



Parallel Processing

Winter Term 2025/26

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

Stand: January 8, 2026

Contents

0 Organisation	2
1 Repetition / Foundations	22
1.1 C/C++ for Java Programmers	24
1.1.1 Fundamentals of C++	24
1.1.2 Data types in C++	29
1.1.3 Pointers	34
1.1.4 Strings and Output	42
1.1.5 Further specifics of C++	43
1.1.6 C/C++ Libraries	46

1.1.7 The C Preprocessor	47
1.2 Threads and Synchronization	49
1.3 C++ Threads	57
2 Basics of Parallel Processing	69
2.1 Motivation	71
2.2 Parallelism	75
2.3 Parallelisation and Data Dependences	84
2.4 Parallel Computer Architectures	87
2.4.1 MIMD: Message Passing Systems	91
2.4.2 MIMD: Shared Memory Systems	95
2.4.3 SIMD	110
2.4.4 High Performance Supercomputers	116
2.5 Parallel Programming Models	119
2.5.1 Shared Memory	120
2.5.2 Message Passing	122
2.5.3 Distributed Objects	124
2.5.4 Data Parallel Languages	125
2.6 Focus of this Lecture	127
2.7 Organisation Forms for Parallel Programs	128
2.7.1 Embarrassingly Parallel	128
2.7.2 Manager/Worker Model (Master/Slave Model)	131
2.7.3 Work Pool Model (Task Pool Model)	134
2.7.4 Divide and Conquer	135

2.7.5	Data parallel Model: SPMD	139
2.7.6	Fork/Join Model	141
2.7.7	Task-Graph Model	142
2.7.8	Pipeline Model	143
2.8	A Design Process for Parallel Programs	144
2.8.1	Partitioning	146
2.8.2	Communication	153
2.8.3	Agglomeration	158
2.8.4	<i>Mapping</i>	159
2.9	Performance Considerations	162
2.9.1	Performance Metrics	163
2.9.2	Reasons for Performance Loss	172

1-4

2.9.3	Load Balancing	174
2.9.4	Performance Analysis of Parallel Software	180
2.9.5	Analytical Performance Modelling	181
2.9.6	Performance Analysis Tools	190
3	Parallel Programming with Shared Memory	196
3.1	OpenMP Basics	199
3.1.1	The <code>parallel</code> directive	204
3.1.2	Library routines	211
3.2	Loop parallelization	213
3.2.1	The <code>for</code> directive: parallel loops	214
3.2.2	Parallelization of Loops	220

1-5

3.2.3	Simple Examples	223
3.2.4	Dependence Analysis in Loops	227
3.3	OpenMP Synchronization	233
3.3.1	Critical sections	235
3.3.2	Atomic operations	236
3.3.3	Reduction operations	238
3.3.4	Execution in program order	240
3.3.5	Barrier	243
3.3.6	Execution using a single thread	248
3.4	Exercise: The Jacobi and Gauss/Seidel Methods	249
3.5	Task Parallelism with OpenMP	266
3.5.1	The <code>sections</code> Directive: Parallel Code Regions	266

1-6

3.5.2	The <code>task</code> Directive: Explicit Tasks	268
3.6	Advanced OpenMP Features	275
3.6.1	Thread Affinity	275
3.6.2	SIMD Vectorization	280
3.6.3	Using External Accelerators	282
3.7	Exercise: A Solver for the Sokoban Game	292
3.8	Excursion: <i>Lock-Free</i> Data Structures	298

4	Parallel Programming with Message Passing	302
4.1	Typical approach	304
4.2	MPI (<i>Message Passing Interface</i>)	308
4.3	MPI Core routines	310

1-7

4.4	Simple MPI programs	321
4.5	Point-to-point communication	329
4.6	Communicators	333
4.7	Collective operations	337
4.8	Exercise: Jacobi and Gauss/Seidel with MPI	349
4.9	Complex data types in messages	356
4.10	Further concepts	362
4.11	Summary	364
5	Optimization Techniques	365
5.1	Cache Optimization	367
5.2	Optimization of Communication	383
5.3	Summary	386
6	Summary / Important Topics	387

Parallel Processing

Winter Term 2025/26

0 Organisation

About Myself



- ➔ Studies in Computer Science, Techn. Univ. Munich
 - ➔ Ph.D. in 1994, state doctorate in 2001
- ➔ Since 2004 Prof. for Operating Systems and Distributed Systems
- ➔ **Research:** Secure component based systems; Using generative AI for teaching; Parallel and distributed systems
- ➔ Head of Examination Board Computer Science
- ➔ **E-mail:** roland.wismueller@uni-siegen.de
- ➔ **Tel.:** 0271/740-4050
- ➔ **Room:** H-B 8404
- ➔ **Office Hour:** Mo., 14:15-15:15



Andreas Hoffmann

andreas.hoffmann@uni-...
0271/740-4047
H-B 8405

- ➔ E-assessment and e-labs
- ➔ IT security
- ➔ Web technologies
- ➔ Mobile applications



Felix Breitweiser

felix.breitweiser@uni-...
0271/740-4719
H-B 8406

- ➔ Operating systems
- ➔ Programming languages
- ➔ Virtual machines



Sven Jacobs

sven.jacobs@uni-...
0271/740-2533
H-B 8407

- ➔ E-assessment and e-labs
- ➔ Generative artificial intelligence
- ➔ Web technologies

Teaching



Lectures/Labs

- ➔ Rechnernetze I, 6 CP (Bachelor, summer term)
- ➔ Rechnernetze Praktikum, 6 CP (Bachelor, winter term)
- ➔ Rechnernetze II, 6 CP (Master, summer term)
- ➔ Betriebssysteme und nebenläufige Programmierung, 6 CP (Bachelor, summer term)
- ➔ Parallel processing, 6 CP (Master, winter term)
- ➔ Distributed systems, 6 CP (Bachelor, winter term)

Project Groups

- ➔ e.g., secure cooperation of software components
- ➔ e.g., concepts for secure management of Linux-based thin clients

Theses (Bachelor, Master)

- ➔ Topic areas: secure virtual machine, parallel computing, pattern recognition in sensor data, e-assessment, ...

Seminars

- ➔ Topic areas: IT security, programming languages, pattern recognition in sensor data, ...
- ➔ Procedure: block seminar (30 min. talk, 5000 words paper)
- ➔ Master: attend the lecture “Scientific Working” beforehand!
 - ➔ block course end of Feb. / beginning of March

Notes for slide 6:

A note on external Master theses: The right to give you a topic for a Master thesis lies with the University only!

This means, if you want to do a thesis at an external company or research institute, you **first** have to find a professor who will supervise you, and then, if she or he is interested, the professor may define a topic together with the company.

Please have a look at our [handout on conducting external theses!](#)^a

^ahttps://www.eti.uni-siegen.de/dekanat/pruefungsamt/dokumente/studien-gaenguebergreifend/externe-abschlussarbeiten-eti_en.pdf



Lecture

- ➔ Mon., 12:15-13:45, H-C 8326
- ➔ on Tue., 14.10., 21.10., 28.10. and 04.11. also in the lab slot: 10:15-11:45, H-C 8326

Practical labs

- ➔ Tue., 10:15-11:45, PC lab room H-C 4111 (and at home)
 - ➔ some exercises rely on software / hardware in the lab room
- ➔ Tutor: Felix Breitweiser (felix.breitweiser@uni-siegen.de)
 - ➔ for questions, help, and discussion of solutions



Information, slides, and announcements

- ➔ See the WWW page for this course
- ➔ <http://www.bs.informatik.uni-siegen.de/lehre/pv/>
- ➔ Annotated slides (PDF) available; maybe slightly modified

Moodle course

- ➔ <https://moodle.uni-siegen.de/course/view.php?id=23366>
- ➔ For students arriving late:
 - ➔ screen casts of the lectures until mid November
- ➔ Screen casts with tutorials on exercises / tools
- ➔ Submission of lab assignments



Learning targets

- ➔ Knowing the basics, techniques, methods, and tools of parallel programming
- ➔ Basic knowledge about parallel computer architectures
- ➔ **Practical knowledge/experience with parallel programming**
- ➔ Knowing and being able to use the most important programming models
- ➔ Knowing about the possibilities, difficulties and limits of parallel processing
- ➔ Being able to identify and select promising strategies for parallelization

- ➔ Focus: high performance computing

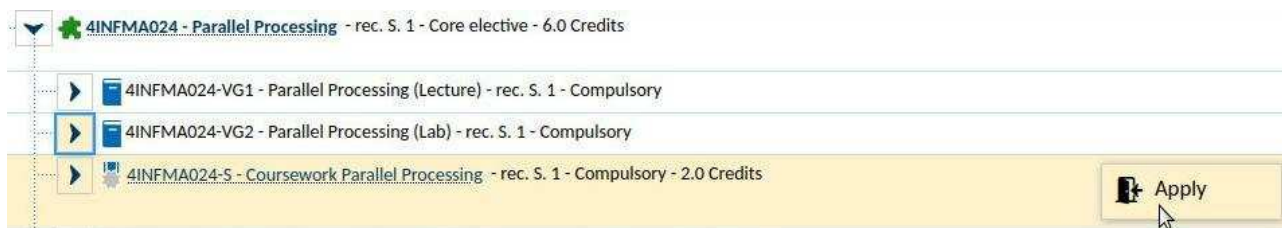


Methodology

- ➔ Lecture: Basics
 - ➔ theoretical knowledge about parallel processing
 - ➔ practical introduction to programming environments
 - ➔ “hands-on” tutorials
- ➔ Lab: practical use
 - ➔ **independent programming work (lab + homework!)**
 - ➔ programming language: C++
 - ➔ practical skills and experiences
 - ➔ in addition: raising questions
 - ➔ main task: parallelization of two representative problems
 - ➔ iterative, numerical method (Jacobi, Gauss/Seidel)
 - ➔ combinatoral search (Sokoban)



- ➔ Passing the course requires successful completion of the lab:
 - ➔ i.e., qualified attempt for all mandatory exercises
- ➔ You must register for '4INFMA024-S Coursework Parallel Processing' in unisono **before you can submit a solution!** (do it right now!)
 - ➔ independent of the registration to the course and the lab!
 - ➔ if you cannot complete the course: **deregister** again!



Notes for slide 11:

If you are not registered for the course achievement, you will not be able to submit any solutions in the Moodle platform (the corresponding section will not be available in Moodle).

Since data is transferred between unisono and Moodle only about once a week, you should register way in advance!



- ➔ Eight exercise sheets
 - ➔ each with 1-2 mandatory exercises of different weight (1-3)
 - ➔ **it'll be hard work!**
 - ➔ 6 LP = 180 hours, that is 60 h presence (lecture + lab), **60 h homework**, 60 h preparation for exam
 - ➔ solve exercises in the lab (H-A 4111) and at home!
 - ➔ solutions must be submitted via **Moodle** in due time
 - ➔ requirement for passing: at least **serious** attempt
 - ➔ codes must at least compile and execute in the lab
 - ➔ sometimes, additional answers / materials are required
 - ➔ **do not copy!** (both will get 0 points)
- ➔ Grading: yes/no
- ➔ You need at least 12 out of 16 possible points to pass

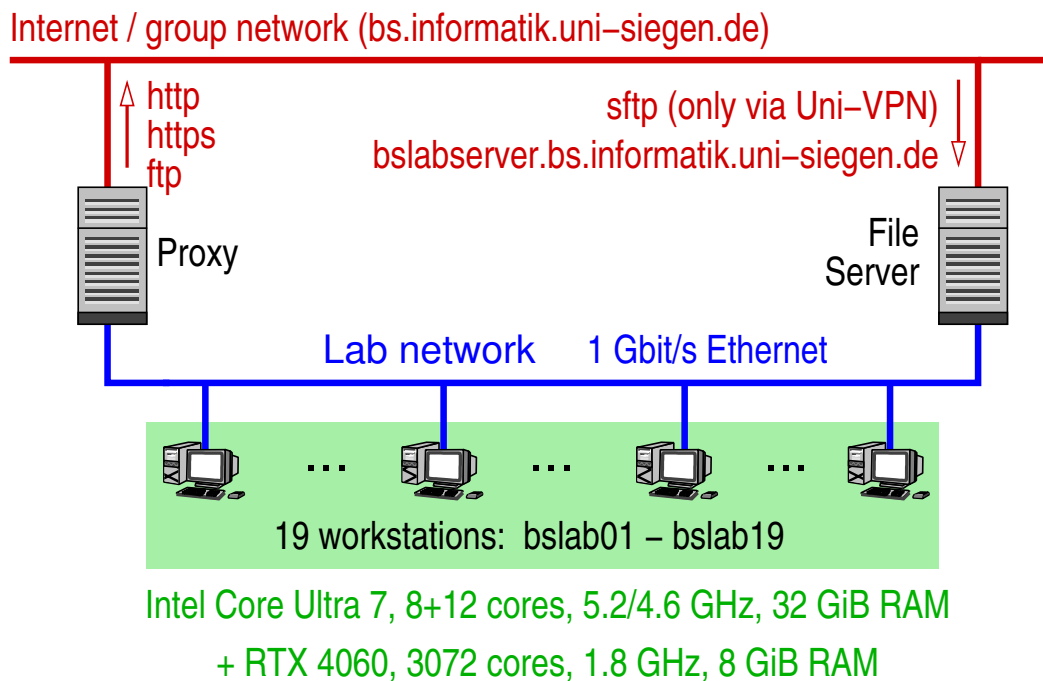
Organisational Issues Regarding the Labs



- ➔ Assignments should be done in the lab (H-A 4111), but can / should also be prepared at home
- ➔ Ideally, you have a Linux-PC with the GNU-compiler (g++)
 - ➔ Windows with MSVC will not work for all exercise sheets
- ➔ In addition, you should have MPI installed, preferable MPICH
 - ➔ see <https://www.mpich.org/downloads>
- ➔ User regulations and key card application form:
 - ➔ <http://www.bs.informatik.uni-siegen.de/lehre/pv>
 - ➔ please let me sign the key card application form and then deliver it directly to Mr. Kiel (AR-P 209)
 - ➔ by using the lab computers you accept the user regulations (see web page)
- ➔ **First lab hour:** Tue. 11.11., H-A 4111



- ➔ Linux-PCs, private IP network, but sftp access via VPN



Examination



- ➔ Written examination (60 minutes)
 - ➔ open book exam
 - ➔ electronic exam, computers provided by university
 - ➔ subject matter: lecture and labs!
 - ➔ examination also covers the practical exercises
- ➔ Application via unisono
 - ➔ **at least two weeks before the exam date (hard deadline!)**
 - ➔ exam date is published via unisono and course web page



- ➔ Repetition / Foundations
 - ➔ C/C++ for Java programmers
 - ➔ Threads and synchronisation
 - ➔ C++ threads
- ➔ Basics of Parallel Processing
 - ➔ Motivation, Parallelism
 - ➔ Parallelization and Data Dependences
 - ➔ Parallel Computers
 - ➔ Programming Models
 - ➔ Organisation Forms for Parallel Programs
 - ➔ Design Process
 - ➔ Performance Considerations



- ➔ Parallel Programming with Shared Memory
 - ➔ Basics
 - ➔ OpenMP
- ➔ Parallel Programming with Message Passing
 - ➔ Approach
 - ➔ MPI
- ➔ Optimization Techniques
 - ➔ Cache Optimization
 - ➔ Optimization of Communication

Time Table of Lecture and Labs



- ➔ Until November, 4th: only lectures (Mon. + Tue.)
 - ➔ no lab (but home work)
- ➔ Then: lectures (Mon.) and lab (Tue. + home work)
- ➔ Last three weeks: only lab (Tue. + home work)

- ➔ Prospective due dates for the assignments:
 - ➔ 11.11.: Exercise sheet 1
 - ➔ ... (see [web page](#))
 - ➔ 03.02.: Exercise sheet 8

General Literature



- ➔ Currently no recommendation for a all-embracing text book

- ➔ Barry Wilkinson, Michael Allen: *Parallel Programming*. internat. ed, 2. ed., Pearson Education international, 2005.
 - ➔ covers most parts of the lecture, many examples
 - ➔ short references for MPI, PThreads, OpenMP

- ➔ A. Grama, A. Gupta, G. Karypis, V. Kumar: *Introduction to Parallel Computing*, 2nd Edition, Pearson, 2003.
 - ➔ much about design, communication, parallel algorithms

- ➔ Thomas Rauber, Gudula Rünger: *Parallele Programmierung*. 2. Auflage, Springer, 2007.
 - ➔ architecture, programming, run-time analysis, algorithms



- ➔ Theo Ungerer: *Parallelrechner und parallele Programmierung*, Spektrum, Akad. Verl., 1997.
 - ➔ much about parallel hardware and operating systems
 - ➔ also basics of programming (MPI) and compiler techniques
- ➔ Ian Foster: *Designing and Building Parallel Programs*, Addison-Wesley, 1995.
 - ➔ design of parallel programs, case studies, MPI
- ➔ Seyed Roosta: *Parallel Processing and Parallel Algorithms*, Springer, 2000.
 - ➔ mostly algorithms (design, examples)
 - ➔ also many other approaches to parallel programming

Literature for Special Topics



- ➔ S. Hoffmann, R. Lienhart: *OpenMP*, Springer, 2008.
 - ➔ handy pocketbook on OpenMP
- ➔ W. Gropp, E. Lusk, A. Skjellum: *Using MPI*, MIT Press, 1994.
 - ➔ the definitive book on MPI
- ➔ D.E. Culler, J.P. Singh: *Parallel Computer Architecture - A Hardware / Software Approach*. Morgan Kaufmann, 1999.
 - ➔ UMA/NUMA systems, cache coherency, memory consistency
- ➔ Michael Wolfe: *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.
 - ➔ details on parallelizing compilers

Parallel Processing

Winter Term 2025/26

1 Repetition / Foundations

1 Repetition / Foundations ...



Contents

- ➔ C/C++ for Java programmers
- ➔ Threads and synchronisation
- ➔ C++ threads

1.1 C/C++ for Java Programmers



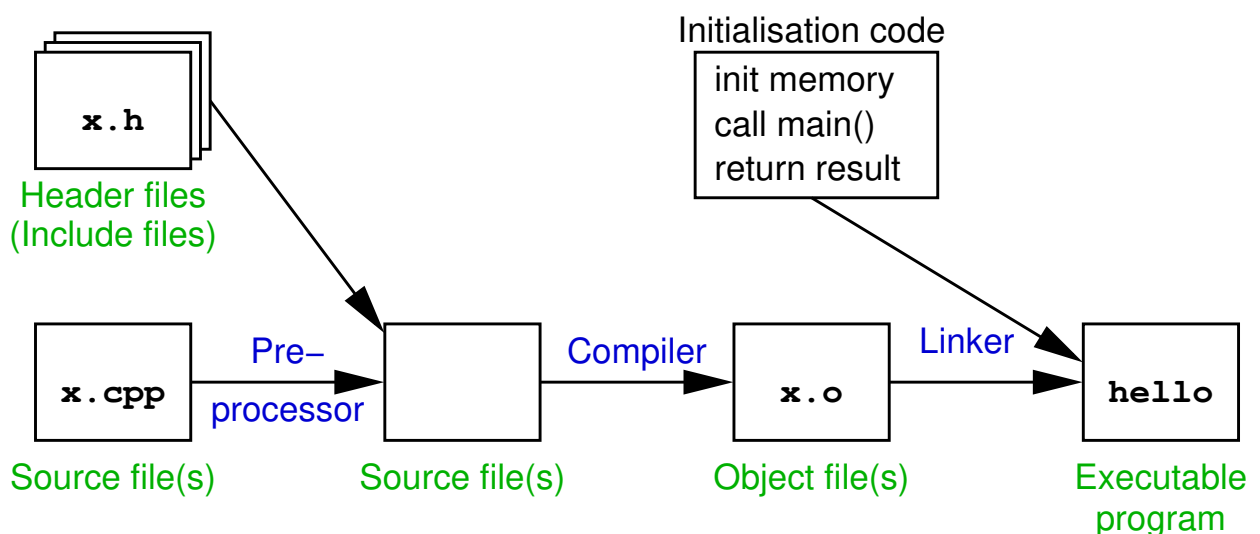
1.1.1 Fundamentals of C++

- ➔ 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

1.1.1 Fundamentals of C++ ...



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 GNU C++ compiler:
 - ➔ `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>`

1.1.1 Fundamentals of C++ ...



An example: *Hello World!* (👉 01/hello.cpp)

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

using namespace std; // Import all names from namespace 'std'

void sayHello() { // Function definition
    cout << "Hello World!\n"; // Print a text to console
}

int main() { // Main program
    sayHello();
    return 0; // Convention for return value: 0 = OK, 1,...,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

1.1.2 Data types in C++



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 (☞ **1.1.3**), 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
```



Templates

- ➔ Somehow similar to generics in Java

- i.e., classes (and methods) may have type parameters
- however, templates are more powerful (and complex) than generics

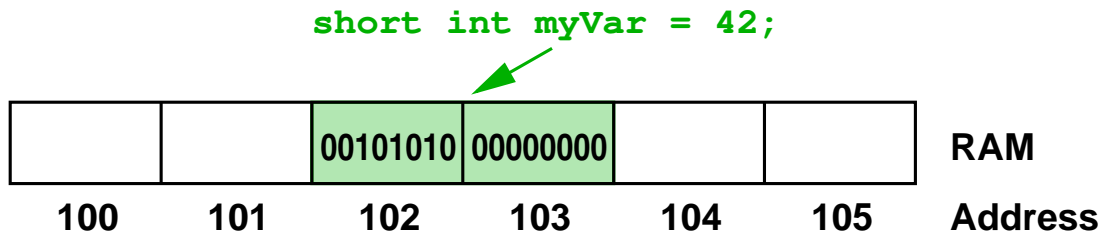
- ➔ Main goal: allow to implement generic classes / data structures, e.g., lists

- ➔ Usage of templates:

```
std::list<int> intl;      // List of integers  
intl.push_back(42);     // Add at the end of the list  
int i = intl.front();   // First element  
std::list<double> dbl;  // List of doubles  
dbl.push_back(3.1415);
```

Variables in memory

➔ Reminder: variables are stored in main memory



➔ 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

Notes for slide 34:

C++ also has the concept of *references* and so called *smart pointers*. Since these concepts are not needed to solve the lab assignments, they are not discussed here.

1.1.3 Pointers ...



(Animated slide)

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 address of a variable
- ➔ The access to *pAge is called **dereferencing** pAge
- ➔ Pointers (nearly) always have a type
 - ➔ e.g. int *, Example *, char **, ...

1.1.3 Pointers ...



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 address 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;
}
```

1.1.3 Pointers ...



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
```

1.1.4 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 exactly 4 decimal places

Parallel Processing

Winter Term 2025/26

20.10.2025

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

Stand: January 8, 2026

1.1.5 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;
}
```

1.1.5 Further specifics of C++ ...



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

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

- ➔ causes the variable/function to be visible 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!)

1.1.5 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	-pthread	threads	pthread_create()
mpi.h	-mpich	MPI	MPI_Init()

1.1.7 The C Preprocessor



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 compilation (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 ...`

1.2 Threads and Synchronization



Threads

- ➔ Activities within processes, concurrent to others
- ➔ Private resources:
 - ➔ CPU registers, including PC and stack pointer
 - ➔ local variables
- ➔ All other resources (esp. memory) are shared
- ➔ Threads are time-multiplexed to available CPU cores by the OS



Synchronization

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



Synchronization using shared variables

- ➔ Example: waiting for a result

Thread 1

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

Thread 2

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

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



Semaphores

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



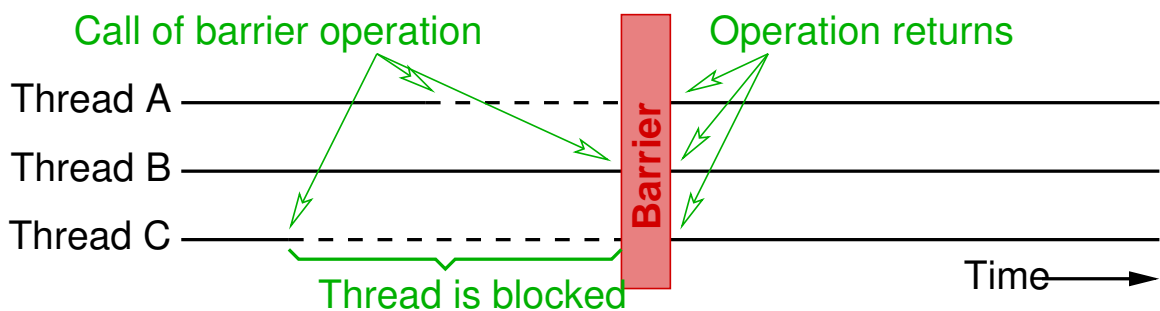
Monitors

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



Barrier

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



- ➔ Used to structure concurrent applications into synchronous phases



Synchronization errors

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



Example for *race conditions*

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

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

Thread 1

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

Thread 2

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

1.3 C++ Threads



- ➔ Part of the C++ language standard since 2011 (C++-11)
 - ➔ implemented by the compiler and the C++ libraries
 - ➔ independent of operating system
- ➔ Programming model:
 - ➔ at program start: exactly one (master) thread
 - ➔ master thread creates other threads and should wait for them to finish
 - ➔ process terminates when master thread terminates
 - ➔ when other threads are still running, an error is raised



Creating threads

- ➔ Class `std::thread`
 - ➔ represents a running thread
- ➔ Creation of a new thread (both C++-object and OS thread):
`std::thread myThread(function, args ...);`
 - ➔ with this declaration, the C++ object (and the OS thread) is automatically destroyed when the current scope is left
 - ➔ *function*: the function that should be executed by the thread
 - ➔ *args* ...: any number of parameters, which will be passed to *function*
 - ➔ *function* cannot have a return value
 - ➔ use result parameters instead



Methods of class `thread` (incomplete)

- ➔ `void join()`
 - ➔ waits until the thread execution has completed
 - ➔ after this method returns, the thread can be destroyed safely
- ➔ `void detach()`
 - ➔ detach the OS thread from the C++ thread object
 - ➔ the OS thread will continue its execution, even when the thread object is destroyed
 - ➔ the thread cannot be joined any more

1.3 C++ Threads ...



Example: Hello world (👉 01/helloThread.cpp)

```
#include <iostream>
#include <thread>

void sayHello()
{
    std::cout << "Hello World!\n";
}

int main(int argc, char **argv)
{
    std::thread t(sayHello);
    t.join();
    return 0;
}
```

1.3 C++ Threads ...



Example: Summation of an array with multiple threads

```
#include <iostream> (👉 01/sum.cpp)
#include <thread>

#define N 5
#define M 1000

/* This function is called by each thread */
void sumRow(int *row, long *res)
{
    int i;
    long sum = 0;
    for (i=0; i<M; i++)
        sum += row[i];

    *res = sum;    /* return the sum. */
}
```

1.3 C++ Threads ...



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

/* Main program */
int main(int argc, char **argv)
{
    int array[N][M];
    int i;
    std::thread threads[N];
    long res[N];
    long sum = 0;

    initArray(array); /* initialize the array */
}
```

1.3 C++ Threads ...



```
/* Create a thread for each row and pass the pointer to the row and the
pointer to the result variable as an argument */
for (i=0; i<N; i++) {
    threads[i] = std::thread(sumRow, array[i], &res[i]);
}

/* Wait for the threads' termination and sum the partial results */
for (i=0; i<N; i++) {
    threads[i].join();
    sum += res[i];
}

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

Compile and link the program

```
➔ g++ -o sum sum.cpp -pthread
```

Notes for slide 63:

In C++, the statement

```
threads[i] = std::thread(...);
```

actually means:

1. create a new (temporary) object of class `std::thread` by calling the proper constructor,
2. copy the object into the array,
3. delete the temporary object from step 1.

For threads, this sequence works due to the special implementation of the `std::thread` class, that overrides the assignment operator in such a way, that the OS thread is *not* copied and/or destroyed along with the C++ object.

A consequence of this assignment is that if the array `threads` is destroyed (because execution leaves the block where it has been declared), the threads are also destroyed.

63-1

1.3 C++ Threads ...



Remarks on the example

- ➔ When creating the thread, any number of parameters can be passed to the thread function
- ➔ Since the thread function has no return value, we pass the address of a result variable (`&res[i]`) as a parameter
 - ➔ the thread function will store its result there
 - ➔ caution: since `res` is a local variable, the threads must be joined before the method exits
- ➔ No synchronization (other than `join()`) is required
 - ➔ each thread stores to a different element of `res`
- ➔ With `join()`, we can only wait for a specific thread
 - ➔ inefficient, when the threads have different execution times

Synchronization: mutex variables

- ➔ Behavior similar to a binary semaphore
 - ➔ states: locked, unlocked; initial state: unlocked
- ➔ Declaration (and initialization):
`std::mutex mutex;`
- ➔ To lock the mutex, create an object of class `std::unique_lock`:
 - ➔ `std::unique_lock<std::mutex> lock(mutex);`
 - ➔ the mutex is automatically unlocked when `lock` is destroyed, i.e., when execution leaves the current block
- ➔ Class `mutex` does not allow recursive locking
 - ➔ i.e., the same thread cannot lock the mutex twice
 - ➔ use class `recursive_mutex` for this purpose

Notes for slide 65:

Please see the C++ reference for more information about the mutex and lock classes, e.g. <http://www.cplusplus.com/reference/mutex/>.

There are, for instance, also classes `timed_mutex` and `recursive_timed_mutex` which enable to have a timeout when trying to lock the mutex.

Synchronization: condition variables

- ➔ Declaration (and initialization):
`std::condition_variable cond;`
- ➔ Important methods:
 - ➔ wait: `void wait(unique_lock<mutex>& lock)`
 - ➔ thread is blocked, the mutex wrapped by `lock` will be unlocked temporarily
 - ➔ signaling thread keeps the mutex, i.e., the signaled condition may no longer hold when `wait()` returns!
 - ➔ typical use: `while (!condition_met) cond.wait(lock);`
 - ➔ signal just one thread: `void notify_one()`
 - ➔ signal all threads: `void notify_all()`

Notes for slide 66:

The syntax `unique_lock<mutex>& lock` indicates that the argument of the method `wait()` is passed by reference rather than by value.



Parallel Processing

Winter Term 2025/26

21.10.2025

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

Stand: January 8, 2026

1.3 C++ Threads ...



Example: simulating a monitor with C++ threads

```
#include <thread> (👉 01/monitor.cpp)
#include <mutex> // Defines std::mutex
#include <condition_variable> // Defines std::condition_variable

std::mutex mutex;
std::condition_variable cond;
volatile int ready = 0;
volatile int result;

void storeResult(int arg) {
    std::unique_lock<std::mutex> lock(mutex);
    result = arg; /* store result */
    ready = 1;
    cond.notify_all();
    // The 'lock' object is destroyed when the method ends, thus unlocking the mutex!
}
```

Notes for slide 67:

The keyword `volatile` at the beginning of the declarations of global variables indicates the **compiler** must actually perform any read and write operation programmed in the source code (in the given order), i.e., the compiler must not apply any optimizations to this variable (especially “caching” the value in a register). This is necessary here, as the variables can be modified at any time by another thread.

Note that `volatile` does **not** imply sequential consistency, since it only imposes a restriction on the compiler, not on the CPU.

67-1

1.3 C++ Threads ...



Example: simulating a monitor with C++ threads ...

```
int readResult()  
{  
    std::unique_lock<std::mutex> lock(mutex);  
    while (ready != 1)  
        cond.wait(lock);  
    return result; // mutex unlocked automatically when 'lock' is destroyed.  
}
```

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

Notes for slide 68:

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

68-1



Parallel Processing

Winter Term 2025/26

2 Basics of Parallel Processing



Contents

- ➔ Motivation
- ➔ Parallelism
- ➔ Parallelism and data dependences
- ➔ Parallel computer architectures
- ➔ Parallel programming models
- ➔ Organisation forms for parallel programs
- ➔ A design process for parallel programs
- ➔ Performance considerations

Literature

- ➔ Ungerer
- ➔ Grama, Gupta, Karypis, Kumar

2.1 Motivation



What is parallelism?

- ➔ In general:
 - ➔ executing more than one action at a time
- ➔ Specifically with respect to execution of programs:
 - ➔ at some point in time
 - ➔ more than one statement is executed
 - and / or
 - ➔ more than one pair of operands is processed
- ➔ Goal: faster solution of the task to be processed
- ➔ Problems: subdivision of the task, coordination overhead



Why parallel processing?

- ➔ Applications with high computing demands, esp. simulations
 - climate, earthquakes, superconductivity, molecular design, ...
- ➔ Example: protein folding
 - 3D structure, function of proteins (Alzheimer, BSE, ...)
 - $1,5 \cdot 10^{11}$ floating point operations (Flop) / time step
 - time step: $5 \cdot 10^{-15} s$
 - to simulate: $10^{-3} s$
 - $3 \cdot 10^{22}$ Flop / simulation
 - \Rightarrow 1 year computation time on a PFlop/s computer!
- ➔ For comparison: world's currently fastest computer: El Capitan (LLNL, USA), 1742 PFlop/s (with 11039616 CPU cores!)

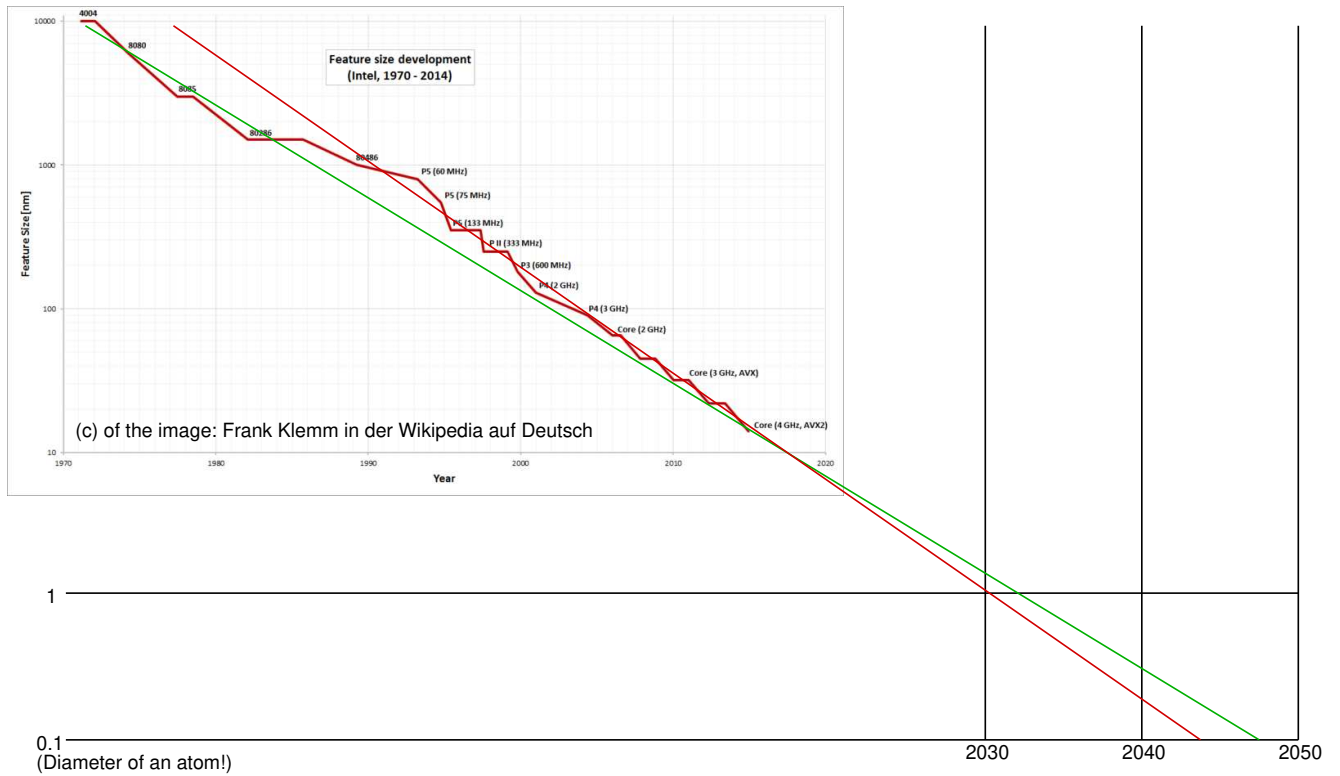


Why parallel processing? ...

- ➔ **Moore's Law:** the computing power of a processor doubles every 18 months
 - but: memory speed increases much slower
 - 2040 the latest: physical limit will be reached
- ➔ Thus:
 - high performance computers are based on parallel processing
 - even standard CPUs use parallel processing internally
 - super scalar processors, pipelining, multicore, ...
- ➔ Economic advantages of parallel computers
 - cheap standard CPUs instead of specifically developed ones

Notes for slide 73:

An estimation of the end of “Moore’s Law”:

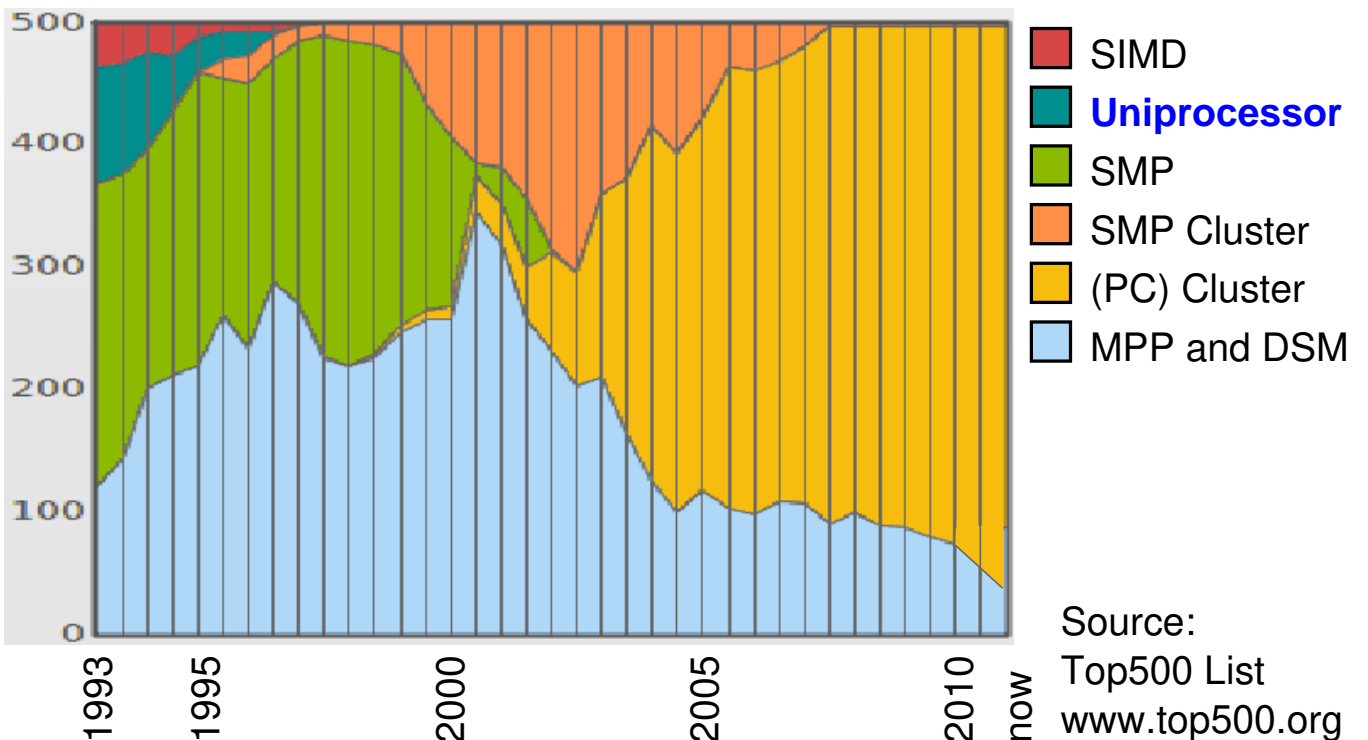


73-1

2.1 Motivation ...



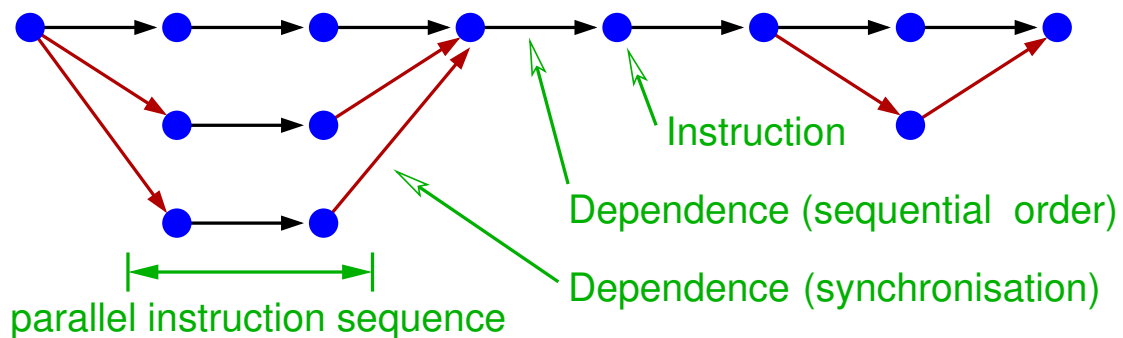
Architecture trend of high performance computers





What is a parallel programm?

- ➔ A parallel program can be viewed as a partially ordered set of instructions (activities)
 - ➔ the order is given by the dependences between the instructions
- ➔ Independent instructions can be executed in parallel



2.2 Parallelism ...

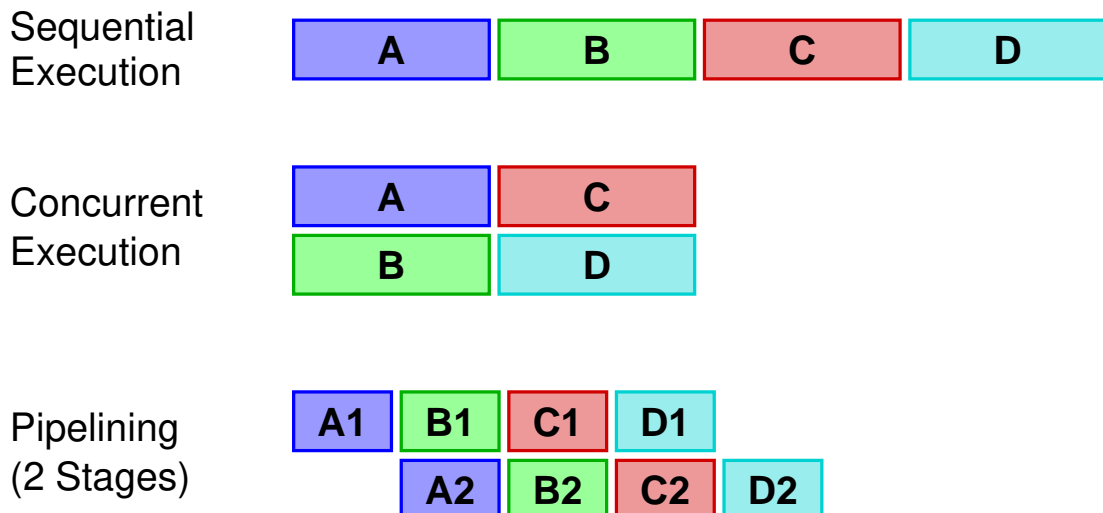


Concurrency vs. pipelining

- ➔ **Concurrency** (*Nebenläufigkeit*): instructions are executed simultaneously in different execution units
- ➔ **Pipelining**: execution of instructions is subdivided into sequential phases. Different phases of **different** instruction **instances** are executed simultaneously.
- ➔ Remark: here, the term “instruction” means a generic compute activity, depending on the layer of abstraction we are considering
 - ➔ e.g., machine instruction, execution of a sub-program



Concurrency vs. pipelining ...



