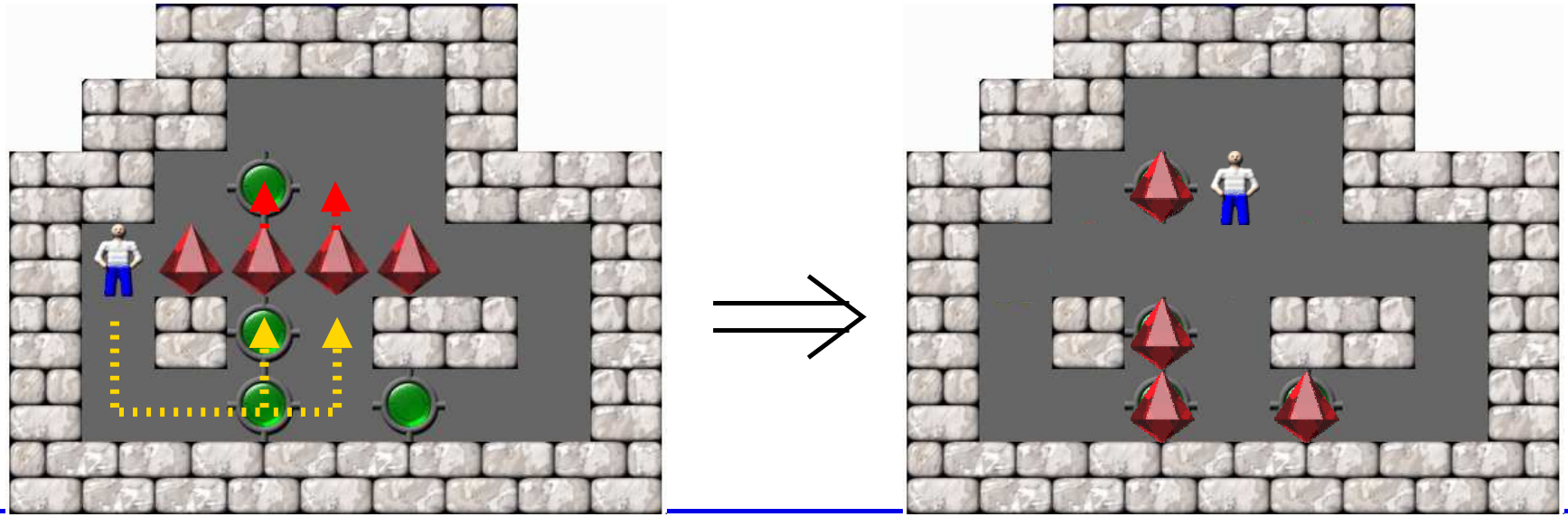
Background

- 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



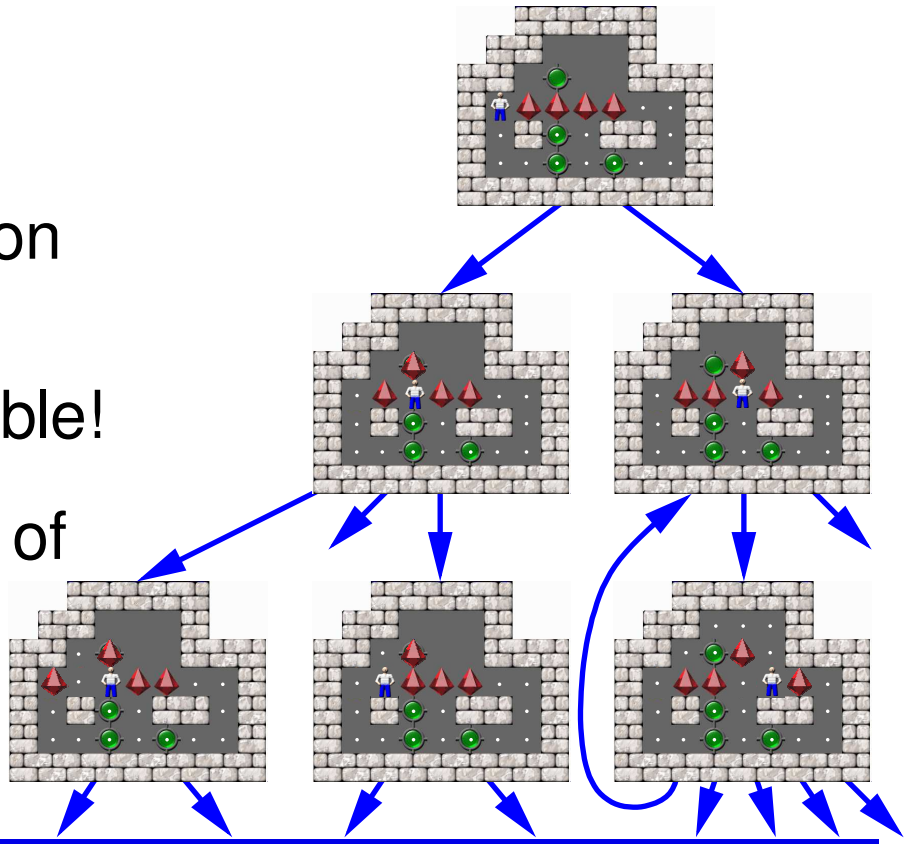
Background

- 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

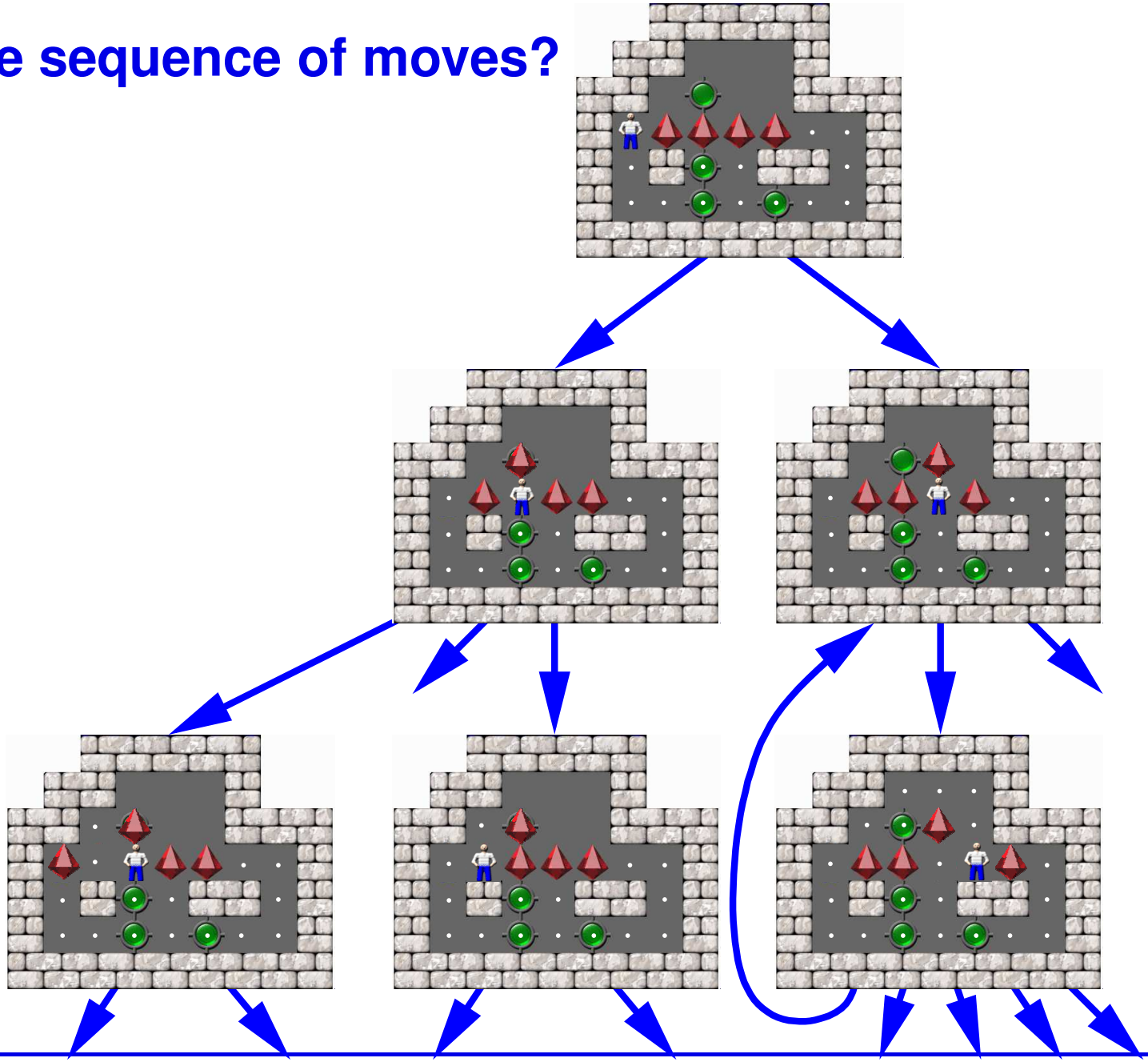


How to find the sequence of moves?

- 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
  - 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 moves

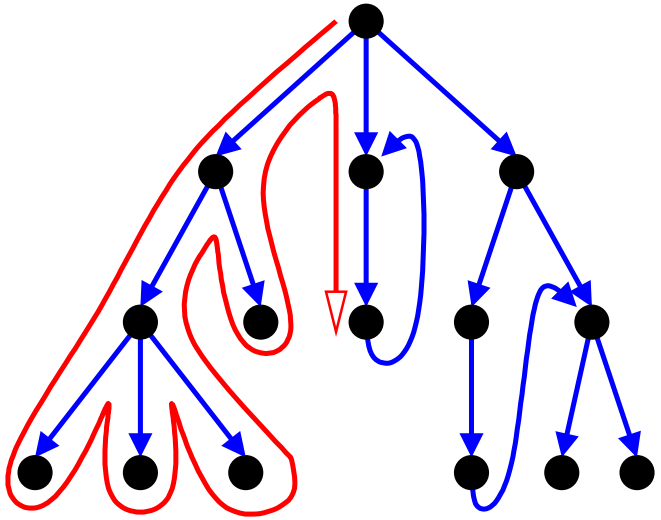


How to find the sequence of moves?

