Parallel Processing

Winter Term 2024/25

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

Stand: January 14, 2025

Contents

Parallel Processing

Winter Term 2024/25

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

- **E-mail:** roland.wismueller@uni-siegen.de
- **Tel.:** 0271/740-4050
- **Room:** H-B 8404
- **Office Hour:** Mo., 14:15-15:15

About the Chair “Operating Systems / Distrib. Sys.”

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)

Teaching ...

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]

[a] https://www.eti.uni-siegen.de/dekanat/pruefungsamt/dokumente/studiengangsuebergreifend/externe-abschlussarbeiten-eti_en.pdf

About the Lecture

Lecture

- ➥ Mon., 12:15-13:45, AR-B 2104/05
- ➥ on 08.10., 15.10., 22.10., and 29.10. also in the lab slot!
  - ➥ Tue., 10:15-11:45, H-C 6321

Practical labs

- ➥ Preferrably at home
  - ➥ if necessary, you can also use the PC lab room H-A 4111
- ➥ Tutor: Felix Breitweiser (felix.breitweiser@uni-siegen.de)
- ➥ Questions and help: via Discord server
  - ➥ `https://discord.gg/UZTv8yptqj`
- ➥ Discussion of solutions: Tue., 10:15-11:45, H-C 6321
  - ➥ only on the due date of an assignment!

About the Lecture ...

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`
- ➥ Recorded screen casts of the lecture (from winter term 2020/21)
- ➥ Submission of lab assignments

About the Lecture ...

Discord invite link

About the Lecture ...

Link to course’s homepage

About the Lecture ...

Learning targets

- Knowing the basics, techniques, methods, and tools of parallel programming
- Basic knowledge about parallel computer architectures
- Practical experiences 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

About the Lecture ...

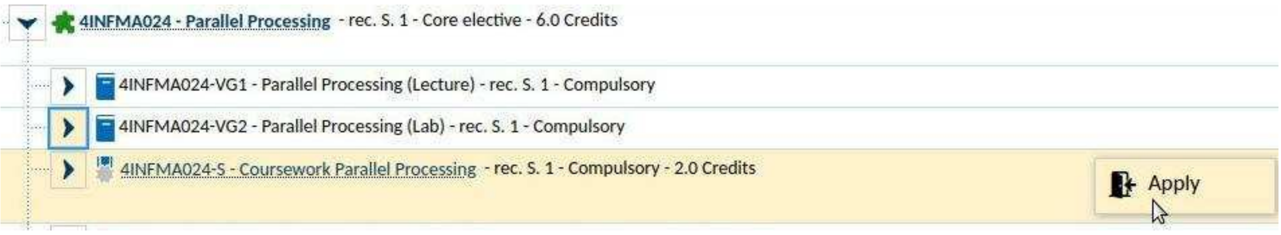
Methodology

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

Registration for “Course Achievement” (Studienleistung)

- ➥ Passing the course requires successful completion of the lab:
  - ➥ i.e., qualified attempt for all mandatory exercises
  - ➥ Exam Regulations 2012: prerequisite for the exam!
- ➥ You must register for the
  - ➥ “Coursework Parallel Processing” 4INFMA024-S, or
  - ➥ “Prüfungsvorleistung” 822120-S

  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!



4INFMA024 - Parallel Processing - rec. S. 1 - Core elective - 6.0 Credits

4INFMA024-VG1 - Parallel Processing (Lecture) - rec. S. 1 - Compulsory

4INFMA024-VG2 - Parallel Processing (Lab) - rec. S. 1 - Compulsory

4INFMA024-S - Coursework Parallel Processing - rec. S. 1 - Compulsory - 2.0 Credits

Notes for slide 13:

If you are not registered for the course achievement, you will not be able to submit any solutions in the Moodle plattform (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!