At which layers of programming can we use parallelism?

- ➔ There is no consistent classification
- ➔ E.g., layers in the book from Waldschmidt, *Parallelrechner: Architekturen - Systeme - Werkzeuge*, Teubner, 1995:
 - ➔ application programs
 - ➔ cooperating processes
 - ➔ data structures
 - ➔ statements and loops
 - ➔ machine instruction

“They are heterogeneous, subdivided according to different characteristics, and partially overlap.”



View of the application developer (design phase):

- ➔ “Natural parallelism”
 - ➔ e.g., computing the forces for all stars of a galaxy
 - ➔ often too fine-grained
- ➔ **Data parallelism** (domain decomposition, *Gebietsaufteilung*)
 - ➔ e.g., sequential processing of all stars in a space region
- ➔ **Task parallelism**
 - ➔ e.g., pre-processing, computation, post-processing, visualisation



View of the programmer:

- ➔ **Explicit parallelism**
 - ➔ exchange of data (communication / synchronisation) must be explicitly programmed
- ➔ **Implicit parallelism**
 - ➔ by the compiler
 - ➔ directive controlled or automatic
 - ➔ loop level / statement level
 - ➔ compiler generates code for communication
 - ➔ within a CPU (that appears to be sequential from the outside)
 - ➔ super scalar processor, pipelining, ...



View of the system (computer / operating system):

- ➔ **Program level (job level)**
 - ➔ independent programs
- ➔ **Process level (task level)**
 - ➔ cooperating processes
 - ➔ mostly with explicit exchange of messages
- ➔ **Block level**
 - ➔ light weight processes (threads)
 - ➔ communication via shared memory
 - ➔ often created by the compiler
 - ➔ parallelisation of loops



View of the system (computer / operating system): ...

- ➔ **Instruction level**
 - ➔ elementary instructions (operations that cannot be further subdivided in the programming language)
 - ➔ scheduling is done automatically by the compiler and/or by the hardware at runtime
 - ➔ e.g., in VLIW (EPIC, e.g. Itanium) and super scalar processors
- ➔ **Sub-operation level**
 - ➔ compiler or hardware subdivide elementary instructions into sub-operations that are executed in parallel
 - ➔ e.g., with vector or array operations



Granularity

- ➔ Defined by the ratio between computation and communication (including synchronisation)
 - intuitively, this corresponds to the length of the parallel instruction sequences in the partial order
 - determines the requirements for the parallel computer
 - especially its communication system
 - influences the achievable acceleration (*speedup*)
- ➔ Coarse-grained: program and process level
- ➔ Mid-grained: block level
- ➔ Fine-grained: instruction level

2.3 Parallelisation and Data Dependences



(Animated slide)

- ➔ Important question: when can two instructions S_1 and S_2 be executed in parallel?
 - Answer: if there are no **dependences** between them
- ➔ Assumption: instruction S_1 can and should be executed **before** instruction S_2 according to the sequential code
 - e.g.: $S_1: x = b + 2 * a;$
 $y = a * (c - 5);$
 $S_2: z = \text{abs}(x - y);$
 - but also in different iterations of a loop
- ➔ **True / flow dependence** (*echte Abhängigkeit*) $S_1 \xrightarrow{\delta^t} S_2$

$S_1: a[1] = a[0] + b[1];$

$S_2: a[2] = a[1] + b[2];$

S_1 ($i=1$) writes to $a[1]$, which is later read by S_2 ($i=2$)

2.3 Parallelisation and Data Dependences ...



(Animated slide)

➔ **Anti dependence** (*Antiabhängigkeit*) $S_1 \xrightarrow{\delta^a} S_2$

S_1 : `a[1] = a[2];`

S_2 : `a[2] = a[3];`

S_1 (i=1) read the value of `a[2]`, which is overwritten by S_2 (i=2)

➔ **Output dependence** (*Ausgabeabhängigkeit*) $S_1 \xrightarrow{\delta^o} S_2$

S_1 : `s = a[1];`

S_2 : `s = a[2];`

S_1 (i=1) writes a value to `s`, which is overwritten by S_2 (i=2)

➔ **Anti** and **Output** dependences can always be removed by consistent renaming of variables

2.3 Parallelisation and Data Dependences ...



(Animated slide)

Data dependences and synchronisation

➔ Two instructions S_1 and S_2 with a data dependence $S_1 \rightarrow S_2$ can be distributed by different threads **only if** a correct synchronisation is performed

➔ S_2 must be executed **after** S_1

➔ e.g., by using `signal/wait` or a message

➔ in the previous example:

Thread 1

`x = b + 2 * a;`

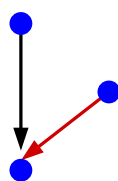
`wait(cond);`

`z = abs(x-y);`

Thread 2

`y = a * (c-5);`

`signal(cond);`





Classification of computer architectures according to Flynn

- ➔ Criteria for differentiation:
 - ➔ how many **instruction streams** does the computer process at a given point in time (single, multiple)?
 - ➔ how many **data streams** does the computer process at a given point in time (single, multiple)?
- ➔ This leads to four possible classes:
 - ➔ SISD: **S**ingle **I**nstruction stream, **S**ingle **D**ata stream
 - ➔ single processor (core) systems
 - ➔ MIMD: **M**ultiple **I**nstruction streams, **M**ultiple **D**ata streams
 - ➔ all kinds of multiprocessor systems
 - ➔ SIMD: vector computers, vector extensions, GPUs
 - ➔ MISD: empty, not really sensible

2.4 Parallel Computer Architectures ...



Classes of MIMD computers

- ➔ Considering two criteria:
 - ➔ physically global vs. distributed memory
 - ➔ shared vs. distributed address space
- ➔ **NORMA: No Remote Memory Access**
 - ➔ distributed memory, distributed address space
 - ➔ i.e., no access to memory modules of non-local nodes
 - ➔ communication is only possible via messages
 - ➔ typical representative of this class:
 - ➔ ***distributed memory systems (DMM)***
 - ➔ also called MPP (massively parallel processor)
 - ➔ in principle also any computer networks (cluster, grid, cloud, ...)



Classes of MIMD computers ...

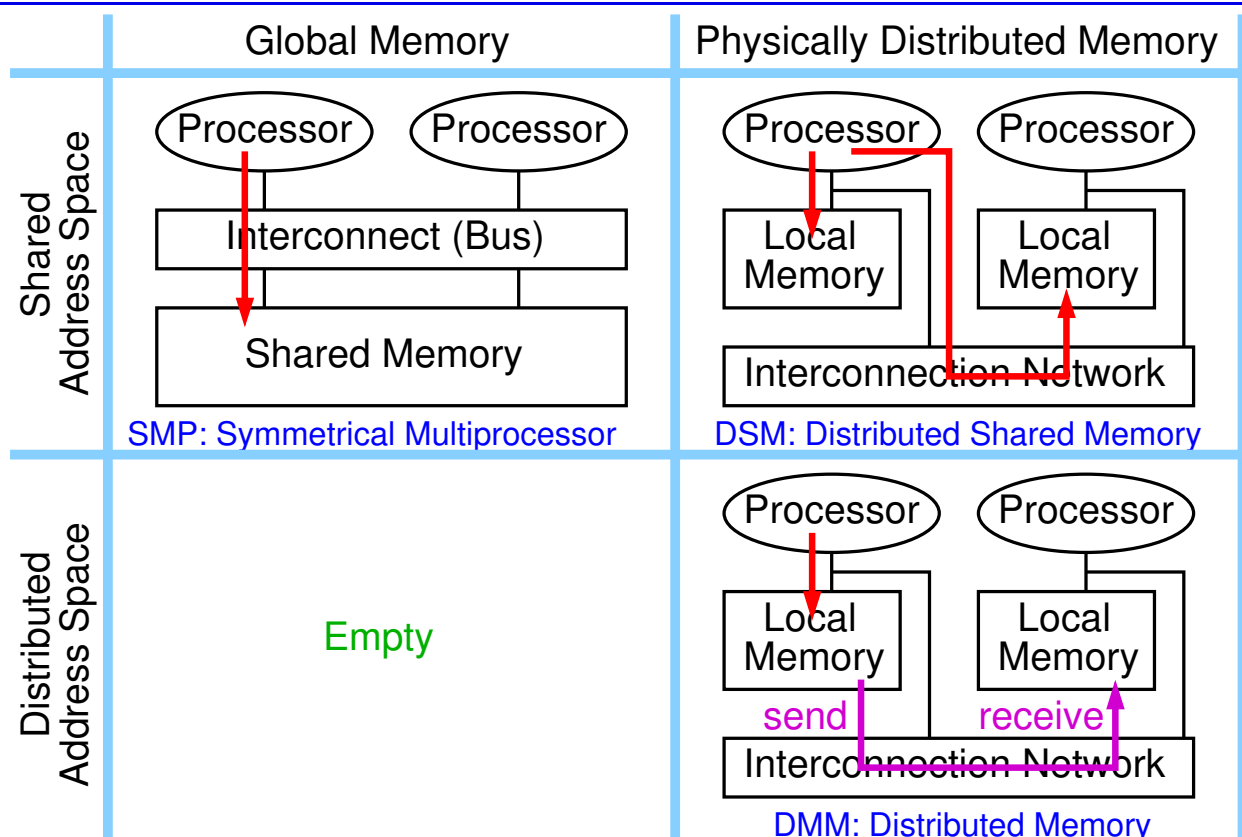
➔ UMA: Uniform Memory Access

- ➔ global memory, shared address space
- ➔ all processors access the memory in the same way
- ➔ access time is equal for all processors
- ➔ typical representative of this class:
symmetrical multiprocessor (SMP), early multicore-CPUs

➔ NUMA: Nonuniform Memory Access

- ➔ distributed memory, shared address space
- ➔ access to local memory is faster than access to remote one
- ➔ typical representative of this class:
distributed shared memory (DSM) systems, modern multicore-CPUs

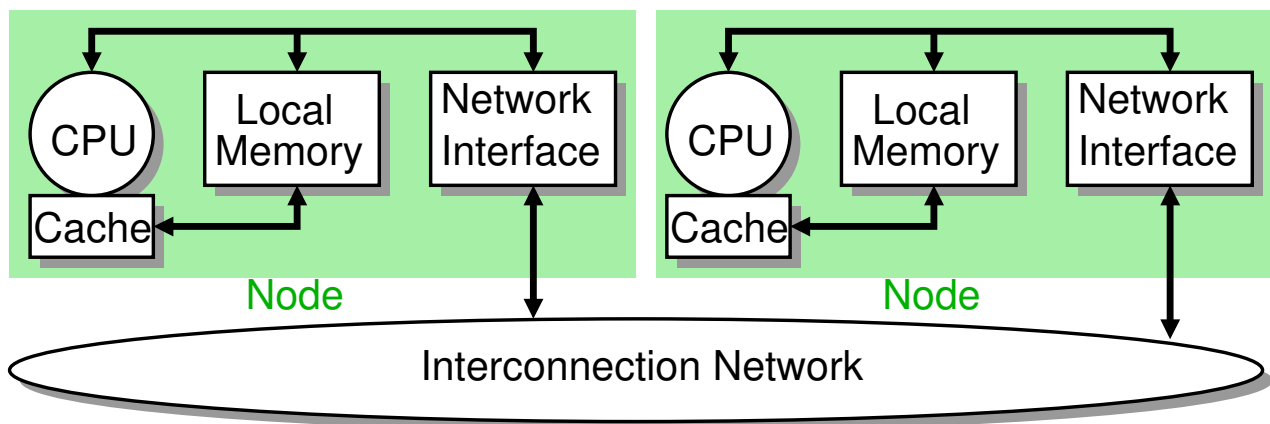
2.4 Parallel Computer Architectures ...



2.4.1 MIMD: Message Passing Systems



Multiprocessor systems with distributed memory



- ➔ **NORMA**: No Remote Memory Access
- ➔ Good scalability (up to several 100000 nodes)
- ➔ Communication and synchronisation via message passing

2.4.1 MIMD: Message Passing Systems ...



Historical evolution

- ➔ In former times: proprietary hardware for nodes and network
 - distinct node architecture (processor, network adapter, ...)
 - often static interconnection networks with *store and forward*
 - often distinct (mini) operating systems
- ➔ Today:
 - cluster with standard components (PC server)
 - usually with SMP (sometimes vector computers) as nodes
 - nodes often use accelerators (GPUs)
 - switched high performance interconnection networks
 - 100Gbit/s Ethernet, Infiniband, ...
 - standard operating systems (UNIX or Linux derivatives)



Properties

- ➔ No shared memory or address areas between nodes
- ➔ Communication via exchange of messages
 - application layer: libraries like e.g., MPI
 - system layer: proprietary protocols or TCP/IP
 - latency caused by software often much larger than hardware latency ($\sim 1 - 50\mu s$ vs. $\sim 20 - 100ns$)
- ➔ In principle unlimited scalability
 - e.g. Frontier: 135936 nodes, (8699904 cores)



Parallel Processing

Winter Term 2025/26

27.10.2025

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

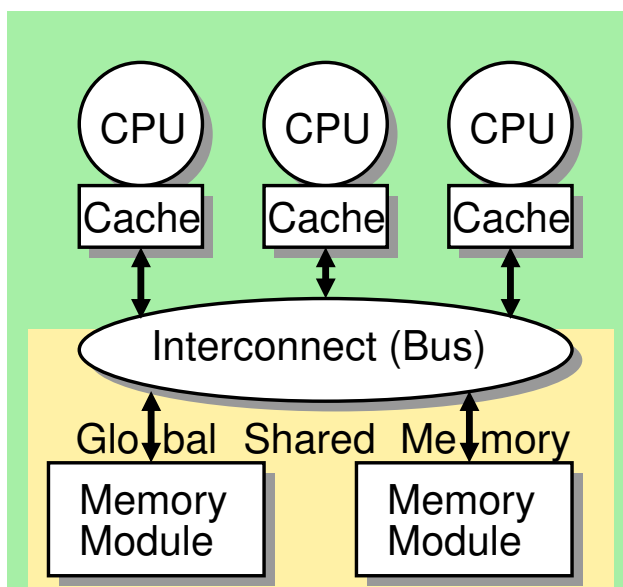
Stand: January 8, 2026

Properties ...

- ➔ Independent operating system on each node
- ➔ Often with shared file system
 - e.g., parallel file system, connected to each node via a (distinct) interconnection network
 - or simply NFS (in small clusters)
- ➔ Usually no *single system image*
 - user/administrator “sees” several computers
- ➔ Often no direct, interactive access to all nodes
 - *batch queueing systems* assign nodes (only) on request to parallel programs
 - often exclusively: *space sharing*, partitioning
 - often small fixed partition for login and interactive use

2.4.2 MIMD: Shared Memory Systems

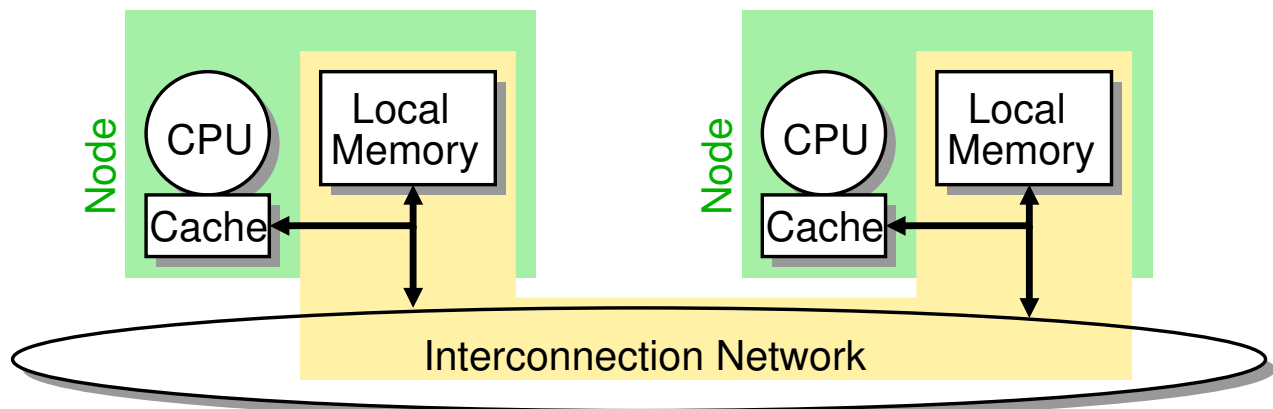
Symmetrical multiprocessors (SMP)



- ➔ Global address space
- ➔ **UMA**: *uniform memory access*
- ➔ Communication and Synchronisation via shared memory
- ➔ only feasible with very few processors (ca. 2 - 32)



Multiprocessor systems with distributed shared memory (DSM)



- ➔ Distributed memory, accessible by all CPUs
- ➔ **NUMA**: *non uniform memory access*
- ➔ Combines shared memory and scalability



Properties

- ➔ All Processors can access all resources in the same way
 - ➔ but: different access times in NUMA architectures
 - ➔ distribute the data such that most accesses are local
- ➔ Only one instance of the operating systems for the whole computer
 - ➔ distributes processes/thread amongst the available processors
 - ➔ all processors can execute operating system services in an equal way
- ➔ *Single system image*
 - ➔ for user/administrator virtually no difference to a uniprocessor system
- ➔ Especially SMPs (UMA) only have limited scalability



Caches in shared memory systems

- ➔ **Cache**: fast intermediate storage, close to the CPU
 - ➔ stores copies of the most recently used data from main memory
 - ➔ when the data is in the cache: no access to main memory is necessary
 - ➔ access is 10-1000 times faster
- ➔ Cache are essential in multiprocessor systems
 - ➔ otherwise memory and interconnection network quickly become a bottleneck
 - ➔ exploiting the property of locality
 - ➔ each process mostly works on “its own” data
- ➔ But: the existence of multiple copies of data can lead to inconsistencies: **cache coherence problem** (BS-1)

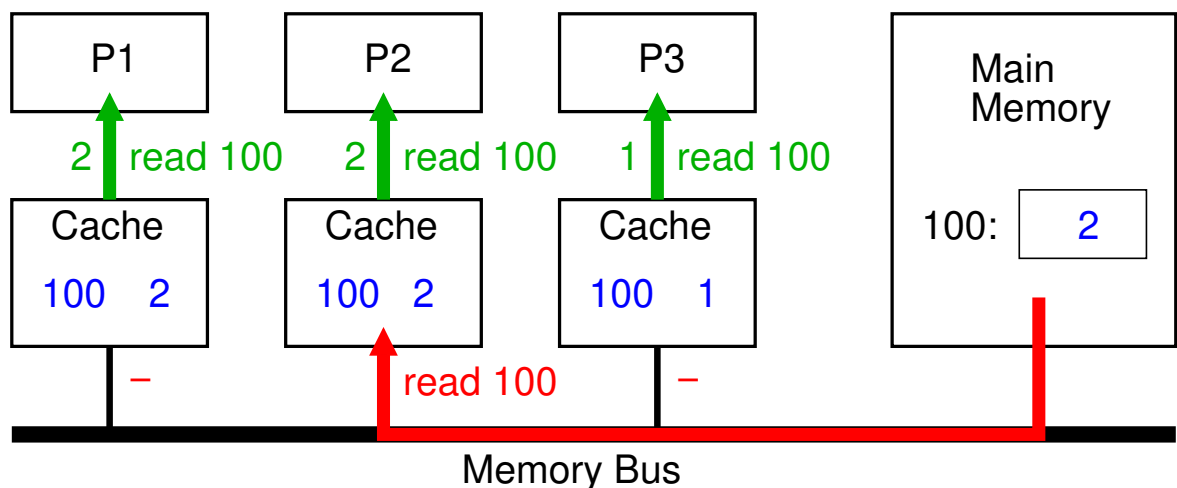
2.4.2 MIMD: Shared Memory Systems ...



(Animated slide)

Cache Coherence Problem: Example

- ➔ Assumption: write access directly updates main memory
- ➔ Three processors access the same memory location and get different results!





Enforcing cache coherency

- ➔ During a write access, all affected caches (= caches with copies) must be notified
 - ➔ caches invalidate or update the affected entry
- ➔ In UMA systems
 - ➔ bus as interconnection network: every access to main memory is visible for everybody (broadcast)
 - ➔ caches “listen in” on the bus (*bus snooping*)
 - ➔ (relatively) simple cache coherence protocols
 - ➔ e.g., MESI protocol
 - ➔ but: bad scalability, since the bus is a shared central resource



Enforcing cache coherency ...

- ➔ In NUMA systems (ccNUMA: *cache coherent NUMA*)
 - ➔ accesses to main memory normally are not visible to other processors
 - ➔ affected caches must be notified explicitly
 - ➔ requires a list of all affected caches (broadcasting to all processors is too expensive)
 - ➔ message transfer time leads to additional consistency problems
 - ➔ cache coherence protocols (*directory protocols*) become very complex
 - ➔ but: good scalability



Memory consistency (*Speicherkonsistenz*)

- ➔ Cache coherence only defines the behavior with respect to **one memory location** at a time
 - ➔ **which values** can a read operation return?
- ➔ Remaining question:
 - ➔ **when** does a processor see the value, which was written by another processor?
 - ➔ more exact: in **which order** does a processor see the write operations on **different** memory locations?



Memory consistency: a simple example

Thread T_1	Thread T_2
A = 0;	B = 0;
...;	...;
A = 1;	B = 1;
print B;	print A;

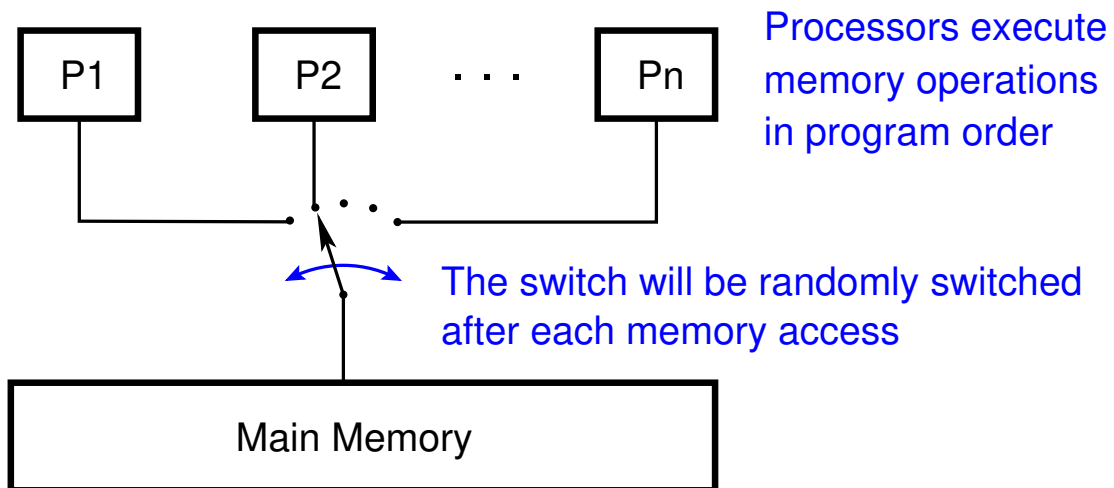
- ➔ Intuitive expectation: the output "0 0" can never occur
- ➔ But: with many SMPs/DSMs the output "0 0" is possible
 - ➔ (CPUs with dynamic instruction scheduling or write buffers)
- ➔ In spite of cache coherency: intuitively inconsistent view on the main memory:

$$T_1: A=1, B=0 \quad T_2: A=0, B=1$$



Definition: sequential consistency

Sequential consistency is given, when the result of each execution of a parallel program can also be produced by the following abstract machine:



Interleavings (*Verzahnungen*) in the example

Some possible execution sequences using the abstract machine:			No sequential consistency:
A = 0	A = 0	A = 0	
B = 0	B = 0	B = 0	
A = 1	A = 1	B = 1	
B = 1	print B	print A	
print B	B = 1	A = 1	
print A	print A	print B	
B=1 A=1	B=0 A=1	B=1 A=0	A = 0
			B = 0
			B = 1
			print A
			A = 1
			print B
			B=0 A=0



Weak consistency models

- ➔ The requirement of sequential consistency leads to strong restrictions for the computer architecture
 - ➔ CPUs can not use instruction scheduling and write buffers
 - ➔ NUMA systems can not be realized efficiently
- ➔ Thus: parallel computers with shared memory (UMA and NUMA) use **weak consistency models!**
 - ➔ allows, e.g., swapping of write operations
 - ➔ however, each processor **always** sees **its own** write operations in program order
- ➔ Remark: also optimizing compilers can lead to weak consistency
 - ➔ swapping of instructions, register allocation, ...
 - ➔ declare the affected variables as `volatile!`



Consequences of weak consistency: examples

- ➔ all variables are initially 0

Possible results with sequential consistency		"unexpected" behavior with weak consistency:	
<pre>A=1; print B;</pre>	<pre>B=1; print A;</pre>	<p>0,1 1,0 1,1</p>	<p>due to swapping of the read and write accesses</p>
<pre>A=1; valid=1;</pre>	<pre>while (!valid); print A;</pre>	<p>1</p>	<p>due to swapping of the write accesses to A and valid</p>



Weak consistency models ...

- ➔ Memory consistency can (and must!) be enforced as needed, using special instructions
 - ➔ *fence / memory barrier (Speicherbarriere)*
 - ➔ all previous memory operations are completed; subsequent memory operations are started only after the barrier
 - ➔ *acquire and release*
 - ➔ *acquire*: subsequent memory operations are started only after the *acquire* is finished
 - ➔ *release*: all previous memory operations are completed
 - ➔ pattern of use is equal to mutex locks

2.4.2 MIMD: Shared Memory Systems ...



Enforcing consistency in the examples

- ➔ Here shown with memory barriers:

<pre>A=1; fence; print B;</pre>	<pre>B=1; fence; print A;</pre>	Fence ensures that the write access is finished before reading
<pre>A=1; fence; valid=1;</pre>	<pre>while (!valid); fence; print A;</pre>	Fence ensures that 'A' is valid before 'valid' is set and that A is read only after 'valid' has been set

Notes for slide 109:

In C++, there are special `atomic` types that allow to realize sequential consistency where needed. In C++, the last example would be (using acquire/release):

➔ Initialization:

```
int A = 0;
std::atomic<int> valid = 0;
```

➔ Code of Thread 1:

```
A = 1;
valid.store(1, std::memory_order_release);
```

➔ Code of Thread 2:

```
while (!valid.load(std::memory_order_acquire));
std::cout << A << std::endl;
```

See, e.g., https://en.cppreference.com/w/cpp/atomic/memory_order for a detailed discussion.

109-1

2.4.3 SIMD



- ➔ Only a single instruction stream, however, the instructions have **vectors** as operands \Rightarrow data parallelism
- ➔ **Vector** = one-dimensional array of numbers
- ➔ Variants:
 - ➔ vector computers
 - ➔ pipelined arithmetic units (vector units) for the processing of vectors
 - ➔ SIMD extensions in processors (SSE, AVX)
 - ➔ Intel: 256 Bit registers with, e.g., four 64 Bit `double` values
 - ➔ graphics processors (GPUs)
 - ➔ multiple streaming multiprocessors
 - ➔ streaming multiprocessor contains several arithmetic units (*CUDA cores*), which all execute the same instruction

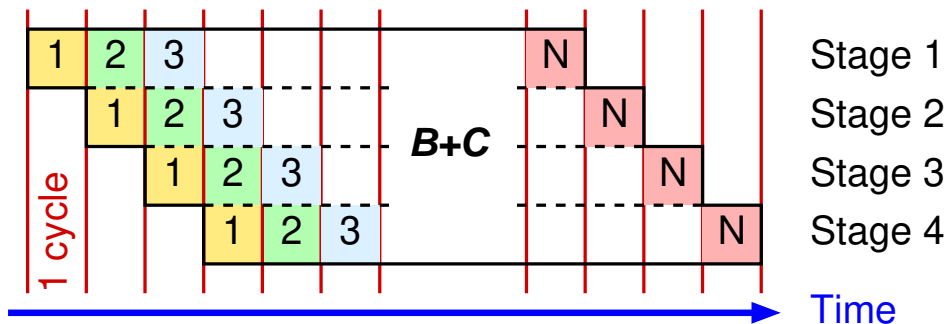
2.4.3 SIMD ...



(Animated slide)

Example: addition of two vectors

- ➔ $A_j = B_j + C_j$, for all $j = 1, \dots, N$
- ➔ Vector computer: the elements of the vectors are added in a pipeline: **sequentially**, but **overlapping**
 - ➔ if a scalar addition takes four clock cycles (i.e., 4 pipeline stages), the following sequence will result:



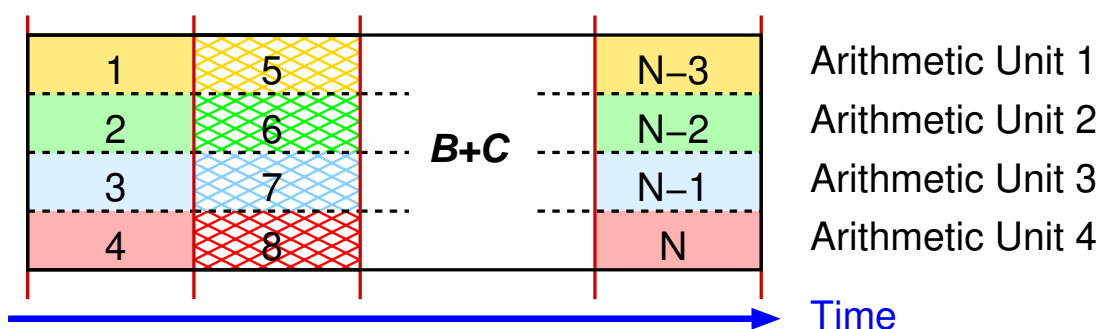
2.4.3 SIMD ...



(Animated slide)

Example: addition of two vectors

- ➔ $A_j = B_j + C_j$, for all $j = 1, \dots, N$
- ➔ AVX and GPU: several elements of the vectors are added **concurrently (in parallel)**
 - ➔ if, e.g., four additions can be done at the same time, the following sequence will result:

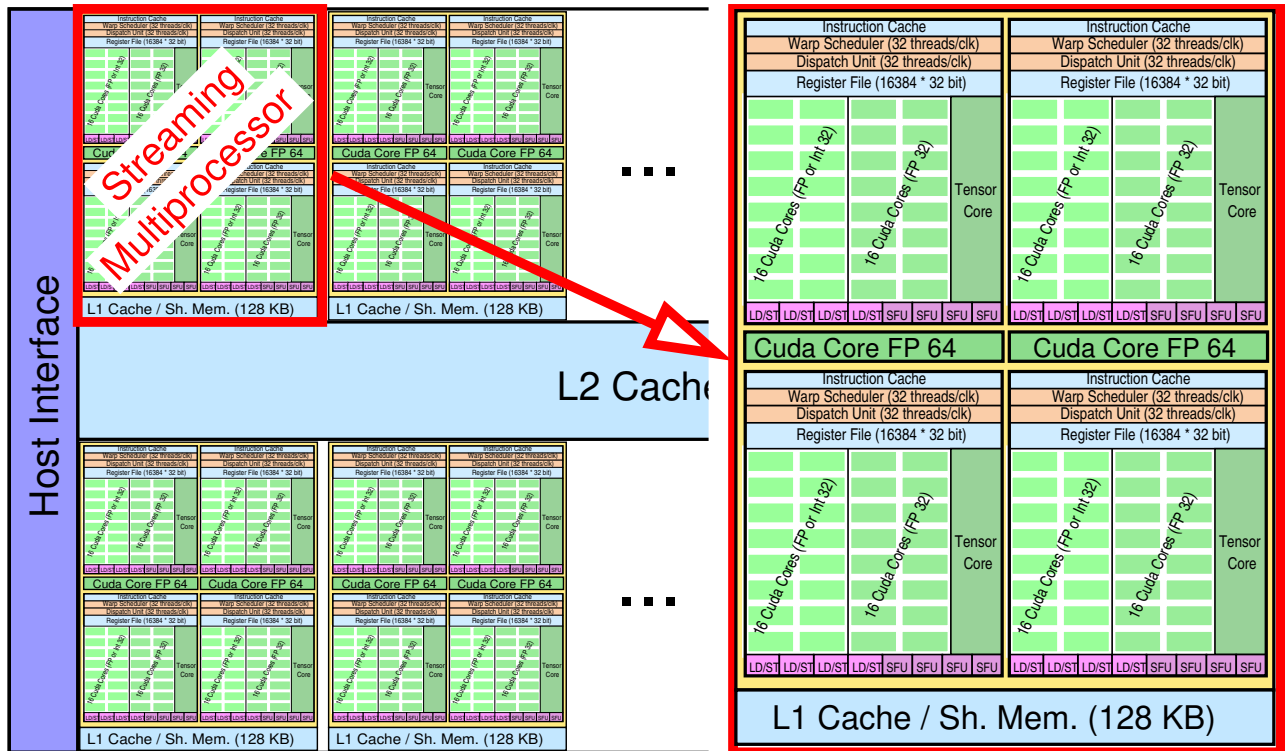


2.4.3 SIMD ...



(Animated slide)

Architecture of a GPU (NVIDIA Ada)



Notes for slide 113:

- ➔ The RTX 4060 (in the lab computers) has 24 Streaming Multiprocessors.
- ➔ The figure does not show graphics-specific components of the architecture, e.g. texture units or ray-tracing cores.



Programming of GPUs (NVIDIA Ada)

- ➔ Partitioning of the code in *blocks* of threads (max. 1024)
- ➔ *Blocks* are distributed between streaming multiprocessors (SMs)
 - ➔ no synchronisation possible between SMs
- ➔ *Warp scheduler* in SM dispatches threads to cores
 - ➔ a *warp* consists of 32 threads
 - ➔ multiple warps per scheduler support latency hiding
 - ➔ one limiting factor: number of registers
 - ➔ on a SM, threads all execute the same instruction (on different data) or none at all (\sim SIMD)
 - ➔ e.g., with `if-then-else`:
 - ➔ first some cores execute the `then` branch,
 - ➔ then the other cores execute the `else` branch



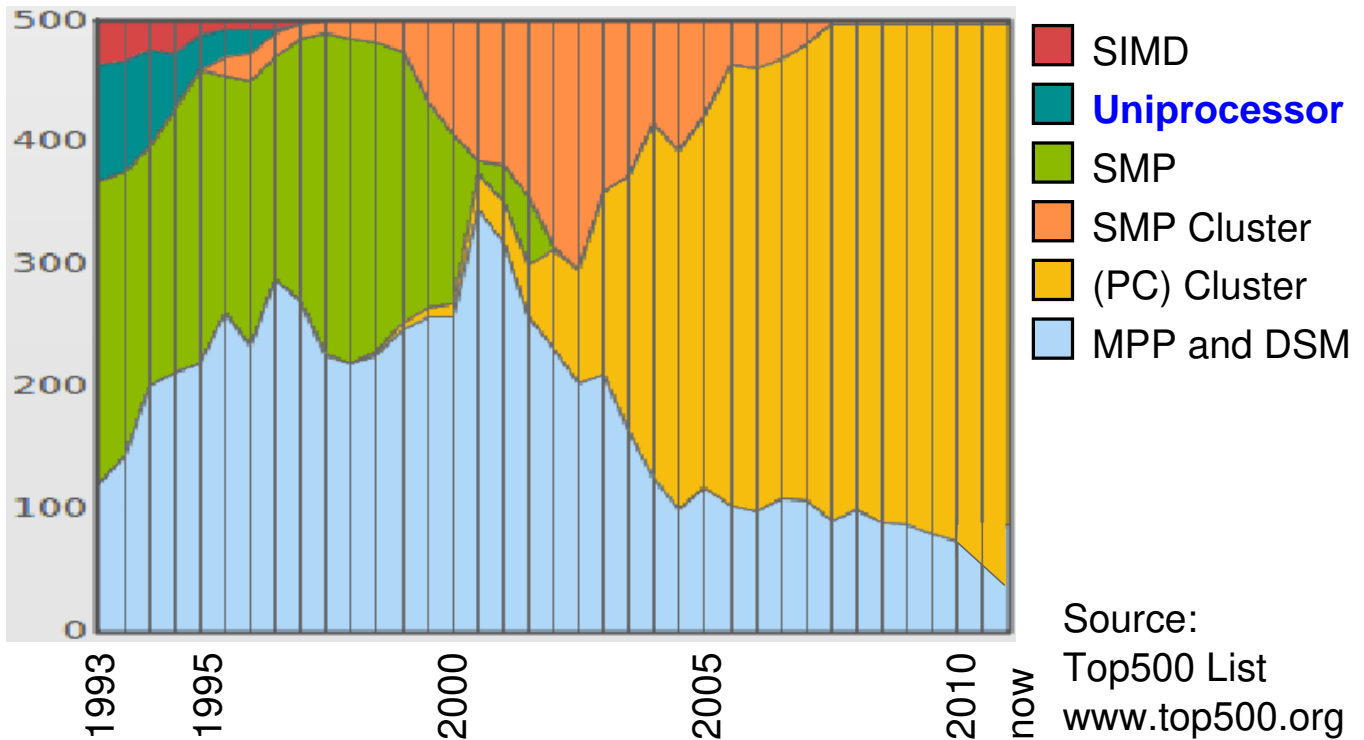
Programming of GPUs (NVIDIA Ada) ...

- ➔ Threads in one block should address subsequent memory locations
 - ➔ only in this case, memory accesses can be merged
- ➔ 32-Bit FP performance is 64 times the 64-Bit FP performance!

2.4.4 High Performance Supercomputers



Trends



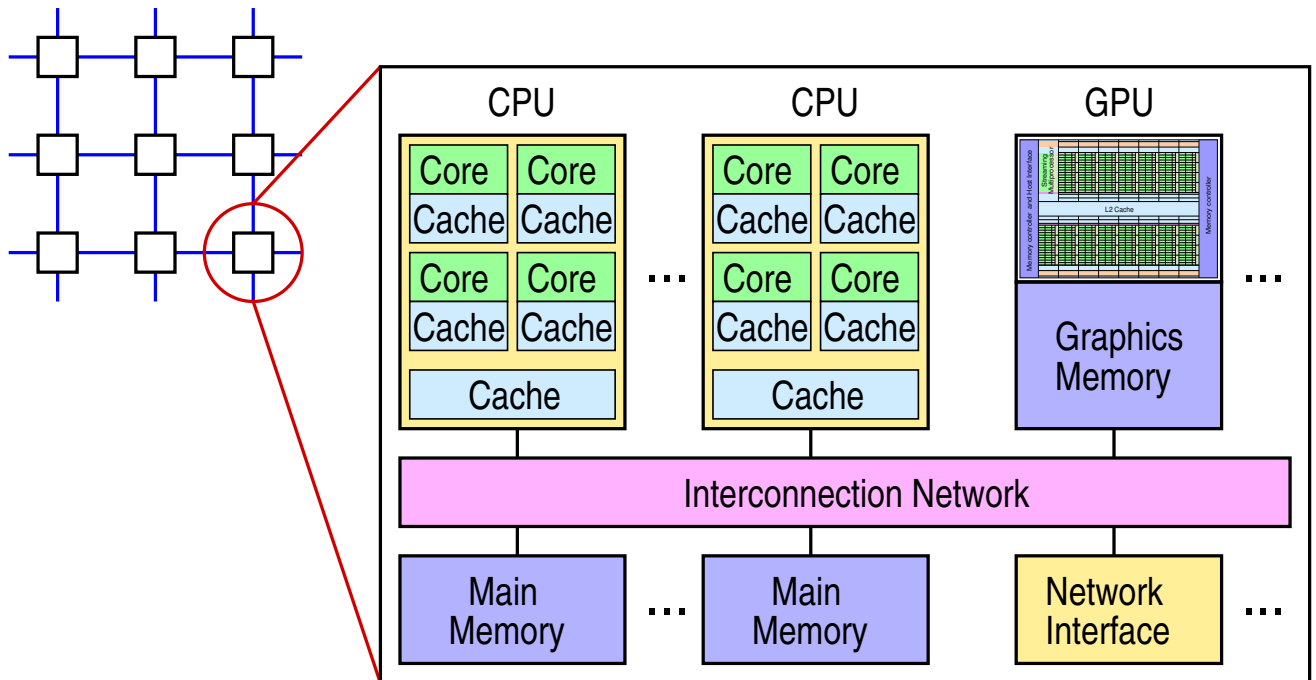
2.4.4 High Performance Supercomputers ...



Typical architecture:

- ➔ Message passing computers with SMP nodes and accelerators (e.g. GPUs)
 - ➔ at the highest layer: systems with distributed memory
 - ➔ nodes: NUMA systems with partially shared cache hierarchy
 - ➔ in addition one or more accelerators per node
- ➔ Compromise between scalability, programmability and performance
- ➔ Programming with hybrid programming model
 - ➔ message passing between the nodes (manually, MPI)
 - ➔ shared memory on the nodes (compiler supported, e.g., OpenMP)
 - ➔ if need be, additional programming model for accelerators

Typical architecture: ...



2.5 Parallel Programming Models

In the following, we discuss:

- ➔ Shared memory
 - ➔ Message passing
 - ➔ Distributed objects
 - ➔ Data parallel languages
- ➔ The list is not complete (e.g., data flow models, PGAS)

Notes for slide 119:

- ➔ In the data flow model, a program is specified as a data flow graph (c.f. 2.7.7), where a node can be computed ('fired') as soon as all its input data is available. The edges in the data flow actually are the true dependences of the program.
- ➔ PGAS = Partitioned Global Address Space. PGAS languages offer a shared memory programming model on a distributed memory computer. In a PGAS language, pointers (or references) can point to data on a remote node. When the pointer is dereferenced, the compiler automatically generates the required message exchange for fetching or storing the data.