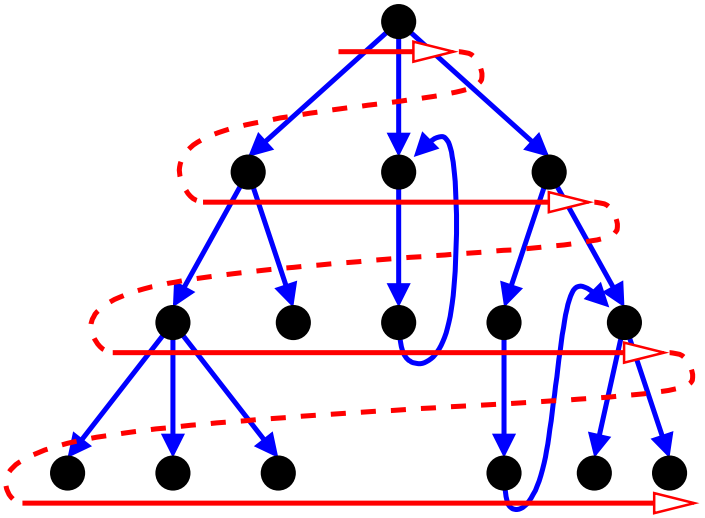


How to find the sequence of moves? ...

- 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:

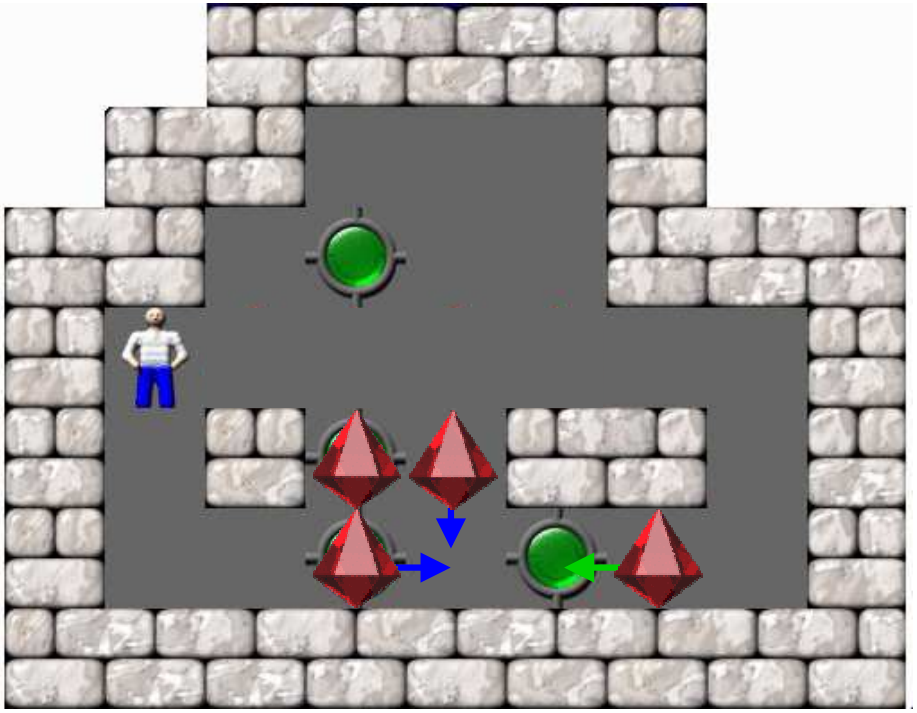
```
DepthFirstSearch(conf):    // conf = current configuration
    append conf to the soultion path
    if conf is a solution configuration:
        found the solution path
        return
    if depth is larger than the depth of the best solution so far:
        remove the last element from the solution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

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
```

Example for the *backtracking* algorithm

Configuration with possible moves

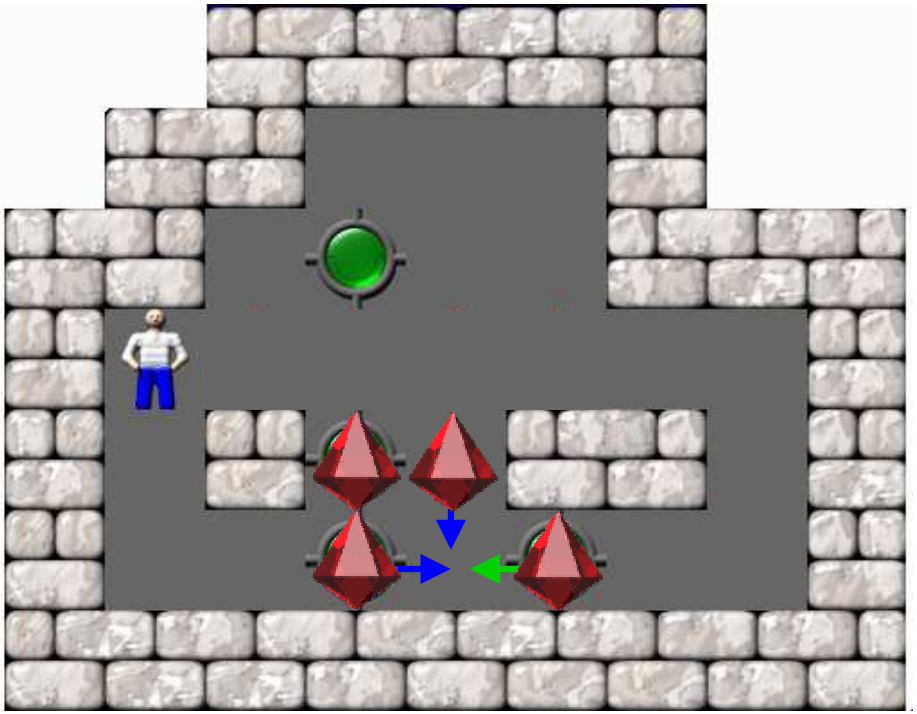


← Possible move

← Chosen move

Example for the *backtracking* algorithm

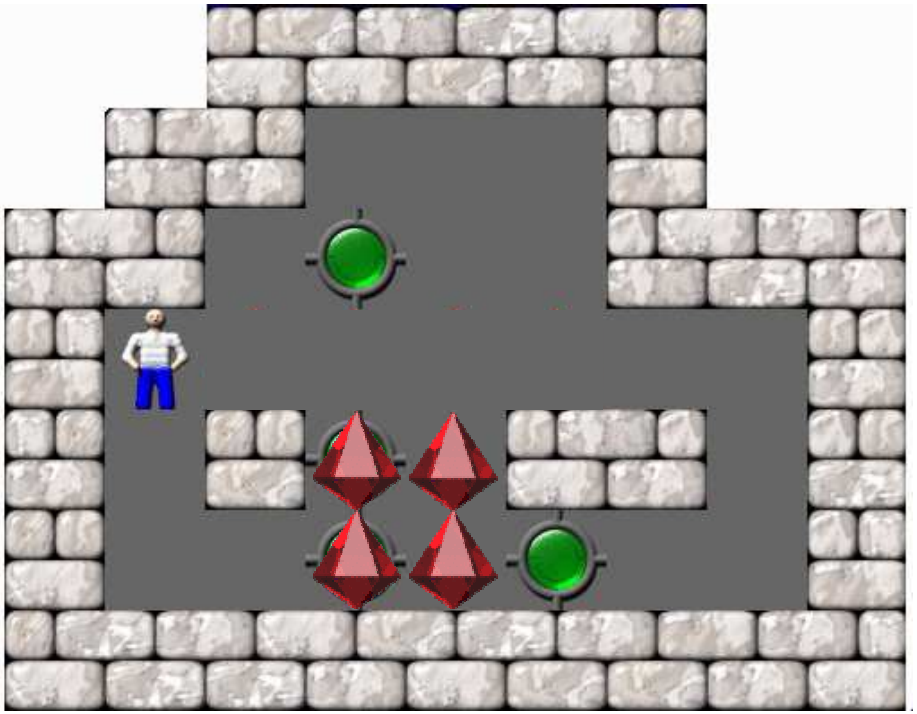
Move has been executed
New configuration with possible moves



← Possible move
← Chosen move

Example for the *backtracking* algorithm

Move has been executed
No further move is possible

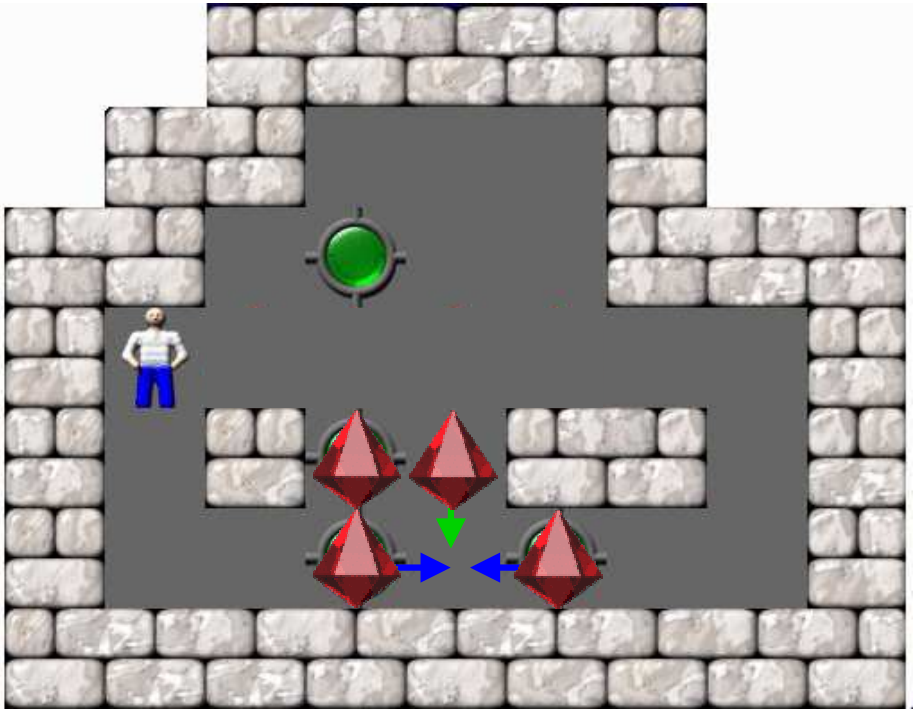


← Possible move

← Chosen move

Example for the *backtracking* algorithm

Backtrack
Back to previous configuration, next move



← Possible move
← Chosen move

5.4.1 Compile and Run

Available MPI implementations

- MPICH2 1.2.1 (in H-A 4111), OpenMPI 1.10.3 (HorUS)
- Portable implementations of the MPI-2 standard
- MPICH2 also provides the visuzlization tool `jumpshot`