Examination

- Written examination (60 minutes)
  - 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
  - if you study Computer Science with Exam Regulations 2012, you first must have your mentor's approval
    - **be sure to meet the deadline!**

Organisational Issues regarding the Labs

- Assignments should be done at home, if possible
- Programming is done in C/C++
- Ideally, you need a Linux-PC with the GNU-compilers (gcc/g++)
  - Windows with `MSVC` will also work, except for one exercise sheet
- In addition, you need to install MPI, preferrable MPICH
  - see `https://www.mpich.org/downloads`
- Four exercise sheets
  - code must be submitted via Moodle in due time
  - different requirements depending on 5 CP vs. 6 CP

Contents of the Lecture

- 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
  - Performance Considerations
  - Design Process

Contents of the Lecture ...

- 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 October, 29th: only lectures (Mon. + Tue.), no lab
- Then: lectures (Mon.) and lab (home work + Tue.)
- Last two weeks: only lab

- Prospective due dates for the assignments:
  - 05.11.: Exercise sheet 1
  - ... (see web page)
  - 28.01.: Exercise sheet 8

  - On due date: presentation and discussion of assignments in H-C 6321

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

General Literature ...

- 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 2024/25

1 Repetition / Foundations

1 Repetition / Foundations ...

Contents

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

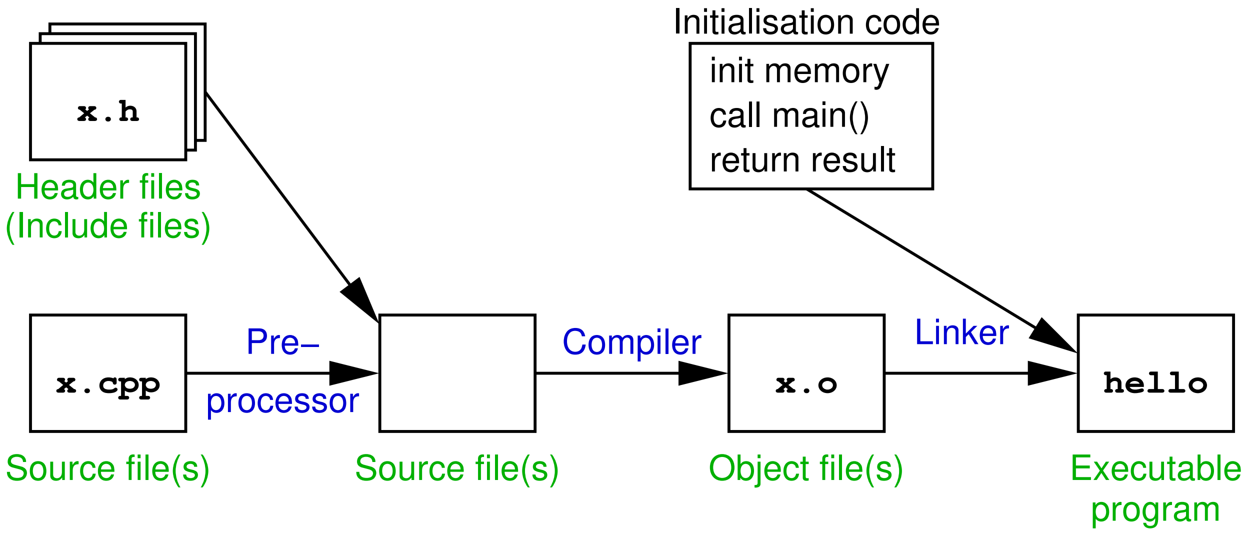
1.1 C/C++ for Java Programmers

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

1.1.1 Fundamentals of C++ ...

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`

1.1.1 Fundamentals of C++ ...

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!!!

1.1.2 Data types in C++ ...

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
};
```

1.1.2 Data types in C++ ...

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

1.1.2 Data types in C++ ...

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
```

1.1.2 Data types in C++ ...

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:

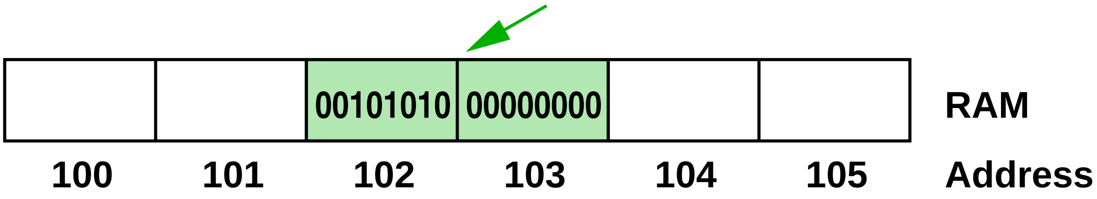
```
   std::list<int> intlist;       // List of integers
   intlist.push_back(42);        // Add at the end of the list
   int i = intlist.front();      // First element
   std::list<double> dbllist;    // List of doubles
   dbllist.push_back(3.1415);
```

1.1.3 Pointers

Variables in memory

- Reminder: variables are stored in main memory

```
short int myVar = 42;
```



  - 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.

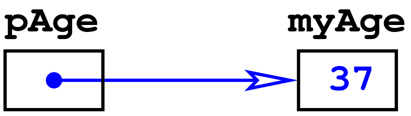
1.1.3 Pointers ...