119-1

2.5.1 Shared Memory



- ➔ Light weight processes (threads) share a common virtual address space
- ➔ The “more simple” parallel programming model
 - ➔ all threads have access to all data
 - ➔ also good theoretical foundation (PRAM model)
- ➔ Mostly with shared memory computers
 - ➔ however also implementable on distributed memory computers (with large performance penalty)
 - ➔ *shared virtual memory* (SVM)
- ➔ Examples:
 - ➔ PThreads, Java Threads, C++ Threads
 - ➔ Intel Threading Building Blocks (TBB)
 - ➔ OpenMP (👉 3.1)



Example for data exchange

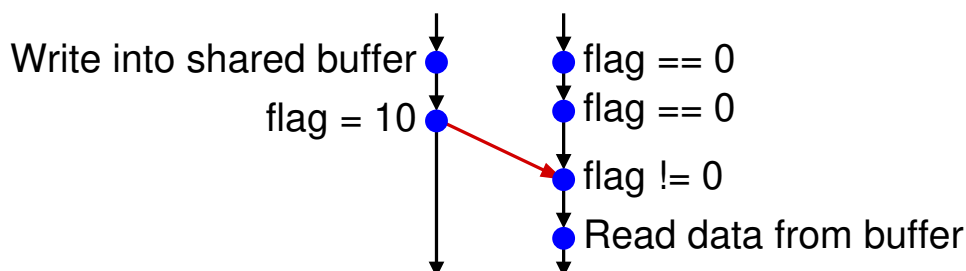
Producer Thread

```
for (i=0; i<size; i++)  
    buffer[i] = produce();  
flag = size;
```

Consumer Thread

```
while(flag==0);  
for (i=0; i<flag; i++)  
    consume(buffer[i]);
```

Execution Sequence:



2.5.2 Message Passing



- ➔ Processes with separate address spaces
- ➔ Library routines for sending and receiving messages
 - ➔ (informal) standard for parallel programming:
MPI (*Message Passing Interface*, ↗ 4.2)
- ➔ Mostly with distributed memory computers
 - ➔ but also well usable with shared memory computers
- ➔ The “more complicated” parallel programming model
 - ➔ explicit data distribution / explicit data transfer
 - ➔ typically no compiler and/or language support
 - ➔ parallelisation is done completely manually



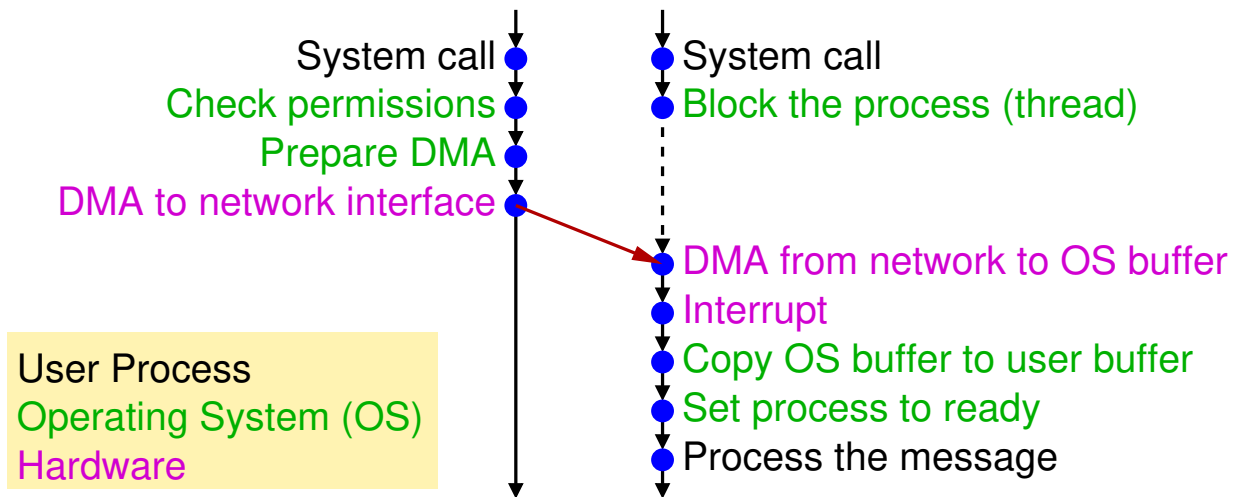
Example for data exchange

Producer Process

```
send(receiver,  
      &buffer, size);
```

Consumer Process

```
receive(&buffer,  
       buffer_length);
```



2.5.3 Distributed Objects



- ➔ Basis: (purely) object oriented programming
 - ➔ access to data **only** via method calls
- ➔ Then: objects can be distributed to different address spaces (computers)
 - ➔ at object creation: additional specification of a node
 - ➔ object reference then also identifies this node
 - ➔ method calls via RPC mechanism
 - ➔ e.g., *Remote Method Invocation* (RMI) in Java
 - ➔ more about this: lecture "Distributed Systems"
- ➔ Distributed objects alone do not yet enable parallel processing
 - ➔ additional concepts / extensions are necessary
 - ➔ e.g., threads, asynchronous RPC, futures

Notes for slide 124:

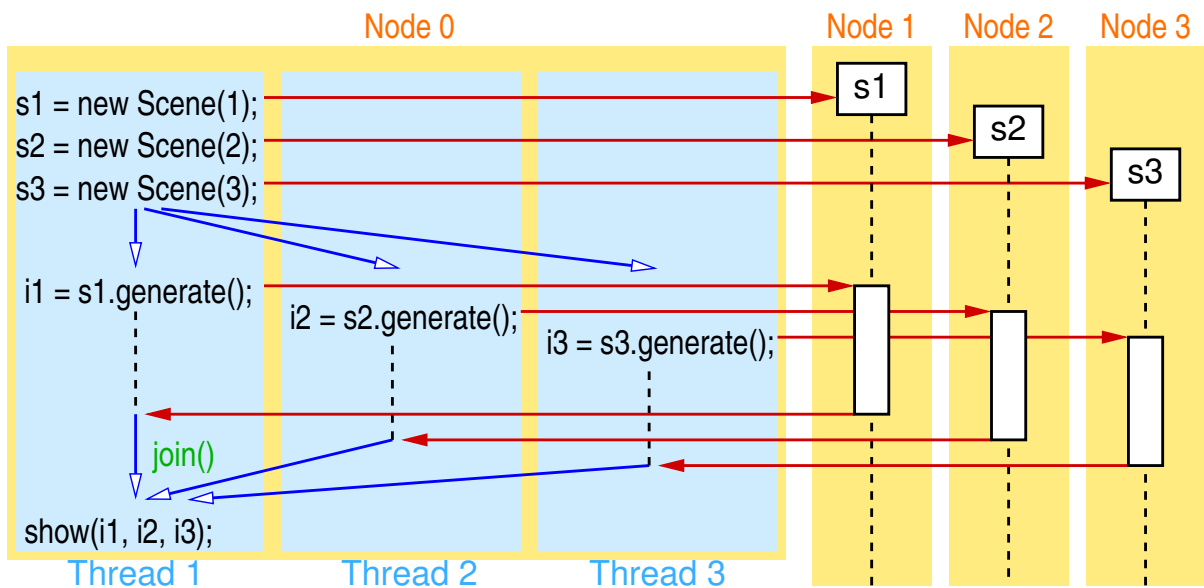
Example

- ➔ Class Scene as description of a scene
 - ➔ constructor Scene(int param)
 - ➔ method Image generate() computes the image
- ➔ Computation of three images with different parameters (sequentially):

```
Scene s1 = new Scene(1);  
Scene s2 = new Scene(2);  
Scene s3 = new Scene(3);  
Image i1 = s1.generate();  
Image i2 = s2.generate();  
Image i3 = s3.generate();  
show(i1, i2, i3);
```

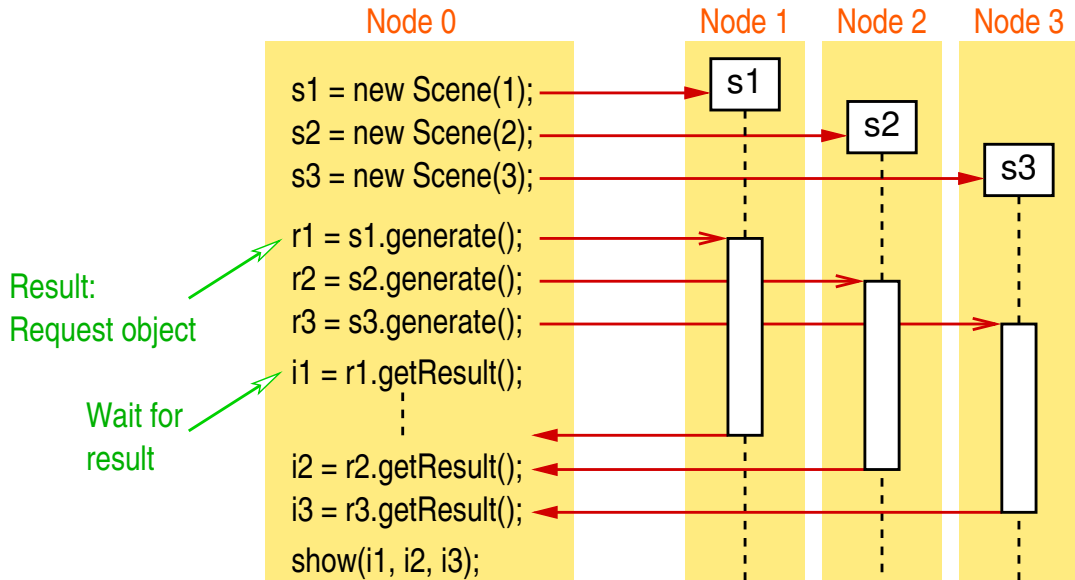
124-1

Parallel computing with threads



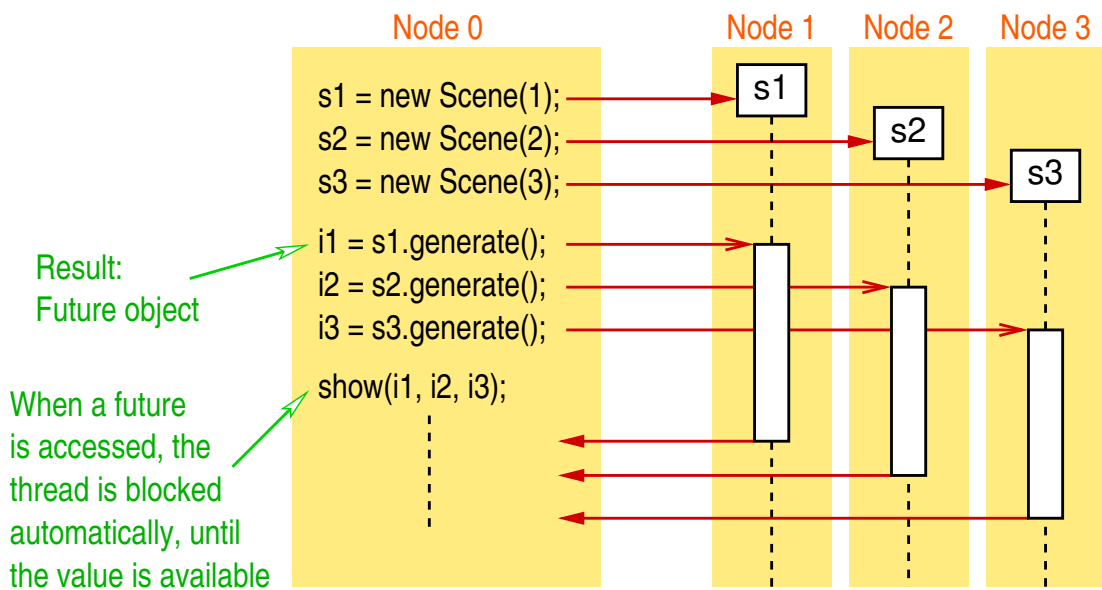
124-2

Parallel computing with asynchronous RPC



124-3

Parallel computing with futures



124-4

2.5.4 Data Parallel Languages



- ➔ Goal: support for data parallelism
- ➔ Sequential code is amended with compiler directives
 - Specification, how to distribute data structures (typically arrays) to processors
- ➔ Compiler automatically generates code for synchronisation or communication, respectively
 - operations are executed on the processor that “possesses” the result variable (*owner computes* rule)
- ➔ Example: HPF (*High Performance Fortran*)
- ➔ Despite easy programming not really successful
 - only suited for a limited class of applications
 - good performance requires a lot of manual optimization

2.5.4 Data Parallel Languages ...

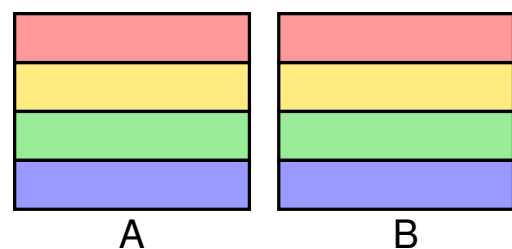


(Animated slide)

Example for HPF

```
REAL A(N,N), B(N,N)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(:, :) WITH A(:, :)
DO I = 1, N
  DO J = 1, N
    A(I,J) = A(I,J) + B(J,I)
  END DO
END DO
```

Distribution with 4 processors:



- ➔ Processor 0 executes computations for $I = 1 .. N/4$
- ➔ Problem in this example: a lot of communication is required
 - B should be distributed in a different way

Notes for slide 126:

For ease of understanding, the example assumes that the matrix is stored in row-major order (i.e., the layout in main memory is row 0, row 1, ...), as it is used by C and C++.

However, Fortran actually stores arrays in column-major order (i.e., the layout in main memory is column 0, column 1, ...). This means that actually A should be distributed with

```
!HPF$ DISTRIBUTE A(*,BLOCK)
```

and the I and J loops should be interchanged.

126-1

2.6 Focus of this Lecture



- ➔ Explicit parallelism
- ➔ Process and block level
- ➔ Coarse and mid grained parallelism

- ➔ MIMD computers (with SIMD extensions)

- ➔ Programming models:
 - ➔ shared memory
 - ➔ message passing

2.7 Organisation Forms for Parallel Programs



- ➔ Models / patterns for parallel programs

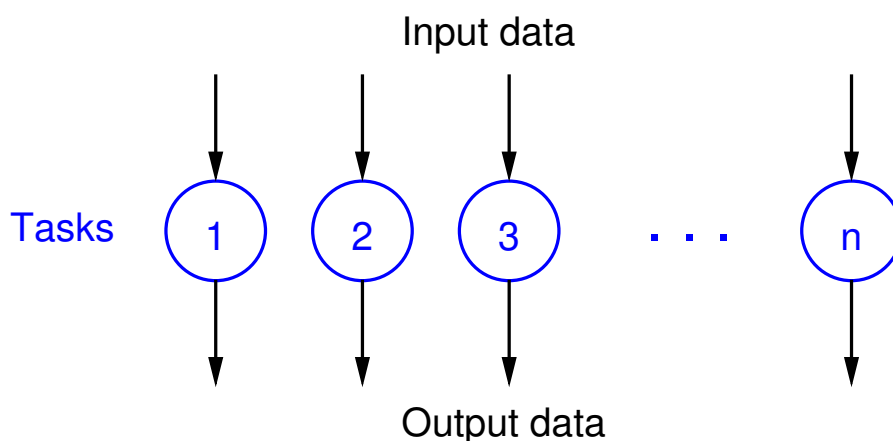
2.7.1 Embarrassingly Parallel

- ➔ The task to be solved can be divided into a set of **completely independent** sub-tasks
- ➔ All sub-tasks can be solved in parallel
- ➔ No data exchange (communication) is necessary between the parallel threads / processes
- ➔ Ideal situation!
 - ➔ when using n processors, the task will (usually) be solved n times faster
 - ➔ (for reflection: why only usually?)

2.7.1 Embarrassingly Parallel ...



Illustration



Parallel Processing

Winter Term 2025/26

28.10.2025

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

Stand: January 8, 2026

2.7.1 Embarrassingly Parallel ...



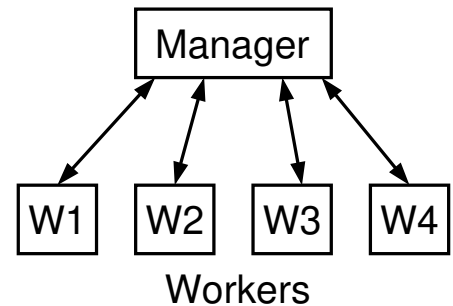
Examples for embarrassingly parallel problems

- ➔ Computation of animations
 - ➔ 3D visualizations, animated cartoons, motion pictures, ...
 - ➔ each image (frame) can be computed independently
- ➔ Parameter studies
 - ➔ multiple / many simulations with different input parameters
 - ➔ e.g., weather forecast with provision for measurement errors, computational fluid dynamics for optimizing an airfoil, ...



2.7.2 Manager/Worker Model (Master/Slave Model)

- ➔ A manager process creates independent tasks and assigns them to worker processes
 - several managers are possible, too
 - a hierarchy is possible, too: a worker can itself be the manager of own workers



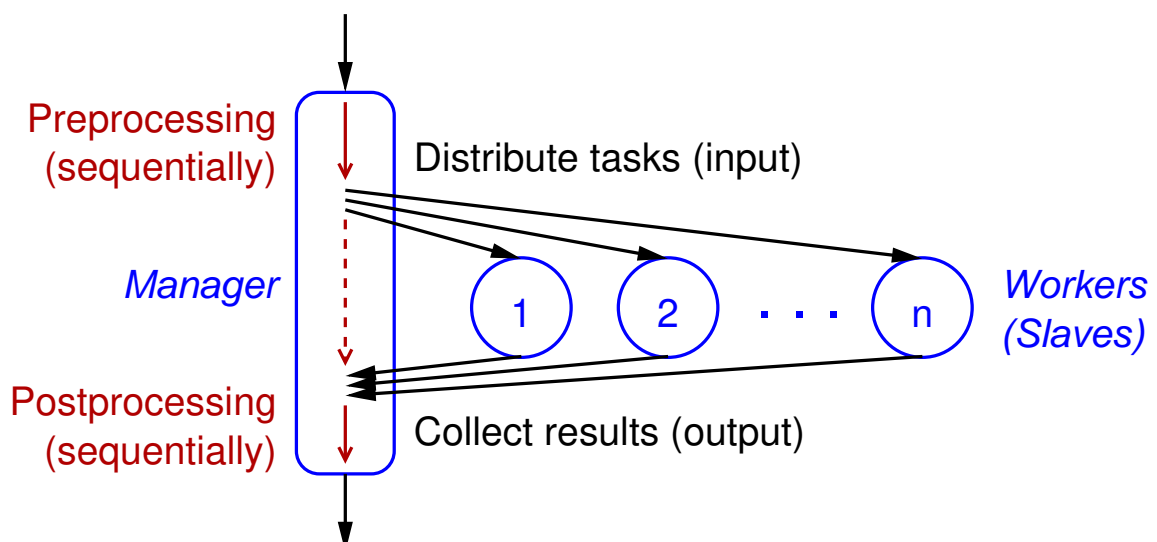
- ➔ The manager (or sometimes also the workers) can create additional tasks, while the workers are working
- ➔ The manager can become a bottleneck
- ➔ The manager should be able to receive the results asynchronously (non blocking)

2.7.2 Manager/Worker Model (Master/Slave Model) ...



Typical application

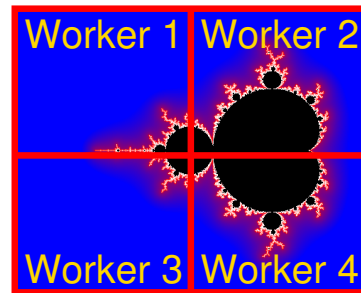
- ➔ Often only a part of a task can be parallelised in an optimal way
- ➔ In the easiest case, the following flow will result:



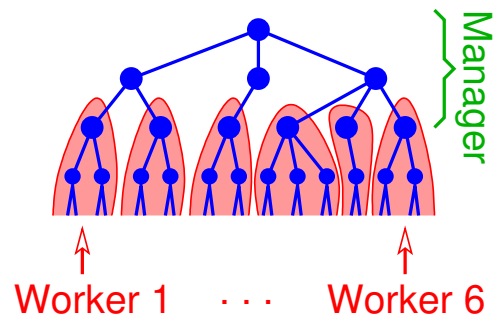


Examples

- ➔ Image creation and processing
 - manager partitions the image into areas; each area is processed by one worker



- ➔ Tree search
 - manager traverses the tree up to a predefined depth; the workers process the sub-trees

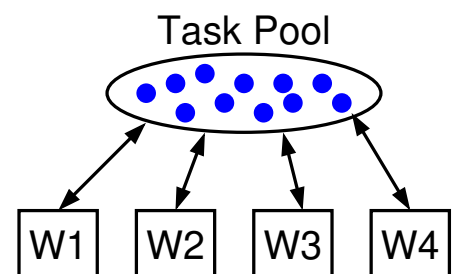


2.7 Organisation Forms for Parallel Programs ...



2.7.3 Work Pool Model (Task Pool Model)

- ➔ Tasks are explicitly specified using a data structure
 - input data + task description, if necessary
- ➔ Centralized or distributed pool (list) of tasks
 - workers (threads or processes) fetch tasks from the pool
 - usually much more tasks than workers
 - good load balancing is possible
 - accesses must be synchronised
- ➔ Workers can put new tasks into the pool, if need be
 - e.g., with divide-and-conquer



Notes for slide 134:

- ➔ The work pool model is somehow similar to the master/worker model. However, the work pool model is more natural when using a shared memory programming model, while the master/worker model is suited for a message passing programming model.

134-1

2.7 Organisation Forms for Parallel Programs ...



2.7.4 Divide and Conquer

- ➔ **Recursive** partitioning of the task into independent sub-tasks
- ➔ Tasks dynamically create new sub-tasks
- ➔ Problem: limiting the number of tasks
 - ➔ esp. if tasks are directly implemented by threads / processes
- ➔ Solutions:
 - ➔ create a new sub-task only, if its size is larger than some minimum
 - ➔ maintain a task pool, which is executed by a fixed number of threads

2.7.4 Divide and Conquer ...



(Animated slide)

Example: parallel quicksort

$Qsort(A_1 .. n)$

If $n = 1$: done.

Else:

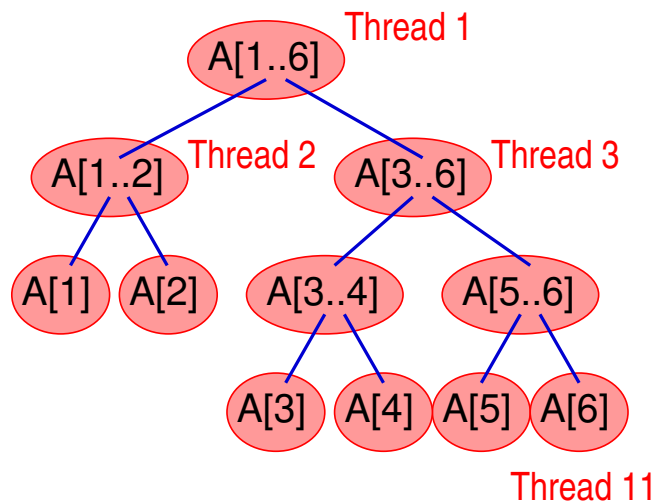
Determine the *pivot* S .

Reorder A such that

$A_i \leq S$ for $i \in [1, k[$ and
 $A_i \geq S$ for $i \in [k, n]$.

Execute $Qsort(A_1 .. k-1)$
and $Qsort(A_k .. n)$
in parallel.

Example:



2.7.4 Divide and Conquer ...



(Animated slide)

Example: parallel quicksort

$Qsort(A_1 .. n)$

If $n = 1$: done.

Else:

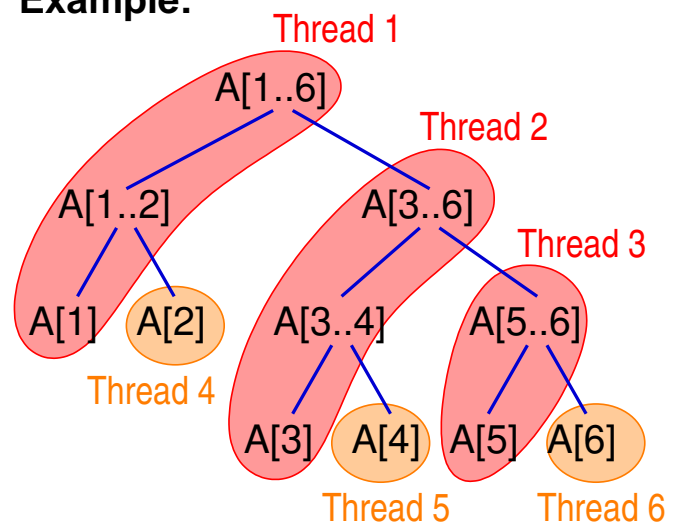
Determine the *pivot* S .

Reorder A such that

$A_i \leq S$ for $i \in [1, k[$ and
 $A_i \geq S$ for $i \in [k, n]$.

Execute $Qsort(A_1 .. k-1)$
and $Qsort(A_k .. n)$
in parallel.

Example:



* Assumption: thread executes first call itself
and creates new thread for the second one

2.7.4 Divide and Conquer ...



(Animated slide)

Example: parallel quicksort

Qsort($A_1 .. n$)

If $n = 1$: done.

Else:

Determine the *pivot* S .

Reorder A such that

$A_i \leq S$ for $i \in [1, k[$ and

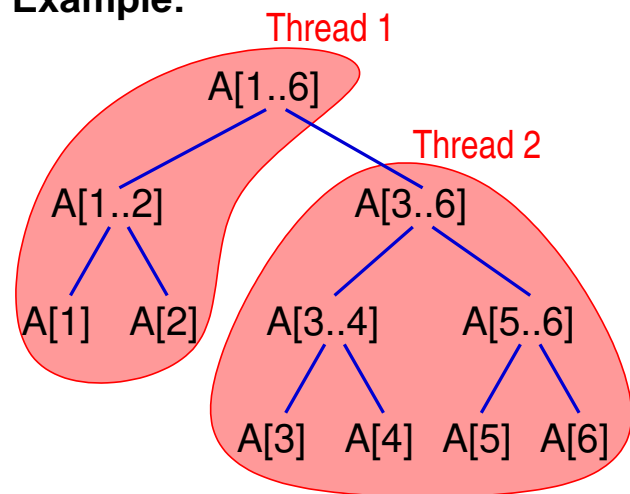
$A_i \geq S$ for $i \in [k, n]$.

Execute Qsort($A_1 .. k-1$)

and Qsort($A_k .. n$)

in parallel.

Example:



* Additional Assumption: new thread is created only if array length > 2

2.7 Organisation Forms for Parallel Programs ...

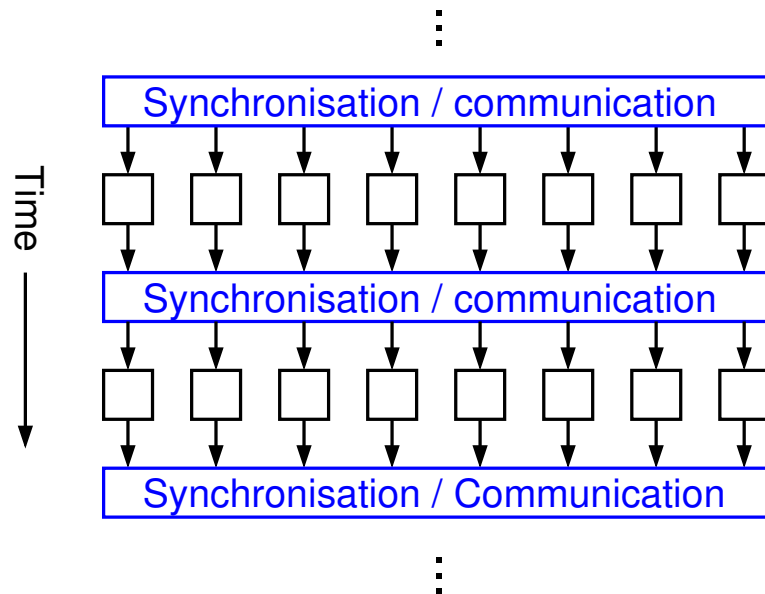


2.7.5 Data parallel Model: SPMD

- ➔ Fixed, constant number of processes (or threads, respectively)
- ➔ One-to-one correspondence between tasks and processes
- ➔ All processes execute the same program code
 - ➔ however: conditional statements are possible ...
- ➔ For program parts which cannot be parallelised:
 - ➔ replicated execution in each process
 - ➔ execution in only one process; the other ones wait
- ➔ Usually loosely synchronous execution:
 - ➔ alternating phases with independent computations and communication / synchronisation



Typical sequence

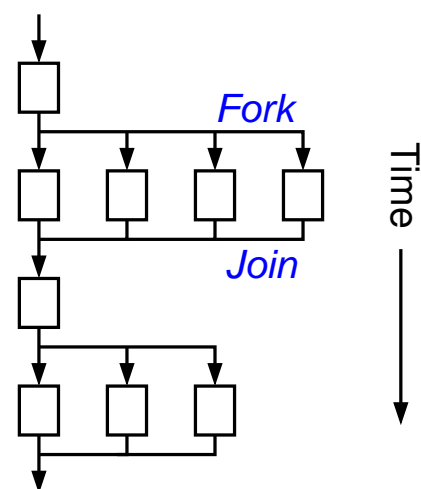


2.7 Organisation Forms for Parallel Programs ...



2.7.6 Fork/Join Model

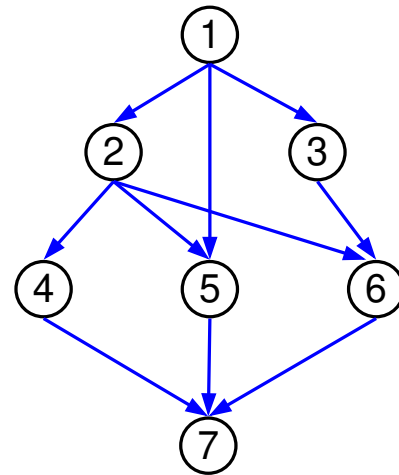
- ➔ Program consists of sequential and parallel phases
- ➔ Thread (or processes, resp.) for parallel phases are created at run-time (*fork*)
 - ➔ one for each task
- ➔ At the end of each parallel phase: synchronisation and termination of the threads (*join*)





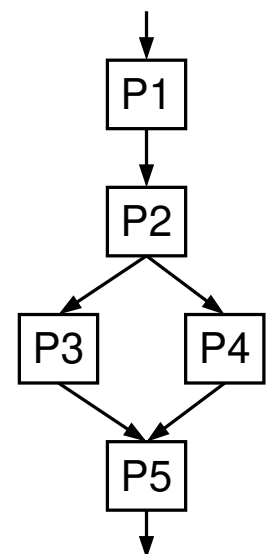
2.7.7 Task-Graph Model

- ➔ Tasks and their dependences (data flow) are represented as a graph
- ➔ An edge in the graph denotes a data flow
 - e.g., task 1 produces data, task 2 starts execution, when this data is entirely available
- ➔ Assignment of tasks to processors usually in such a way, that the necessary amount of communication is as small as possible
 - e.g., tasks 1, 5, and 7 in one process



2.7.8 Pipeline Model

- ➔ A *stream* of data elements is directed through a sequence of processes
- ➔ The execution of a task starts as soon as a data element arrives
- ➔ Pipeline needs not necessarily be linear
 - general (acyclic) graphs are possible, as with the task-graph model
- ➔ Producer/consumer synchronisation between the processes



2.8 A Design Process for Parallel Programs

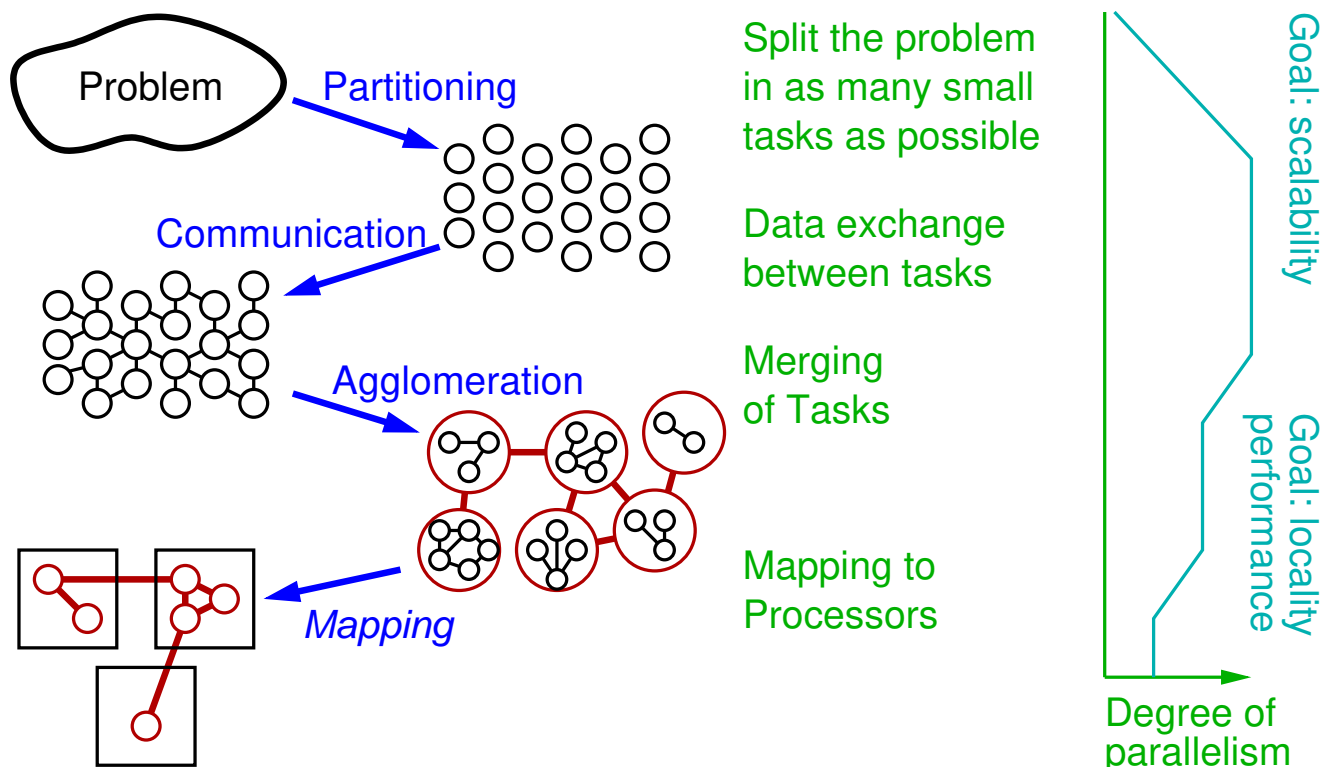


Four design steps:

1. Partitioning
 - ➔ split the problem into many tasks
2. Communication
 - ➔ specify the information flow between the tasks
 - ➔ determine the communication structure
3. Agglomeration
 - ➔ evaluate the performance (tasks, communication structure)
 - ➔ if need be, aggregate tasks into larger tasks
4. Mapping
 - ➔ map the tasks to processors

(See Foster: *Designing and Building Parallel Programs*, Ch. 2)

2.8 A Design Process for Parallel Programs ...



2.8.1 Partitioning



- ➔ Goal: split the problem into as many small tasks as possible

Data partitioning (data parallelism)

- ➔ Tasks specify **identical computaions** for a **part** of the data
- ➔ In general, high degree of parallelism is possible
- ➔ We can distribute:
 - input data
 - output data
 - intermediate data
- ➔ In some cases: recursive partitioning (*divide and conquer*)
- ➔ Special case: partitioning of search space in search problems

2.8.1 Partitioning ...



Example: matrix multiplication

- ➔ Product $C = A \cdot B$ of two square matrices

- $$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}, \text{ for all } i, j = 1 \dots n$$

- ➔ This formula also holds when square sub-matrices A_{ik}, B_{kj}, C_{ij} are considered instead of single scalar elements

- block matrix algorithms:

$$\begin{array}{|c|c|} \hline A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \\ \hline \end{array} = \begin{array}{|c|c|} \hline C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \\ \hline \end{array} \quad \begin{array}{l} C_{1,1} = A_{1,1} \cdot B_{1,1} \\ \quad + A_{1,2} \cdot B_{2,1} \end{array}$$

2.8.1 Partitioning ...



Example: matrix multiplication ...

➔ Distribution of output data: each task computes a sub-matrix of C

➔ E.g., distribution of C into four sub-matrices

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

➔ Results in four independent tasks:

1. $C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$
2. $C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$
3. $C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$
4. $C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$

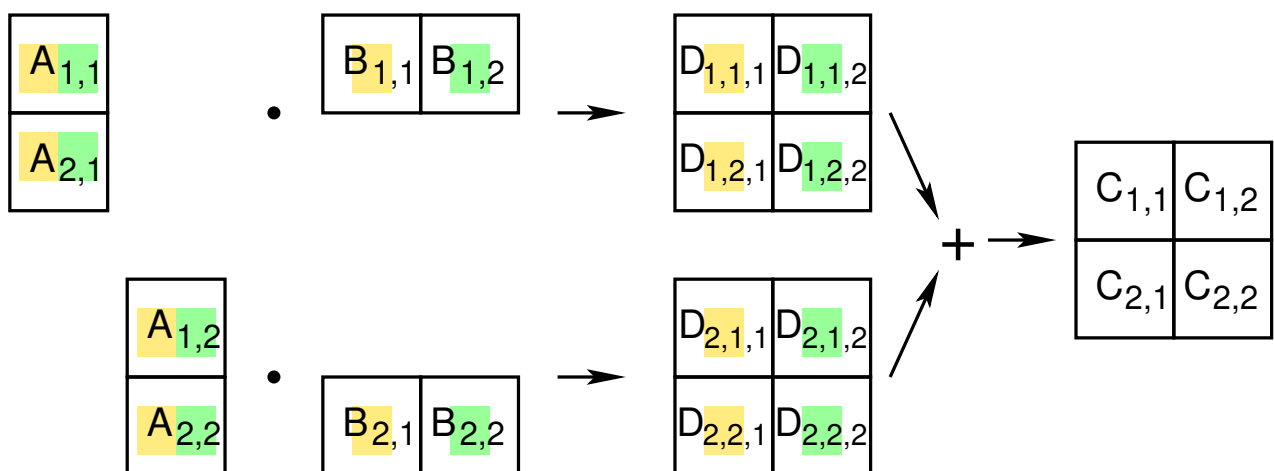
2.8.1 Partitioning ...



Example: matrix multiplication $A \cdot B \rightarrow C$

➔ Distribution of intermediate data (higher degree of parallelism)

➔ here: 8 multiplications of sub-matrices

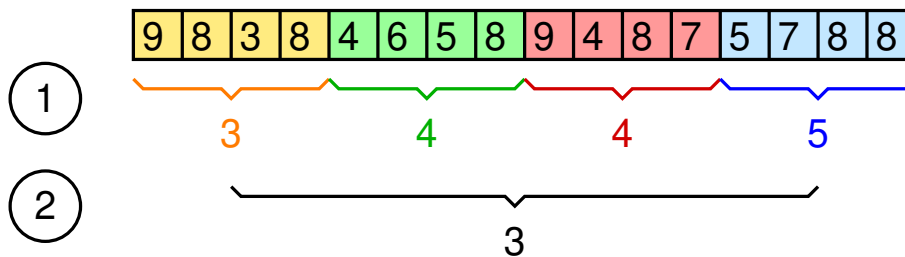


2.8.1 Partitioning ...



Example: minimum of an array

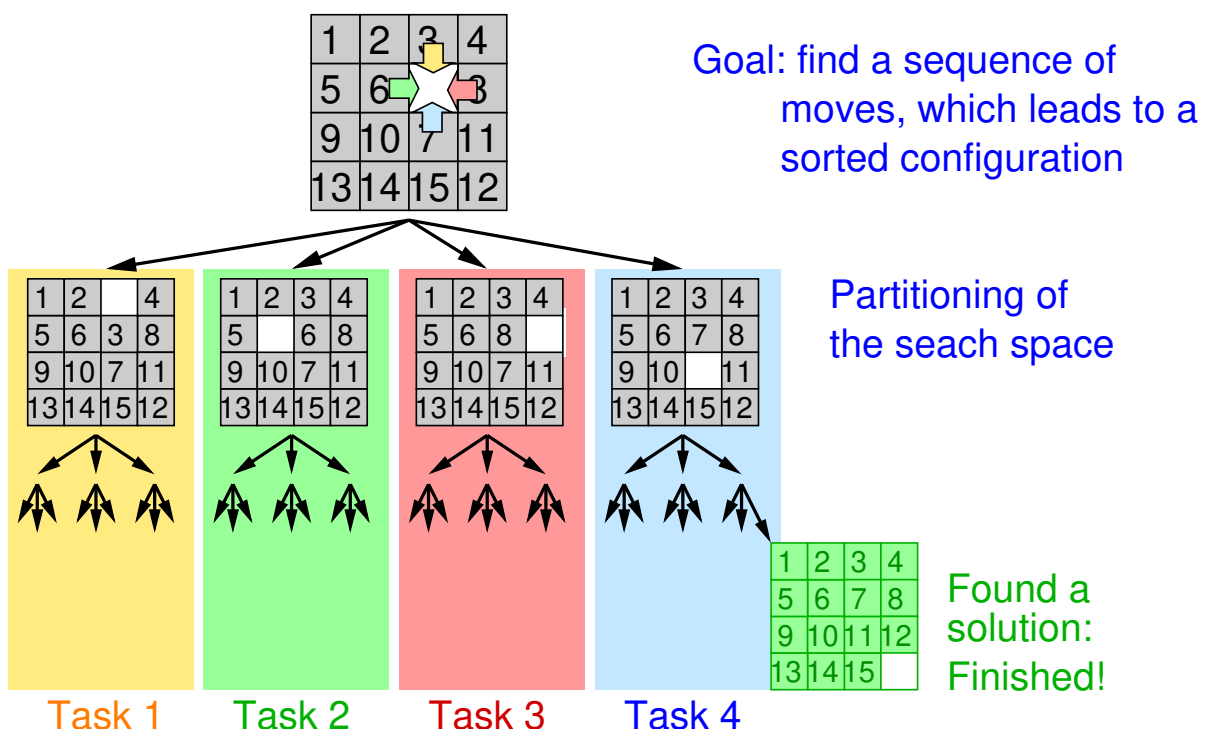
- ➔ Distribution of input data
 - each threads computates its local minimum
 - afterwards: computation of the global minimum



2.8.1 Partitioning ...



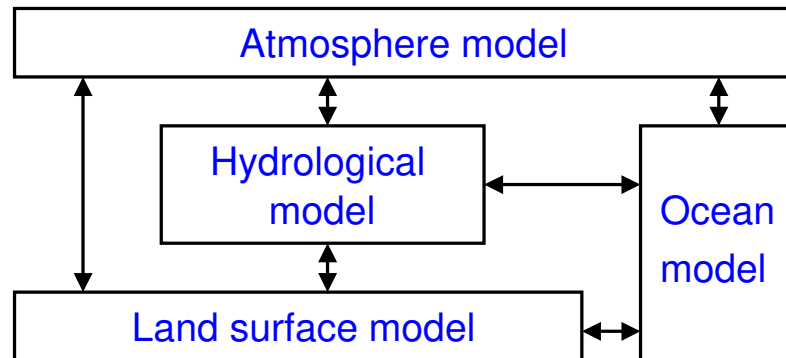
Example: sliding puzzle (partitioning of search space)





Task partitioning (task parallelism)

- ➔ Tasks are **different** sub-problems (execution steps) of a problem
- ➔ E.g., climate model



- ➔ Tasks can work concurrently or in a pipeline
- ➔ max. gain: number of sub-problems (typically small)
- ➔ often in addition to data partitioning

2.8.2 Communication



- ➔ Two step approach
 - definition of the communication structure
 - who must exchange data with whom?
 - sometimes complex when using data partitioning
 - often simple when using task partitioning
 - definition of the messages to be sent
 - which data must be exchanged when?
 - taking data dependences into account

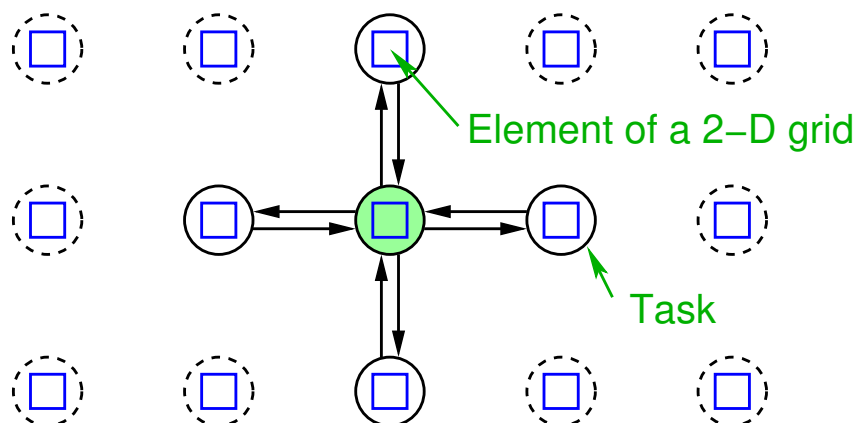


Different communication patterns:

- ➔ Local vs. global communication
 - lokal: task communicates only with a small set of other tasks (its “neighbors”)
 - global: task communicates with many/all other tasks
- ➔ Structured vs. unstructured communication
 - structured: regular structure, e.g., grid, tree
- ➔ Static vs. dynamic communication
 - dynamic: communication structure is changing during run-time, depending on computed data
- ➔ Synchronous vs. asynchronous communication
 - asynchronous: the task owning the data does not know, when other tasks need to access it

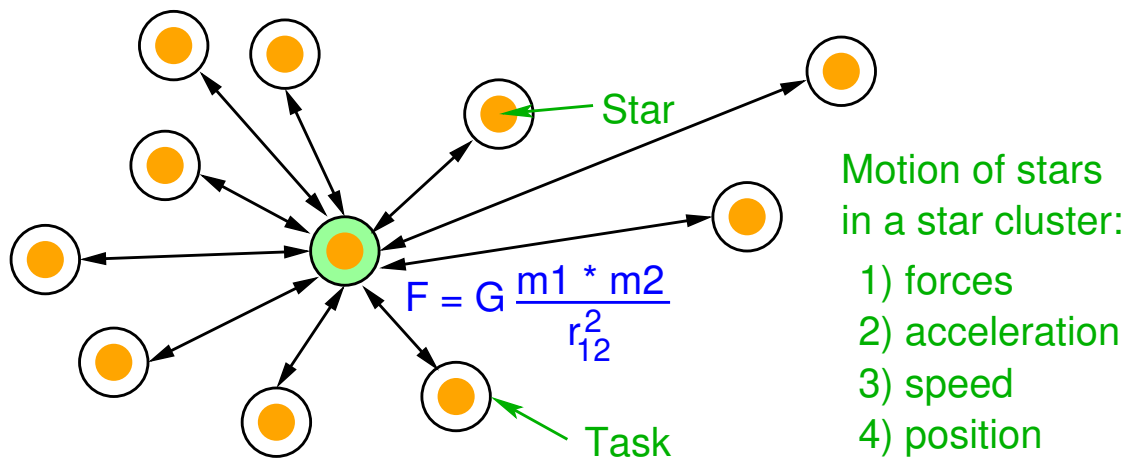


Example for local communication: stencil algorithms



- ➔ Here: 5-point stencil (also others are possible)
- ➔ Examples: Jacobi or Gauss-Seidel methods, filters for image processing, ...