Compiling MPI programs: `mpic++`

- `mpic++ -o myProg myProg.cpp`
- Not a separate compiler for MPI, but just a script that defines additional compiler options:
  - include und linker paths, MPI libraries, ...
  - option `-show` shows the invocations of the compiler

Running MPI programs: `mpiexec`

- `mpiexec -n 3 myProg arg1 arg2`
  - starts `myProg arg1 arg2` with 3 prozesses
  - `myProg` must be on the command search path or must be specified with (absolute or relative) path name
- On which nodes do the processes start?
  - depends on the implementation and the platform
  - in MPICH2 (with Hydra process manager): specification is possible via a configuration file:

    ```
    mpiexec -n 3 -machinefile machines myProg arg1 arg2
    ```

    - configuration file contains a list of node names, e.g.:

      ```
      bslab01      ← start one process on bslab03
      bslab05:2    ← start two processes on bslab05
      ```

Using MPICH2 in the lab H-A 4111

- If necessary, set the path (in `~/.bashrc`):
  - `export PATH=/opt/dist/mpich2-1.2.1/bin:$PATH`
- Remarks on `mpiexec`:
  - on the other nodes, the MPI programs start in the same directory, in which `mpiexec` was invoked
  - `mpiexec` uses `ssh` to start an auxiliary process on each node that is listed in the configuration fiile
    - even when less MPI processes should be started
  - avoid an entry `localhost` in the configuration file
    - it results in problems with `ssh`
  - sporadically, programs with relative path name (`./myProg`) will not be found; in these cases, specify `$PWD/myProg`

5.4.2 Debugging

- MPICH2 and OpenMPI support `gdb` and `totalview`
  - however, in MPICH2 only with the MPD process manager
- "Hack" in H-A 4111: start a `gdb` / `ddd` for each process
  - for `gdb`: `mpiexec -enable-x -n ...`
    `/opt/dist/mpidebug/mpigdb myProg args`
    - each `gdb` starts in its own console window
  - for `ddd`: `mpiexec -enable-x -n ...`
    `/opt/dist/mpidebug/mpiddd myProg args`
  - start the processes using `continue` (not `run ...`)
- Prerequisite: compilation with debugging information
  - `mpic++ -g -o myProg myProg.cpp`

5.4.3 Performance Analysis using Scalasca

- In principle, in the same way as for OpenMP
- Compiling the program:
  - `scalasca -instrument mpic++ -o myprog myprog.cpp`
- Running the programms:
  - `scalasca -analyze mpiexec -n 4 ...  ./myprog`
  - creates a directory `scorep_myprog_4_sum`
    - `4` indicates the number of processes
    - directory must not previously exist; delete it, if necessary
- Interactive analysis of the recorded data:
  - `scalasca -examine scorep_myprog_4_sum`

5.4.4 Performance Analysis and Visualization using Jumpshot

- MPICH supports the recording of event traces
  - Events: call/return of MPI routines, send, receive, ...
- Enable tracing via an option of `mpic++`
- `mpic++ -mpe=mpitrace -o myProg myProg.cpp`
  - prints the events to the console
- `mpic++ -mpe=mpilog -o myProg myProg.cpp`
  - stores the events in a trace file `myProg.clog2`
    - one shared trace file for all processes
  - analysis of the frace file using `jumpshot`:
    - format conversion: `clog2TOslog2 myProg.clog2`
    - visualization: `jumpshot myProg.slog2`

Example: ping-pong programm

- Modification in the program: process 0 performs a computation between sending and reveiving
  - just to make the example more interesting ...
- Commands:

```
mpic++ -mpe=mpilog -o pingpong pingpong.cpp
```

  - add the additional code for creating a trace file

```
mpiexec -n 2 -machinefile machines ./pingpong 5 100000
```

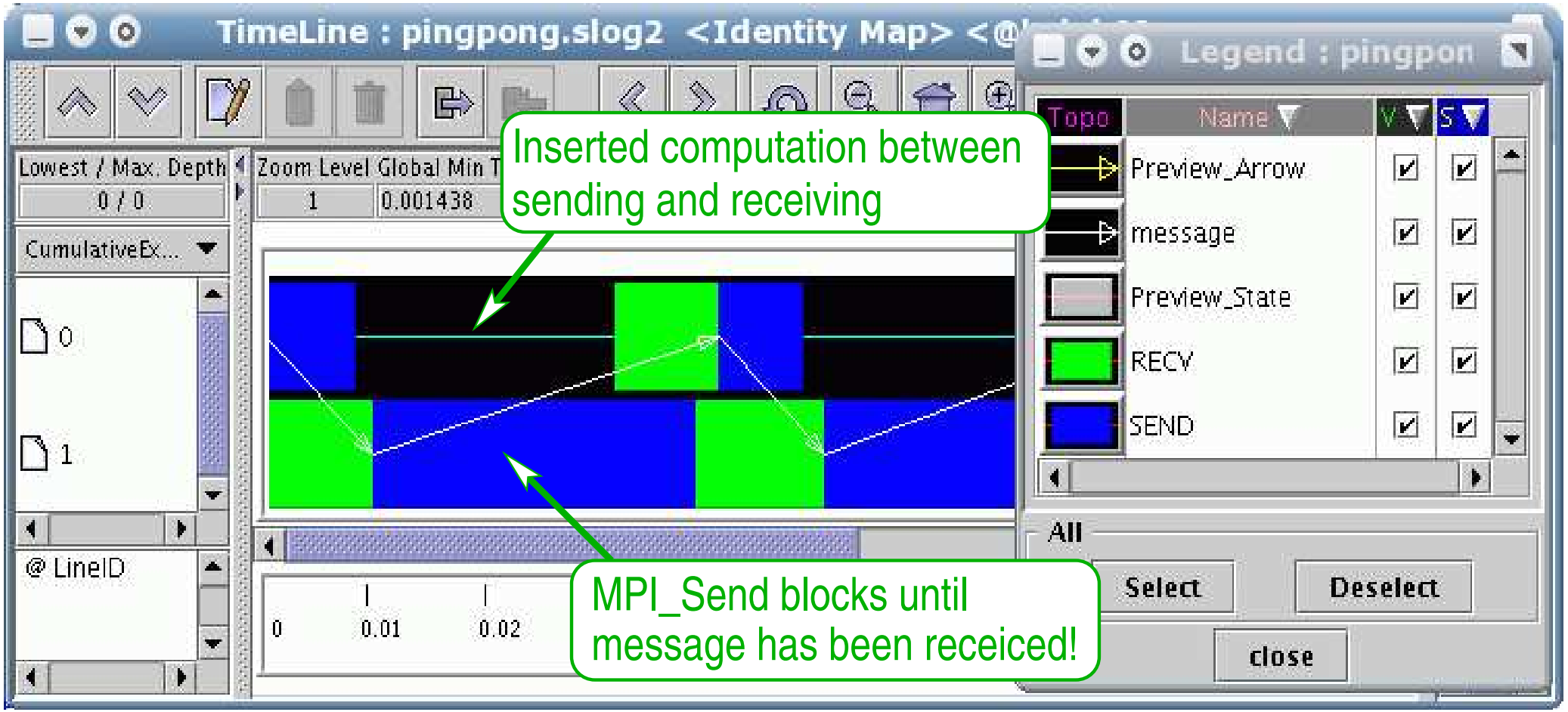
  - creates a trace file `pingpong.clog2`

```
clog2TOslog2 pingpong.clog2
```

  - converts `pingpong.clog2` to `pingpong.slog2`

```
jumpshot pingpong.slog2
```

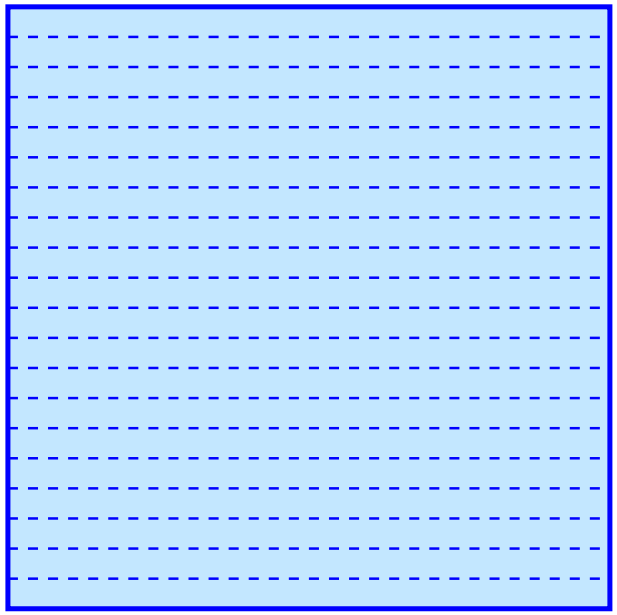
Example: ping-pong programm ...