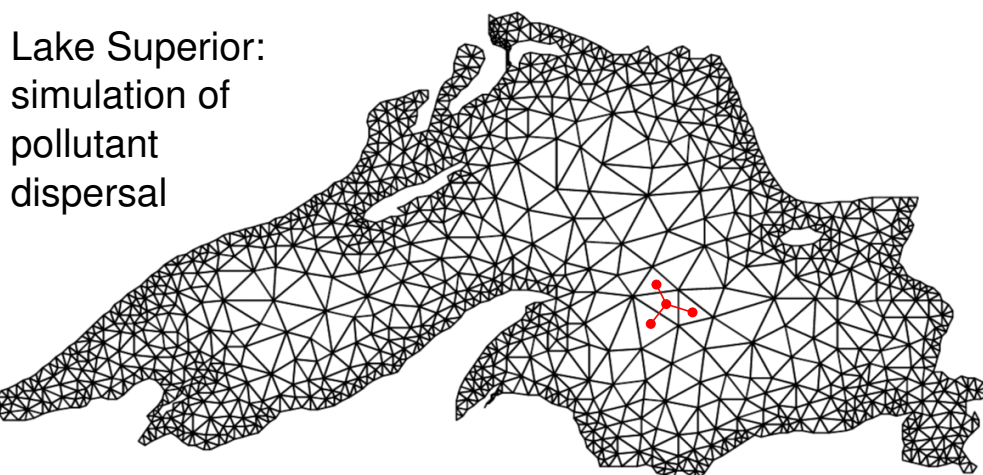
Example for global communication: N-body problem



- ➔ The effective force on a star in a star cluster depends on the masses and locations of all other stars
 - ➔ possible approximation: restriction to relatively close stars
 - ➔ will, however, result in dynamic communication

Example for structured / unstructured communication

- ➔ Structured: stencil algorithms
- ➔ Unstructured: “unstructured grids”



- ➔ grid points are defined at different density
- ➔ edges: neighborhood relation (communication)

2.8.3 Agglomeration



- ➔ So far: abstract parallel algorithms
- ➔ Now: concrete formulation for real computers
 - limited number of processors
 - costs for communication, process creation, process switching, ...
- ➔ Goals:
 - reducing the communication costs
 - aggregation of tasks
 - replication of data and/or computation
 - retaining the flexibility
 - sufficiently fine-grained parallelism for mapping phase

2.8.4 Mapping

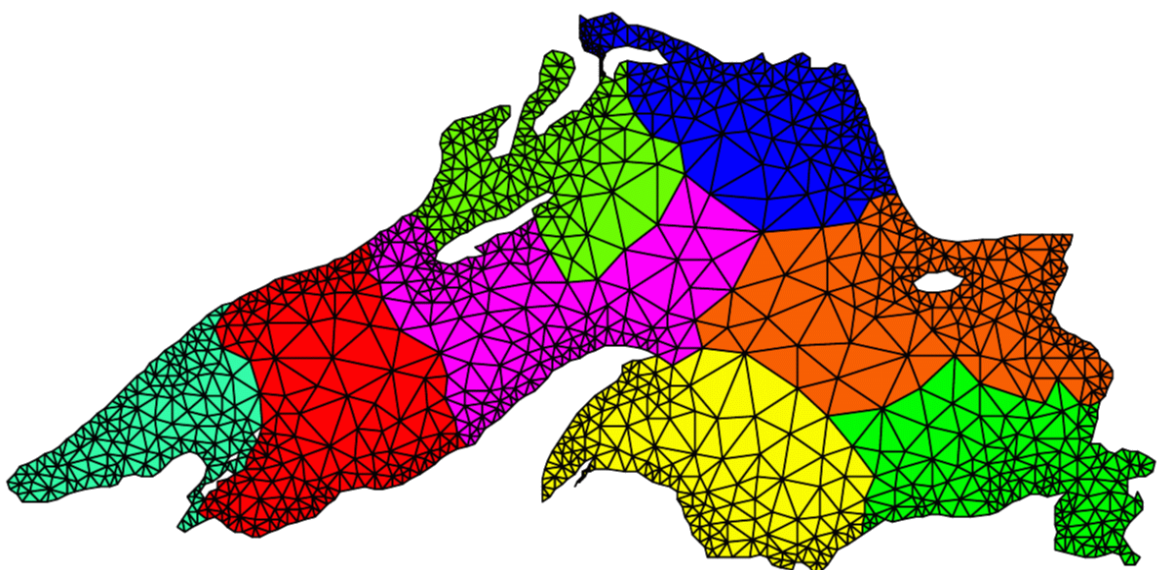


- ➔ Task: assignment of tasks to available processors
- ➔ Goal: minimizing the execution time
- ➔ Two (conflicting) strategies:
 - map concurrently executable tasks to different processors
 - high degree of parallelism
 - map communicating tasks to the same processor
 - higher locality (less communication)
- ➔ Constraint: load balancing
 - (roughly) the same computing effort for each processor
- ➔ The mapping problem is NP complete

Variants of mapping techniques

- ➔ Static mapping
 - ➔ fixed assignment of tasks to processors when program is started
 - ➔ for algorithms on arrays or Cartesian grids:
 - ➔ often manually, e.g., block wise or cyclic distribution
 - ➔ for unstructured grids:
 - ➔ graph partitioning algorithms, e.g., greedy, recursive coordinate bisection, recursive spectral bisection, ...
- ➔ Dynamic mapping (dynamic load balancing)
 - ➔ assignment of tasks to processors at runtime
 - ➔ variants:
 - ➔ tasks stay on their processor until their execution ends
 - ➔ task migration is possible during runtime

Example: static mapping with unstructured grid



- ➔ (Roughly) the same number of grid points per processor
- ➔ Short boundaries: small amount of communication

Parallel Processing

Winter Term 2025/26

03.11.2025

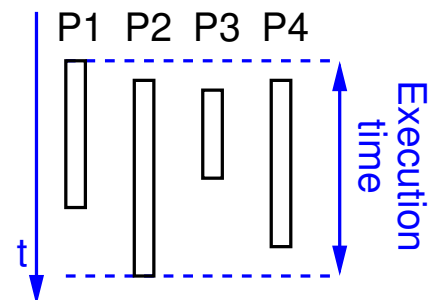
Roland Wismüller
 Universität Siegen
 roland.wismueller@uni-siegen.de
 Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 8, 2026

2.9 Performance Considerations



- ➔ Which performance gain results from the parallelisation?
- ➔ Possible performance metrics:
 - ➔ execution time, throughput, memory requirements, processor utilisation, development cost, maintenance cost, ...
- ➔ In the following, we consider execution time
 - ➔ **execution time** of a parallel program: time between the start of the program and the end of the computation on the last processor





Speedup (*Beschleunigung*)

➔ Reduction of execution time due to parallel execution

➔ **Absolute speedup**

$$S(p) = \frac{T_s}{T(p)}$$

- ➔ T_s = execution time of the sequential program (or the best sequential algorithm, respectively)
- ➔ $T(p)$ = execution time of the parallel program (algorithm) with p processors



Speedup (*Beschleunigung*) ...

➔ **Relative speedup** (for “sugarcoated” results ...)

$$S(p) = \frac{T(1)}{T(p)}$$

- ➔ $T(1)$ = execution time of the parallel program (algorithm) with one processor
- ➔ Optimum: $S(p) = p$
- ➔ Often: with **fixed** problem size, $S(p)$ declines again, when p increases
 - ➔ more communication, less computing work per processor

Notes for slide 164:

Sometimes, the relative speedup is computed with respect to the parallel execution on a small number of processors. This is necessary, when the problem cannot be solved sequentially, e.g., due to time or memory constraints. In such cases, the absolute speedup cannot be determined.

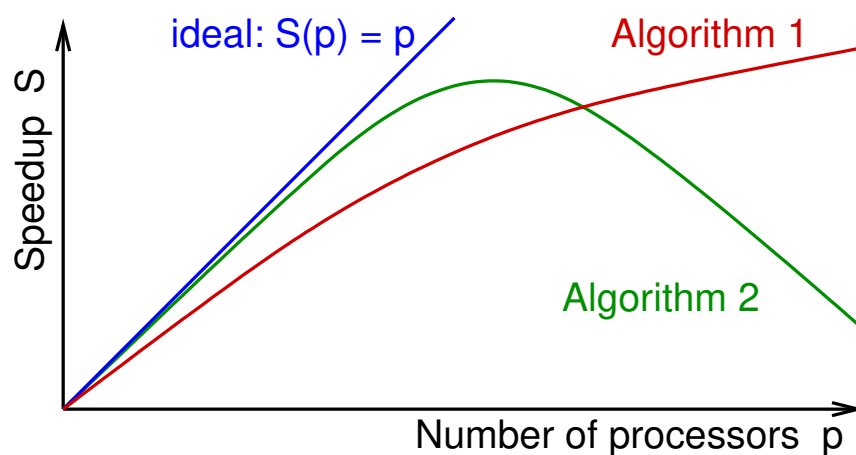
164-1

2.9.1 Performance Metrics ...



Speedup (*Beschleunigung*) ...

➔ Typical trends:



➔ Statements like “speedup of 7.5 with 8 processors” can not be extrapolated to a larger number of processors

Amdahl's Law

- ➔ Defines an upper limit for the achievable speedup
- ➔ Basis: usually, not all parts of a program can be parallelized
 - ➔ due to the programming effort
 - ➔ due to data dependences
- ➔ Let a be the **portion of time** of these program parts in the **sequential** version of the program. Then:

$$S(p) = \frac{T_s}{T(p)} \leq \frac{1}{a + (1 - a)/p} \leq \frac{1}{a}$$

- ➔ With a 10% sequential portion, this leads to $S(p) \leq 10$

Notes for slide 166:

If a portion a of the sequential execution time is not parallelizable, then the parallel execution time in the best case is

$$T(p) = a \cdot T_s + (1 - a) \cdot \frac{T_s}{p}$$

Thus

$$S(p) = \frac{T_s}{T(p)} \leq \frac{T_s}{a \cdot T_s + (1 - a) \cdot \frac{T_s}{p}} = \frac{1}{a + (1 - a)/p}$$



Superlinear speedup

- ➔ Sometimes we observe $S(p) > p$, although this should actually be impossible
- ➔ Causes:
 - ➔ cache effects
 - ➔ with p processors, the amount of cache is p times higher than that with one processor
 - ➔ thus, we also have higher cache hit rates
 - ➔ implicit change in the algorithm
 - ➔ e.g., with parallel tree search: several paths in the search tree are traversed simultaneously
 - ➔ limited breadth-first search instead of depth-first search



Efficiency

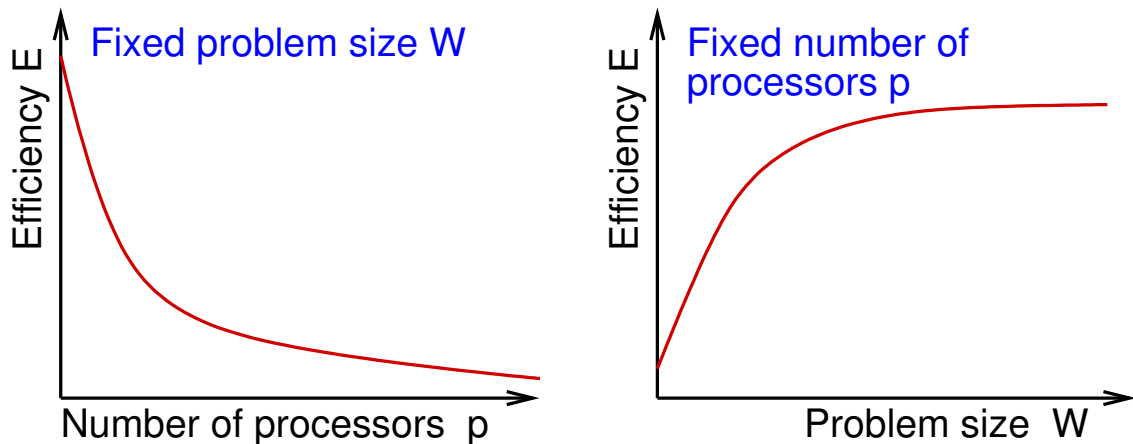
$$E(p) = \frac{S(p)}{p}$$

- ➔ Metrics for the utilisation of a parallel computer
- ➔ $E(p) \leq 1$, the optimum would be $E(p) = 1$



Scalability

➔ Typical observations:



➔ Reason: with increasing p : less work per processor, but the same amount of (or even more) communication



Scalability ...

➔ How must the problem size W increase with increasing number of processors p , such that the efficiency stays the same?

➔ Answer is given by the **isoefficiency function**

➔ Parallel execution time

$$T(p) = \frac{W + T_o(W, p)}{p}$$

➔ $T_o(W, p)$ = overhead of parallel execution

➔ T and W are measured as the number of elementary operations

➔ Thus:

$$W = \frac{E(p)}{1 - E(p)} \cdot T_o(W, p)$$

Notes for slide 170:

With

$$T(p) = \frac{W + T_o(W, p)}{p}$$

we get

$$S(p) = \frac{W}{T(p)} = \frac{W \cdot p}{W + T_o(W, p)}$$

and

$$E(p) = \frac{S(p)}{p} = \frac{W}{W + T_o(W, p)} = \frac{1}{1 + T_o(W, p)/W}$$

Thus:

$$\frac{T_o(W, p)}{W} = \frac{1 - E(p)}{E(p)}$$

and

$$W = \frac{E(p)}{1 - E(p)} \cdot T_o(W, p)$$

170-1

2.9.1 Performance Metrics ...



Scalability ...

- ➔ Isoefficiency function $I(p)$
 - solution of the equation $W = K \cdot T_o(W, p)$ w.r.t. W
 - $K = \text{constant}$, depending on the required efficiency
- ➔ Good scalability: $I(p) = \mathcal{O}(p)$ or $I(p) = \mathcal{O}(p \log p)$
- ➔ Bad scalability: $I(p) = \mathcal{O}(p^k)$ for $k > 1$
- ➔ $T_o(W, p)$ is determined by analysing the parallel algorithm
 - how much time is needed for communication / synchronisation and potentially additional computations?
 - more details and examples in chapter 2.9.5

2.9.2 Reasons for Performance Loss



- ➔ **Access losses** due to data exchange between tasks
 - ➔ e.g., message passing, remote memory access
- ➔ **Conflict losses** due to shared use of resources by multiple tasks
 - ➔ e.g., conflicts when accessing the network, mutual exclusion when accessing data
- ➔ **Utilisation losses** due to insufficient degree of parallelism
 - ➔ e.g., waiting for data, load imbalance
- ➔ **Complexity losses** due to additional work necessary for the parallel execution
 - ➔ e.g., partitioning of unstructured grids

2.9.2 Reasons for Performance Loss ...

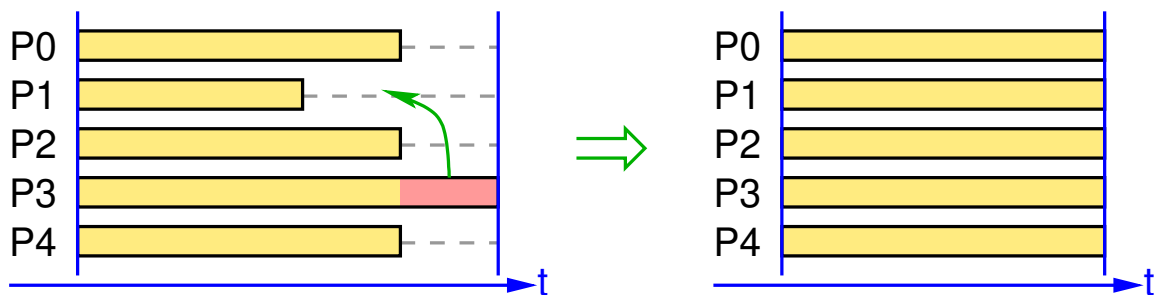


- ➔ **Algorithmic losses** due to modifications of the algorithms during the parallelisation
 - ➔ e.g., worse convergence of an iterative method
- ➔ **Dumping losses** due to computations, which are executed redundantly but not used later on
 - ➔ e.g., lapsed search in branch-and-bound algorithms
- ➔ **Breaking losses** when computations should end
 - ➔ e.g., with search problems: all other processes must be notified that a solution has been found



Introduction

- ➔ For optimal performance: processors should compute equally long between two (global) synchronisations
- synchronisation: includes messages and program start / end



- ➔ Load in this context: execution time between two synchronisations
- other load metrics are possible, e.g., communication load
- ➔ Load balancing is one of the goals of the mapping phase

2.9.3 Load Balancing ...



Reasons for load imbalance

- ➔ Unequal computational load of the tasks
 - e.g., atmospheric model: areas over land / water
- ➔ Heterogeneous execution platform
 - e.g., processors (cores) with different performance
- ➔ Computational load of the tasks changes dynamically
 - e.g., in atmospheric model, depending on the simulated time of day (solar radiation)
- ➔ Background load on the processors
 - e.g., in a PC cluster

static

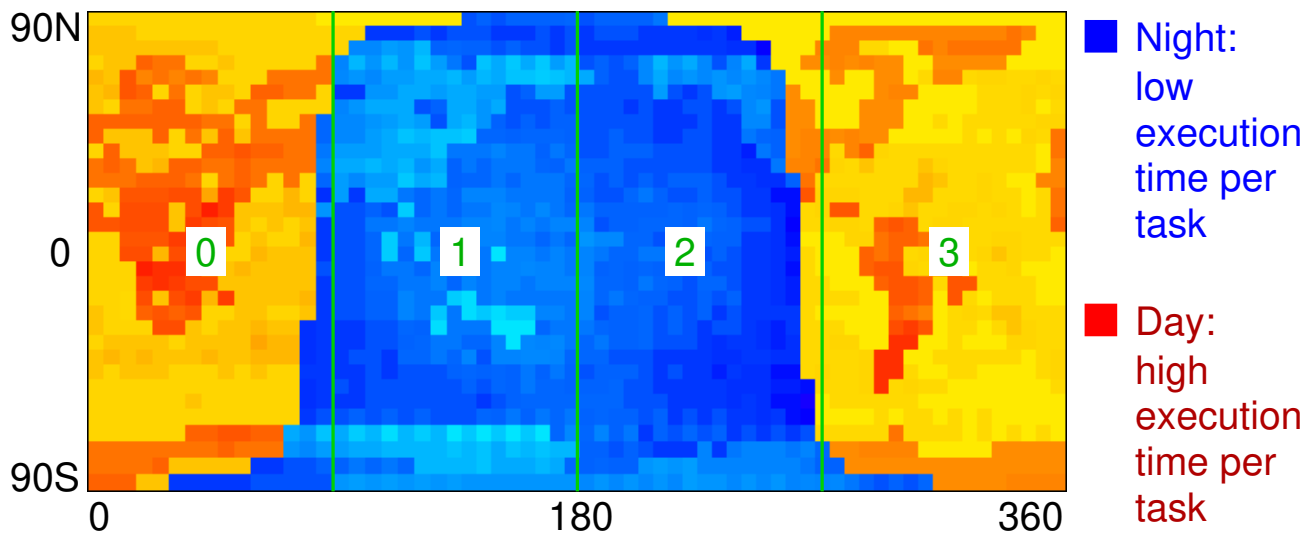
dynamic

2.9.3 Load Balancing ...



(Animated slide)

Example: atmospheric model



- ➔ Continents: static load imbalance
- ➔ Border between day and night: dynamic load imbalance

2.9.3 Load Balancing ...



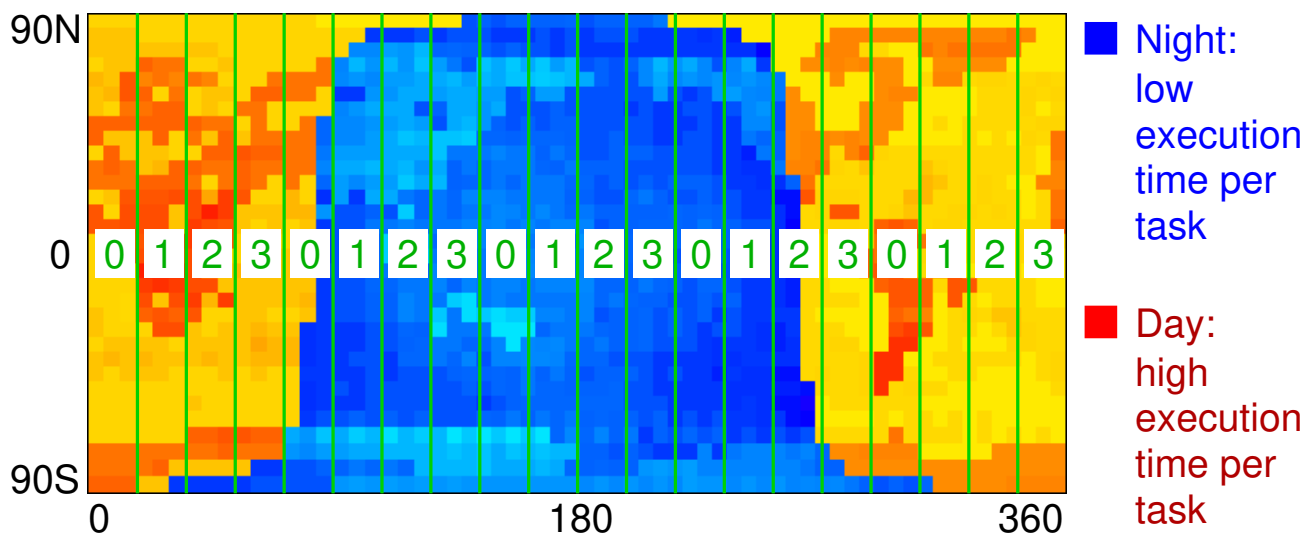
Static load balancing

- ➔ Goal: distribute the tasks to the processors at / before program start, such that the computational load of the processors is equal
- ➔ Two fundamentally different approaches:
 - take into account the tasks' different computational load when mapping them to processors
 - extension of graph partitioning algorithms
 - requires a good estimation of a task's load
 - no solution, when load changes dynamically
 - fine grained cyclic or random mapping
 - results (most likely) in a good load balancing, even when the load changes dynamically
 - price: usually higher communication cost

2.9.3 Load Balancing ...



Example: atmospheric model, cyclic mapping



➔ Each processor has tasks with high and low computational load

2.9.3 Load Balancing ...



Dynamic load balancing

- ➔ Independent (often dyn. created) tasks (e.g., search problem)
 - goal: processors do not idle, i.e., always have a task to process
 - even at the end of the program, i.e., all processes finish at the same time
 - tasks are dynamically **allocated** to processors and stay there until their processing is finished
 - optimal: allocate task with highest processing time first
- ➔ Communicating tasks (SPMD, e.g., stencil algorithm)
 - goal: equal computing time between synchronisations
 - if necessary, tasks are **migrated** between processors during their execution



How to determine performance metrics

- ➔ Analytical model of the algorithm
 - approach: determine computation and communication time
 - $T(p) = t_{comp} + t_{comm}$
 - computation/communication ratio t_{comp}/t_{comm} allows a rough estimation of performance
 - requires a computation model (model of the computer hardware)
- ➔ Measurement with the real programm
 - explicit timing measurement in the code
 - performance analysis tools

2.9.5 Analytical Performance Modelling

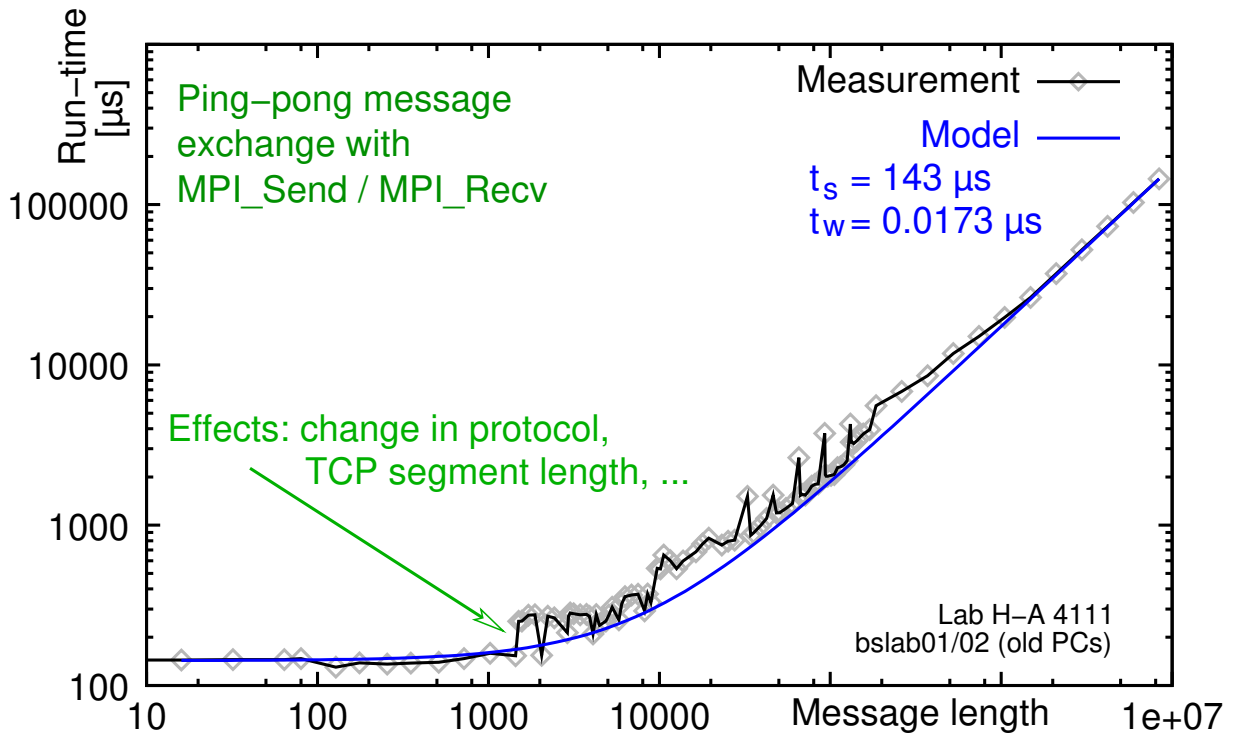


Models for communication time

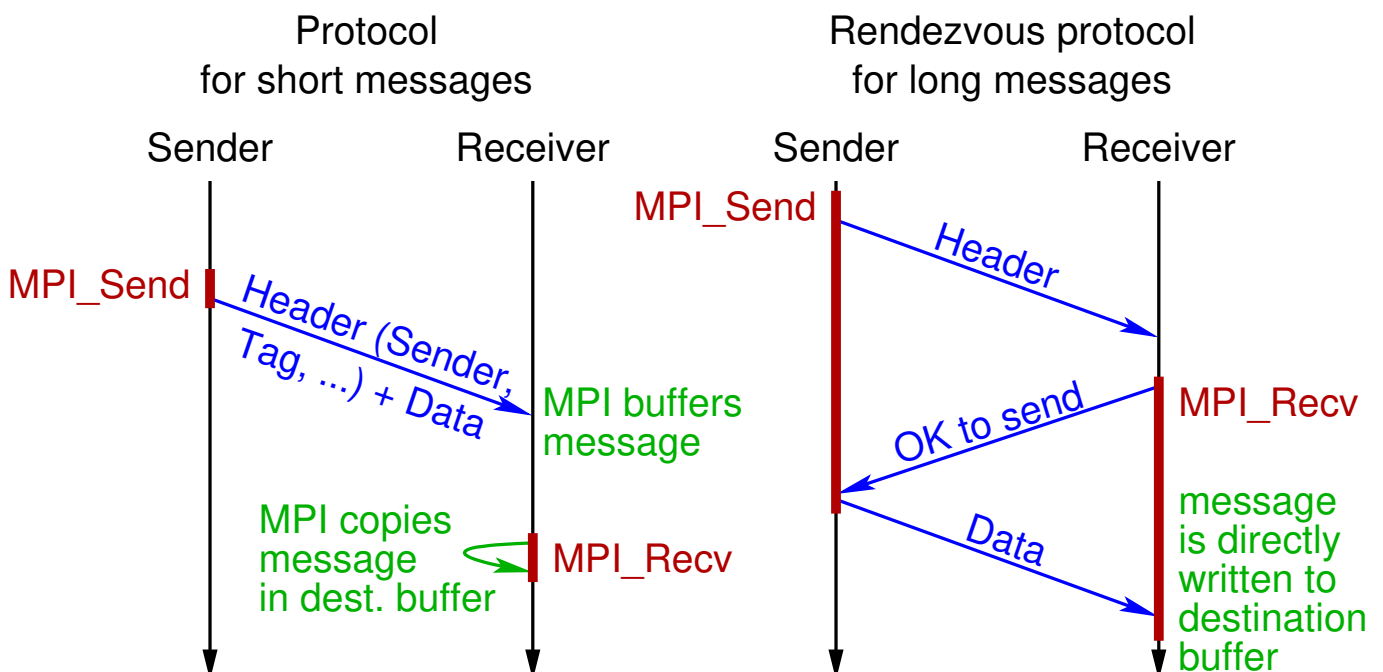
- ➔ E.g., for MPI (following Rauber: “*Parallele und verteilte Programmierung*”)
 - point-to-point send: $t(m) = t_s + t_w \cdot m$
 - broadcast: $t(p, m) = \tau \cdot \log p + t_w \cdot m \cdot \log p$
- ➔ Parameters (t_s, t_w, τ) are obtained via **micro benchmarks**
 - selectively measure a single aspect of the system
 - also allow the deduction of implementation characteristics
 - fitting, e.g., using the least square method
 - e.g., for point-to-point send:
 - PCs H-A 4111 (remote): $t_s = 75.3 \mu s, t_w = 9 ns$
 - PCs H-A 4111 (local): $t_s = 0.45 \mu s, t_w = 0.11 ns$



Example: results of the micro benchmark SKaMPI



Communication protocols in MPI





Example: matrix multiplication

- ➔ Product $C = A \cdot B$ of two square matrices
- ➔ Assumption: A, B, C are distributed blockwise on p processors
 - processor P_{ij} has A_{ij} and B_{ij} and computes C_{ij}
- ➔ P_{ij} needs A_{ik} and B_{kj} for $k = 1 \dots \sqrt{p}$
- ➔ Chosen approach:
 - all-to-all broadcast of the A blocks in each row of processors
 - all-to-all broadcast of the B blocks in each column of processors

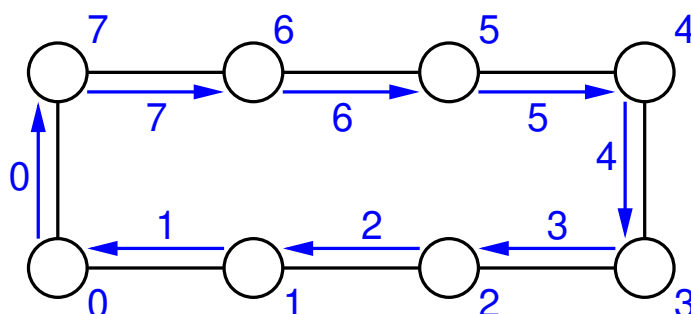
➔ computation of $C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{ik} \cdot B_{kj}$



(Animated slide)

All-to-all broadcast

- ➔ Required time depends on selected communication structure
- ➔ This structure may depend on the network structure of the parallel computer
 - who can directly communicate with whom?
- ➔ Example: ring topology



$p-1$ steps:
send "newest"
data element to
successor in ring

- ➔ Cost: $t_s(p - 1) + t_w m(p - 1)$ (m : data length)

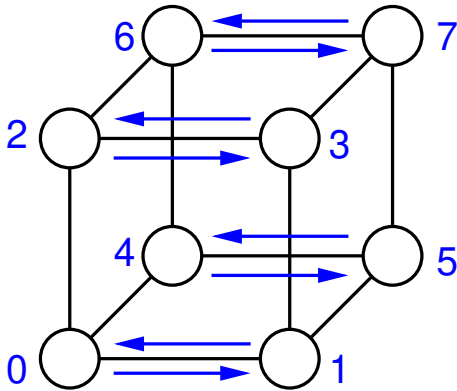
2.9.5 Analytical Performance Modelling ...



(Animated slide)

All-to-all broadcast ...

- ➔ Example: communication along a hyper cube
- ➔ requires only $\log p$ steps with p processors



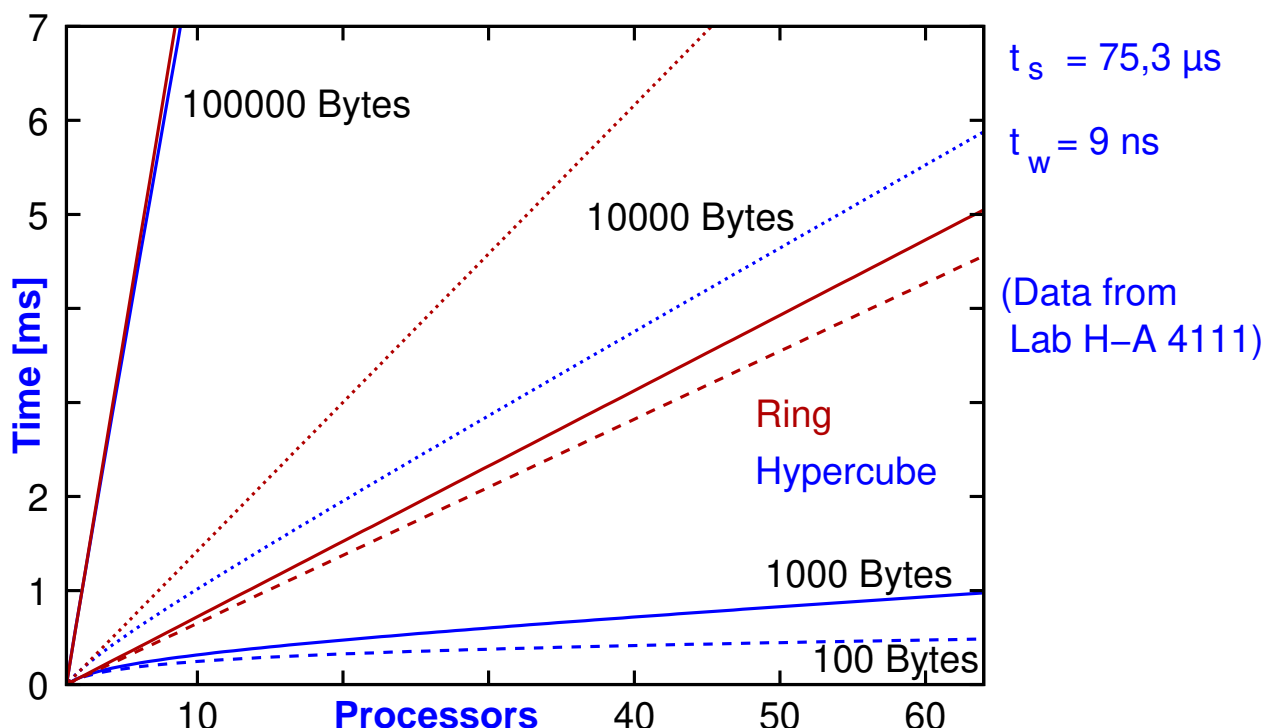
1. Pairwise exchange in x direction
2. Pairwise exchange in y direction
3. Pairwise exchange in z direction

➔ Cost:
$$\sum_{i=1}^{\log p} (t_s + 2^{i-1} t_w m) = t_s \log p + t_w m (p - 1)$$

2.9.5 Analytical Performance Modelling ...



All-to-all broadcast ...



Complete analysis of matrix multiplication

- ➔ Two all-to-all broadcast steps between \sqrt{p} processors
 - ➔ each step concurrently in \sqrt{p} rows / columns
- ➔ Communication time: $2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p} - 1))$
- ➔ \sqrt{p} multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ sub-matrices
- ➔ Computation time: $t_c \sqrt{p} \cdot (n/\sqrt{p})^3 = t_c n^3 / p$
- ➔ Parallel run-time: $T(p) \approx t_c n^3 / p + t_s \log p + 2t_w(n^2 / \sqrt{p})$
- ➔ Sequential run-time: $T_s = t_c n^3$

Notes for slide 188:

When we compare the result for $T(p)$ with the formula $T(p) = \frac{W + T_o(W, p)}{p}$ slide from 170, we get:

$$T_o(n, p) = p \cdot (t_s \log p + 2t_w(n^2 / \sqrt{p}))$$

(We use the problem size n instead of the work W here).

Now, to get the isoefficiency function of our matrix multiplication, we must solve the equation $n = K \cdot T_o(n, p)$ w.r.t. n . We can do this in approximation by neglecting the term $t_s \log p$. Then we get:

$$n = K \cdot p \cdot (2t_w(n^2 / \sqrt{p})) = 2 \cdot K \cdot t_w \cdot n^2 \cdot p^{\frac{3}{2}}$$

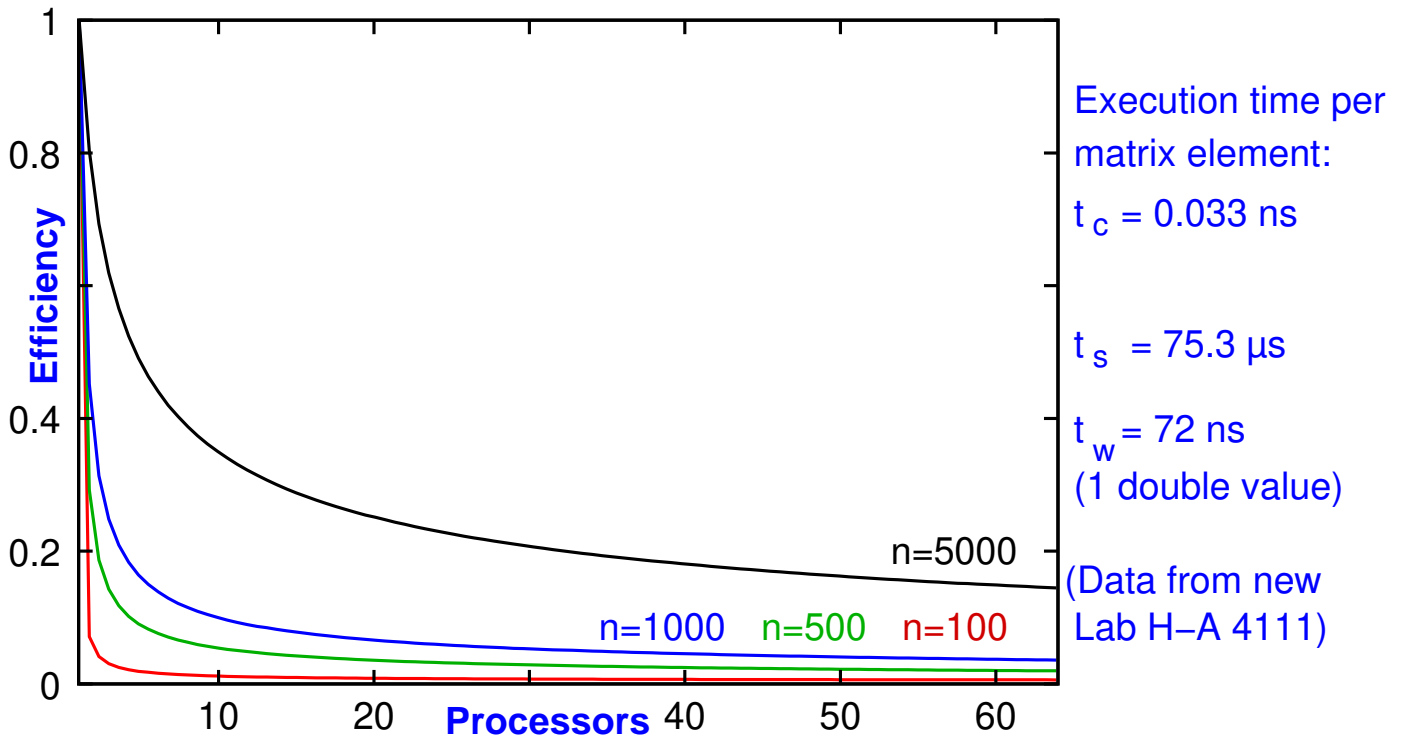
Or (by dividing both sizes by n and reordering):

$$n = \frac{1}{2 \cdot K \cdot t_w} \cdot p^{\frac{2}{3}}$$

That is $n(p) = \mathcal{O}(p^{\frac{2}{3}})$, which implies $n(p) = \mathcal{O}(p)$. Thus, matrix multiplication offers (very) good scalability.

(Animated slide)

Efficiency of matrix multiplication



2.9.6 Performance Analysis Tools

- ➔ Goal: performance debugging, i.e., finding and eliminating performance bottlenecks
- ➔ Method: measurement of different quantities (metrics), if applicable separated according to:
 - ➔ execution unit (compute node, process, thread)
 - ➔ source code position (procedure, source code line)
 - ➔ time
- ➔ Tools are very different in their details
 - ➔ method of measurement, required preparation steps, processing of information, ...
- ➔ Some tools are also usable to visualise the program execution



Metrics for performance analysis

- ➔ CPU time (assessment of computing effort)
- ➔ Wall clock time (includes times where thread is blocked)
- ➔ Communication time and volume
- ➔ Metrics of the operating system:
 - page faults, process switches, system calls, signals
- ➔ Hardware metrics (only with hardware support in the CPU):
 - CPU cycles, floating point operations, memory accesses
 - cache misses, cache invalidations, ...



Parallel Processing

Winter Term 2025/26

04.11.2025

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

Stand: January 8, 2026



Sampling (sample based performance analysis)

- ➔ Program is interrupted periodically
- ➔ Current value of the program counter is read (and maybe also the call stack)
- ➔ The full measurement value is assigned to this place in the program, e.g., when measuring CPU time:
 - periodic interruption every $10ms$ CPU time
 - `CPU_time[current_PC_value] += 10ms`
- ➔ Mapping to source code level is done offline
- ➔ Result: measurement value for each function / source line



Profiling and tracing (event based performance analysis)

- ➔ Requires an **instrumentation** of the programs, e.g., insertion of measurement code at interesting places
 - often at the beginning and end of library routines, e.g., `MPI_Recv`, `MPI_Barrier`, ...
- ➔ Tools usually do the instrumentation automatically
 - typically, the program must be re-compiled or re-linked
- ➔ Analysis of the results is done during the measurement (profiling) or after the program execution (tracing)
- ➔ Result:
 - measurement value for each measured function (profiling, tracing)
 - development of the measurement value over time (tracing)



Example: measurement of cache misses

- ➔ Basis: hardware counter for cache misses in the processor
- ➔ Sampling based:
 - when a certain counter value (e.g., 419) is reached, an interrupt is triggered
 - `cache_misses[current_PC_value] += 419`
- ➔ Event based:
 - insertion of code for reading the counters:

```
old_cm = read_hw_counter(25);
for (j=0;j<1000;j++)
    d += a[i][j];
cache_misses += read_hw_counter(25)-old_cm;
```



Pros and cons of the methods

- ➔ Sampling
 - low and predictable overhead; reference to source code
 - limited precision; no resolution in time
- ➔ Tracing
 - acquisition of all relevant data with high resolution in time
 - relatively high overhead; large volumes of data
- ➔ Profiling
 - reduced volume of data, but less flexible

Parallel Processing

Winter Term 2025/26

3 Parallel Programming with Shared Memory

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**)

3.1 OpenMP Basics



Background

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

Notes for slide 199:

The discussion of OpenMP focuses on version 4.0. In the meantime, there is OpenMP 6.0 with a lot of extensions as compared to 4.0. However, compiler support for newer OpenMP standards is often missing.

199-1

3.1 OpenMP Basics ...



Parallelization using directives

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



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`

Notes for slide 203:

On the lab computers in H-A 4111, you can also use the Nvidia `nvc++` compiler. It does not support all OpenMP directives supported by `g++`, but sometimes achieves much better performance. In addition, `nvc++` provides a much better support for GPU offloading than `g++`.

In order to use the `nvc++` compiler, you have to load the module `nvhpc` using the command:

```
module load nvhpc.
```

The typical call of `nvc++` is:

```
nvc++ -mp myProg.cpp -o myProg
```

To unload the module again (which is required for working with `g++`), use:

```
module unload nvhpc.
```

3.1.1 The parallel directive



(Animated slide)

An example (👉 03/firstprog.cpp)

Program

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

Compilation

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

Execution

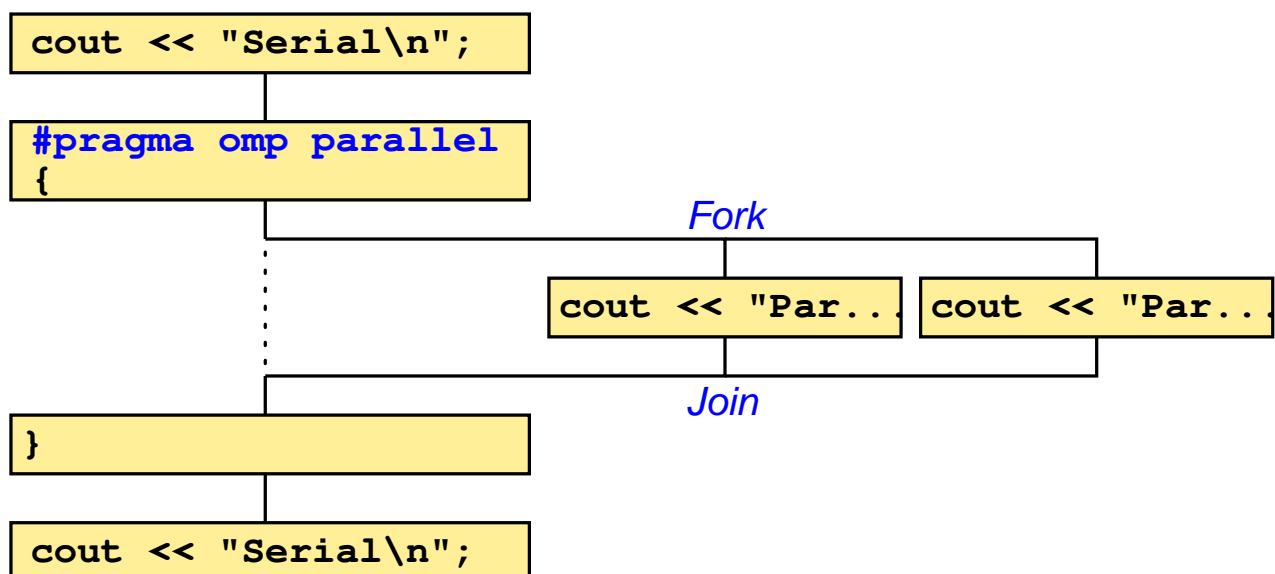
```
% export OMP_NUM_THREADS=2  
% ./firstprog  
Serial  
Parallel  
Parallel  
Serial
```

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



Shared and private variables ...

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

Notes for slide 209:

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

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

209-1

3.1.1 The `parallel` directive ...



Shared and private variables: an example (03/private.cpp)

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

Each thread has a (non-initialized) copy of `i`

Each thread has an initialized copy of `j`

`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

3.2 Loop parallelization



Motivation

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

3.2 Loop parallelization ...



3.2.1 The `for` directive: parallel loops

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

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

Notes for slide 214:

- ➔ The option `nowait` is not accepted in a `#pragma omp parallel for` (as at the end of a parallel region, there always is a global synchronisation)
- ➔ Besides the option `nowait`, the following additional options can be specified in the `<clause_list>` of a `for` directive:
 - ➔ `private`, `firstprivate`, `lastprivate`, `shared`: see slides 209 and 219 (These options are only accepted in a `#pragma omp parallel for`, not in a `#pragma omp for` inside a parallel region)
 - ➔ `schedule`: see slide 216
 - ➔ `ordered`: see slide 240
 - ➔ `reduction`: see slide 238
 - ➔ `collapse(<num>)`: this option tells the compiler that the next `<num>` (perfectly) nested loops should be collapsed into a single loop, whose iterations will then be distributed.

214-1

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



Example: vector addition

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

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

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

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



Scheduling of loop iterations

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

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



Scheduling example (👉 03/loops.cpp)

```
int i, j;
double x;

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

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

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



Scheduling example: results

➔ Runtime with 4 threads on the lab computers:

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

➔ Load imbalance when using "static"

➤ thread 1: $i=0..9999$, thread 4: $i=30000..39999$

➔ "static,1" and "dynamic" use a block size of 1

➤ each thread executes every 4th iteration of the i loop

➤ can be very inefficient due to caches (*false sharing*, [5.1](#))

➤ remedy: use larger block size (e.g.: "dynamic,100")

➔ "guided" often is a good compromise between load balancing and locality (cache usage)

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



Shared and private variables in loops

➔ The `parallel for` directive can be supplemented with the options `private`, `shared` and `firstprivate` (see slide 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: January 8, 2026

3.2.2 Parallelization of Loops



(Animated slide)

When can a loop be parallelized?

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

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**)

3.2.2 Parallelization of Loops ...



(Animated slide)

Handling of data dependences in loops

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

```
#pragma omp parallel
{
    #pragma omp for
    for (i=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!

3.2.2 Parallelization of Loops ...



Handling of data dependences in loops ...

- ➔ True dependence:
 - ➔ introduce proper synchronization between the threads
 - ➔ e.g., using the `ordered` directive (☞ 3.3):

```
#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++)
    a[i] = a[i-4] + b[i];
```

3.2.3 Simple Examples



(Animated slide)

Matrix addition

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

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

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

3.2.3 Simple Examples ...



(Animated slide)

Matrix addition

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

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

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

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

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

Outer loop can be executed in parallel

Advantage: less overhead!



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



(Animated slide)

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 dependence between loop and environment ➔ lastprivate(i) + barriers

3.2.4 Dependence Analysis in Loops



(Animated slide)

Direction vectors

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

```

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

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

Direction vector: \nearrow
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];
  }
}
    
```

S1 in earlier iteration of 'i' and 'j' loop than S2

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

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

3.2.4 Dependence Analysis in Loops ...



Formal computation of dependences

- ➔ Basis: Look for an integer solution of a system of (in-)equations

- ➔ Example:

```

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

Equation system:

$$0 \leq i_1 < 10$$

$$0 \leq i_2 < 10$$

$$0 \leq j_1 \leq i_1$$

$$0 \leq j_2 \leq i_2$$

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

- ➔ 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 $(\dots, <, >, \dots)$

Notes for slide 229:

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

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

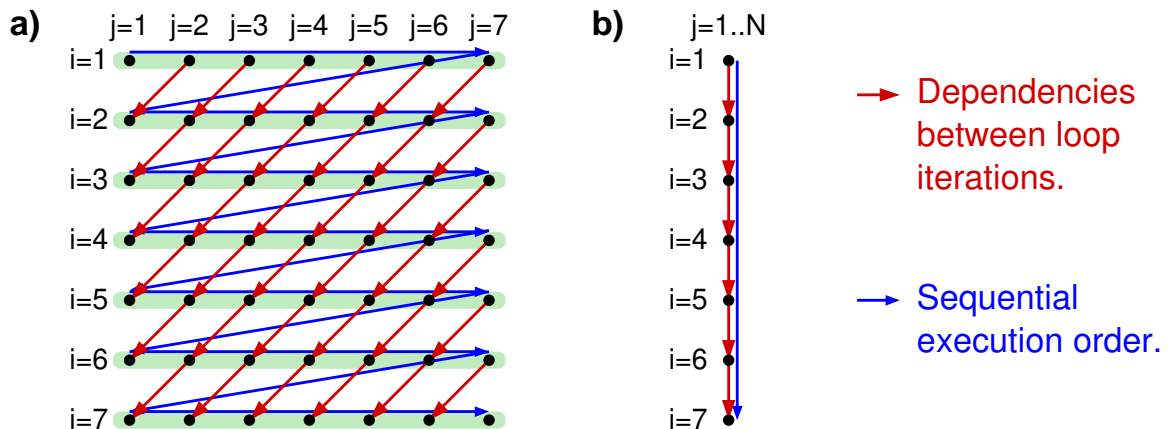
There is an anti-dependence from S2 to S1 (consider e.g. $a[3][3]$: it is read in iteration $i=2, j=4$ and is written later in iteration $i=3, j=3$).

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

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

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

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



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

229-2

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

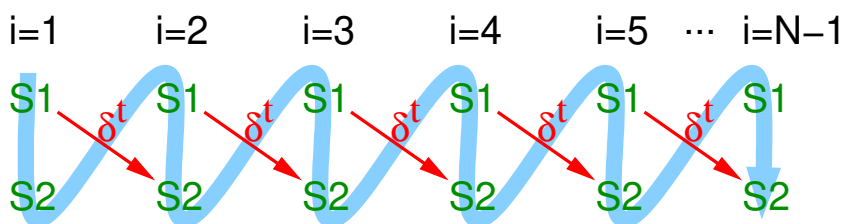
➔ However, since we do *not* have any dependence $S2 \xrightarrow{\delta_{(<)}} S1$, loop splitting is permitted, which results in:

```
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
```

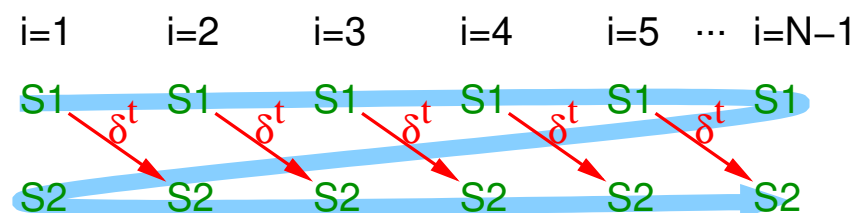


Example: loop splitting ...

➔ 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!

3.3 OpenMP Synchronization ...



Synchronization in OpenMP

➔ Higher-level, easy to use constructs

➔ Implementation using directives:

- ➔ `critical`: critical section
- ➔ `atomic`: atomic operations
- ➔ `ordered`: execution in program order
- ➔ `barrier`: barrier
- ➔ `single` and `master`: execution by a single thread
- ➔ `taskwait` and `taskgroup`: wait for tasks (➔ [3.5.2](#))
- ➔ `flush`: make the memory consistent
 - ➔ memory barrier (➔ [2.4.2](#))
 - ➔ implicitly executed with the other synchronization directives



3.3.1 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`)

Notes for slide 236:

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

236-1

3.3.2 Atomic operations ...



Examples

➔ Atomic adding:

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

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

➔ Atomic *fetch-and-add*:

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

➔ Instead of `+`, all other binary operators are possible, too

➔ With OpenMP 4, an atomic *compare-and-swap* can not yet be implemented

➔ use builtin functions of the compiler, if necessary

➔ (OpenMP 5.1 introduces a `compare` clause)

Notes for slide 237:

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

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

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

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

237-1

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

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

OpenMP 5.1 introduces a `compare` option, which allows (among others) to implement an atomic compare-and-swap operation (see slide 300). However, most compilers do not yet support OpenMP 5.1.

237-2



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 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);
```

Add local partial sum to the global sum



3.3.4 Execution in program order

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

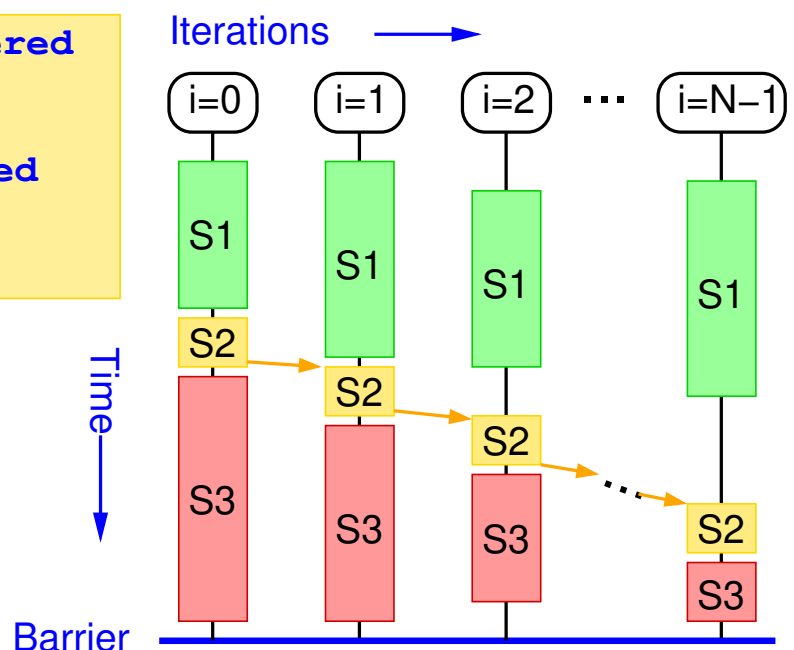
- ➔ The ordered directive is only allowed in the dynamic extent of a for directive with option ordered
 - recommendation: use option `schedule(static,1)`
 - or `schedule(static,n)` with small n
- ➔ The threads will execute the instances of the statement / block exactly in the same order as in the sequential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

Notes for slide 242:

Example for a nested loop with dependencies:

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

In an analogous way, the `ordered` directive allows to parallelize the Gauss/Seidel-method in a pipeline style (👉 [page 255](#)).




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: January 8, 2026

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";
}
```



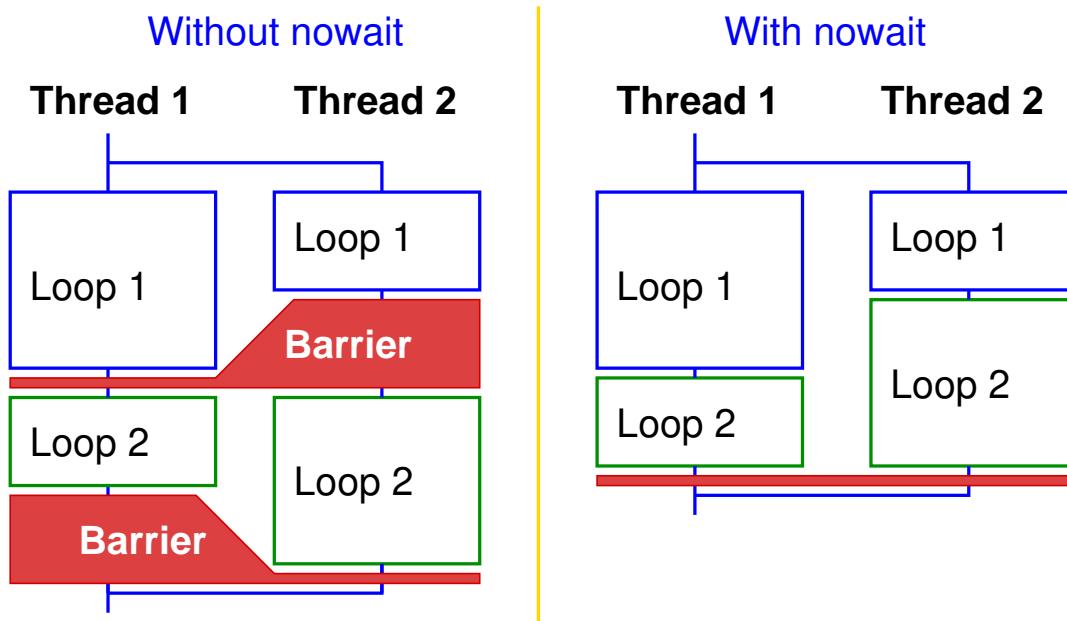
Example ...

- ➔ 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



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

Notes for slide 248:

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

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

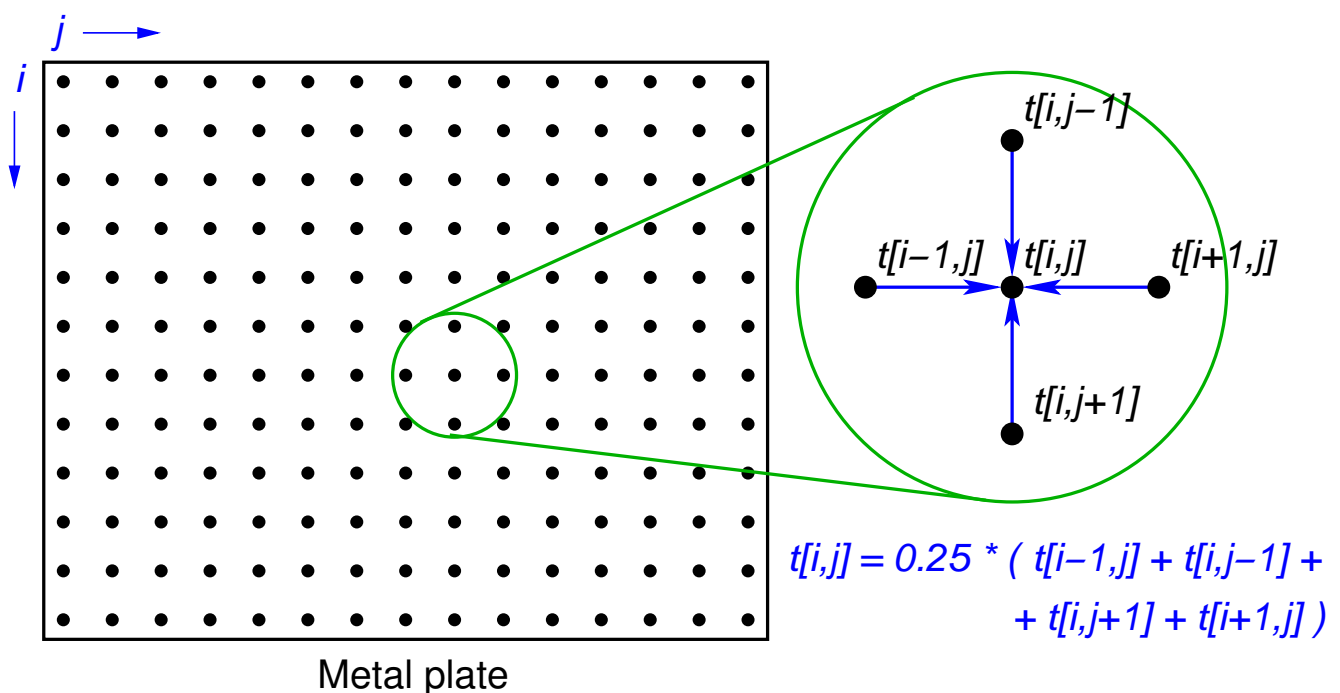


Numerical solution of the equations for thermal conduction

- ➔ 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



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

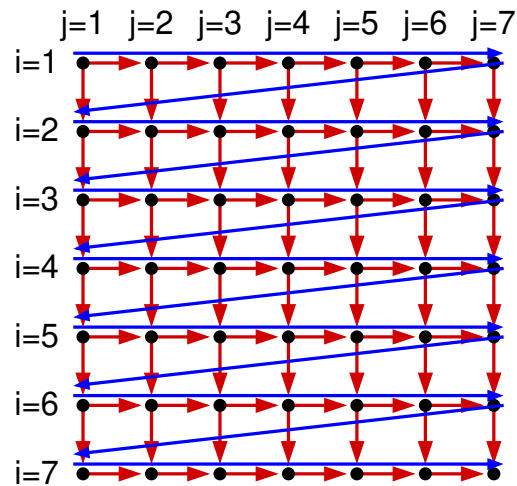
3.4 Exercise: The Jacobi and Gauss/Seidel Methods ...



(Animated slide)

Dependences 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!

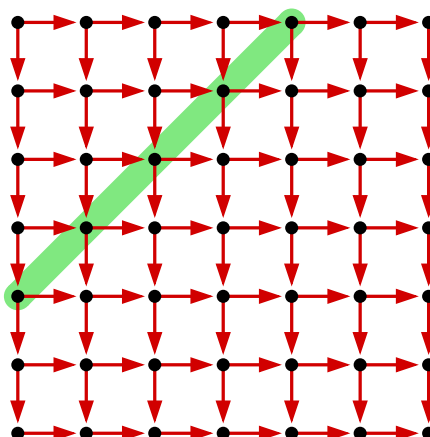
3.4 Exercise: The Jacobi and Gauss/Seidel Methods ...



(Animated slide)

Parallelisation of the Gauss/Seidel method

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



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



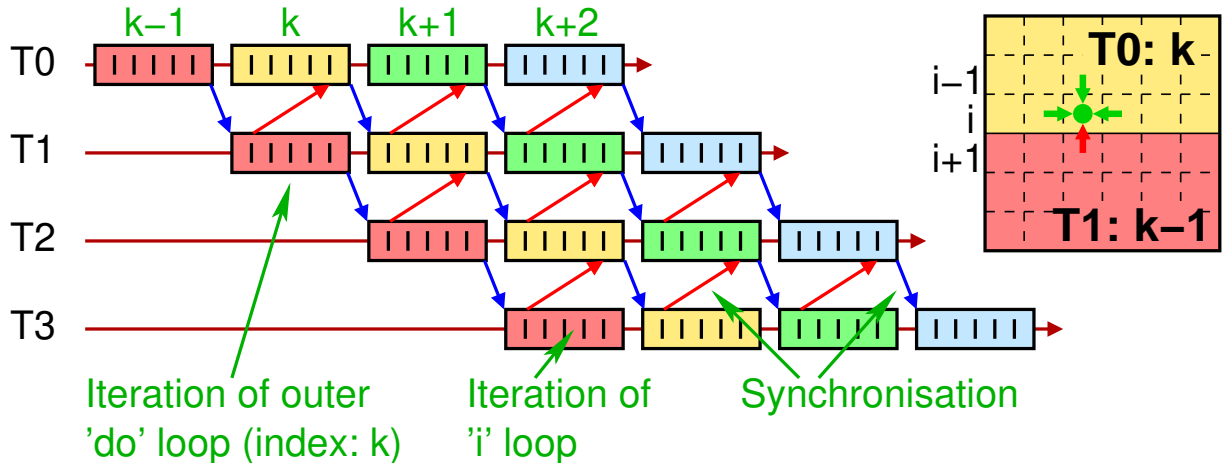
(Animated slide)

Alternative parallelization of the Gauss/Seidel method

- ➔ Requirement: number of iterations is known in advance
 - (or: we are allowed to execute a few more iterations after convergence)

➔ Then we can use a pipeline-style parallelization

- synchronisation via ordered (see 3.3.4)



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



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

Notes for slide 256:

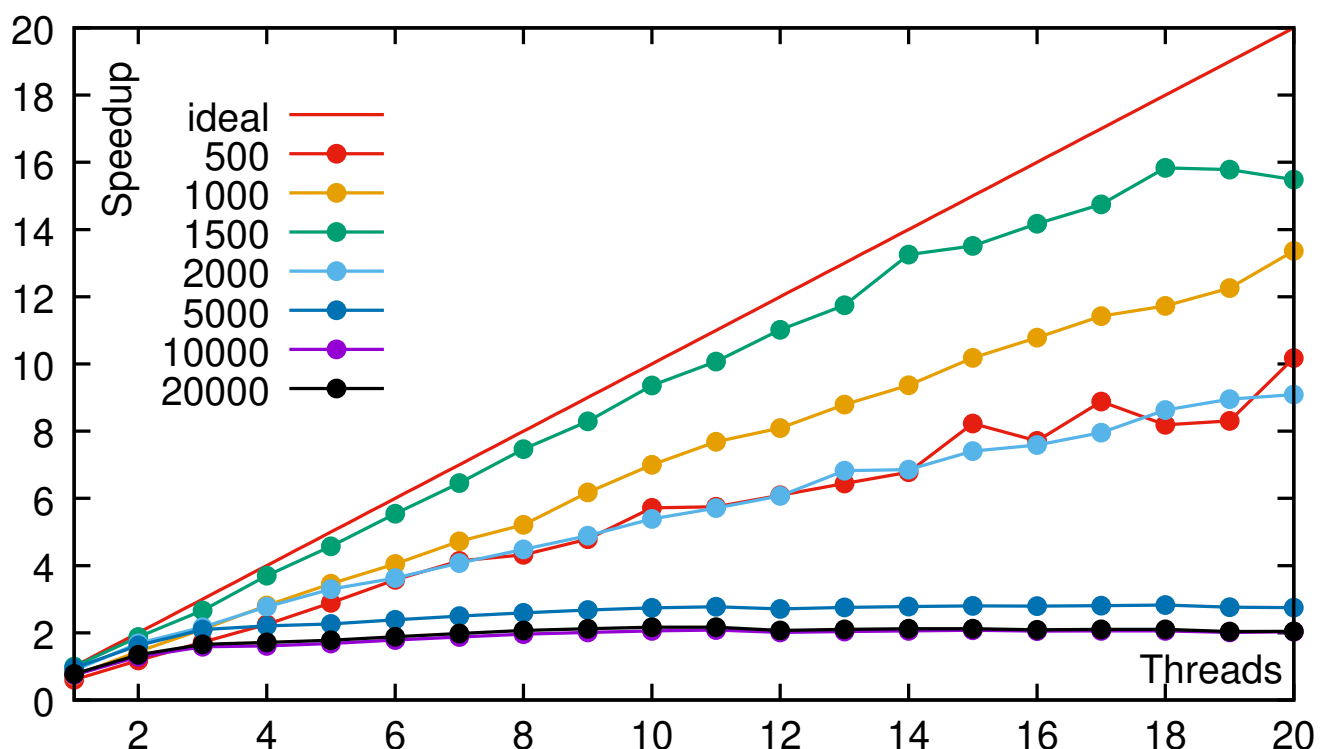
In the following graphs

- ➔ the runtime of the sequential program has been determined as the minimum of the runtimes on a performance core and an efficiency core,
- ➔ the parallel programs used performance cores for the first 8 threads, and then efficiency cores.

256-1

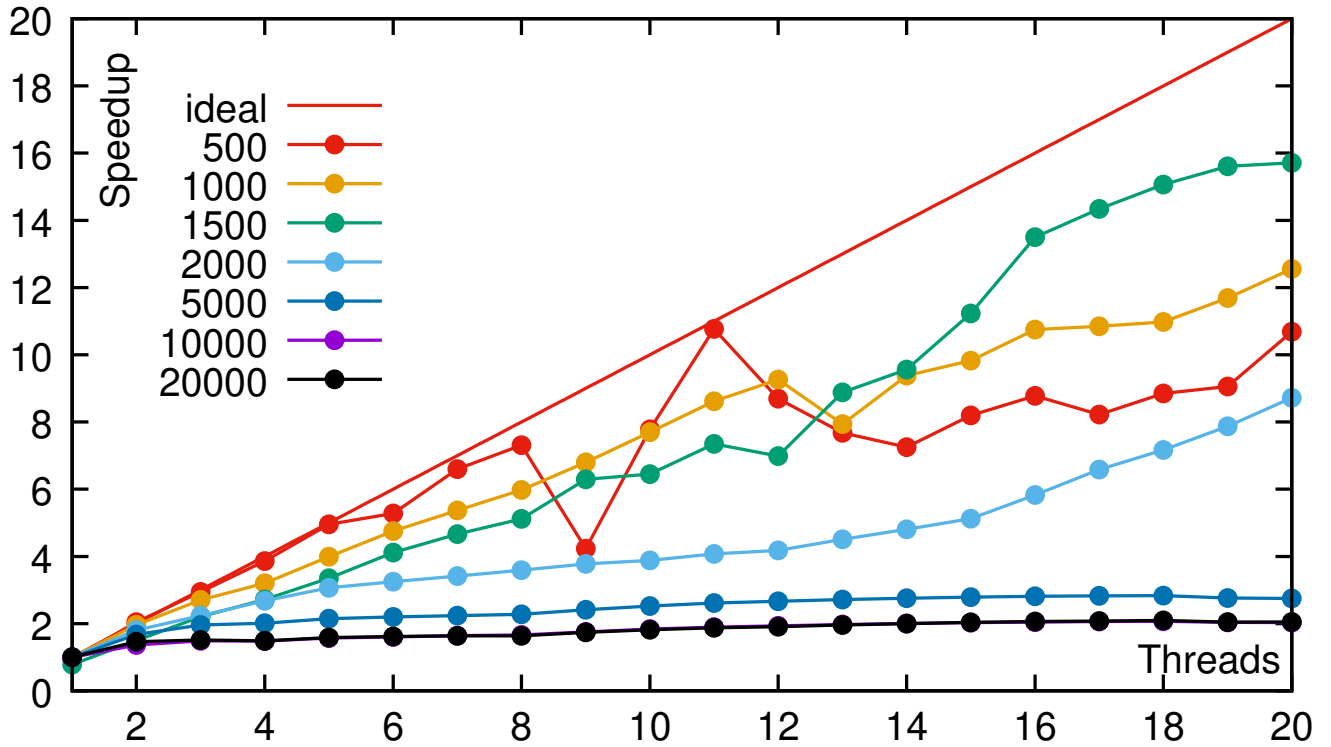
3.4 Exercise: The Jacobi and Gauss/Seidel Methods ...

Jacobi: Speedup (g++)



Notes for slide 257:

When efficiency cores are preferred for the threads, the following speedup results:

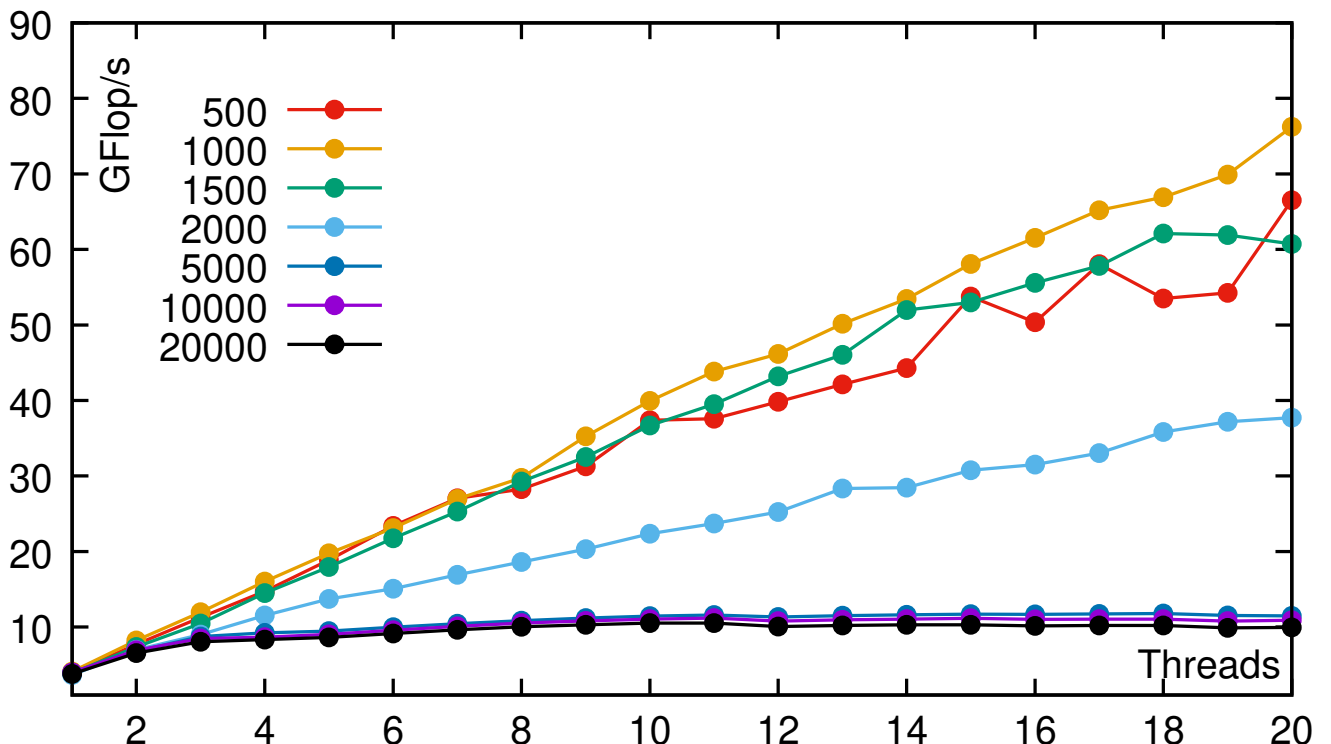


257-1

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



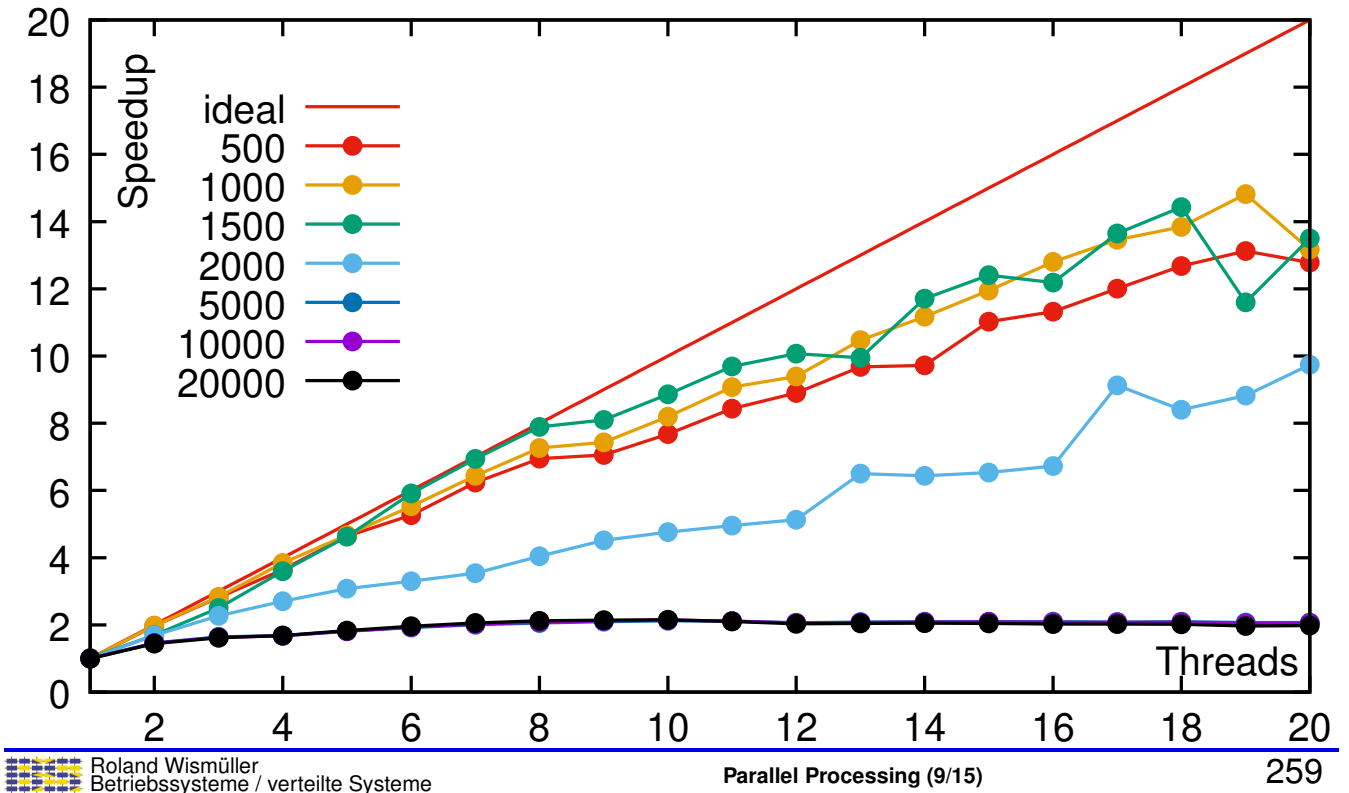
Jacobi: Performance (g++)



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



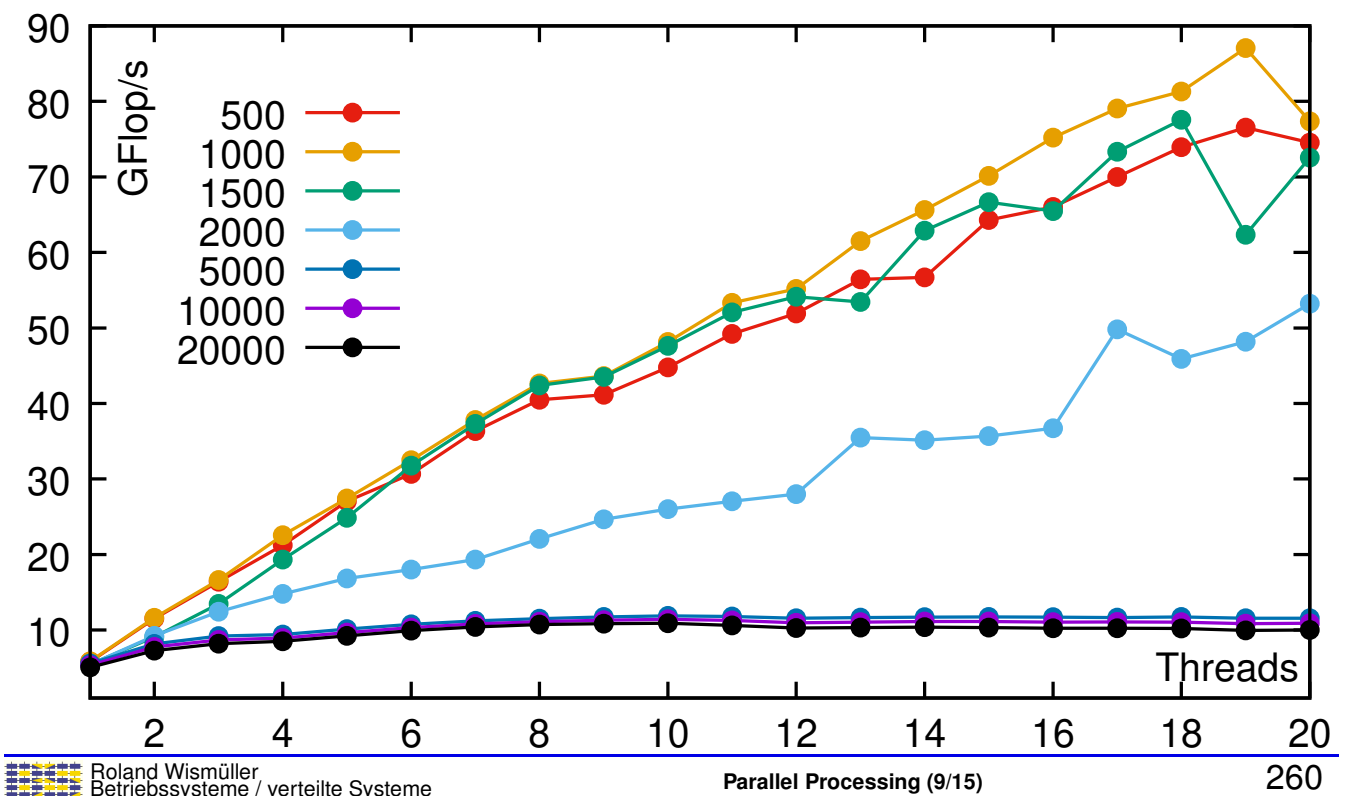
Jacobi: Speedup (nvc++)



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



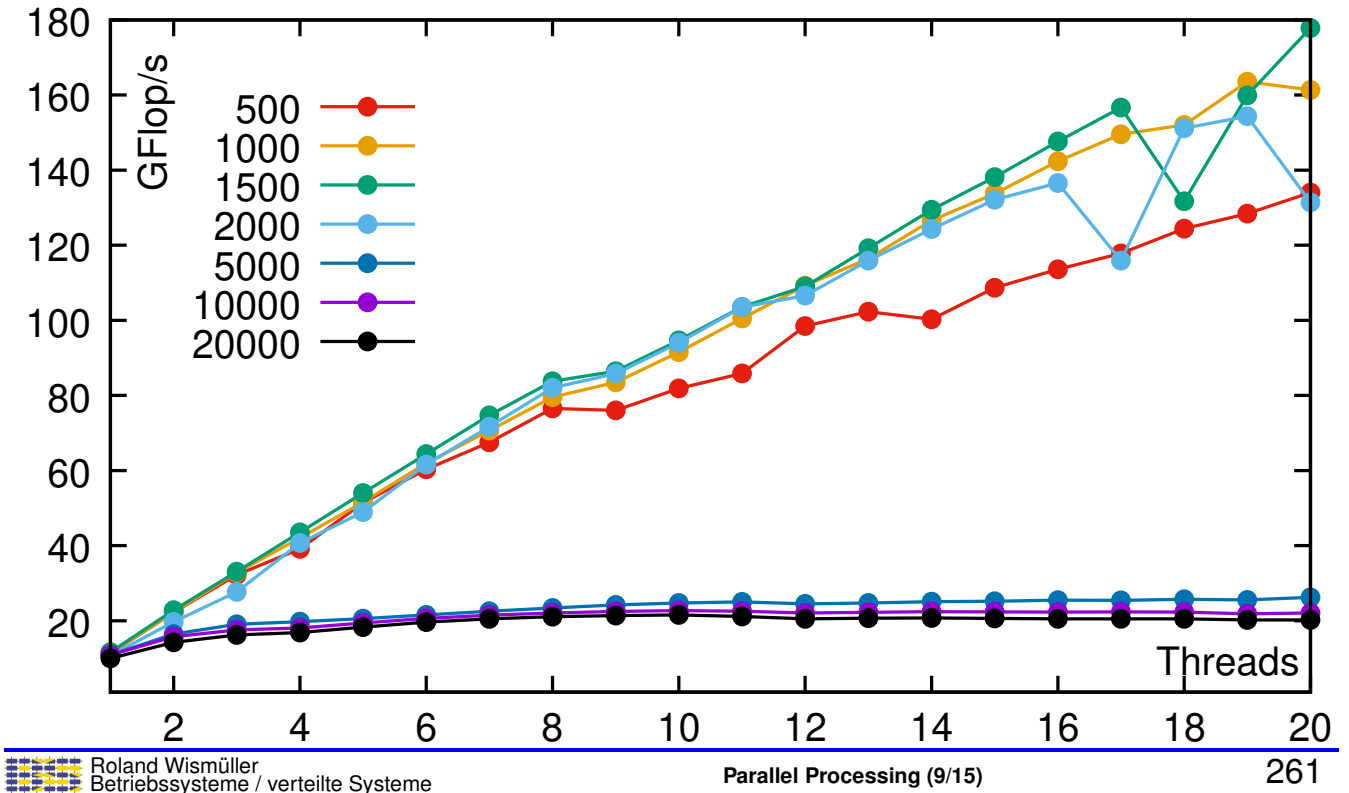
Jacobi: Performance (nvc++)



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



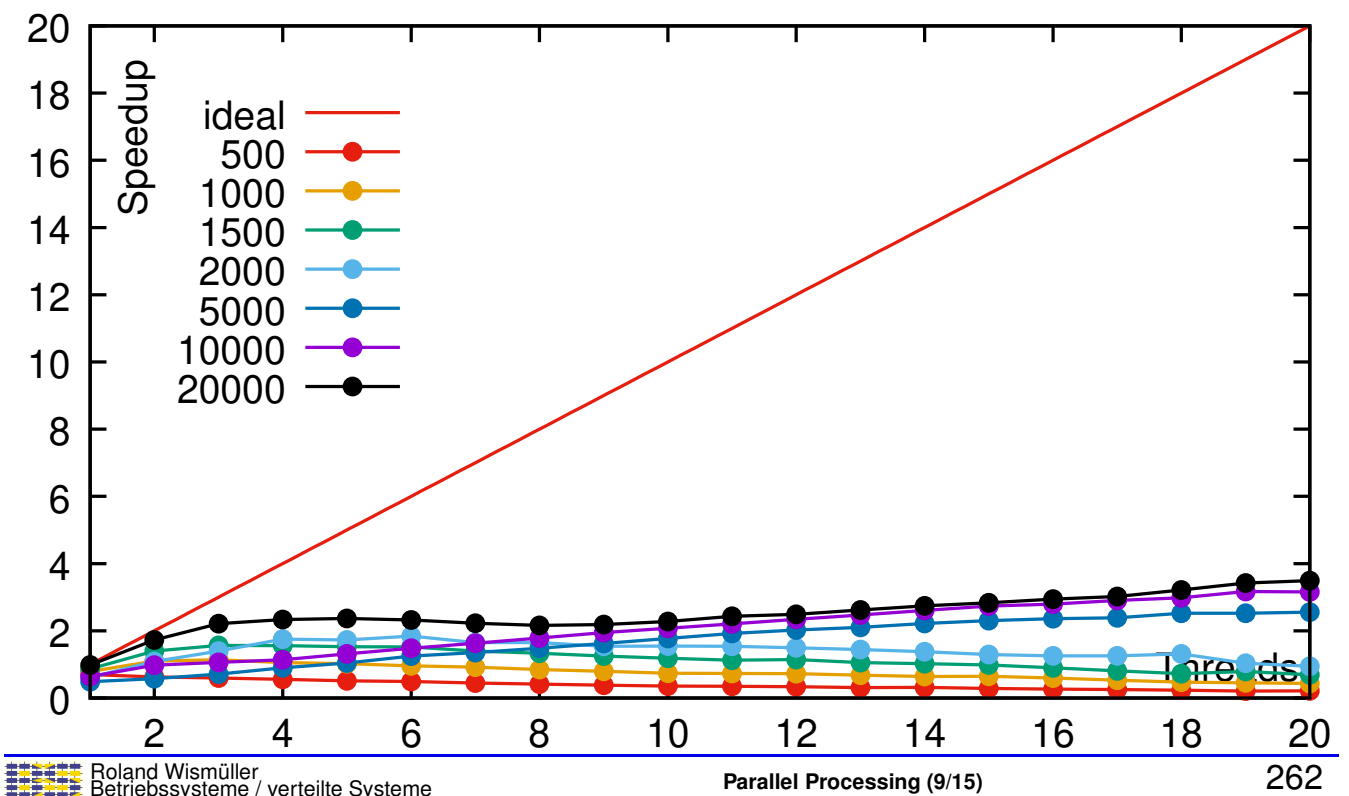
Jacobi: Performance (nvc++ with 32-Bit floating point)



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



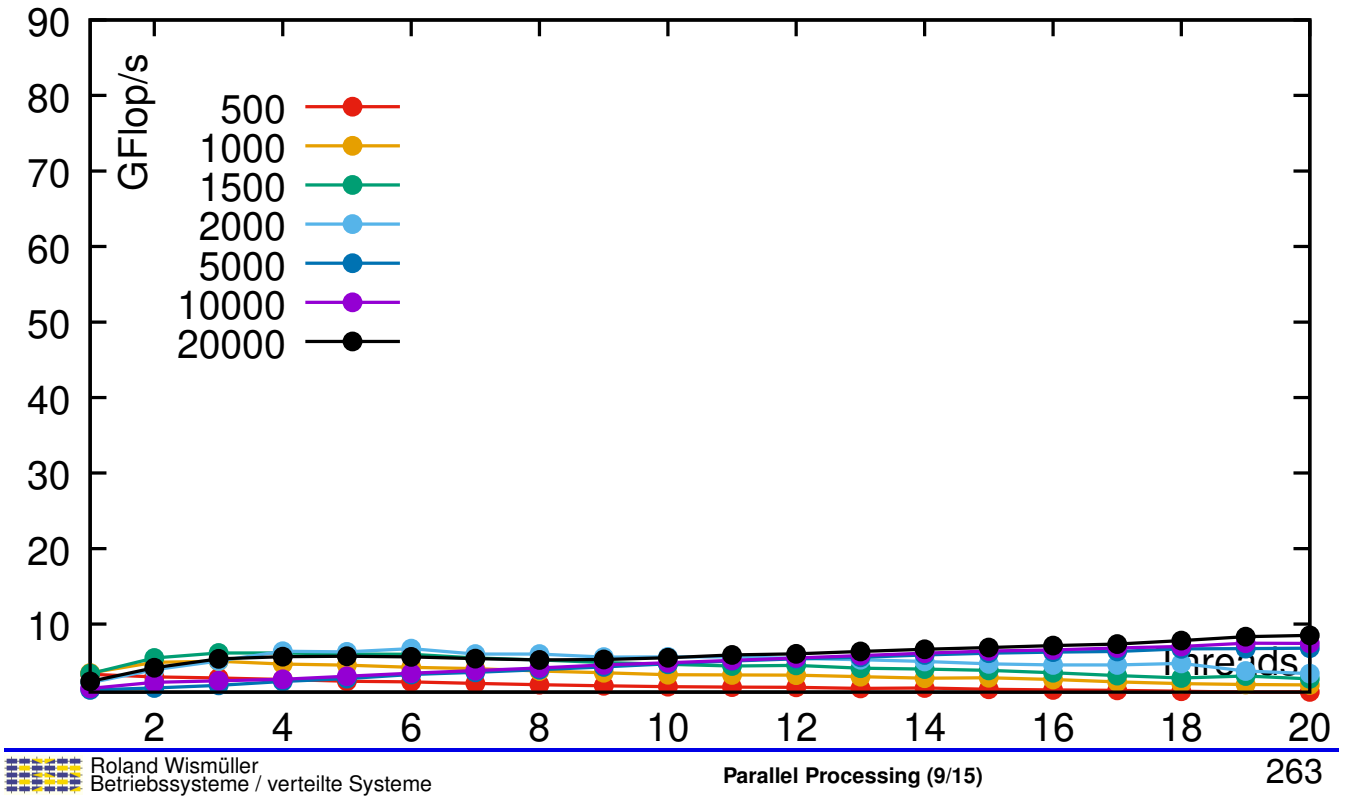
Gauss/Seidel: Speedup (diagonal, g++)



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



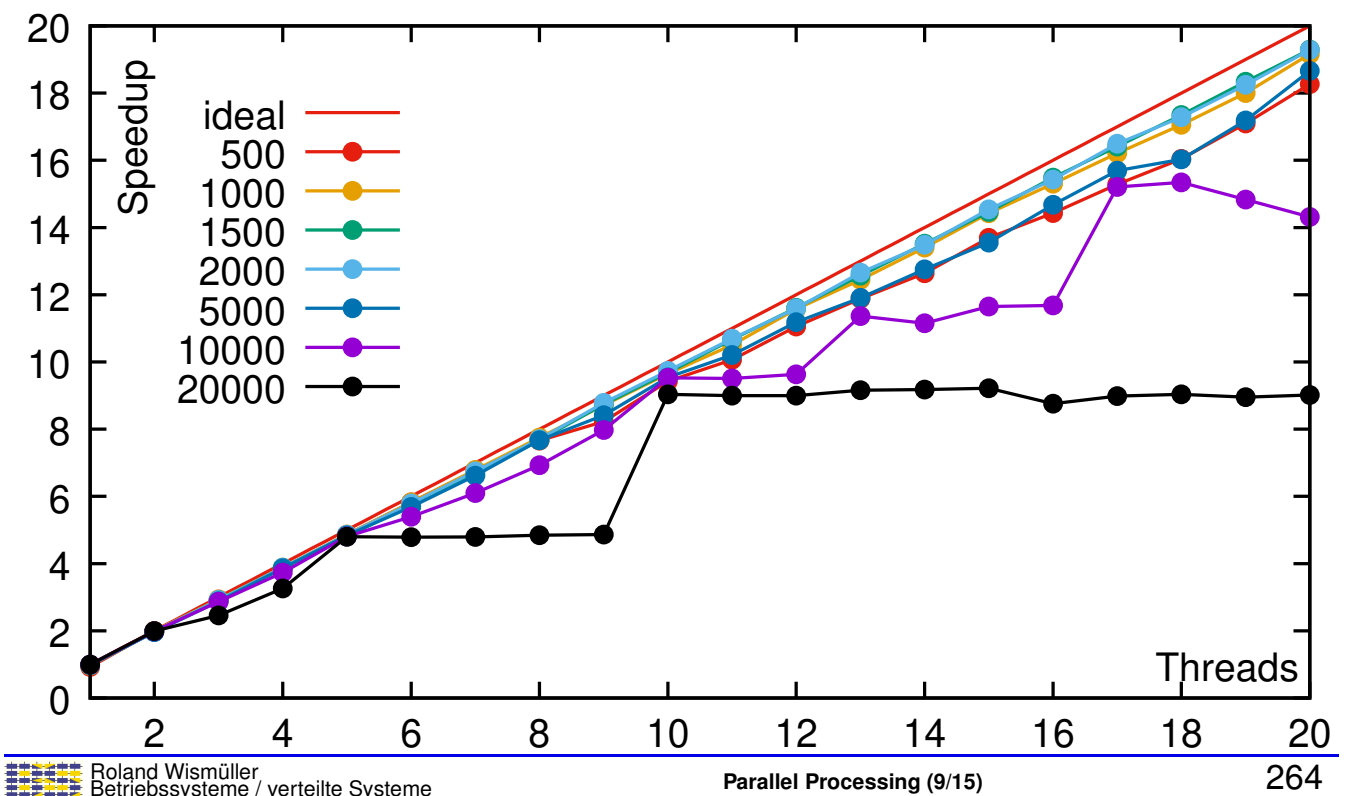
Gauss/Seidel: Performance (diagonal, g++)



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



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



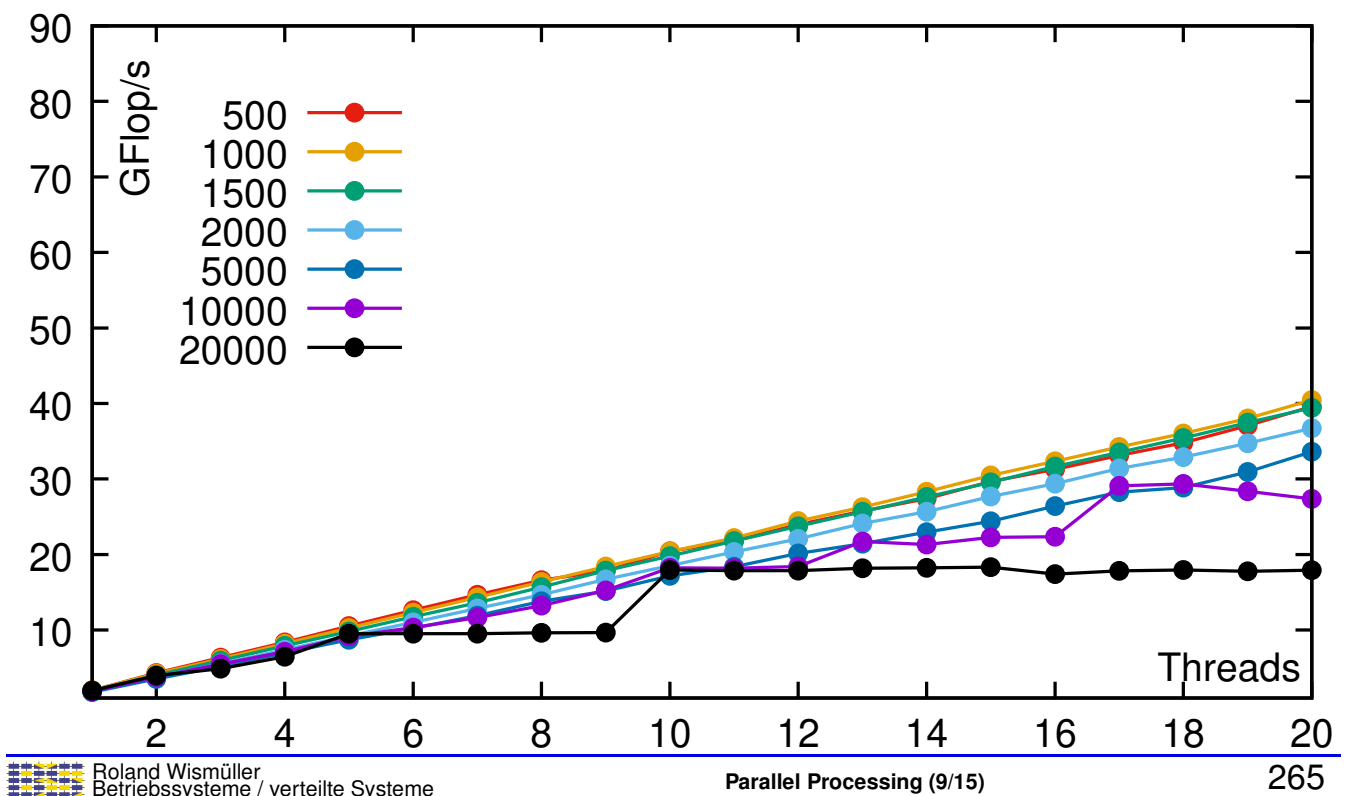
Notes for slide 264:

The nvc++ compiler does not yet support the required OpenMP features.

264-1

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

3.5.1 The sections directive ...



Example: independent code parts

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

Important!!

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



3.5.2 The `task` Directive: Explicit Tasks

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

- ➔ Creates an explicit task from the statement / the block
- ➔ Tasks will be executed by the available threads (*work pool* model)
- ➔ Options `private`, `firstprivate`, `shared` determine, which variables belong to the data environment of the task
 - ➔ the default for local variables is `firstprivate`, i.e., local variables declared outside but used inside the block are the task's input arguments
- ➔ Option `if` allows to determine, when an explicit task should be created

3.5.2 The `task` Directive ...



Example: parallel quicksort ( 03/qsort.cpp)