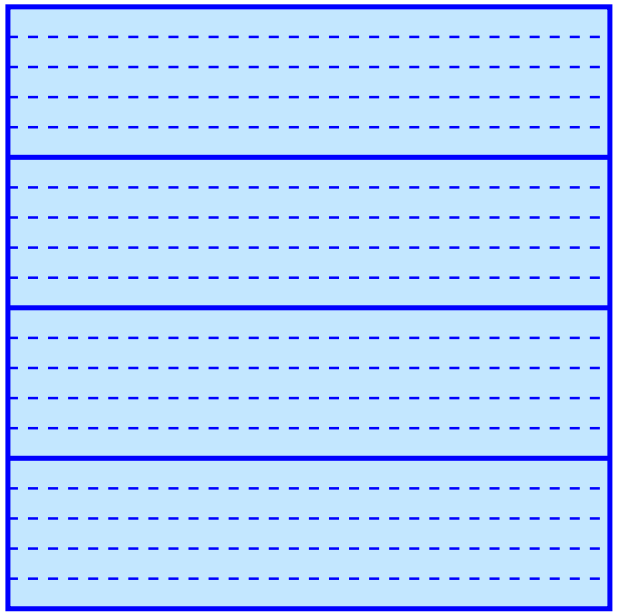
5.4.5 Using the HorUS cluster

- Start MPI programs using SLURM, e.g., with:
  - `salloc --exclusive --partition short --nodes=4 \`
    `--ntasks=16 mpiexec $HOME/GAUSS/heat 500`
  - important options of `salloc`:
    - `--nodes=4`: number of nodes to be allocated
    - `--ntasks=16`: total number of processes to be started
- For performance analysis: Scalasca
  - `salloc ... scalasca -analyze mpiexec $HOME/GAUSS...`
- For debugging: Totalview
  - `salloc ... mpiexec -debug $HOME/GAUSS/heat 500`

Gerneral approach



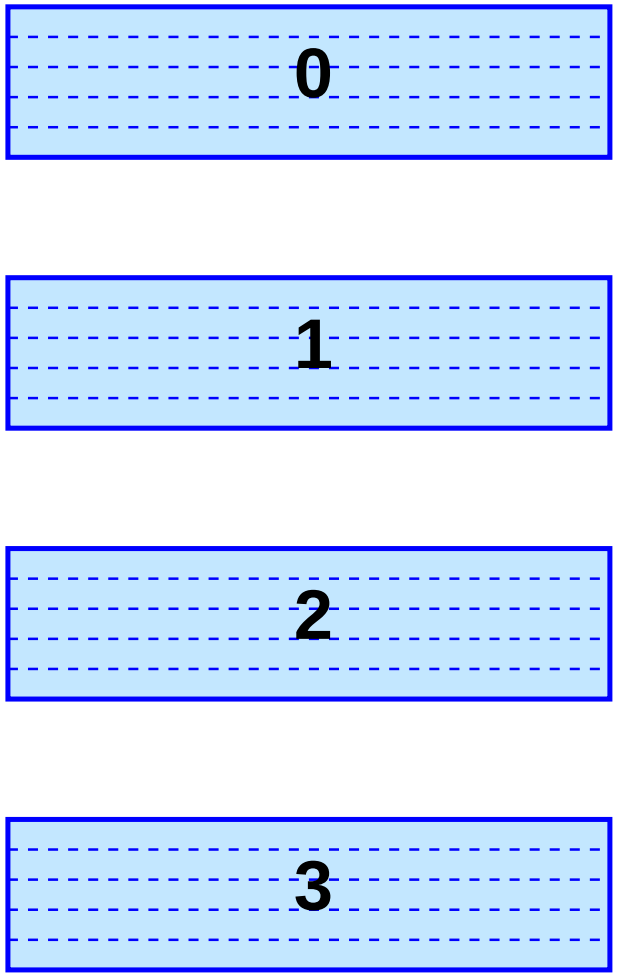
0. Matrix with temperature values



Gerneral approach

0. Matrix with temperature values
1. Distribute the matrix into stripes

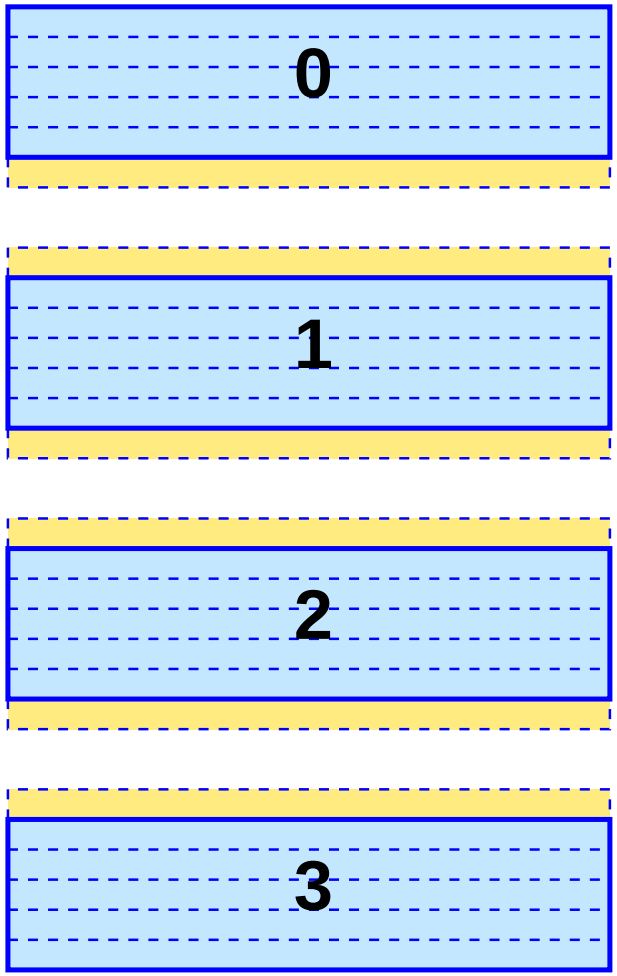
Gerneral approach



0. Matrix with temperature values
1. Distribute the matrix into stripes

   Each process only stores a part of the matrix

Gerneral approach

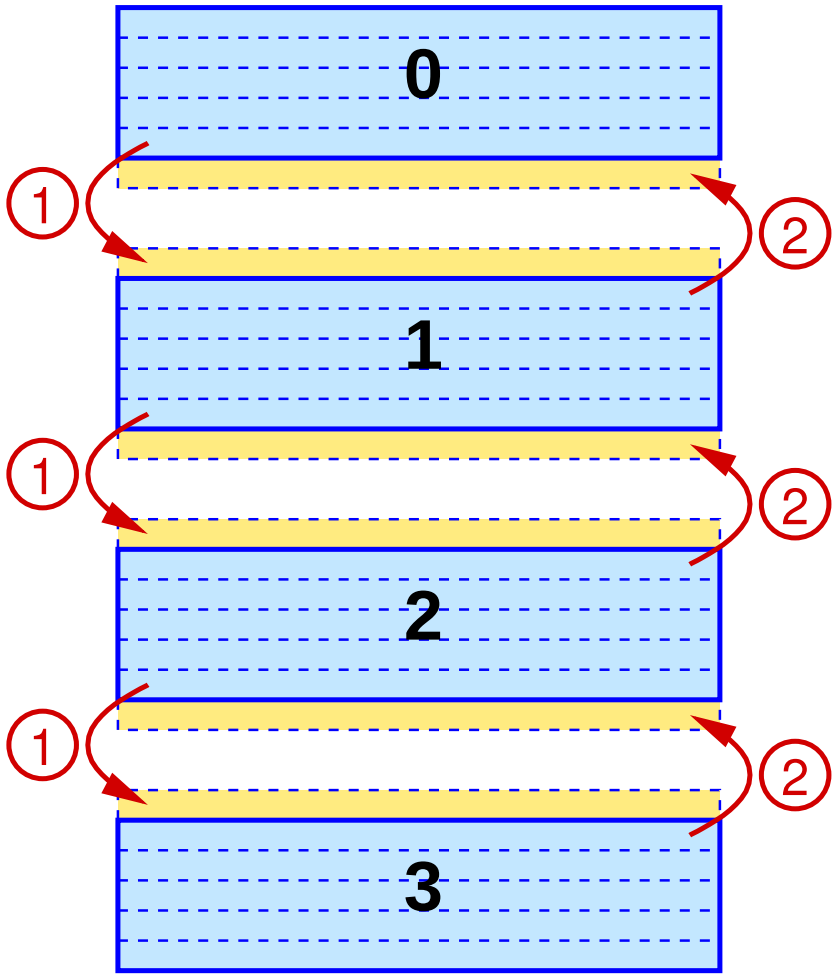


0. Matrix with temperature values
1. Distribute the matrix into stripes

   Each process only stores a part of the matrix
2. Introduce ghost zones

   Each process stores an additional row at the cutting edges

Gerneral approach



0. Matrix with temperature values
1. Distribute the matrix into stripes

   Each process only stores a part of the matrix
2. Introduce ghost zones

   Each process stores an additional row at the cutting edges
3. After each iteration the ghost zones are exchanged with the neighbor processes

   E.g., first downwards (1), then upwards (2)

Gerneral approach ...

```
int nprocs, myrank;
double a[LINES][COLS];
MPI_Status status;

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

/* Step 1: Send downwards, recieve from above */
if (myrank != nprocs-1)
   MPI_Send(a[LINES-2], COLS, MPI_DOUBLE, myrank+1, 0,
            MPI_COMM_WORLD);
if (myrank != 0)
   MPI_Recv(a[0], COLS, MPI_DOUBLE, myrank-1, 0,
            MPI_COMM_WORLD, &status);
```

Distribution of data

- Close formula for the uniform distribution of an array of length n to np processes:
  - $\mathbf{start}(p) = n \div np \cdot p + \max(p - (np - n \bmod np), 0)$
  - $\mathbf{size}(p) = (n + p) \div np$
  - process p receives $\mathbf{size}(p)$ elements starting at index $\mathbf{start}(p)$
- This results in the following index transformation:
  - $\mathbf{tolocal}(i) = (p, i - \mathbf{start}(p))$
    mit $p \in [0, np - 1]$ so, daß $0 \leq i - \mathbf{start}(p) < \mathbf{size}(p)$
  - $\mathbf{toglobal}(p, i) = i + \mathbf{start}(p)$
- In addition, you have to consider the ghost zones for Jacobi and Gauss/Seidel!