```
void quicksort(int *a, int lo, int hi) {  
    ...  
    // Variables are 'firstprivate' by default  
    #pragma omp task if (j-lo > 10000)  
    quicksort(a, lo, j);  
    quicksort(a, i, hi);  
}  
  
int main() {  
    ...  
    #pragma omp parallel  
    #pragma omp single nowait // Execution by a single thread  
    quicksort(array, 0, n-1);  
    // Before the parallel region ends, we wait for the termination of all threads
```

Notes for slide 269:

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

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

269-1

3.5.2 The `task` Directive ...



Task synchronization

```
#pragma omp taskwait
```

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

- ➔ `taskwait`: waits for the completion of all direct subtasks of the current task
- ➔ `taskgroup`: at the end of the block, the program waits for all tasks, which have been created within the block by the current task or one of its subtasks
 - ➔ available since OpenMP 4.0
 - ➔ caution: older compilers silently ignore this directive!

3.5.2 The task Directive ...



Example: parallel quicksort (👉 03/qsort.cpp)

➔ Imagine the following change when calling quicksort:

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

➔ Problem:

- ➔ quicksort() starts new tasks
- ➔ tasks are not yet finished, when quicksort() returns

3.5.2 The task Directive ...



Example: parallel quicksort ...

➔ Solution 1:

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

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

Notes for slide 272:

In this example, an additional overhead is created by always waiting for the subtasks after the recursive calls, even if none were generated (because $j-1 \leq 10000$). For the `taskwait` directive, there is no `if` option, so you might need to include a conditional statement here.

272-1

3.5.2 The `task` Directive ...



Example: parallel quicksort ...

➔ Solution 2:

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

← wait for all tasks created in the block

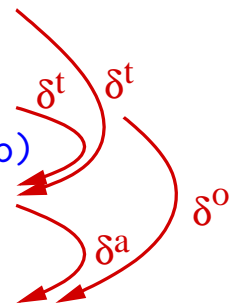
- ➔ advantage: only wait at one single place
- ➔ disadvantage: semantics of `quicksort()` must be very well documented

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

Notes for slide 274:

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

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

Array sections can be specified using the notation:

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

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

3.6 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

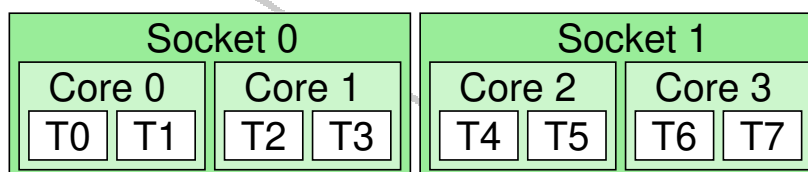
3.6.1 Thread Affinity ...



(Animated slide)

Environment variable `OMP_PLACES`

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



- ➔ To consider each 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"`

Notes for slide 276:

A single place is defined as a set of hardware execution environments (on a standard CPU, these are typically the hardware threads). In order to abbreviate the enumeration, a length and an optional step size can be specified for the simplified definition of intervals (separated by “:”). E.g., $\{0, 1\}:4:2$ is equivalent to $\{0, 1\}, \{2, 3\}, \{4, 5\}, \{6, 7\}$.

In addition to explicitly specifying the hardware threads, the list of places in `OMP_PLACES` can also be defined by a symbolic name (`threads`, `cores`, `sockets` or an implementation-dependent name).

276-1

3.6.1 Thread Affinity ...



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

Notes for slide 277:

In the example of slide 276, if the master thread is executed by hardware thread T0, `OMP_PLACES = threads` is specified, and a parallel region with 4 threads is created, the following happens:

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

277-1

3.6.1 Thread Affinity ...



Example: nested parallel regions

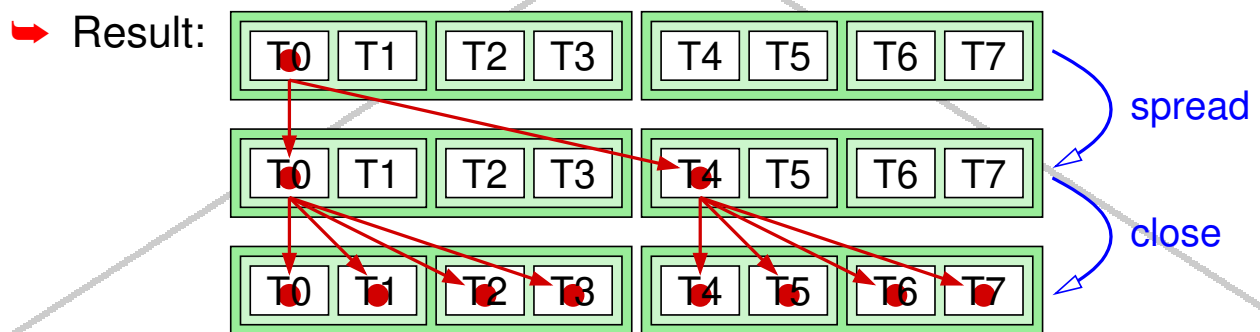
```
double f1(double x)
{
    #pragma omp parallel for proc_bind(close)
    for (i=0; i<N; i++) {
        ...
    }
    ...
    #pragma omp parallel proc_bind(spread)
    #pragma omp single
    {
        #pragma omp task shared(a)
        a = f1(x);
        #pragma omp task shared(b)
        b = f1(y);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

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`



Notes for slide 279:

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

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

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

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

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

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



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.:

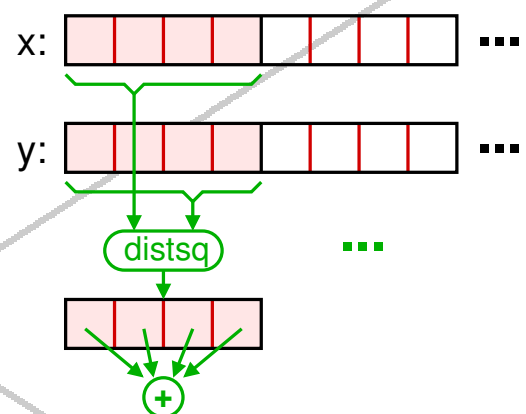
```
for (i=0; i<N; i++)
    a[i] = b[i] + a[i-4]; // safelen = 4
```

3.6.2 SIMD Vectorization ...



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)

3.6.3 Using External Accelerators ...



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`

Notes for slide 284:

- ➔ It is possible to allocate variables only on the target by using the OpenMP library. In order to allocate an array on the target, use the following code on the host (not inside a target directive!):

```
int device = omp_get_default_device();  
double *a = (double *)omp_target_alloc(N * sizeof(double), device);  
...  
omp_target_free(a, device);
```

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

Example:

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

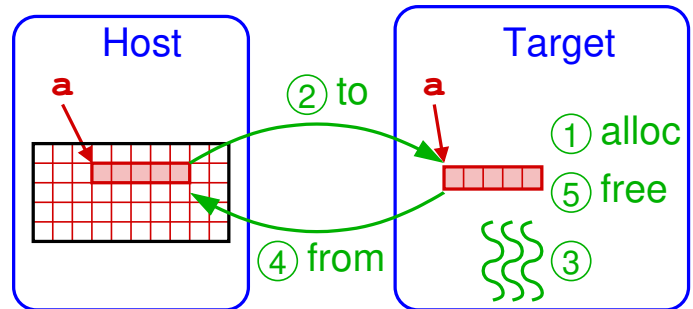
```
#pragma omp end declare target
```



Target data environment

➔ Course of actions for the target construct:

```
#pragma omp target \
    map(tofrom: 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: January 8, 2026

3.6.3 Using External Accelerators ...



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)

3.6.3 Using External Accelerators ...



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

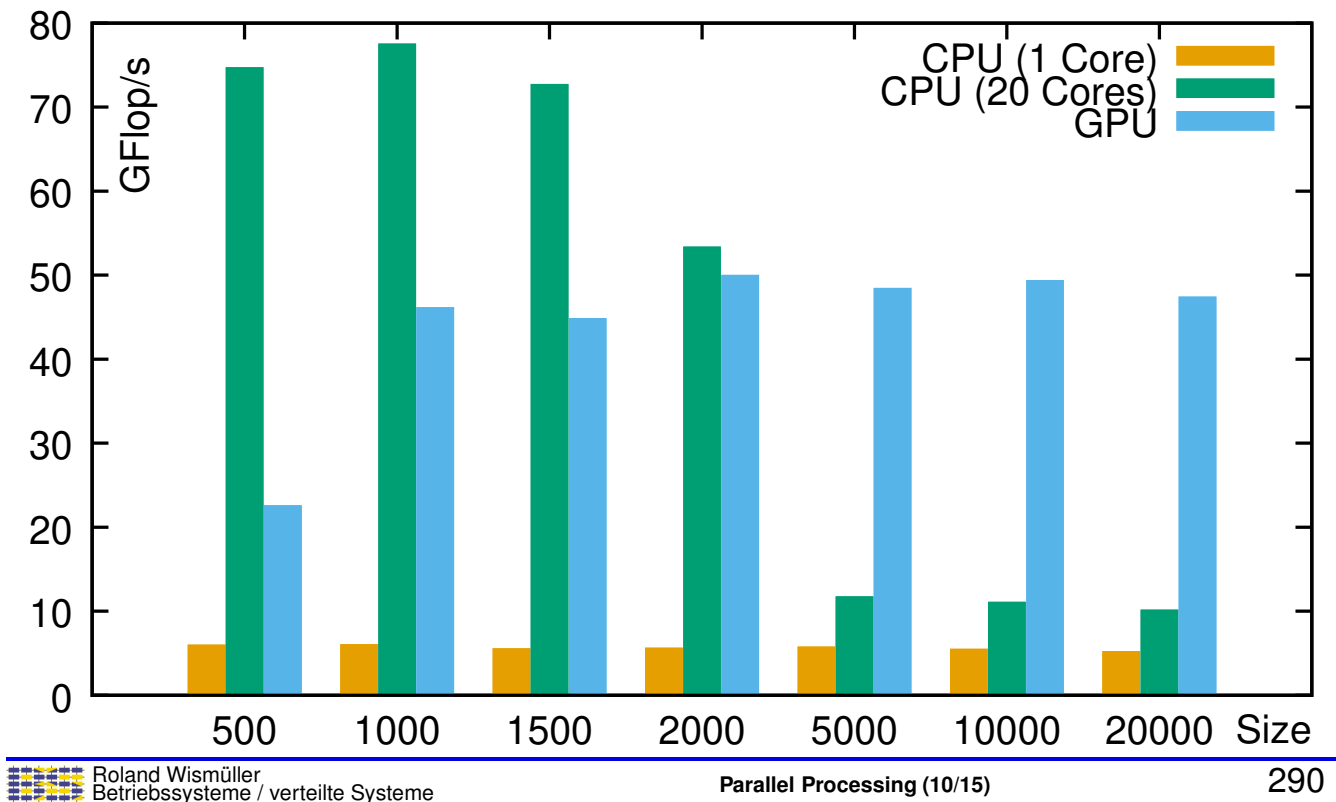
➔ Important clauses:

- ➔ `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 ...



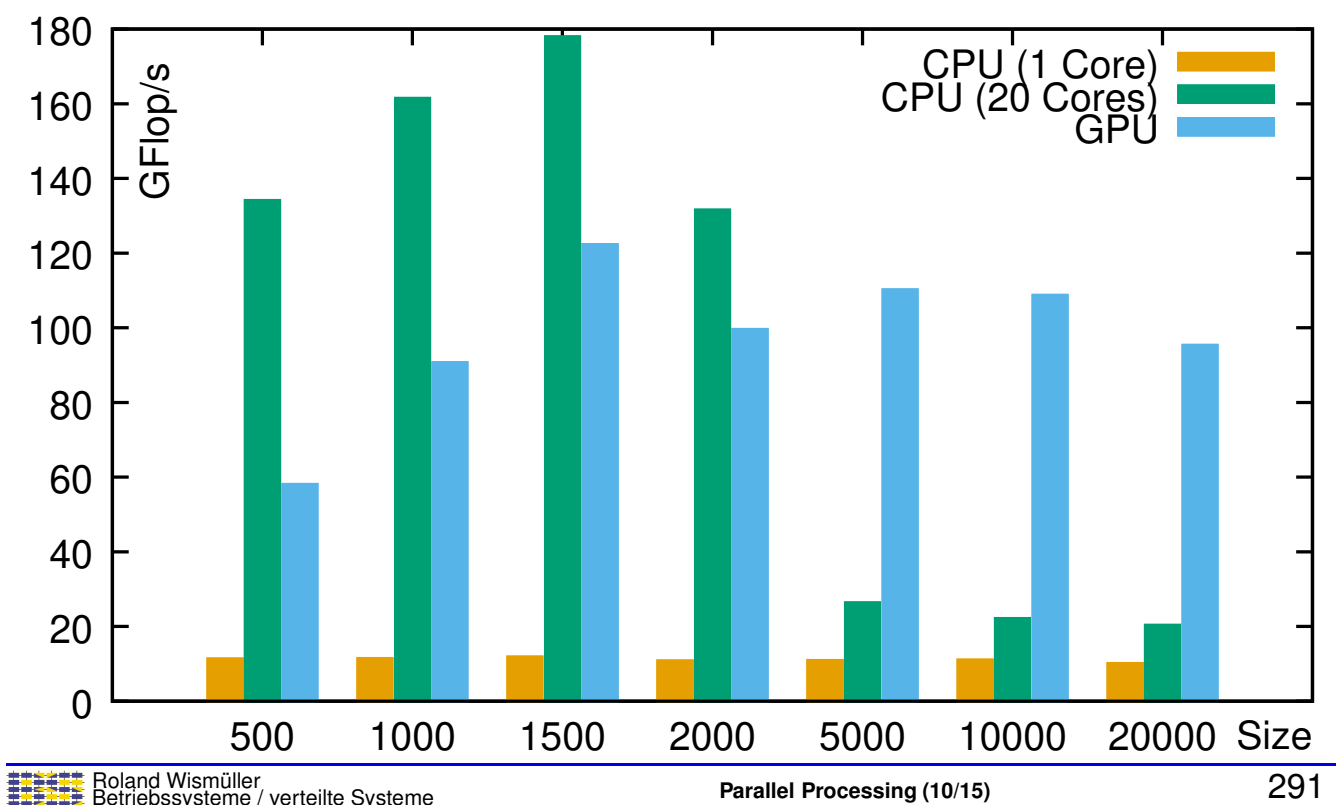
Jacobi: Performance Comparison (nvc)



3.6.3 Using External Accelerators ...



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



Notes for slide 291:

For comparison, the relevant performance data of the RTX 4060 are:

- ➔ Peak performance single precision (32 Bit): 15 TFlop/s
- ➔ Peak performance double precision (64 Bit): 236 GFlop/s
- ➔ Memory bandwidth: 272 GB/s

291-1

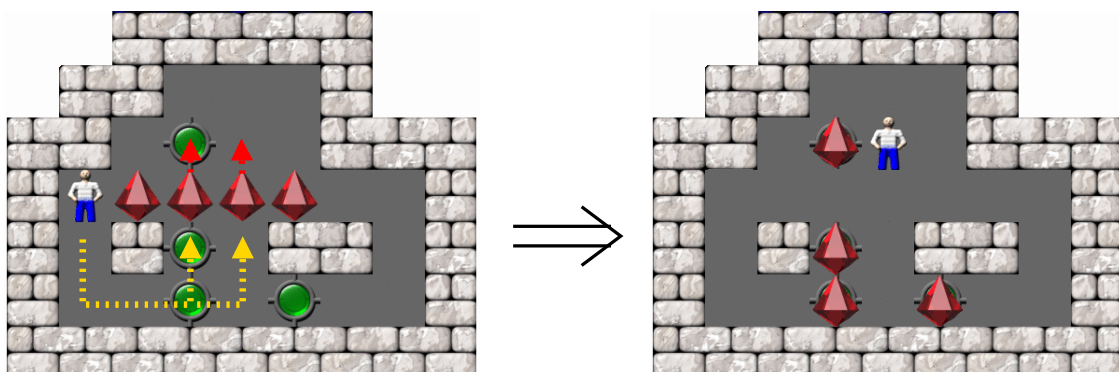
3.7 Exercise: A Solver for the Sokoban Game



(Animated slide)

Background

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



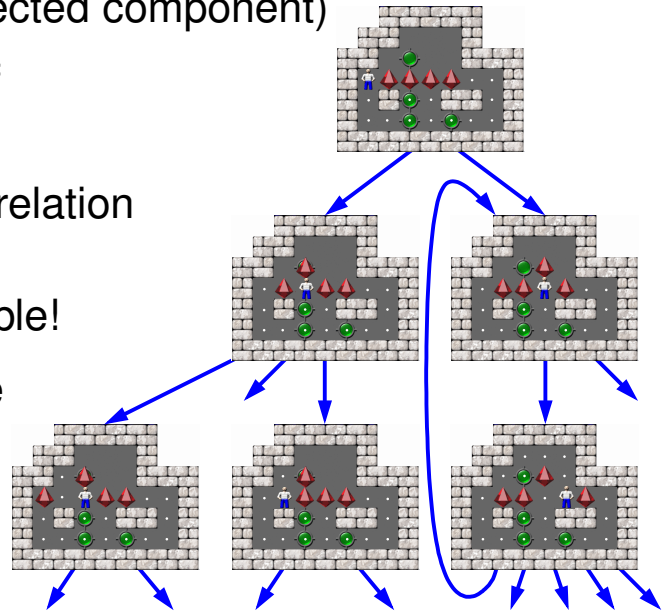
3.7 Exercise: A Solver for the Sokoban Game ...



(Animated slide)

How to find the sequence of moves?

- ➔ Configuration: state of the play field
 - ➔ positions of the boxes
 - ➔ position of the player (connected component)
- ➔ Each configuration has a set of successor configurations
- ➔ Configurations with successor relation build a directed graph
 - ➔ 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



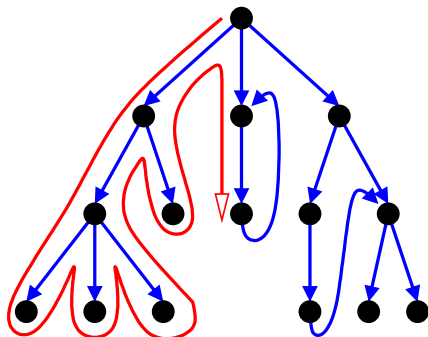
3.7 Exercise: A Solver for the Sokoban Game ...



How to find the sequence of moves? ...

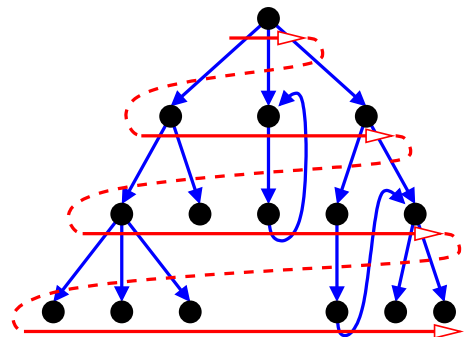
- ➔ Two alternatives:

- ➔ 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:

```

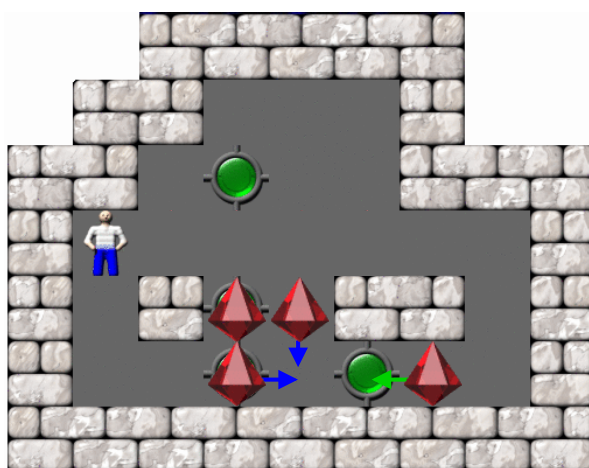
DepthFirstSearch(conf): // conf = current configuration
  append conf to the solution path
  if conf is a solution configuration:
    found the solution path
    return
  if current depth ≥ depth of the best solution so far:
    remove the last element from the solution path
    return // cancel the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



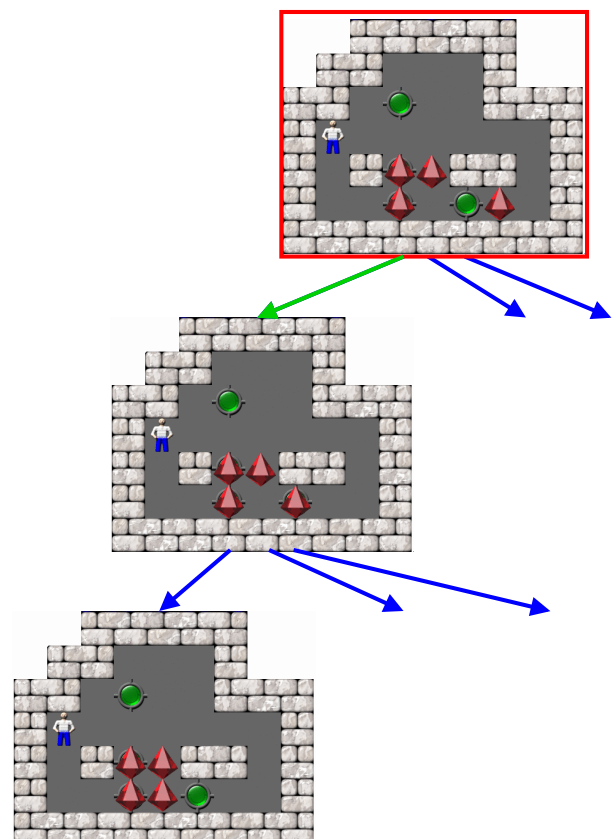
(Animated slide)

Example

Configuration with possible moves



← 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 Parallel Programming with Shared Memory ...



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`

Notes for slide 301:

Starting with OpenMP 5.1, the atomic functions used in the examples can be implemented like this:

```
int fetch_and_add(int *addr, int val) {  
    int tmp;  
    #pragma omp atomic capture  
    {  
        tmp = *addr;  
        *addr += val;  
    }  
    return tmp;  
}
```

```
bool compare_and_swap(void **addr, void *exp, void *val) {  
    bool res;  
    #pragma omp atomic compare capture  
    {  
        res = *addr == exp;  
        if (res)  
            *addr = val;  
    }  
    return res;  
}
```

Parallel Processing

Winter Term 2025/26

4 Parallel Programming with Message Passing

4 Parallel Programming with Message Passing ...



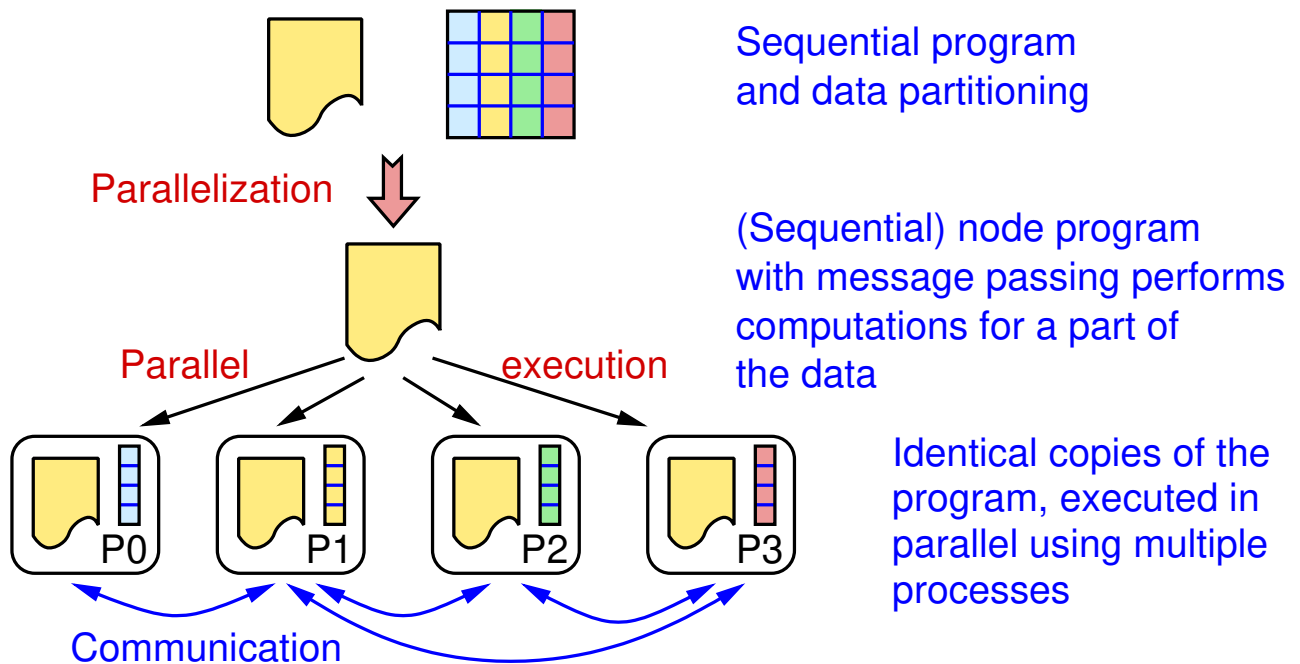
Contents

- ➔ Typical approach
- ➔ MPI (*Message Passing Interface*)
- ➔ MPI core routines
- ➔ Simple MPI programs
- ➔ Point-to-point communication
- ➔ Complex data types in messages
- ➔ Communicators
- ➔ Collective operations
- ➔ Exercise: Jacobi and Gauss/Seidel with MPI
- ➔ Further concepts

4.1 Typical approach



Data partitioning with SPMD model



4.1 Typical approach ...



Activities when creating a node program

- ➔ Adjustment of array declarations
 - ➔ node program stores only a part of the data
 - ➔ (assumption: data are stored in arrays)
- ➔ Index transformation
 - ➔ global index \leftrightarrow (process number, local index)
- ➔ Work partitioning
 - ➔ each process executes the computations on its part of the data
- ➔ Communication
 - ➔ when a process needs non-local data, a suitable message exchange must be programmed

4.1 Typical approach ...



About communication

- ➔ When a process needs data: the owner of the data must send them explicitly
 - ➔ exception: one-sided communication (👉 4.10)
- ➔ Communication should be merged as much as possible
 - ➔ one large message is better than many small ones
 - ➔ however, data dependences must not be violated

Sequential execution

```
a[1] = ...;  
a[2] = ...;  
a[3] = a[1]+...;  
a[4] = a[2]+...;
```

Parallel execution

Process 1

```
a[1] = ...;  
a[2] = ...;  
send(a[1], a[2]);
```

Process 2

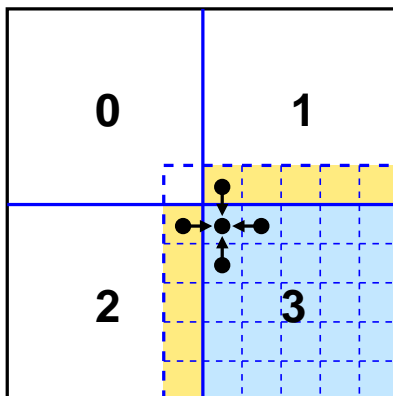
```
recv(a[1], a[2]);  
a[3] = a[1]+...;  
a[4] = a[2]+...;
```

4.1 Typical approach ...



About communication ...