Distribution of computation

- In general, using the *owner computes* rule
  - the process that writes a data element also performs the corresponding calculations
- Two approaches for technically realizing this:
  - index transformation and conditional execution
    - e.g., when printing the verification values of the matrix:

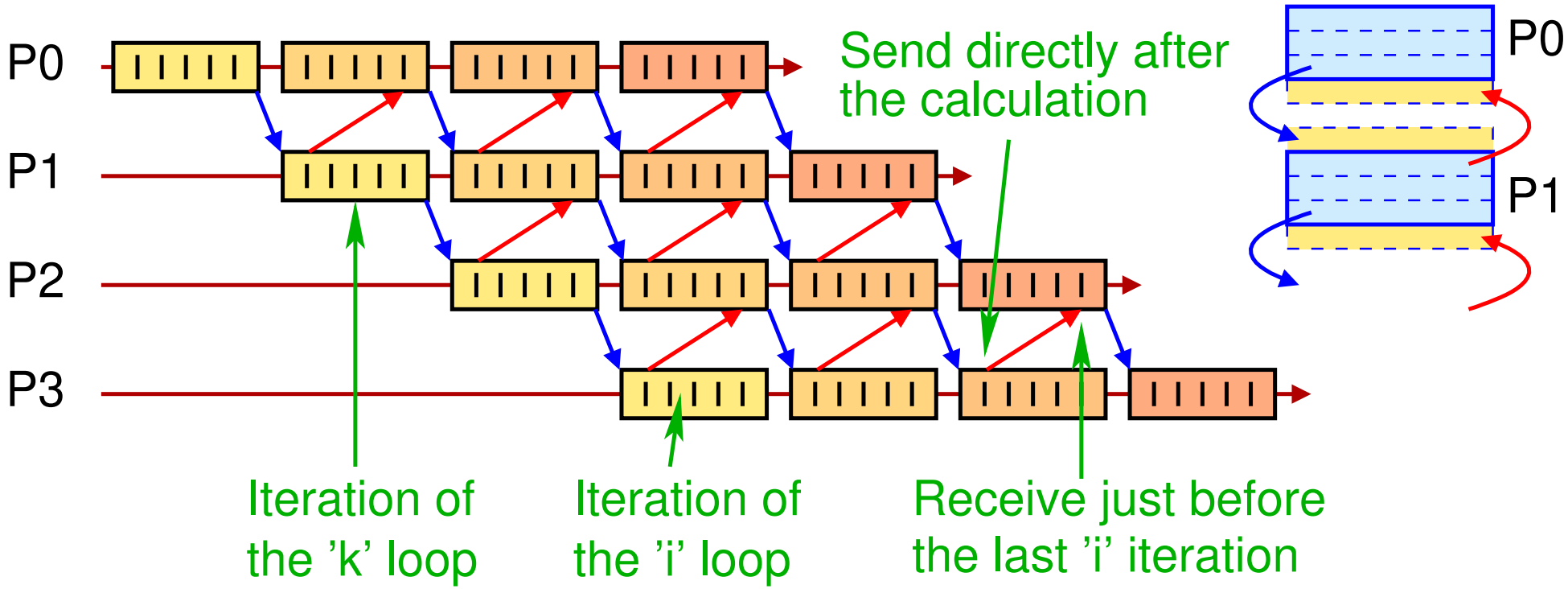
```
if ((x-start >= 0) && (x-start < size))
    cout << "a[" << x << "]=" << a[x-start] << "\n";
```

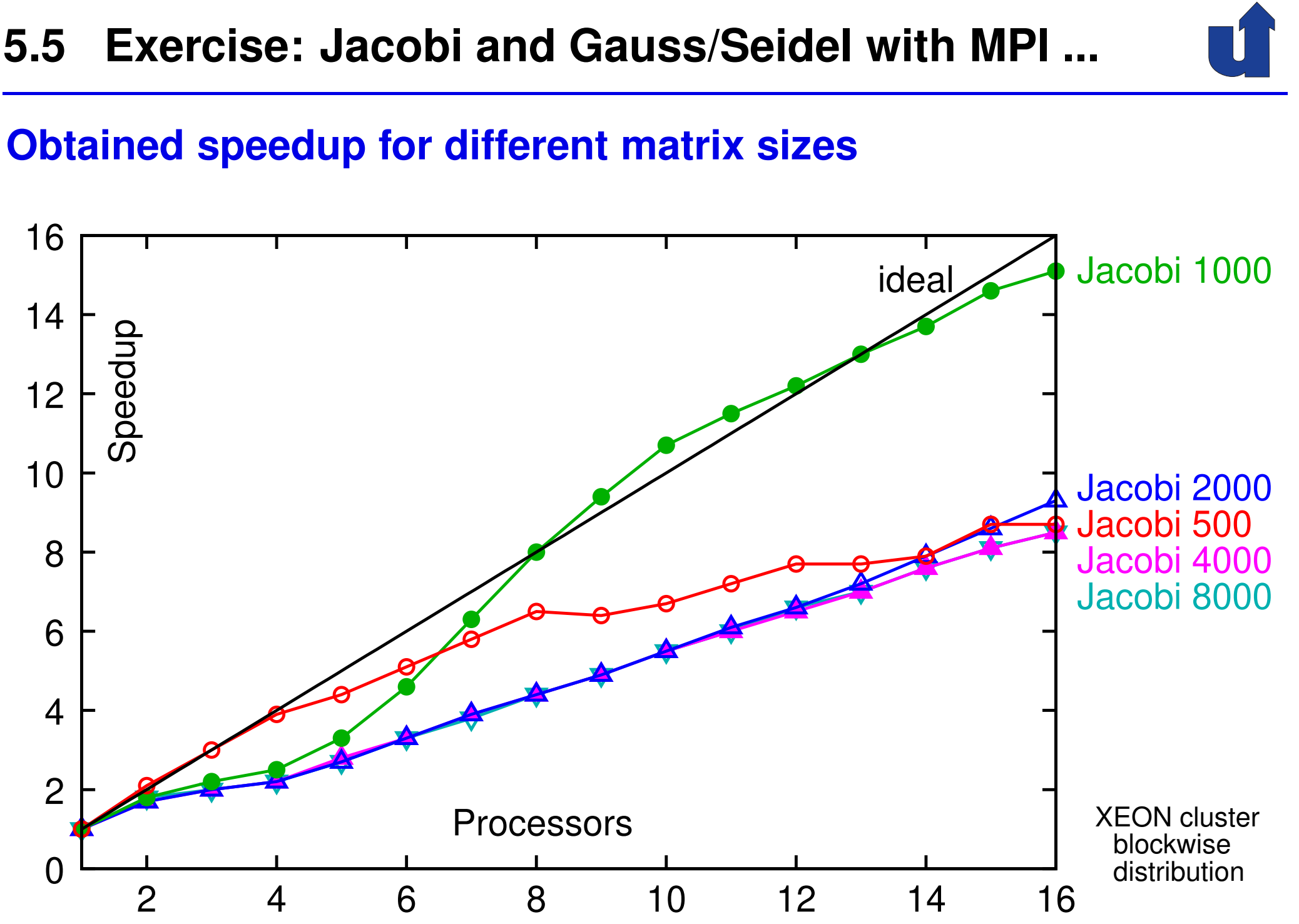
  - adjustment of the enclosing loops
    - e.g., during the iteration or when initializing the matrix:

```
for (i=0; i<size; i++)
    a[i] = 0;
```

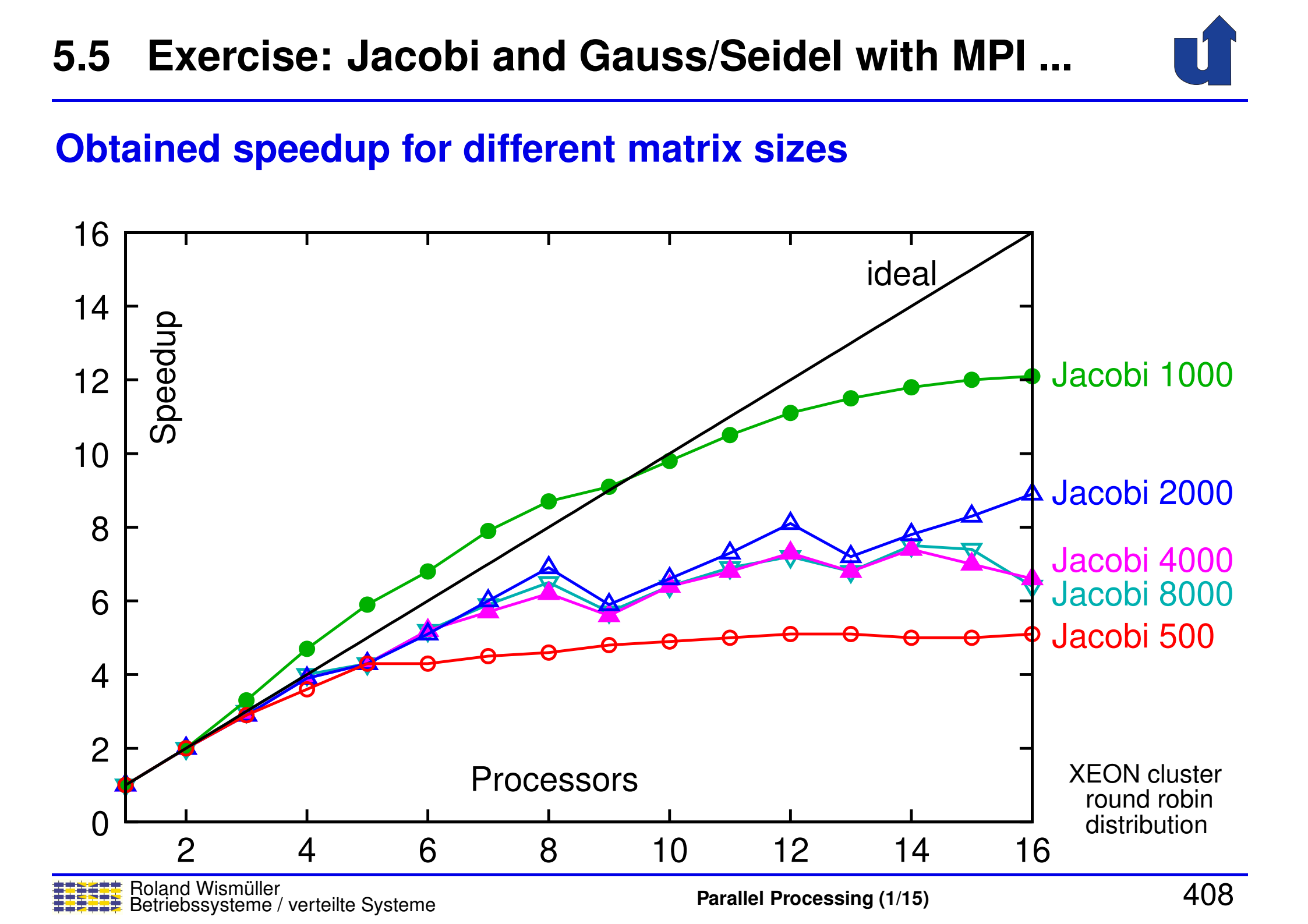
On the parallelization of the Gauss/Seidel method

- Similar to the pipelined parallelization with OpenMP (☞ **2.5**)



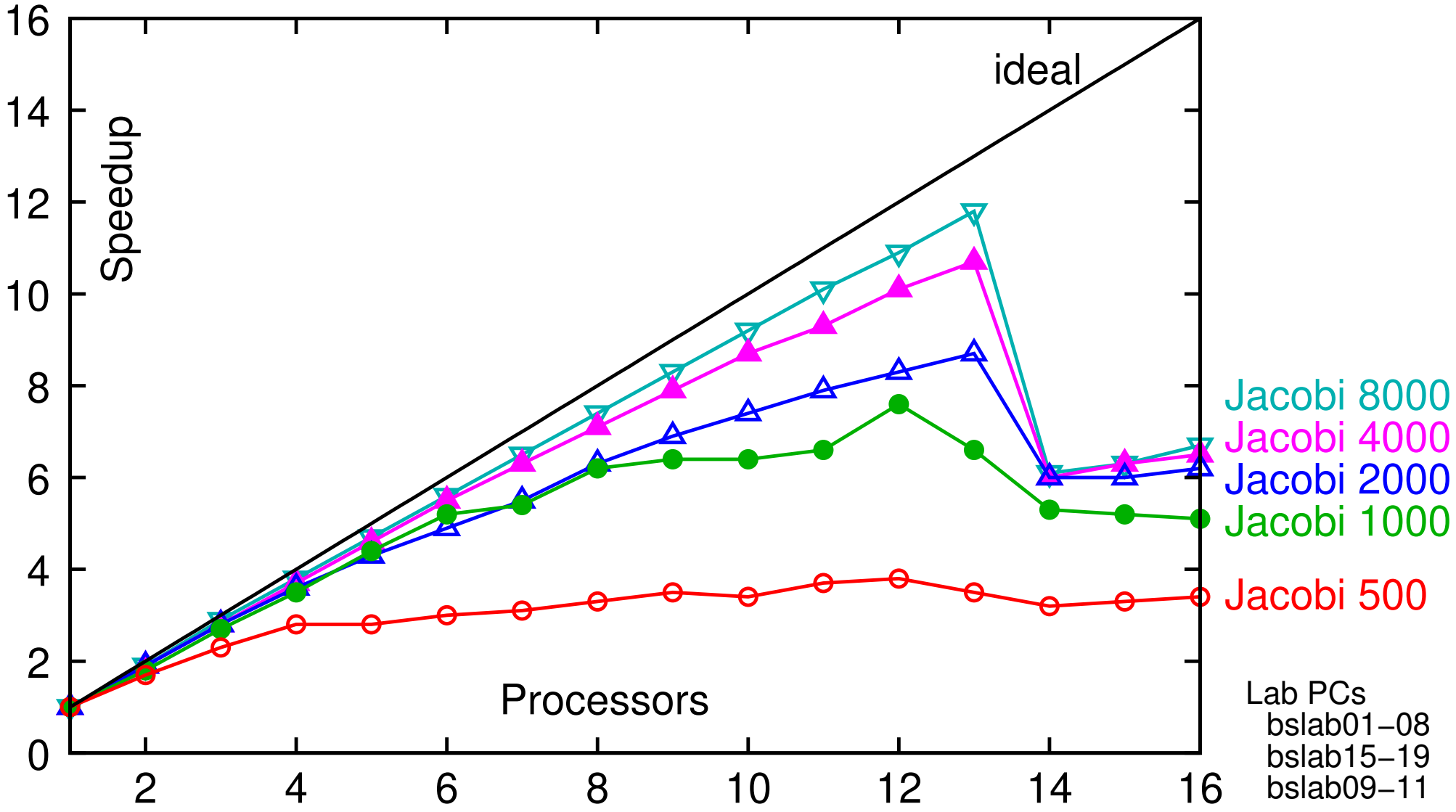


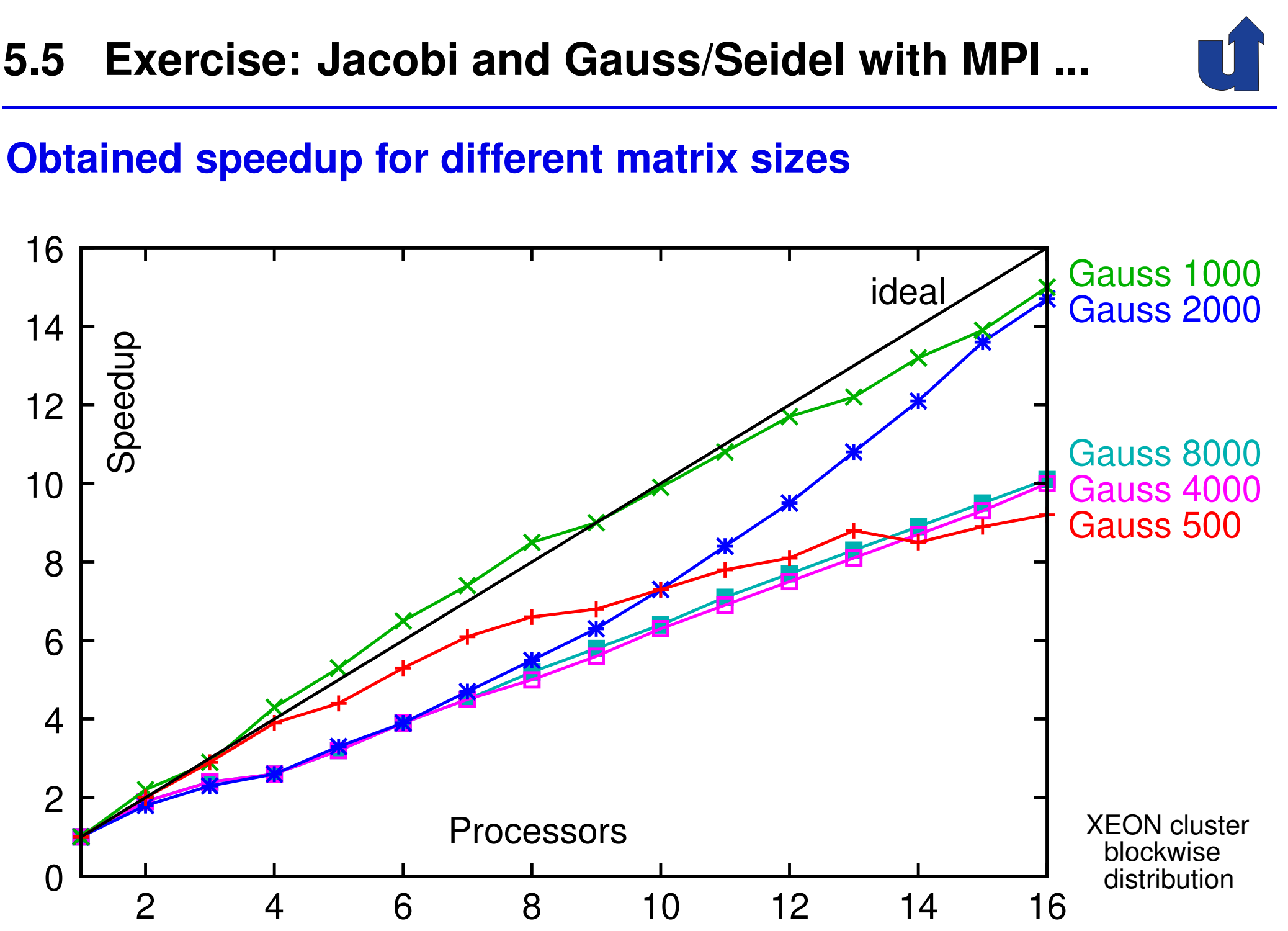
Obtained speedup for different matrix sizes