- ➔ Often the node program allocates an overlapping buffer region (**ghost region** / **ghost cells**) for non-local data
- ➔ Example: Jacobi iteration



Partitioning of the matrix into 4 parts

Each process allocates an additional row/column at the borders of its sub-matrix

Data exchange at the end of each iteration



History and background

- ➔ At the beginning of the parallel computer era (late 1980's):
 - many different communication libraries (NX, PARMACS, PVM, P4, ...)
 - parallel programs are not easily portable
- ➔ Definition of an informal standard by the MPI forum
 - 1994: MPI-1.0
 - 1997: MPI-1.2 and MPI-2.0 (considerable extensions)
 - 2009: MPI 2.2 (clarifications, minor extensions)
 - 2012/15: MPI-3.0 und MPI-3.1 (considerable extensions)
 - documents at <http://www.mpi-forum.org/docs>
- ➔ MPI only defines the API (i.e., the programming interface)
 - different implementations, e.g., MPICH2, OpenMPI, ...



Programming model

- ➔ Distributed memory, processes with message passing
- ➔ SPMD: one program code for all processes
 - but different program codes are also possible
- ➔ MPI-1: static process model
 - all processes are created at program start
 - program start is standardized since MPI-2
 - MPI-2 also allows to create new processes at runtime
- ➔ MPI is thread safe: a process is allowed to create additional threads
 - hybrid parallelization using MPI and OpenMP is possible
- ➔ Program terminates when all its processes have terminated



- ➔ MPI-1.2 has 129 routines (and MPI-2 even more ...)
- ➔ However, often only 6 routines are sufficient to write relevant programs:
 - ➔ `MPI_Init` – MPI initialization
 - ➔ `MPI_Finalize` – MPI cleanup
 - ➔ `MPI_Comm_size` – get number of processes
 - ➔ `MPI_Comm_rank` – get own process number
 - ➔ `MPI_Send` – send a message
 - ➔ `MPI_Recv` – receive a message



MPI_Init

```
int MPI_Init(int *argc, char ***argv)
```

INOUT `argc` Pointer to `argc` of `main()`

INOUT `argv` Pointer to `argv` of `main()`

Result `MPI_SUCCESS` or error code

- ➔ Each MPI process must call `MPI_Init`, before it can use other MPI routines
- ➔ Typically:

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    ...
}
```
- ➔ `MPI_Init` may also ensure that all processes receive the command line arguments



Parallel Processing

Winter Term 2025/26

01.12.2025

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

Stand: January 8, 2026

Announcements (1)



Exam

- ➔ Expected date: **Mon., March 2nd, 2026, 09:00, Room PB-C 101**
 - ➔ Duration: 60 minutes
- ➔ Written **electronic** exam **in presence**
- ➔ Open book exam
 - ➔ you can use books, scripts, etc.
 - ➔ but no communication with other students, no Internet
 - ➔ you also get a summary of important **OpenMP pragmas** and **MPI routines**
- ➔ Link to a demo exam: [see web page](#)
 - ➔ choose “Demo Prüfung Parallelverarbeitung”
 - ➔ length of demo exam is less than the length of the real exam
- ➔ **Registration deadline:** 14 days before the exam!



Evaluation

- ➔ You can evaluate this course (lecture + lab), starting today until Dec. 19th
- ➔ Time for evaluation:
 - in the lecture on Dec. 8th
 - be sure to bring a suitable device
- ➔ Or fill the evaluation form at home
- ➔ Link: <https://evasys.zv.uni-siegen.de/evasys/online.php?p=2H2Q1>



4.3 MPI Core routines ...



MPI_Finalize

```
int MPI_Finalize()
```

- ➔ Each MPI process must call `MPI_Finalize` at the end
- ➔ Main purpose: deallocation of resources
 - e.g.: closing communication links
- ➔ After that, no other MPI routines must be used
 - in particular, no further `MPI_Init`
- ➔ `MPI_Finalize` does **not** terminate the process!



MPI_Comm_size

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

IN **comm** Communicator

OUT **size** Number of processes in **comm**

- ➔ Typically: `MPI_Comm_size(MPI_COMM_WORLD, &nprocs)`
- ➔ returns the number of MPI processes in `nprocs`

MPI_Comm_rank

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

IN **comm** Communicator

OUT **rank** Number of processes in **comm**

- ➔ Process number (“rank”) counts upward, starting at 0
- ➔ only differentiation of the processes in the SPMD model



Communicators

- ➔ A communicator consists of
 - ➔ a process group
 - ➔ a subset of all processes of the parallel application
 - ➔ a communication context
 - ➔ to allow the separation of different communication relations (☞ 4.6)
- ➔ There is a predefined communicator `MPI_COMM_WORLD`
 - ➔ its process group contains all processes of the parallel application
- ➔ Additional communicators can be created as needed (☞ 4.6)



MPI_Send

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
             int dest, int tag, MPI_Comm comm)
```

IN **buf** (Pointer to) the data to be sent (send buffer)
IN **count** Number of data elements (of type **dtype**)
IN **dtype** Data type of the individual data elements
IN **dest** Rank of destination process in communicator **comm**
IN **tag** Message tag
IN **comm** Communicator

- ➔ Specification of data type: for format conversions
- ➔ Destination process is always relative to a communicator
- ➔ *Tag* allows to distinguish different messages (or message types) in the program



MPI_Send ...

- ➔ MPI_Send blocks the calling thread* at least until all data has been read from the send buffer
 - ➔ send buffer can be reused (i.e., modified) immediately after MPI_Send returns
- ➔ The MPI implementation decides whether the thread is blocked until
 - a) the data has been copied to a system buffer, or
 - b) the data has been received by the destination process.
 - ➔ in some cases, this decision can influence the correctness of the program! (👉 slide 329)

* Remember that MPI is thread safe!



MPI_Recv

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

OUT **buf** (Pointer to) receive buffer
IN **count** Buffer size (number of data elements of type **dtype**)
IN **dtype** Data type of the individual data elements
IN **source** Rank of source process in communicator **comm**
IN **tag** Message tag
IN **comm** Communicator
OUT **status** Status (among others: actual message length)

- ➔ Calling thread is blocked until the message has been completely received and stored in the receive buffer



MPI_Recv ...

- ➔ MPI_Recv only receives a message where
 - ➔ sender,
 - ➔ message tag, and
 - ➔ communicatormatch the parameters
- ➔ For source process (sender) and message tag, wild-cards can be used:
 - ➔ MPI_ANY_SOURCE: sender doesn't matter
 - ➔ MPI_ANY_TAG: message tag doesn't matter



MPI_Recv ...

- ➔ Message must not be larger than the receive buffer
 - ➔ but it may be smaller; the unused part of the buffer remains unchanged
- ➔ From the return value `status` you can determine:
 - ➔ the sender of the message: `status.MPI_SOURCE`
 - ➔ the message tag: `status.MPI_TAG`
 - ➔ the error code: `status.MPI_ERROR`
 - ➔ the actual length of the received message (number of data elements): `MPI_Get_count(&status, dtype, &count)`



Simple data types (MPI_Datatype)

MPI	C/C++	MPI	C/C++
MPI_CHAR	char	MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	short	MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	int	MPI_UNSIGNED	unsigned int
MPI_LONG	long	MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float		
MPI_DOUBLE	double	MPI_LONG_DOUBLE	long double
MPI_BYTE	Byte with 8 bits	MPI_PACKED	Packed data*

*  4.9

4.4 Simple MPI programs



Example: typical MPI program skeleton (📄 04/rahmen.cpp)

```
#include <iostream>
#include <mpi.h>
using namespace std;

int main (int argc, char **argv)
{
    int i;
    int myrank, nprocs;
    int namelen;
    char name [MPI_MAX_PROCESSOR_NAME];

    // Initialize MPI and set the command line arguments
    MPI_Init(&argc, &argv);

    // Determine the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

4.4 Simple MPI programs ...



```
    // Determine the own rank
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Determine the node name
    MPI_Get_processor_name(name, &namelen);

    // 'flush' is used to enforce immediate output
    cout << "Process " << myrank << "/" << nprocs
         << "started on " << name << "\n" << flush;

    cout << "-- Arguments: ";
    for (i = 0; i<argc; i++)
        cout << argv[i] << " ";
    cout << "\n";

    // Finish MPI
    MPI_Finalize();

    return 0;
}
```

4.4 Simple MPI programs ...



Starting MPI programs: `mpiexec`

- ➔ `mpiexec -n 3 myProg arg1 arg2`
 - ➔ starts `myProg arg1 arg2` with 3 processes
 - ➔ the specification of the nodes to be used depends on the MPI implementation and the hardware/OS platform

- ➔ Starting the example program using MPICH:

```
mpiexec -n 3 -machinefile machines ./rahmen a1 a2
```

- ➔ Output:

```
Process 0/3 started on bslab02.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
Process 2/3 started on bslab03.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
Process 1/3 started on bslab06.lab.bvs
Args: /home/wismueller/LEHRE/pv/CODE/04/rahmen a1 a2
```

4.4 Simple MPI programs ...



Example: ping pong with messages (04/pingpong.cpp)

```
int main (int argc, char **argv)
{
    int i, passes, size, myrank;
    char *buf;
    MPI_Status status;
    double start, end;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    passes = atoi(argv[1]); // Number of repetitions
    size = atoi(argv[2]); // Message length
    buf = new char[size];
```

4.4 Simple MPI programs ...



```
if (myrank == 0) {      // Process 0
    start = MPI_Wtime(); // Get the current time
    for (i=0; i<passes; i++) {
        // Send a message to process 1, tag = 42
        MPI_Send(buf, size, MPI_CHAR, 1, 42, MPI_COMM_WORLD);

        // Wait for the answer, tag is not relevant
        MPI_Recv(buf, size, MPI_CHAR, 1, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);
    }

    end = MPI_Wtime(); // Get the current time

    cout << "Time for one message: "
          << ((end - start) * 1e6 / (2 * passes)) << "us\n";
    cout << "Bandwidth: "
          << (size*2*passes/(1024*1024*(end-start))) << "MB/s\`
}
}
```

4.4 Simple MPI programs ...



```
else { // Process 1
    for (i=0; i<passes; i++) {
        // Wait for the message from process 0, tag is not relevant
        MPI_Recv(buf, size, MPI_CHAR, 0, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &status);

        // Send back the answer to process 0, tag = 24
        MPI_Send(buf, size, MPI_CHAR, 0, 24, MPI_COMM_WORLD);
    }
}

MPI_Finalize();
return 0;
}
```

4.4 Simple MPI programs ...



Example: ping pong with messages ...

➔ Results (in the lab H-A 4111):

- `mpiexec -n 2/pingpong 1000 1`
Time for one message: 85.1829 us
Bandwidth: 0.0111956 MB/s
- `mpiexec -n 2/pingpong 1000 1000`
Time for one message: 155.584 us
Bandwidth: 6.12966 MB/s
- `mpiexec -n 2/pingpong 100 1000000`
Time for one message: 8809.63 us
Bandwidth: 108.254 MB/s

➔ (Only) with large messages the bandwidth of the interconnection network is reached

➤ Lab: 1 GBit/s Ethernet ($\hat{=}$ 119.2 MB/s)

4.4 Simple MPI programs ...



Additional MPI routines in the examples:

```
int MPI_Get_processor_name(char *name, int *len)
```

OUT **name** Pointer to buffer for node name

OUT **len** Length of the node name

Result **MPI_SUCCESS** or error code

➔ The buffer for node name should have the length
`MPI_MAX_PROCESSOR_NAME`

```
double MPI_Wtime()
```

Result Current wall clock time in seconds

➔ for timing measurements

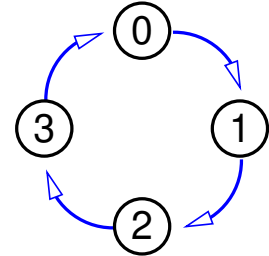
➔ in MPICH: time is synchronized between the nodes

4.5 Point-to-point communication



Example: sending in a closed cycle (04/ring.cpp)

```
int a[N];
...
MPI_Send(a, N, MPI_INT, (myrank+1) % nprocs,
         0, MPI_COMM_WORLD);
MPI_Recv(a, N, MPI_INT,
        (myrank+nprocs-1) % nprocs,
        0, MPI_COMM_WORLD, &status);
```



- ➔ Each process first attempts to send, before it receives
- ➔ This works **only if** MPI buffers the messages
- ➔ But MPI_Send can also block until the message is received
 - ➔ deadlock!

4.5 Point-to-point communication ...



Example: sending in a closed cycle (correct)

- ➔ Some processes must first receive, before they send

```
int a[N];
...
if (myrank % 2 == 0) {
    MPI_Send(a, N, MPI_INT, (myrank+1)%nprocs, ...
    MPI_Recv(a, N, MPI_INT, (myrank+nprocs-1)%nprocs, ...
}
else {
    MPI_Recv(a, N, MPI_INT, (myrank+nprocs-1)%nprocs, ...
    MPI_Send(a, N, MPI_INT, (myrank+1)%nprocs, ...
}
```

- ➔ Better: use non-blocking operations




Non-blocking communication

- ➔ MPI_Isend and MPI_Irecv return immediately
 - ➔ before the message actually has been sent / received
 - ➔ result: request object (MPI_Request)
 - ➔ send / receive buffer must not be modified / used, until the communication is completed
- ➔ MPI_Test checks whether communication is completed
- ➔ MPI_Wait blocks, until communication is completed
- ➔ Allows to overlap communication and computation
- ➔ can be “mixed” with blocking communication
 - ➔ e.g., send usgin MPI_Send, receive using MPI_Irecv

4.5 Point-to-point communication ...



Example: sending in a closed cycle with MPI_Irecv

( 04/ring2.cpp)

```
int sbuf [N];
int rbuf [N];
MPI_Status status;
MPI_Request request;
...

// Set up the receive request
MPI_Irecv(rbuf, N, MPI_INT, (myrank+nprocs-1) % nprocs, 0,
          MPI_COMM_WORLD, &request);

// Sending
MPI_Send(sbuf, N, MPI_INT, (myrank+1) % nprocs, 0,
         MPI_COMM_WORLD);

// Wait for the message being received
MPI_Wait(&request, &status);
```

Notes for slide 332:

MPI offers many different variants for point-to-point communication:

- ➔ For sending, there are four modes:
 - **synchronous**: send operation blocks, until message is received
 - rendez-vous between sender and receiver
 - **buffered**: message will be buffered by the sender
 - application must allocate and register the buffer
 - **ready**: the programmer must guarantee that the receiver process already waits for the message (allows optimized sending)
 - **standard**: MPI decides whether synchronous or buffered
 - in this case, MPI provides the buffer itself
- ➔ In addition: sending can be blocking or non-blocking
- ➔ For receiving of messages: only blocking and non-blocking variant

332-1

➔ The following table summarizes all routines:

		synchronous	asynchronous
Sending	synchronous	MPI_Ssend()	MPI_Issend()
	buffered	MPI_Bsend()	MPI_Ibsend()
	ready	MPI_Rsend()	MPI_Irsend()
	standard	MPI_Send()	MPI_Isend()
Receiving		MPI_Recv()	MPI_Irecv()

332-2

- ➔ In addition, MPI also has a routine `MPI_Sendrecv`, which allows to send and receive at the same time, without the possibility of a deadlock. Using this function, the example from (👉 04/ring1.cpp) looks like:

```

int sbuf[N];
int rbuf[N];
MPI_Status status;
...

MPI_Sendrecv(sbuf, N, MPI_INT, (myrank+1) % nprocs, 0,
             rbuf, N, MPI_INT, (myrank+nprocs-1) % nprocs, 0,
             MPI_COMM_WORLD, &status);

```

- ➔ When using `MPI_Sendrecv`, send and receive buffer must be different, when using `MPI_Sendrecv_replace` the send buffer is overwritten with the received message.

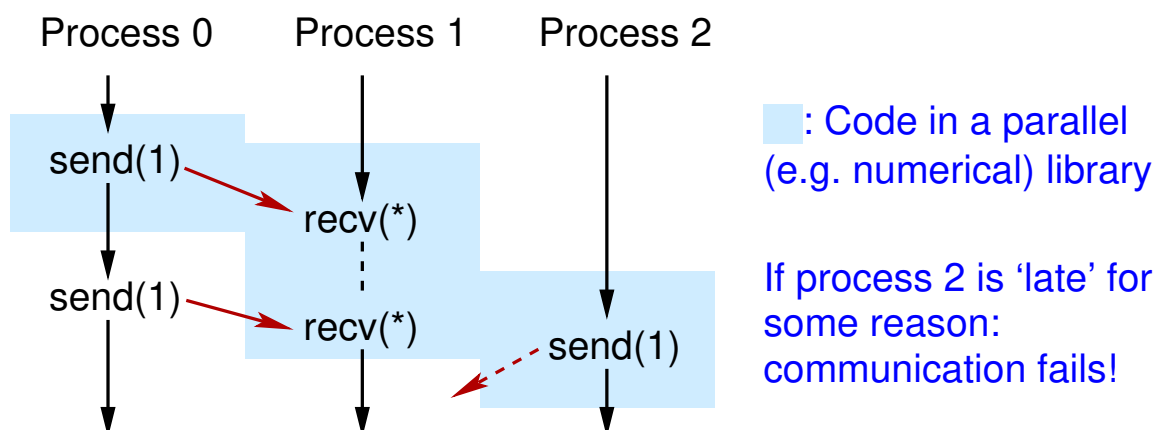
332-3

4.6 Communicators



(Animated slide)

Motivation: problem of earlier communication libraries



- ➔ Message tags are not a reliable solution
 - ➔ tags might be chosen identically by chance!
- ➔ Required: different communication contexts



- ➔ Communicator = process group + context
- ➔ Communicators support
 - working with process groups
 - task parallelism
 - coupled simulations
 - collective communication with a subset of all processes
 - communication contexts
 - for parallel libraries
- ➔ A communicator represents a communication domain
 - communication is possible only within the same domain
 - no wild-card for communicator in `MPI_Recv`
 - a process can belong to several domains at the same time



Creating new communicators

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
int MPI_Comm_split(MPI_Comm comm, int color
                  int key, MPI_Comm *newcomm)
```

- ➔ Collective operations (👉 4.7)
 - all processes in `comm` must execute them concurrently
- ➔ `MPI_Comm_dup` creates a copy with a new context
- ➔ `MPI_Comm_split` splits `comm` into several communicators
 - one communicator for each value of `color`
 - as the result, each process receives the communicator to which it was assigned
 - `key` determines the order of the new process ranks



Example for MPI_Comm_split

- ➔ *Multi-physics code*: air pollution
 - one half of the processes computes the airflow
 - the other half computes chemical reactions
- ➔ Creation of two communicators for the two parts:

```
MPI_Comm_split(MPI_COMM_WORLD, myrank%2, myrank, &comm)
```

Process	myrank	Color	Result in comm	Rank in C_0	Rank in C_1
P0	0	0	C_0	0	–
P1	1	1	C_1	–	0
P2	2	0	C_0	1	–
P3	3	1	C_1	–	1

4.7 Collective operations



- ➔ Collective operations in MPI
 - must be executed concurrently by all processes of a process group (a communicator)
 - are blocking
 - do not necessarily result in a global (barrier) synchronisation, however
- ➔ Collective synchronisation and communication functions
 - barriers
 - reductions (communication with aggregation)
 - global communication: broadcast, scatter, gather, ...

Notes for slide 337:

Note that “concurrently” (German: “nebenläufig”) does not mean that the operations must be executed at the same time, or in an overlapping way. It just means that (1) all processes in the communicator execute the operation and (2) there is no synchronization that enforces any restriction on the ordering of the operations. (In other words: it must be **possible** that the operations can be executed at the same time, but this is not required)

337-1

4.7 Collective operations ...



MPI_Barrier

```
int MPI_Barrier(MPI_Comm comm)
```

- ➔ Barrier synchronization of all processes in `comm`
- ➔ With message passing, barriers are actually not really necessary
 - ➔ synchronization is achieved by message exchange
- ➔ Reasons for barriers:
 - ➔ more easy understanding of the program
 - ➔ timing measurements, debugging output
 - ➔ console output ??
 - ➔ MPI-2: MPI I/O, one-sided communication

Notes for slide 338:

In principle, a barrier may be used to order the console output of different processes, as in:

```
if (myrank == 0)
    cout << "This is some output of process 0" << endl;
MPI_Barrier(MPI_COMM_WORLD);
if (myrank == 1)
    cout << "This is some output of process 1" << endl;
```

However, the barrier only ensures that the execution of the 'cout << ...' statements is ordered. Since the output is sent to the console as messages, there is still a chance that these messages arrive in a different order, i.e., the output is not ordered as expected.

338-1

4.7 Collective operations ...



Reduction: MPI_Reduce

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
               int count, MPI_Datatype dtype,
               MPI_Op op, int root,
               MPI_Comm comm)
```

- ➔ Each element in the receive buffer is the result of a reduction operation (e.g., the sum) of the corresponding elements in the send buffer
- ➔ op defines the operation
 - ➔ predefined: minimum, maximum, sum, product, AND, OR, XOR, ...
 - ➔ in addition, user defined operations are possible, too

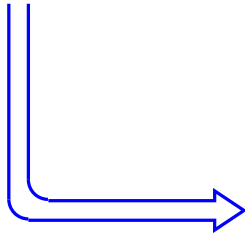
4.7 Collective operations ...



Example: summing up an array

Sequential

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



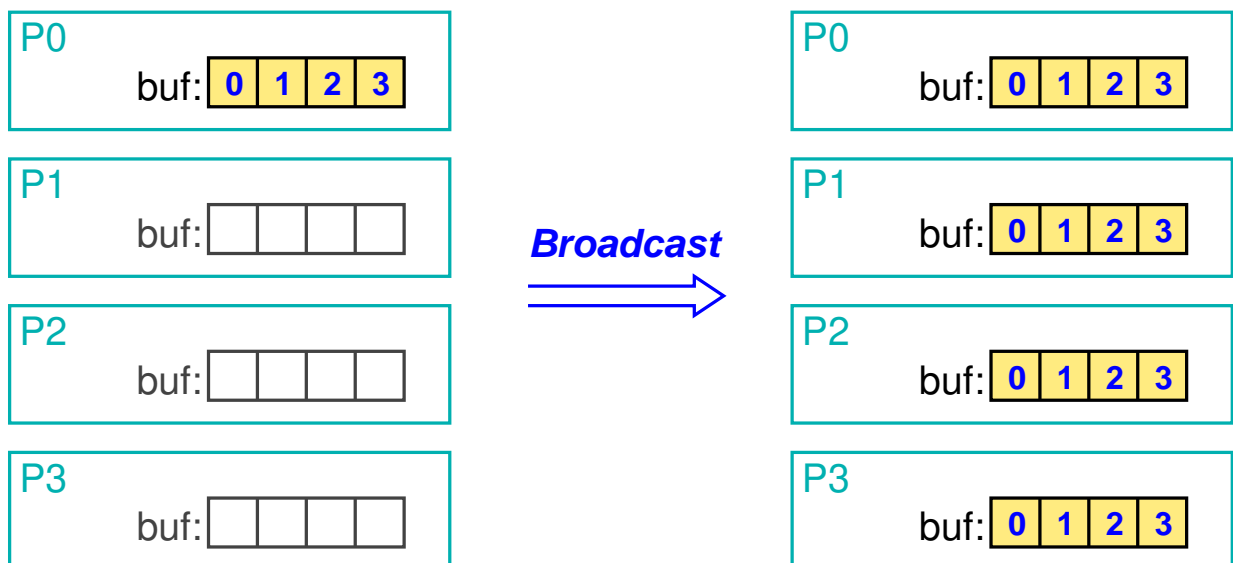
Parallel

```
local_s = 0;  
for (i=0; i<local_size; i++)  
    local_s += a[i];  
  
MPI_Reduce(&local_s, &s,  
          1, MPI_INT,  
          MPI_SUM,  
          0, MPI_COMM_WORLD);
```

4.7 Collective operations ...



Collective communication: broadcast





MPI_Bcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dtype,  
             int root, MPI_Comm comm)
```

IN *root* Rank of the sending process

- ➔ Buffer is sent by process *root* and received by all others
- ➔ Collective, blocking operation: no tag necessary
- ➔ *count*, *dtype*, *root*, *comm* must be the same in all processes



Parallel Processing

Winter Term 2025/26

08.12.2025

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

Stand: January 8, 2026

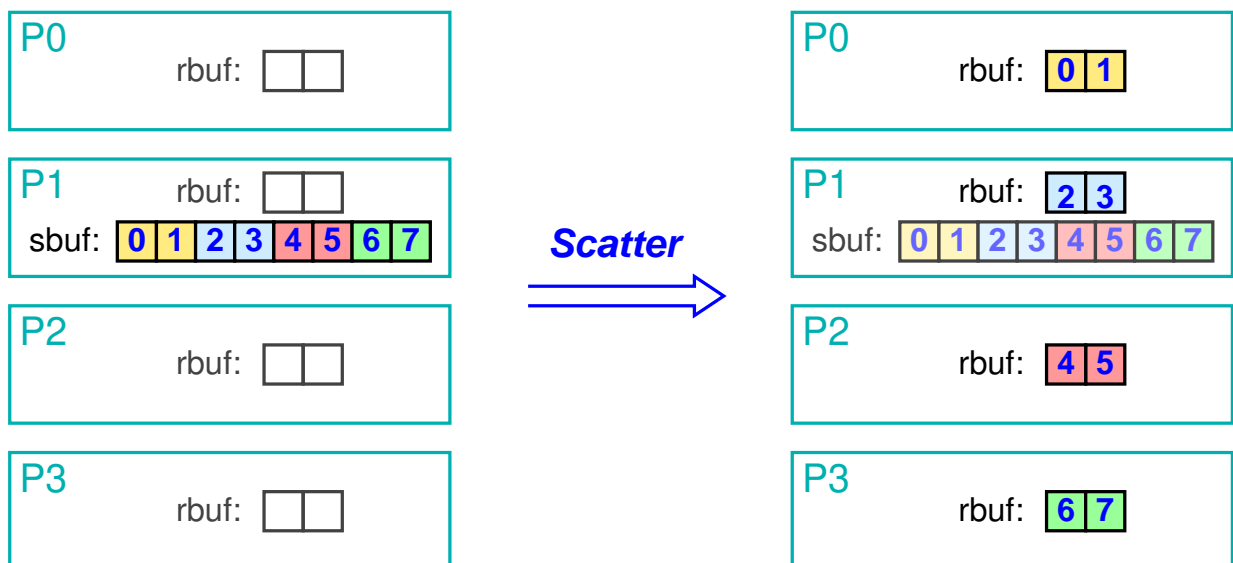


<https://evasys.zv.uni-siegen.de/evasys/online.php?p=2H2Q1>

4.7 Collective operations ...



Collective communication: scatter



MPI_Scatter

```
int MPI_Scatter(void *sendbuf, int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf, int recvcount,
               MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

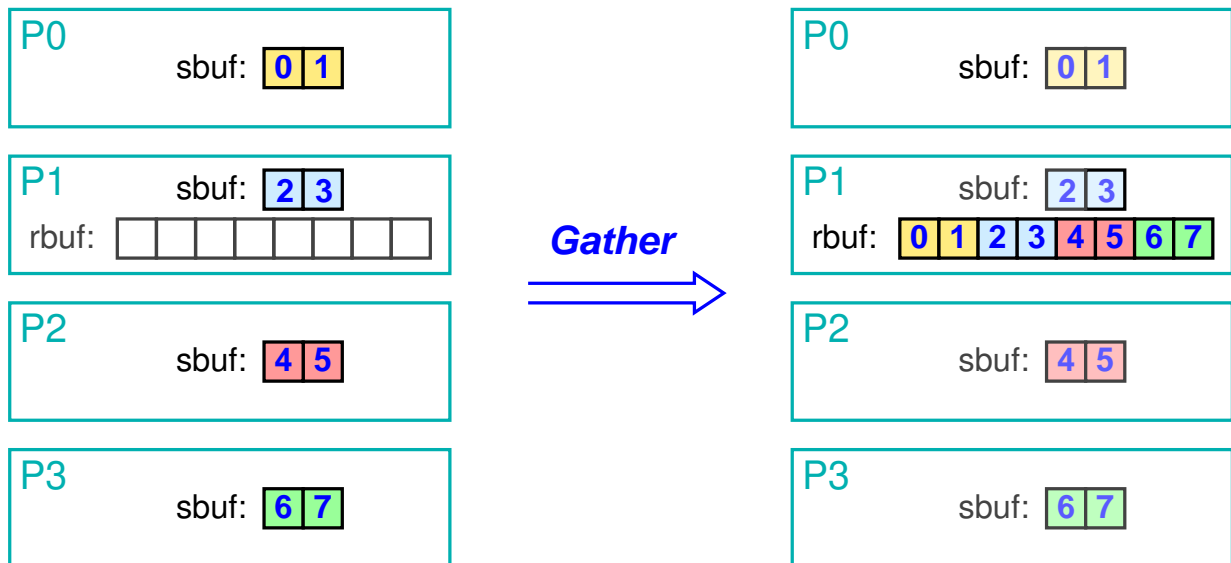
- ➔ Process `root` sends a part of the data to each process
 - ➔ including itself
- ➔ `sendcount`: data length for each process (not the total length!)
- ➔ Process `i` receives `sendcount` elements of `sendbuf` starting from position `i * sendcount`
- ➔ Alternative `MPI_Scatterv`: length and position can be specified individually for each receiver

Notes for slide 344:

- ➔ A problem that may arise when using `MPI_Scatter` is that the data cannot be distributed evenly, e.g., if an array with 1000 elements should be distributed to 16 processes.
- ➔ In `MPI_Scatterv`, the argument `sendcount` is replaced by two arrays `sendcounts` and `displacements`
 - ➔ process `i` then receives `sendcounts[i]` elements of `sendbuf`, starting at position `displacements[i]`



Collective communication: gather



MPI_Gather

```
int MPI_Gather(void *sendbuf, int sendcount,  
              MPI_Datatype sendtype,  
              void *recvbuf, int recvcount,  
              MPI_Datatype recvtype,  
              int root, MPI_Comm comm)
```

- ➔ All processes send sendcount elements to process root
 - ➔ even root itself
- ➔ Important: each process must send the same amount of data
- ➔ root stores the data from process i starting at position $i * \text{recvcount}$ in recvbuf
- ➔ recvcount: data length for each process (not the total length!)
- ➔ Alternative MPI_Gatherv: analogous to MPI_Scatterv

4.7 Collective operations ...



Example: multiplication of vector and scalar (👉 04/vecmult.cpp)

```
double a[N], factor, local_a[LOCAL_N];
... // Process 0 reads a and factor from file
MPI_Bcast(&factor, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(a, LOCAL_N, MPI_DOUBLE, local_a, LOCAL_N,
           MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i=0; i<LOCAL_N; i++)
    local_a[i] *= factor;
MPI_Gather(local_a, LOCAL_N, MPI_DOUBLE, a, LOCAL_N,
           MPI_DOUBLE, 0, MPI_COMM_WORLD);
... // Process 0 writes a into file
```

- ➔ **Caution:** LOCAL_N must have the same value in all processes!
 - ➔ otherwise: use MPI_Scatterv / MPI_Gatherv (👉 04/vecmult3.cpp)

4.7 Collective operations ...



More collective communication operations

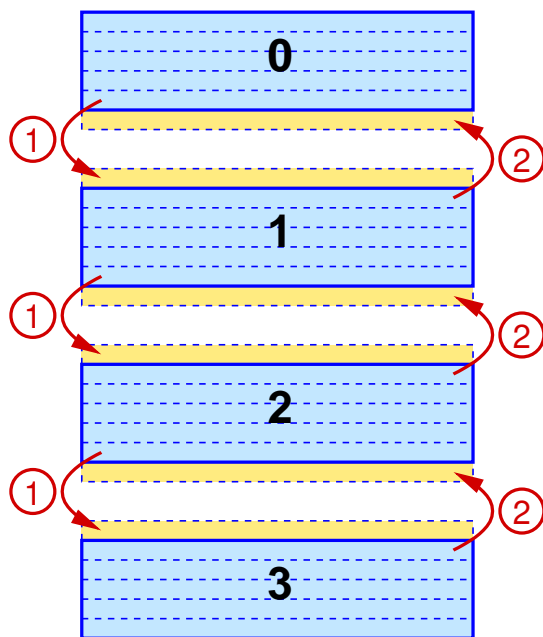
- ➔ MPI_Alltoall: all-to-all broadcast (👉 2.9.5)
- ➔ MPI_Allgather and MPI_Allgatherv: at the end, all processes have the gathered data
 - ➔ corresponds to a gather with subsequent broadcast
- ➔ MPI_Allreduce: at the end, all processes have the result of the reduction
 - ➔ corresponds to a reduce with subsequent broadcast
- ➔ MPI_Scan: prefix reduction
 - ➔ e.g., using the sum: process i receives the sum of the data from processes 0 up to and including i

4.8 Exercise: Jacobi and Gauss/Seidel with MPI



(Animated slide)

General 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)

4.8 Exercise: Jacobi and Gauss/Seidel with MPI ...



General approach ...

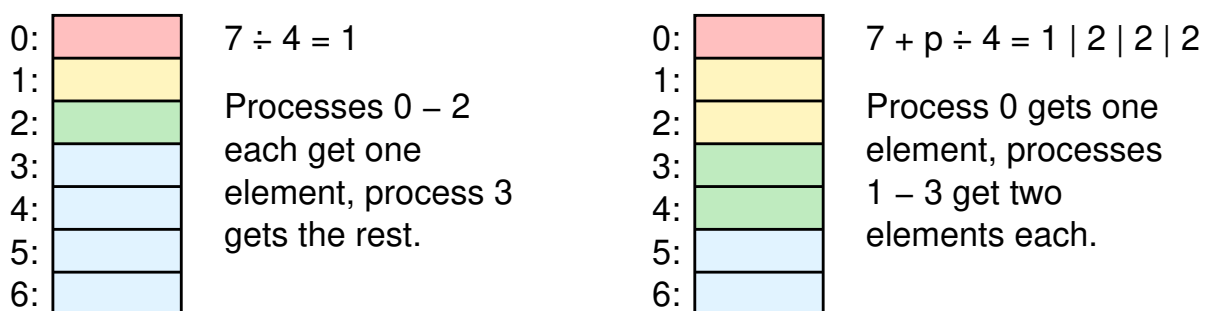
```
int nprocs, myrank;  
double a[LINES][COLS];  
MPI_Status status;  
  
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);  
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
  
// Step 1: Send downwards, receive from above  
if (myrank != nprocs-1)  
    MPI_Send(a[LINES-2], COLS, MPI_DOUBLE, myrank+1, 0,  
      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

Distribution of data

- ➔ For a uniform distribution of an array of length n to np processes:
 - $\text{size}(p) = (n + p) \div np$
 - $\text{start}(p) = \sum_{i=0}^{p-1} \text{size}(i)$
 $= n \div np \cdot p + \max(p - (np - n \bmod np), 0)$
 - process p receives $\text{size}(p)$ elements starting at index $\text{start}(p)$
- ➔ This results in the following index transformation:
 - $\text{tlocal}(i) = (p, i - \text{start}(p))$
 with $p \in [0, np - 1]$ such that $0 \leq i - \text{start}(p) < \text{size}(p)$
 - $\text{tglobal}(p, i) = i + \text{start}(p)$
- ➔ In addition, you have to consider the ghost zones for Jacobi and Gauss/Seidel!

Notes for slide 351:

As a motivation for the formula $\text{size}(p) = (n + p) \div np$, consider the simple example of $n = 7$ and $np = 4$:



When `nprocs` contains the number of processes and `myrank` is the rank of the MPI process, the following code will compute the start row (`start`) and the number of rows (`size`) for the current process:

```
size = (n + myrank) / nprocs;
start = n / nprocs * myrank;
if (myrank > nprocs - n % nprocs)
    start += myrank - (nprocs - n % nprocs);
```

Note that after this computation, you will have to modify these numbers a little, since you also have to account for the ghost rows.

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


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

Notes for slide 352:

For the initialization of the border values it is the easiest method to use conditional execution. So the original loop

```
for (i=0; i<n; i++) {
    double x = (double)i / (n-1);
    a[i][0] = x;
    a[n-1-i][n-1] = x;
    a[0][i] = x;
    a[n-1][n-1-i] = x;
}
```

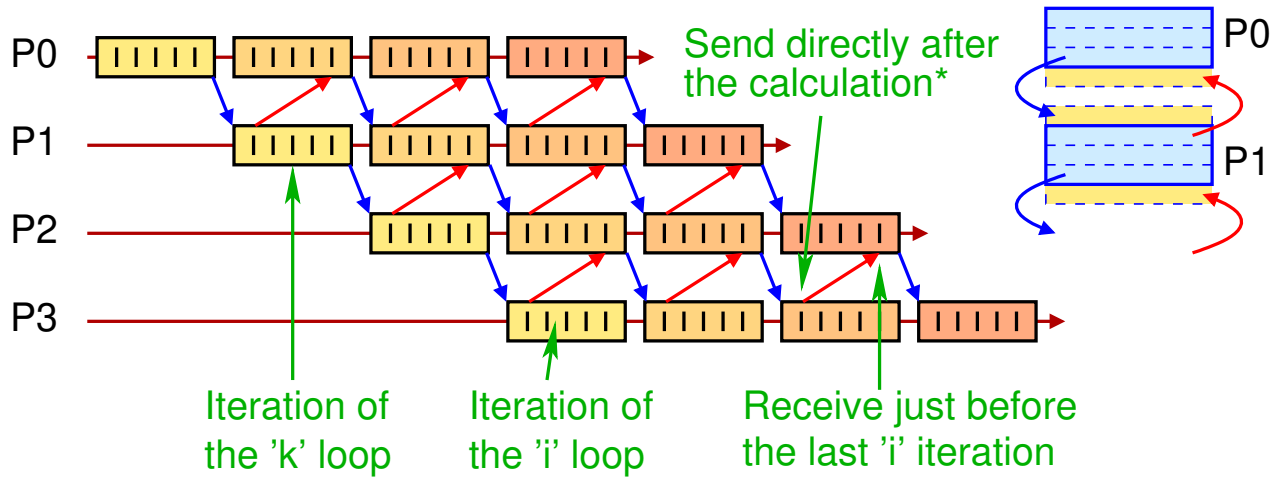
becomes

```
for (i=0; i<n; i++) {
    double x = (double)i / (n-1);
    if ((i-start >= 0) && (i-start < size))
        a[i-start][0] = x;
    if ((n-1-i-start >= 0) && (n-1-i-start < size))
        a[n-1-i-start][n-1] = x;
    if ((0-start >= 0) && (0-start < size))
        a[0-start][i] = x;
    if ((n-1-start >= 0) && (n-1-start < size))
        a[n-1-start][n-1-i] = x;
}
```



On the parallelization of the Gauss/Seidel method

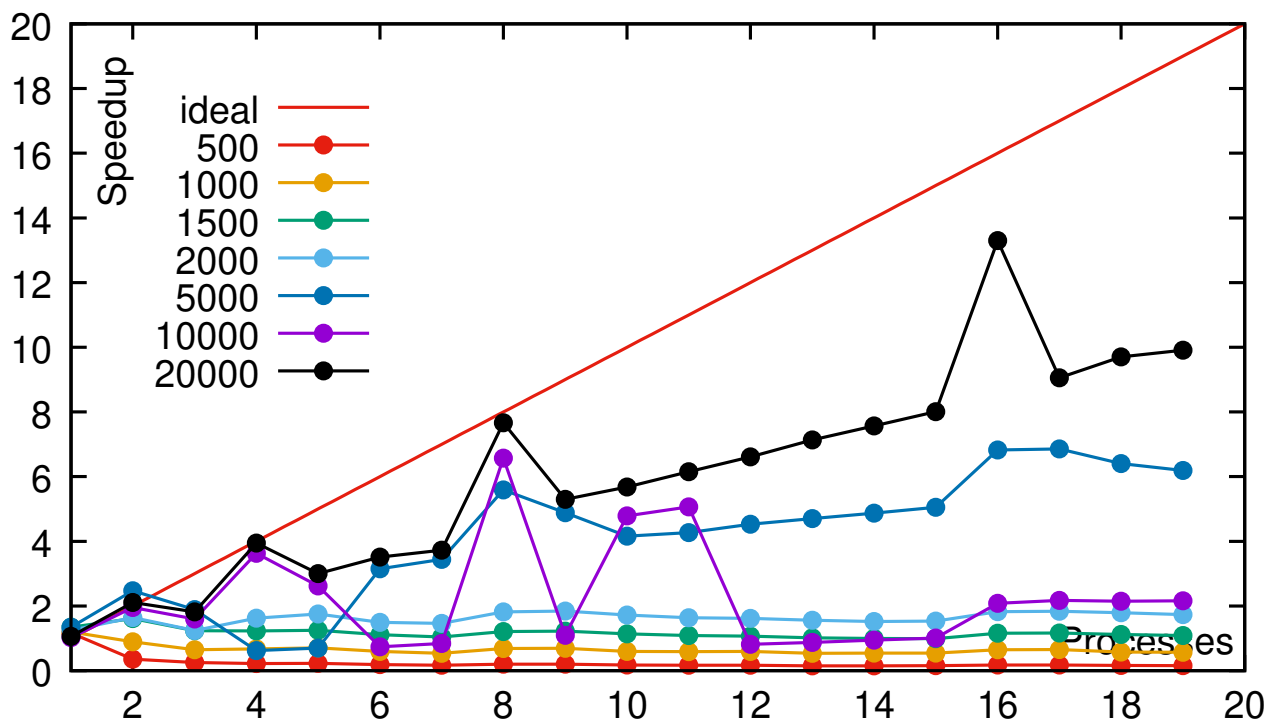
➔ Similar to the pipelined parallelization with OpenMP (👉 3.4)



* If you can ensure that sending doesn't block. E.g., you can post the receive event (MPI_irecv) already after the first 'i' iteration. As an alternative, send later, i.e., just before the last 'i' iteration.