Obtained speedup for different matrix sizes

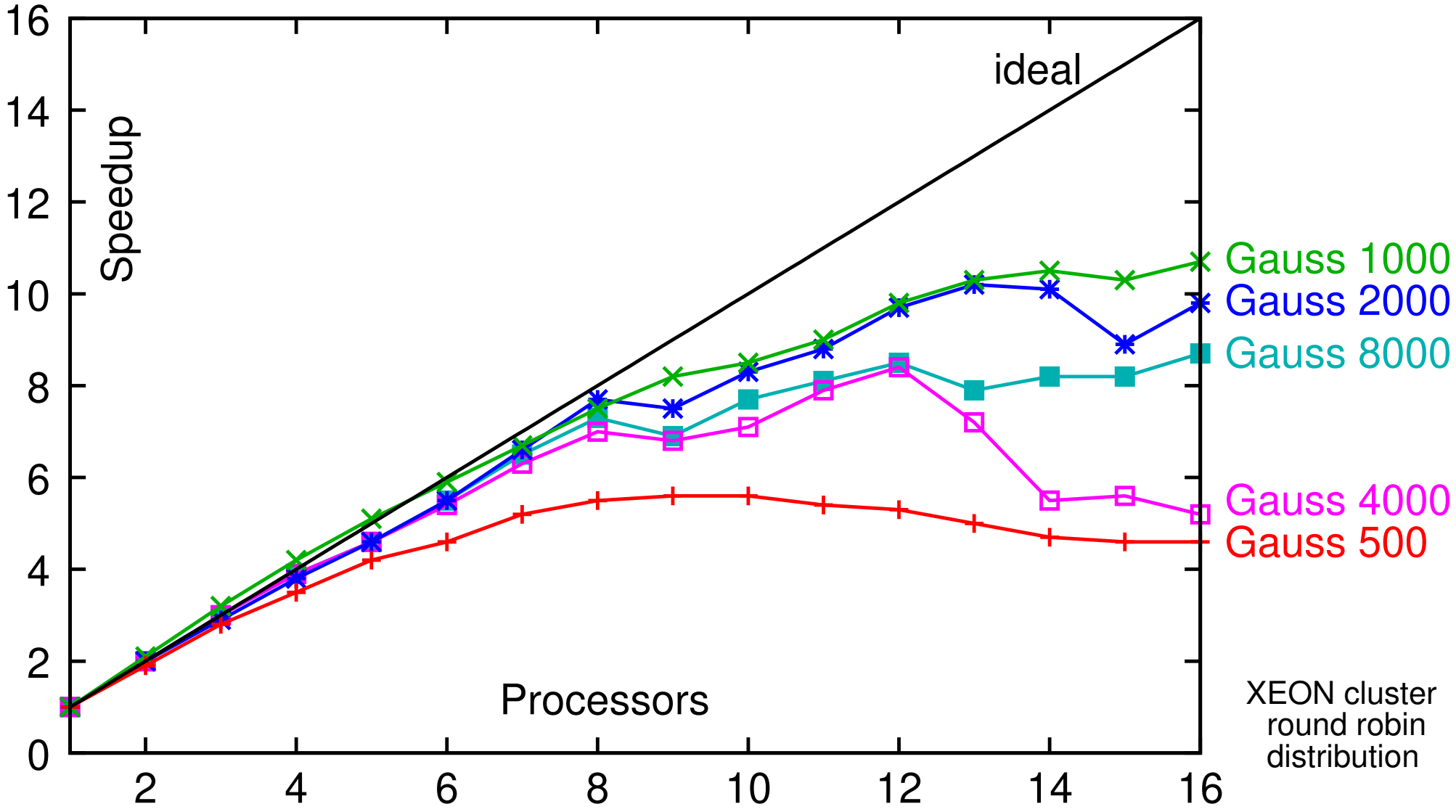
Obtained speedup for different matrix sizes

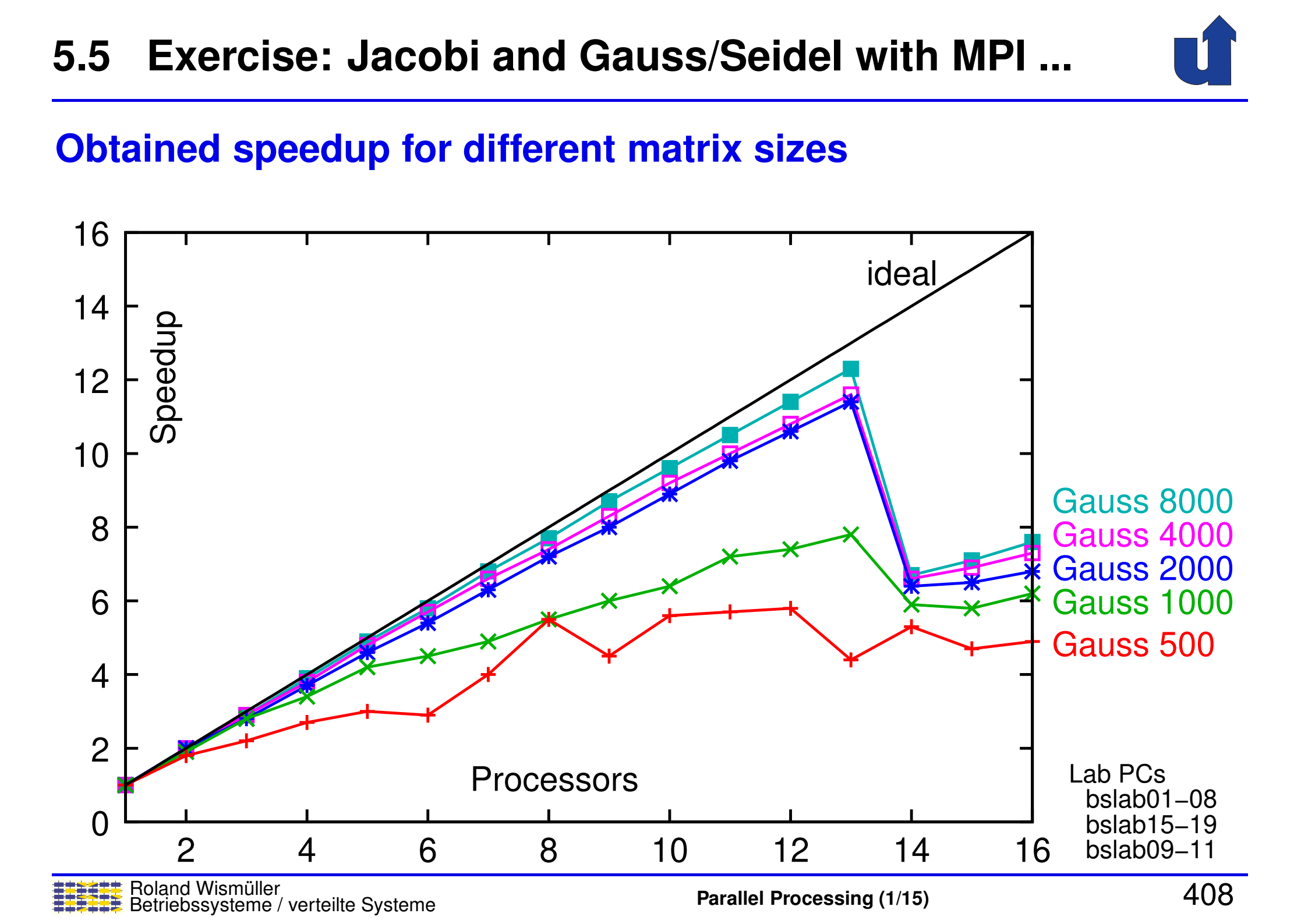




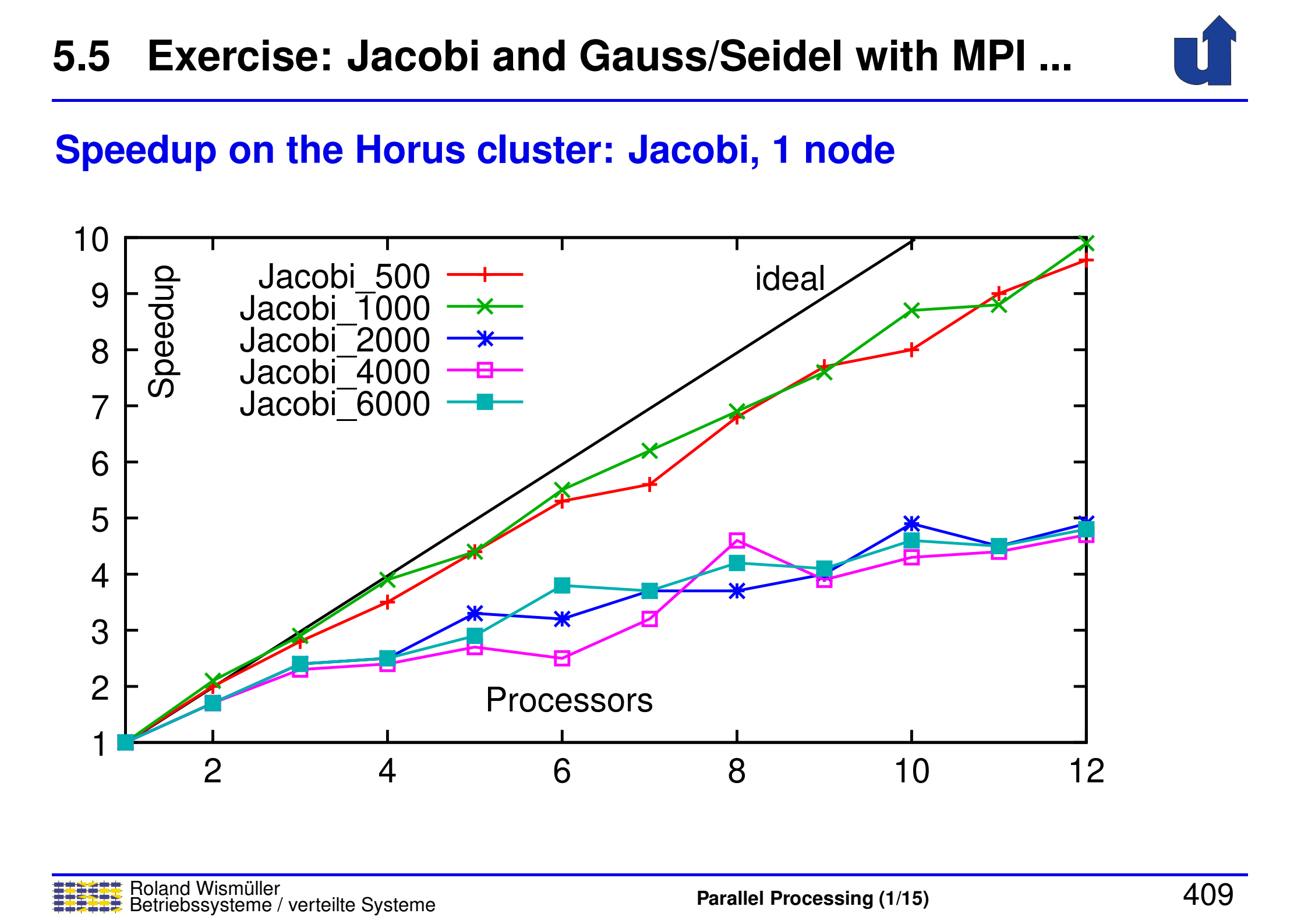
Obtained speedup for different matrix sizes