Jacobi: Speedup



Notes for slide 354:

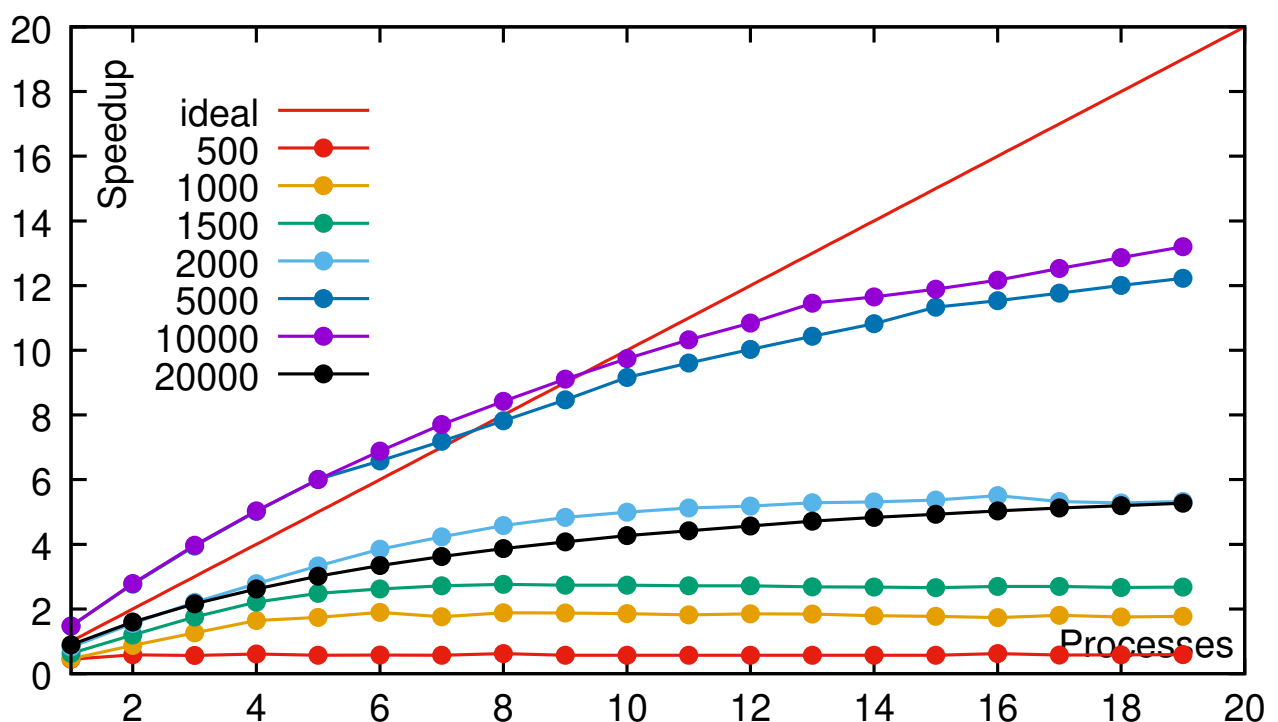
- Results were obtained with optimization level '-O3'.
- The MPI processes were bound to core 0 (performance core) using the taskset command.
- The sequential solver was bound to core 8 (efficiency core) as interestingly it shows a higher performance there as on core 0.
- The MPI program with one process actually runs faster than the sequential program. The reasons are unknown.
- The performance peaks for matrix size 20000 and processor numbers that are a power of two are due to the fact that the MPI_AllReduce function is much faster for these processor numbers.

354-1

4.8 Exercise: Jacobi and Gauss/Seidel with MPI ...



Jacobi: Gauss/Seidel



Notes for slide 355:

- ➔ Results were obtained with optimization level '-O3'.
- ➔ The MPI processes were bound to core 0 (performance core) using the `taskset` command.
- ➔ The sequential solver was bound to core 0 (efficiency core) as for most matrix sizes this gives the higher performance.
- ➔ For some matrix sizes, the MPI program with one process actually runs faster than the sequential program. The reasons are unknown.

355-1

4.9 Complex data types in messages



- ➔ So far: only arrays can be send as messages
- ➔ What about complex data types (e.g., structures)?
 - ➔ z.B. `struct bsp { int a; double b[3]; char c; };`
- ➔ MPI offers two mechanisms
 - ➔ **packing and unpacking** the individual components
 - ➔ use `MPI_Pack` to pack components into a buffer one after another; send as `MPI_PACKED`; extract the components again using `MPI_Unpack`
 - ➔ **derived data types**
 - ➔ `MPI_Send` gets a pointer to the data structure as well as a description of the data type
 - ➔ the description of the data type must be created by calling MPI routines

Notes for slide 356:

Example for packing and unpacking using MPI_Pack and MPI_Unpack:

```
// C structure (or likewise C++ object), which should be sent
struct bsp { int a; double b[3]; char c; } str;

char buf[100]; // buffer, must be large enough!!
int pos; // position in the buffer
...

pos = 0;
MPI_Pack(&str.a, 1, MPI_INT, buf, 100, &pos, MPI_COMM_WORLD);
MPI_Pack(&str.b, 3, MPI_DOUBLE, buf, 100, &pos, MPI_COMM_WORLD);
MPI_Pack(&str.c, 1, MPI_CHAR, buf, 100, &pos, MPI_COMM_WORLD);
MPI_Send(buf, pos, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
...

MPI_Recv(buf, 100, MPI_PACKED, 1, 0, MPI_COMM_WORLD, &status);
pos = 0;
MPI_Unpack(buf, 100, &pos, &str.a, 1, MPI_INT, MPI_COMM_WORLD);
MPI_Unpack(buf, 100, &pos, &str.b, 3, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buf, 100, &pos, &str.c, 1, MPI_CHAR, MPI_COMM_WORLD);
```

356-1

The MPI standard requires that a message always must be packed as shown in successive calls to MPI_Pack (pack unit), where buffer, buffer length and communicator are identical.

In this way, the standard allows that an implementation also packs a header into the message (e.g., for an architecture tag). For this, information from the communicator may be used, if required.

356-2



Derived data types

- ➔ MPI offers constructors, which can be used to define own (derived) data types:
 - ➔ for contiguous data: `MPI_Type_contiguous`
 - ➔ allows the definition of array types
 - ➔ for non-contiguous, strided data: `MPI_Type_vector`
 - ➔ e.g., for a column of a matrix or a sub-matrix
 - ➔ for other non-contiguous data: `MPI_Type_indexed`
 - ➔ for structures: `MPI_Type_create_struct`
- ➔ After a new data type has been created, it must be “announced”: `MPI_Type_commit`
- ➔ After that, the data type can be used like a predefined data type (e.g., `MPI_INT`)



`MPI_Type_vector`: non-contiguous arrays

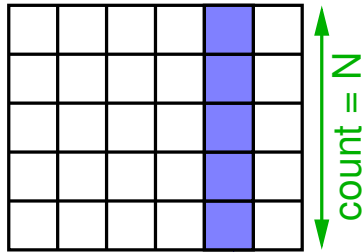
```
int MPI_Type_vector(int count, int blocklen, int stride,
                   MPI_Datatype oldtype,
                   MPI_Datatype *newtype)
```

<i>IN</i>	count	Number of data blocks
<i>IN</i>	blocklen	Length of the individual data blocks
<i>IN</i>	stride	Distance between successive data blocks
<i>IN</i>	oldtype	Type of the elements in the data blocks
<i>OUT</i>	newtype	Newly created data type

- ➔ Summarizes a number of data blocks (described as arrays) into a new data type
- ➔ However, the result is more like a new **view** onto the existing data than a new data **type**

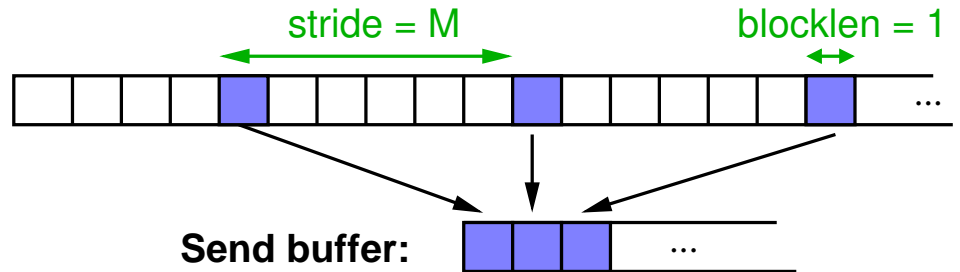
Example: transferring a column of a matrix

Matrix: $a[N][M]$



This column should be sent

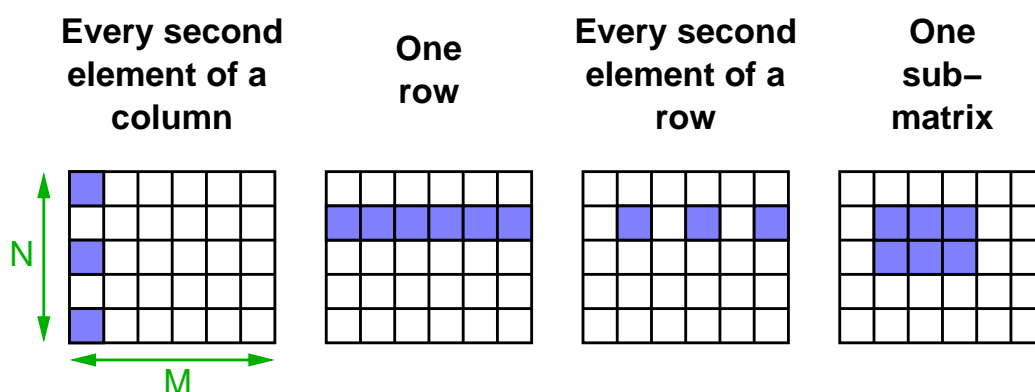
Memory layout of the matrix:



```
MPI_type_vector(N, 1, M, MPI_INT, &column);
MPI_Type_commit(&column);
// Transfer the column
if (rank==0) MPI_Send(&a[0][4], 1, column, 1, 0, comm);
else MPI_Recv(&a[0][4], 1, column, 0, 0, comm, &status);
```

Notes for slide 359:

Additional options of MPI_Type_vector



	Every second element of a column	One row	Every second element of a row	One sub-matrix
count	$N / 2$	1	$M / 2$	2
blocklen	1	M	1	3
stride	$2 * M$	x	2	M

Remarks on `MPI_Type_vector`

- ➔ The receiver can use a different data type than the sender
- ➔ It is only required that the number of elements and the sequence of their types is the same in the send and receive operations
- ➔ Thus, e.g., the following is possible:
 - sender transmits a column of a matrix
 - receiver stores it in a one-dimensional array

```
int a[N][M], b[N];  
MPI_type_vector(N, 1, M, MPI_INT, &column);  
MPI_Type_commit(&column);  
if (rank==0) MPI_Send(&a[0][4], 1, column, 1, 0, comm);  
else MPI_Recv(b, N, MPI_INT, 0, 0, comm, &status);
```

Notes for slide 360:

Strided arrays that have been created using `MPI_Type_vector` can usually be transmitted as efficiently as contiguous arrays (i.e., with stride 1) with modern network interface cards. These cards support the transmission of non-contiguous memory areas in hardware.



How to select the best approach

- ➔ Homogeneous data (elements of the same type):
 - contiguous (stride 1): standard data type and count parameter
 - non-contiguous:
 - stride is constant: `MPI_Type_vector`
 - stride is irregular: `MPI_Type_indexed`
- ➔ Heterogeneous data (elements of different types):
 - large data, often transmitted: `MPI_Type_create_struct`
 - few data, rarely transmitted: `MPI_Pack` / `MPI_Unpack`
 - structures of variable length: `MPI_Pack` / `MPI_Unpack`

4.10 Further concepts



- ➔ Topologies
 - the application's communication structure is stored in a communicator
 - e.g., cartesian grid
 - allows to simplify and optimize the communication
 - e.g., "send to the left neighbor"
 - the communicating processes can be placed on neighboring nodes
- ➔ Dynamic process creation (since MPI-2)
 - new processes can be created at run-time
 - process creation is a collective operation
 - the newly created process group gets its own `MPI_COMM_WORLD`
 - communication between process groups uses an *intercommunicator*

Parallel Processing

Winter Term 2025/26

15.12.2025

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

Stand: January 8, 2026

4.10 Further concepts ...



- ➔ One-sided communication (since MPI-2)
 - ➔ access to the address space of other processes
 - ➔ operations: read, write, atomic update
 - ➔ weak consistency model
 - ➔ explicit *fence* and *lock/unlock* operations for synchronisation
 - ➔ useful for applications with irregular communication
 - ➔ one process alone can execute the communication
- ➔ Parallel I/O (since MPI-2)
 - ➔ processes have individual views to a file
 - ➔ specified by an MPI data type
 - ➔ file operations: individual / collective, private / shared file pointer, blocking / non-blocking

4.11 Summary



- ➔ Basic routines:
 - ➔ Init, Finalize, Comm_size, Comm_rank, Send, Recv
- ➔ Communicators: process group + communication context
- ➔ Non-blocking communication: Isend, Irecv, Test, Wait
- ➔ Collective operations
 - ➔ Barrier, Bcast, Scatter(v), Gather(v), Reduce, ...
- ➔ Complex data types in messages
 - ➔ Pack and Unpack
 - ➔ user defined data types
 - ➔ also for non-contiguous data (e.g., column of a matrix)



Parallel Processing

Winter Term 2025/26

5 Optimization Techniques



- ➔ In the following: examples for important techniques to optimize parallel programs
- ➔ Shared memory:
 - ➔ cache optimization: improve the locality of memory accesses
 - ➔ loop interchange, tiling
 - ➔ array padding
 - ➔ false sharing
- ➔ Message passing:
 - ➔ combining messages
 - ➔ latency hiding

5.1 Cache Optimization



Example: summation of a matrix in C++ (👉 05/sum.cpp)

```
double a[N][N];  
...  
for (j=0; j<N; j++) {  
    for (i=0; i<N; i++) {  
        s += a[i][j];  
    }  
} column-wise traversal
```

```
double a[N][N];  
...  
for (i=0; i<N; i++) {  
    for (j=0; j<N; j++) {  
        s += a[i][j];  
    }  
} row-wise traversal
```

N=8192: Run time: 930ms

N=8193: Run time: 140 ms

Run time: 80ms

Run time: 80ms

(bspc02,
g++ -O3)

- ➔ Reason: caches
 - ➔ higher hit rate when matrix is traversed row-wise
 - ➔ although each element is used only once ...
- ➔ Remark: C/C++ stores a matrix row-major, Fortran column-major



Details on caches: cache lines

- ➔ Storage of data in the cache and transfer between main memory and cache are performed using larger blocks
 - ➔ reason: after a memory cell has been addressed, the subsequent cells can be read very fast
 - ➔ size of a cache line: 32-128 Byte
- ➔ In the example:
 - ➔ row-wise traversal: after the cache line for $a[i][j]$ has been loaded, the values of $a[i+1][j]$, $a[i+2][j]$, ... are already in the cache, too
 - ➔ column-wise traversal: the cache line for $a[i][j]$ has already been evicted, when $a[i+1][j]$, ... are used
- ➔ **Rule:** traverse memory in linearly increasing order, if possible!



Details on caches: set-associative caches

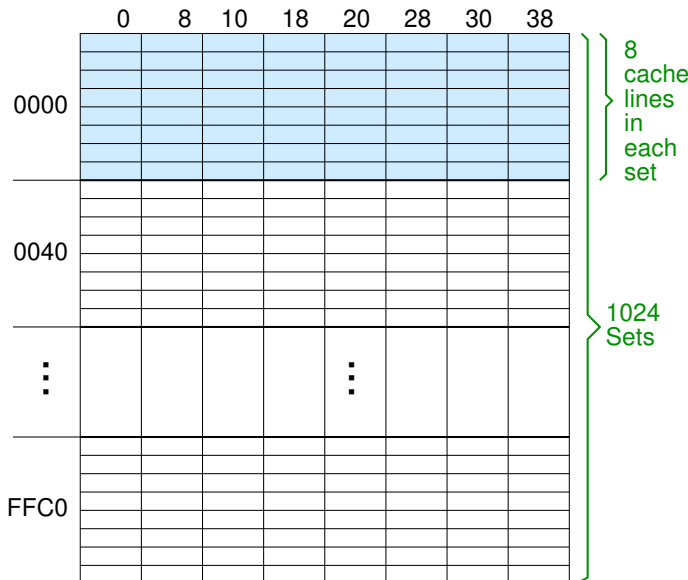
- ➔ A memory block (with given address) can be stored only at a few places in the cache
 - ➔ reason: easy retrieval of the data in hardware
 - ➔ usually, a set has 2 to 16 entries
 - ➔ the entry within a set is determined using the LRU strategy
- ➔ The lower k Bits of the address determine the set (k depends on cache size and degree of associativity)
 - ➔ for all memory locations, whose lower k address bits are the same, there are only 2 - 16 possible cache entries!

5.1 Cache Optimization ...

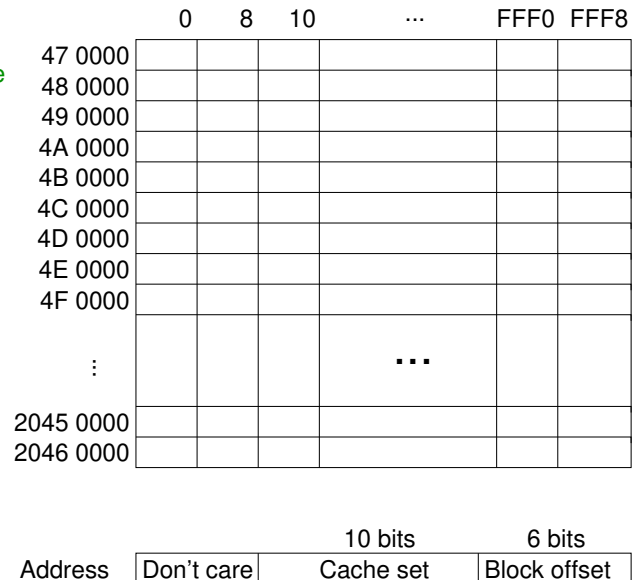


Details on caches: set-associative caches ...

Cache: 512 KByte, 8-way set associative, line size 64 Byte



Matrix 8192 x 8192 at address 0x470000:



Notes for slide 370:

In the figure shown on the slide, the address of each row and the offset of each column of the matrix is indicated (all addresses are shown as hexadecimal numbers).

When a thread traverses the first column of the matrix, the lower 16 address bits of the element being read will always be 0000. Thus, when the data is loaded into the cache, only the first set (with address 0000) is used, since the set address in the cache is determined by bits 6..15 of the memory address (Bits 0..5 determine the offset in the cache line). Now when the element in row 8 is read, one of the cache lines in set 0000 must be evicted, since each set contains only 8 cache lines. This means when the next column is traversed, the data is no longer in the cache.

A detailed explanation of the example is given in the lecture.



Details on caches: set-associative caches ...

- ➔ In the example: with $N = 8192$ and column-wise traversal
 - ➔ a cache entry is guaranteed to be evicted after a few iterations of the i -loop (address distance is a power of two)
 - ➔ cache hit rate is very close to zero
- ➔ **Rule:** when traversing memory, avoid address distances that are a power of two!
 - ➔ (avoid powers of two as matrix size for large multi-dimensional matrices)



Important cache optimizations

- ➔ **Loop interchange:** swapping of loops
 - ➔ such that memory is traversed in linearly increasing order
 - ➔ with C/C++: traverse matrices row-wise
 - ➔ with Fortran: traverse matrices column-wise
- ➔ **Array padding**
 - ➔ if necessary, allocate matrices larger than necessary, in order to avoid a power of two as the length of each row
- ➔ **Tiling:** blockwise partitioning of loop iterations
 - ➔ restructure algorithms in such a way that they work as long as possible with sub-matrices, which fit completely into the caches

5.1 Cache Optimization ...



Example: Matrix multiply

(05/matmult.c)

➔ Naive code:

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

➔ Performance with different compiler optimization levels:
(N=384, g++ 12.2.0, Intel Core i7 @ 3.4 GHz (bspc02))

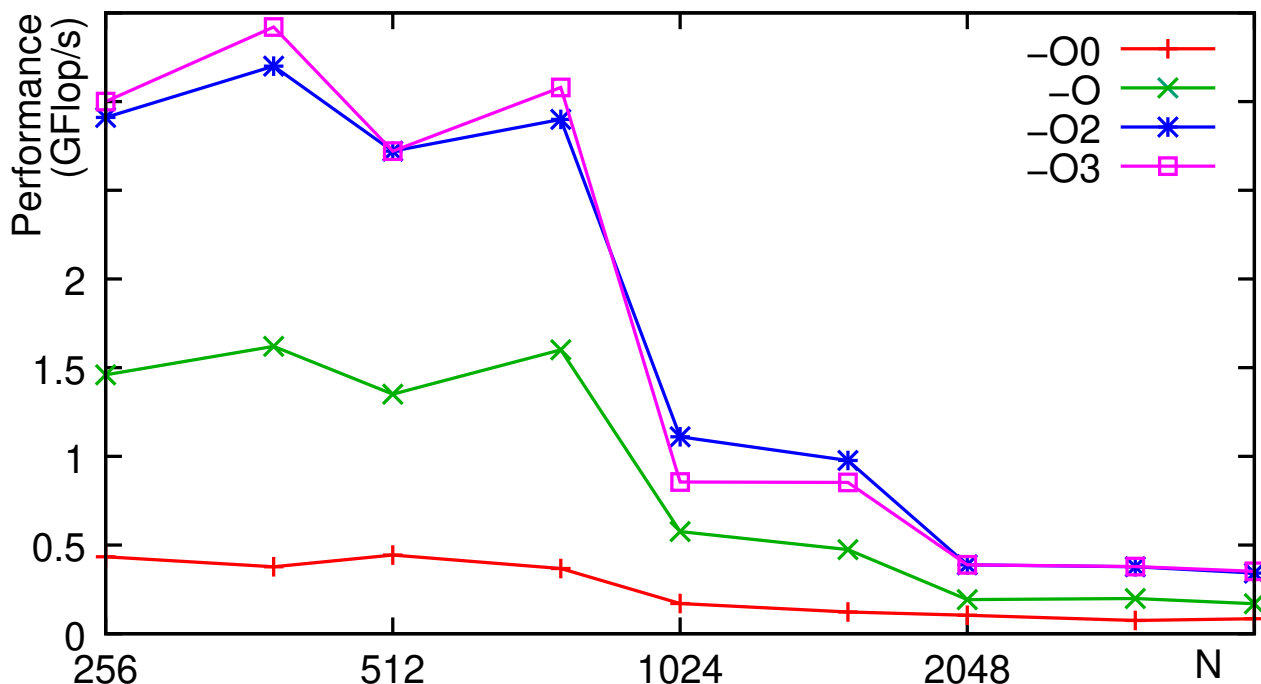
- ➔ -O0: 0.4 GFlop/s
- ➔ -O: 1.6 GFlop/s
- ➔ -O2: 3.2 GFlop/s
- ➔ -O3: 3.4 GFlop/s

5.1 Cache Optimization ...



Example: Matrix multiply ...

➔ Scalability of the performance for different matrix sizes:



Example: Matrix multiply ...

➔ Optimized order of the loops:

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

➔ Matrix b now is traversed row-wise

➤ considerably less L1 cache misses

➤ substantially higher performance:

➤ N=384, -O3: 9.8 GFlop/s instead of 3.4 GFlop/s

➤ considerably better scalability

Notes for slide 375:

In the code on slide 373, the statement $c[i][j] += a[i][k] * b[k][j]$ has a true dependence, an anti dependence and an output dependence between different iterations of the k -loop. Thus, the dependence vector for all these dependences is $(=, =, <)$.

So according to slide 229, interchanging the j - and k -loops is permitted, since the loops are perfectly nested, the loop bounds are independent, and there is no dependence with a direction vector of $(*, <, >)$.

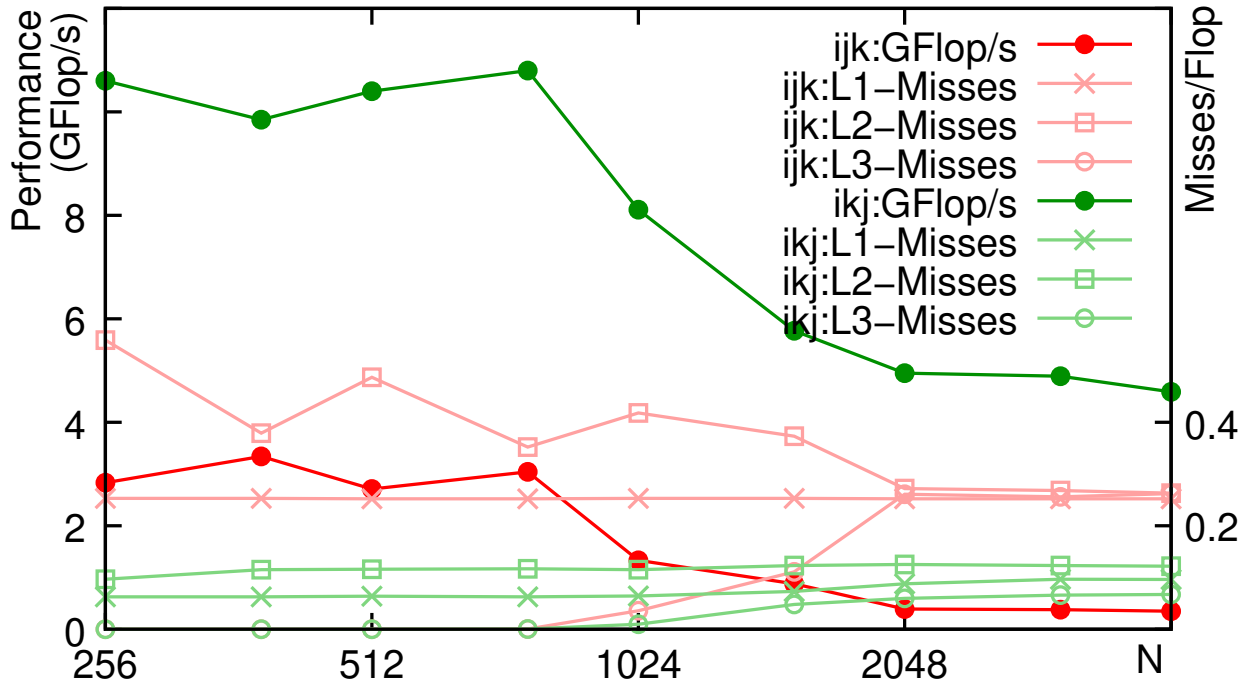
5.1 Cache Optimization ...



(Animated slide)

Example: Matrix multiply ...

→ Comparison of both loop orders:



Notes for slide 376:

The decrease in performance for $N > 768$ is due to a large increase in the L3 misses from $3 \cdot 10^{-4}$ to about 0.25 for the 'ijk' loop and 0.06 for the 'ikj' loop.

Example: Matrix multiply ...

- ➔ Block algorithm (tiling) with array padding:

```
double a[N][N+8], b[N][N+8], c[N][N+8]
for (int ii=0; ii<N; ii+=16)
  for (int kk=0; kk<N; kk+=16)
    for (int jj=0; jj<N; jj+=16)
      for (int i=0; i<16; i++)
        for (int k=0; k<16; k++)
          for (int j=0; j<16; j++)
            c[i+ii][j+jj] += a[i+ii][k+kk] * b[k+kk][j+jj];
```

- ➔ Matrix is viewed as a matrix of 16×16 sub-matrices
 - ➔ multiplication of sub-matrices fits into the L1 cache
- ➔ Achieves a performance of 13 GFlop/s even with $N=2048$

Notes for slide 377:

See slide [230](#) for the details of the code transformation (strip mining followed by loop interchange) used to create this version of the code.

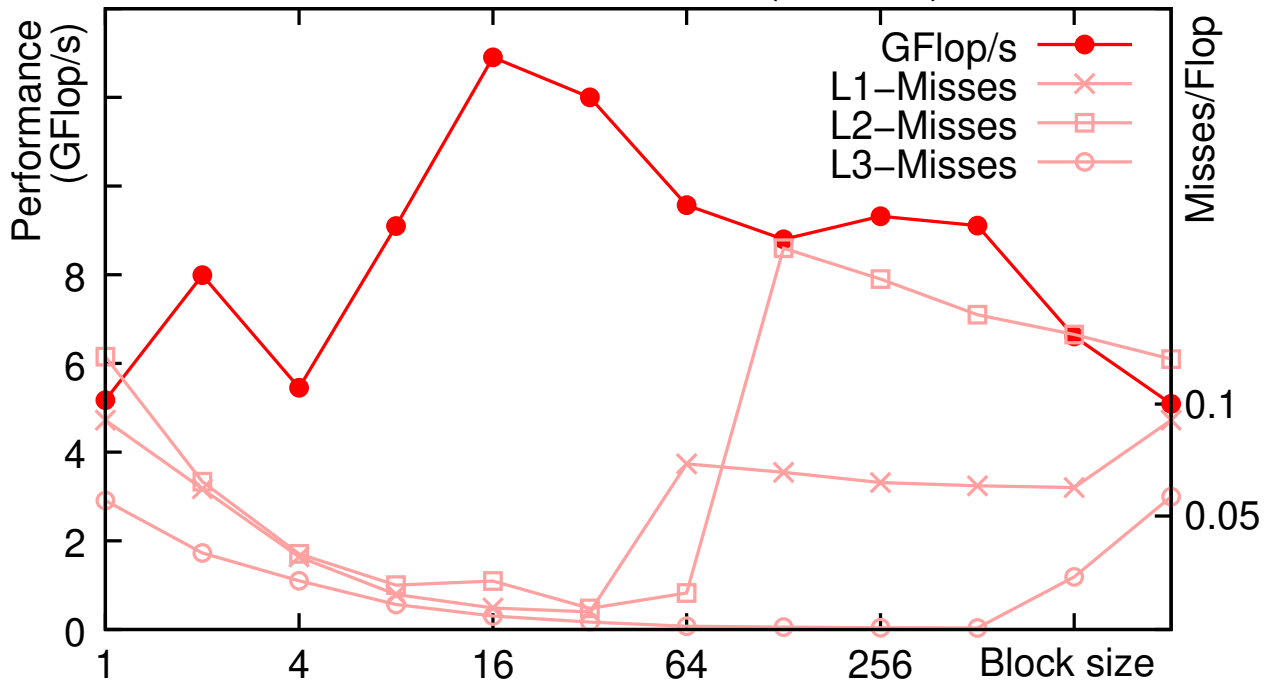
The block size (16) and the padding (8) have been determined experimentally.

5.1 Cache Optimization ...



Example: Matrix multiply ...

➔ Performance as a function of block size (N=2048):

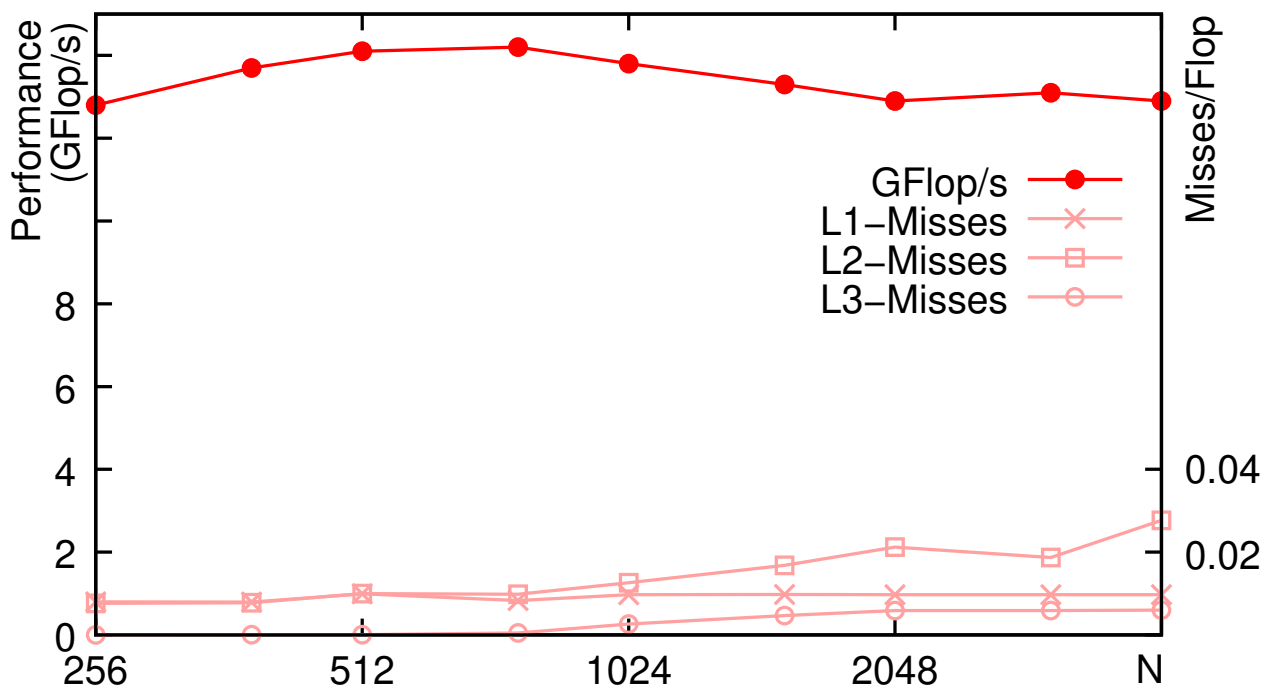


5.1 Cache Optimization ...



Example: Matrix multiply ...

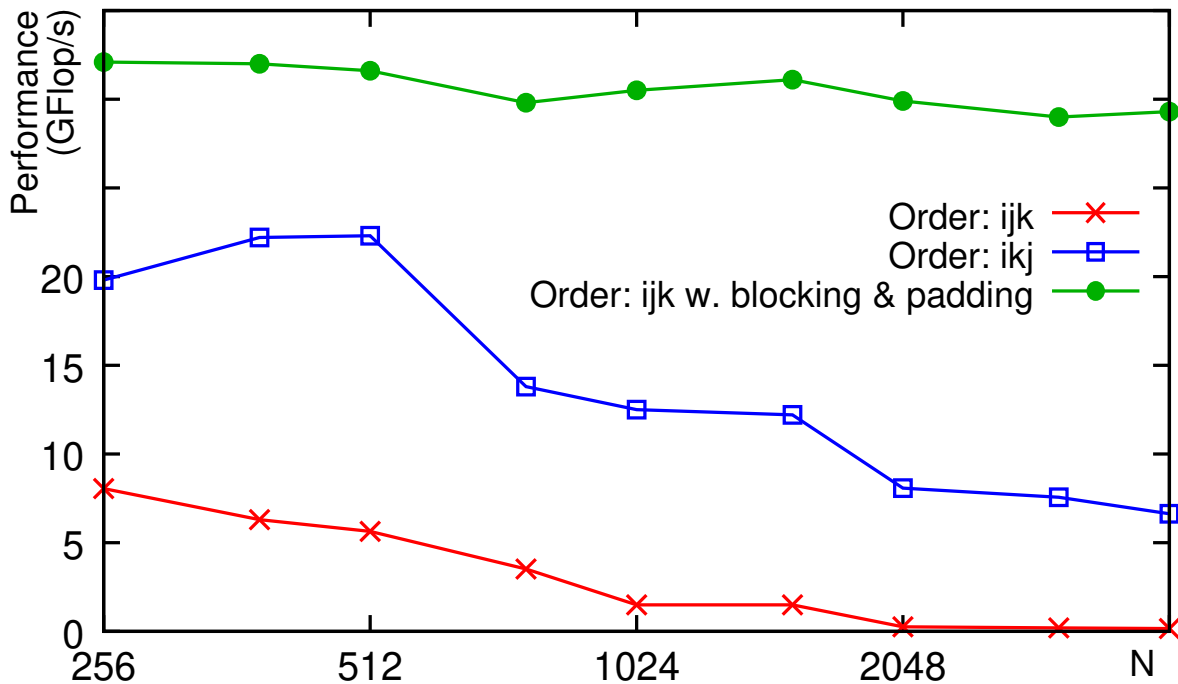
➔ Scalability of performance for different matrix sizes:





Example: Matrix multiply ...


➔ Results in the lab: (Intel Core Ultra 7 265, max. 5.2 GHz; -O3):



Cache optimization for parallel computers

- ➔ Cache optimization is especially important for parallel computers (UMA and NUMA)
 - ➔ larger difference between the access times of cache and main memory
 - ➔ concurrency conflicts when accessing main memory
- ➔ Additional problem with parallel computers: **false sharing**
 - ➔ several variables, which do not have a logical association, can (by chance) be stored in the same cache line
 - ➔ write accesses to these variables lead to frequent cache invalidations (due to the cache coherence protocol)
 - ➔ performance degrades drastically

Example for false sharing: parallel summation of an array

( 05/false.cpp)

- ➔ Global variable `double sum[NUM_THREADS]` for the partial sums
- ➔ Version 1: thread `i` adds to `sum[i]`
 - run-time^(*) with 4 threads: 0.21 s, sequentially: 0.17 s !
 - performance loss due to false sharing: the variables `sum[i]` are located in the same cache line
- ➔ Version 2: thread `i` first adds to a local variable and stores the result to `sum[i]` at the end
 - run-time^(*) with 4 threads: 0.043 s
- ➔ **Rule:** variables that are used by different threads should be separated in main memory (e.g., use padding)!

(*) 8000 x 8000 matrix, Intel Core i7, 2.8 GHz, without compiler optimization

Notes for slide 382:

When compiler optimization is enabled with `gcc`, the run-time of the parallel program in version 1 is reduced to 0.045 s (version 2: 0.043 s, sequentially: 0.16 s), i.e., in this code, `gcc` is smart enough to detect and solve the problem with false sharing.



Combining messages

- ➔ The time for sending short messages is dominated by the (software) latency
 - ➔ i.e., a long message is “cheaper” than several short ones!
- ➔ Example: PC cluster in the lab H-A 4111 with MPICH2
 - ➔ 32 messages with 32 Byte each need $32 \cdot 145 = 4640\mu s$
 - ➔ one message with 1024 Byte needs only $159\mu s$
- ➔ Thus: combine the data to be sent into as few messages as possible
 - ➔ where applicable, this can also be done with communication in loops (hoisting)

5.2 Optimization of Communication ...



Hoisting of communication calls

```
for (i=0; i<N; i++) {      for (i=0; i<N; i++) {
    b = f(..., i);          recv(&b, 1, P1);
    send(&b, 1, P2);        a[i] = a[i] + b;
}                            }

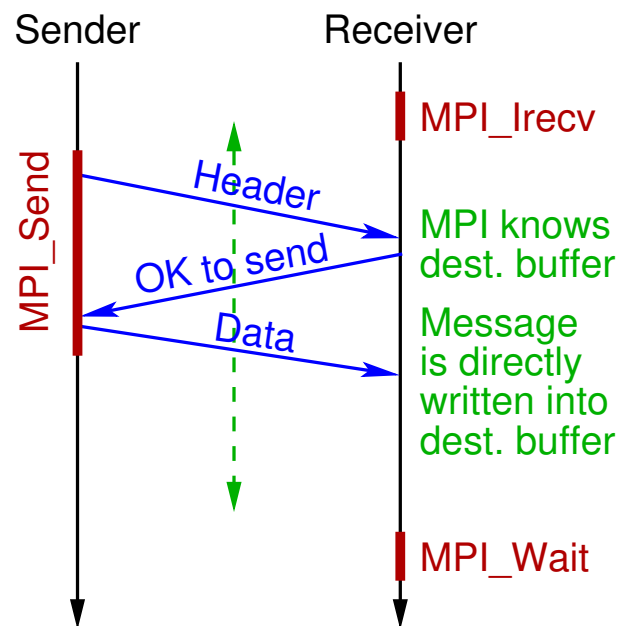
for (i=0; i<N; i++) {      recv(b, N, P1);
    b[i] = f(..., i);      for (i=0; i<N; i++) {
}                            a[i] = a[i] + b[i];
send(b, N, P2);            }
```

- ➔ Send operations are hoisted past the end of the loop, receive operations are hoisted before the beginning of the loop
- ➔ Prerequisite: variables are not modified in the loop (sending process) or not used in the loop (receiving process)



Latency hiding

- ➔ Goal: hide the communication latency, i.e., overlap it with computations
- ➔ As early as possible:
 - post the receive operation (MPI_Irecv)
- ➔ Then:
 - send the data
- ➔ As late as possible:
 - finish the receive operation (MPI_Wait)



5.3 Summary



- ➔ Take care of good locality (caches)!
 - traverse matrices in the order in which they are stored
 - avoid powers of two as address increment when sweeping through memory
 - use block algorithms
- ➔ Avoid false sharing!
- ➔ Combine messages, if possible!
- ➔ Use latency hiding when the communication library can execute the receipt of a message “in background”
- ➔ If send operations are blocking: execute send and receive operations as synchronously as possible

Parallel Processing

Winter Term 2025/26

6 Summary / Important Topics

6 Summary / Important Topics ...



2 Basics of Parallel Processing

- ➔ Parallelism: concurrency/pipelining, data/task parallelism
- ➔ **Data dependences** (true, anti, output) and synchronisation
- ➔ SIMD computers
- ➔ **MIMD computers:** UMA, NUMA, NORMA
 - ➔ architectural properties, programming
- ➔ **Caches**, cache coherency (👉 5.1)
- ➔ **Organisation forms** (manager/worker, task pool, divide and conquer, SPMD, fork/join, ...)
- ➔ Design process (classes of partitioning, communication, mapping)
- ➔ **Performance** (speedup, efficiency, performance modeling)



3 Parallel Programming with Shared Memory

- ➔ **OpenMP programming model (fork/join)**
- ➔ **parallel directive**: syntax, semantics
 - ➔ shared, private, firstprivate variables
- ➔ **for directive**: syntax, semantics
 - ➔ scheduling and scheduling options
- ➔ **Parallelization of loops**
 - ➔ condition, handling of dependences
- ➔ **Parallelization of Jacobi and Gauss/Seidel**
- ➔ **Synchronization**: barrier, critical/atomic, ordered, reduction
- ➔ **Task parallelism**: sections / task directive, task synchronization



4 Parallel Programming with Message Passing

- ➔ **MPI programming model (SPMD)**
- ➔ **Point-to-point communication**: Send, Recv
- ➔ Nonblocking communication
- ➔ Derived data types
- ➔ **Communicators**
- ➔ **Collective operations**: Bcast, Scatter, Gather, Reduce

5 Optimization Techniques

- ➔ **Organization of caches**
- ➔ **Rules for optimal use of caches**
- ➔ **False sharing**