Obtained speedup for different matrix sizes



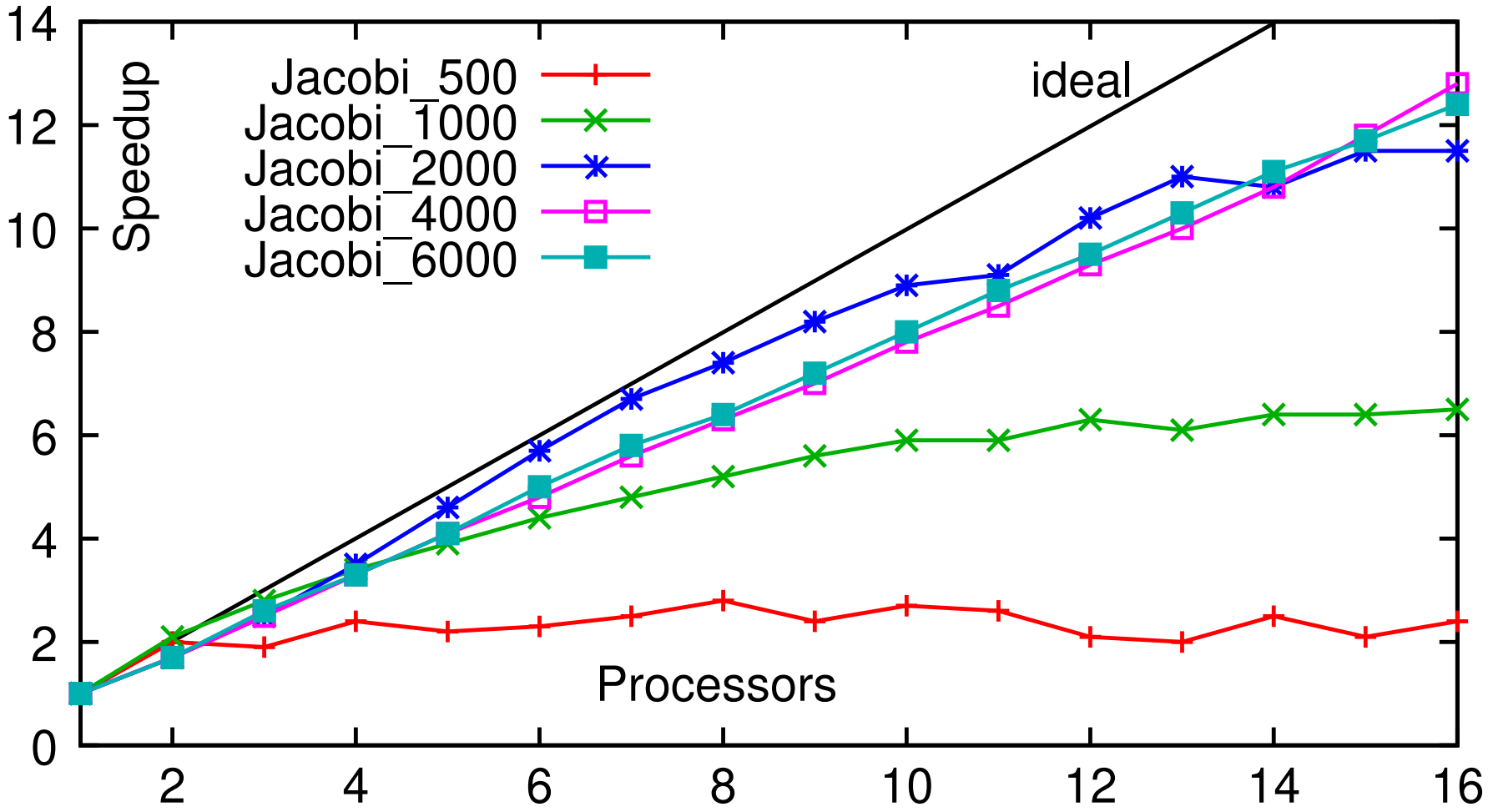


Obtained speedup for different matrix sizes

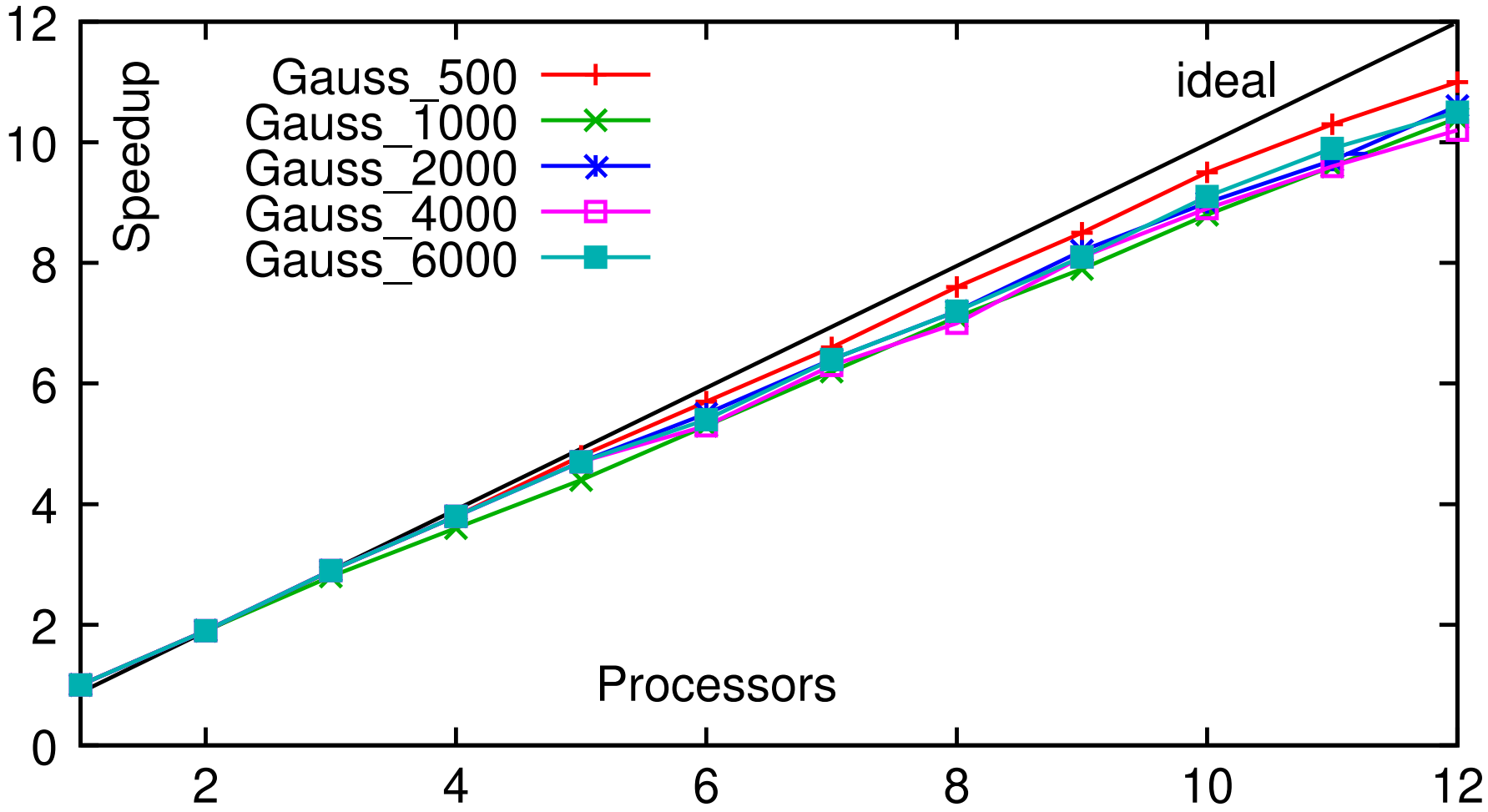


Speedup on the Horus cluster: Jacobi, 1 node

Speedup on the Horus cluster: Jacobi, 2 processes/node



Speedup on the Horus cluster: Gauss/Seidel, 1 node



Speedup on the Horus cluster: Gauss/Seidel, 2 processes/node