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

Parallel Processing

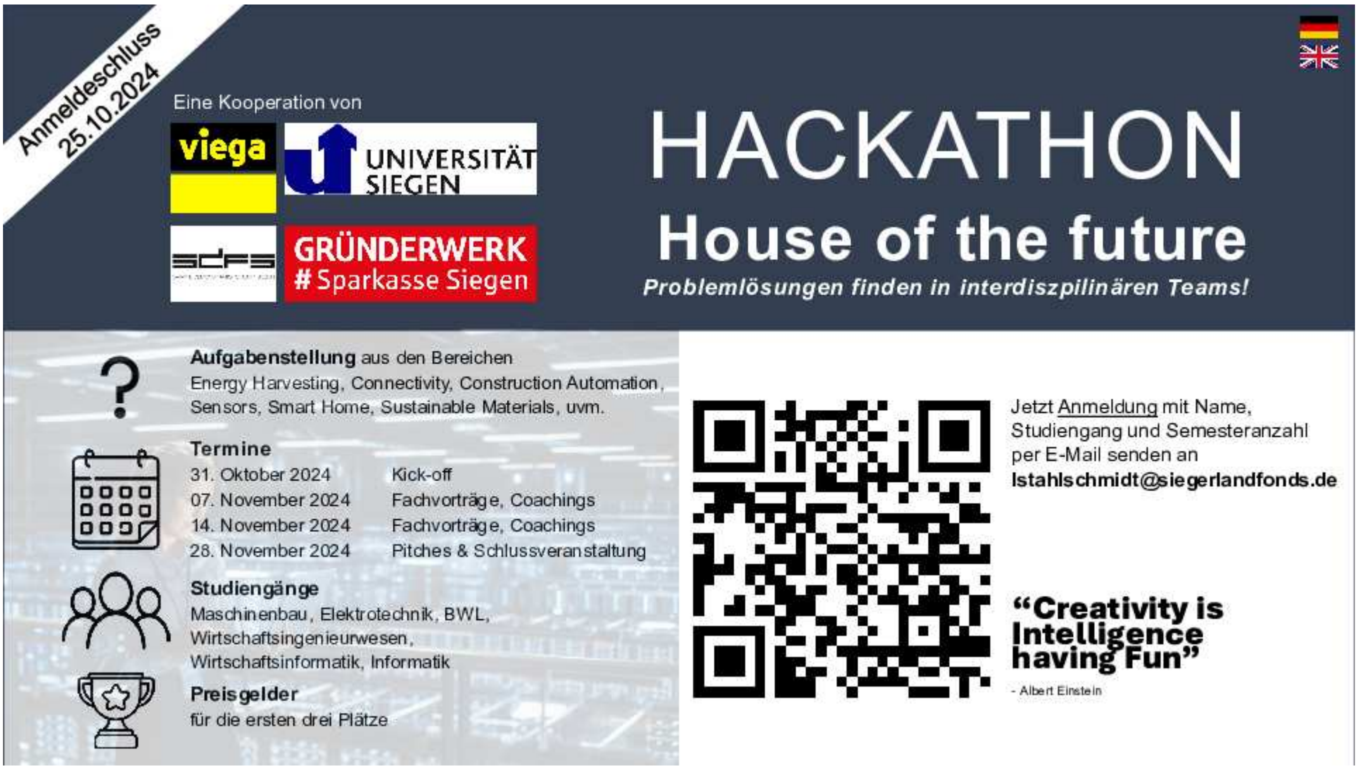
Winter Term 2024/25

14.10.2024

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

Stand: January 14, 2025

Discord invite link



Anmeldeschluss 25.10.2024

Eine Kooperation von

viega

UNIVERSITÄT SIEGEN

GRÜNDERWERK #Sparkasse Siegen

HACKATHON

House of the future

Problemlösungen finden in interdiszpilinären Teams!

Aufgabenstellung aus den Bereichen
Energy Harvesting, Connectivity, Construction Automation, Sensors, Smart Home, Sustainable Materials, uvm.

Termine

31. Oktober 2024	Kick-off
07. November 2024	Fachvorträge, Coachings
14. November 2024	Fachvorträge, Coachings
28. November 2024	Pitches & Schlussveranstaltung

Studiengänge
Maschinenbau, Elektrotechnik, BWL, Wirtschaftsingenieurwesen, Wirtschaftsinformatik, Informatik

Preisgelder
für die ersten drei Plätze

Jetzt Anmeldung mit Name, Studiengang und Semesteranzahl per E-Mail senden an
lstahlschmidt@siegerlandfonds.de

"Creativity is Intelligence having Fun"
- Albert Einstein

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`

1.1.3 Pointers ...

`void` pointers and type conversion

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

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

1.1.3 Pointers ...

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]
```

1.1.3 Pointers ...

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!

1.1.3 Pointers ...

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 exacly 4 decimal places

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 gloabal 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`

1.1.6 C/C++ Libraries

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	-lmpich	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);
```

1.1.7 The C Preprocessor ...

Functions of the preprocessor: ...

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

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

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

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

1.2 Threads and Synchronization ...

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

1.2 Threads and Synchronization ...

Synchronization using shared variables

- Example: waiting for a result

Thread 1

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

Thread 2

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

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

Semaphores

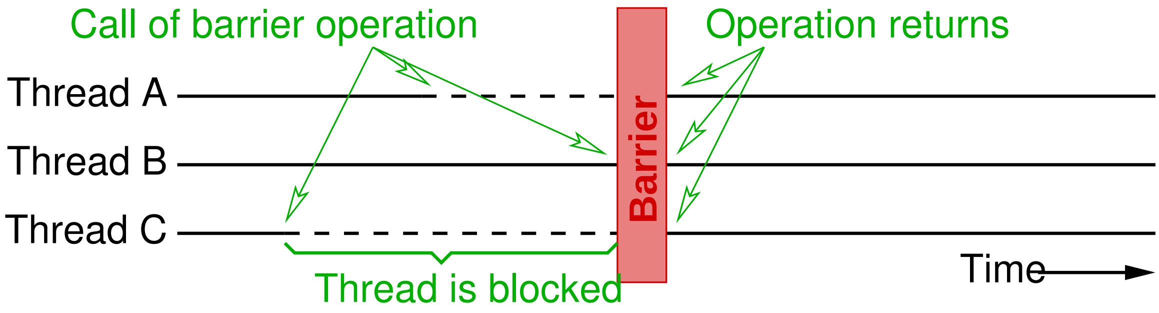
- Components: counter, queue of blocked threads
- **Atomic** operations:
  - `P()` (also `acquire`, `wait` or `down`)
    - decrements the counter by 1
    - if counter $<$ 0: block the thread
  - `V()` (also `release`, `signal` or `up`)
    - increments counter by 1
    - if counter $\leq$ 0: wake up 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

1.2 Threads and Synchronization ...

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                    Thread 2

while (true) {              while (true) {
   P(s1);                      P(s2);
   print("1");                 P(s1);
   V(s2);                      print("2");
   V(s1);                      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

1.3 C++ Threads ...

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

1.3 C++ Threads ...

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;
}
```

Parallel Processing

Winter Term 2024/25

15.10.2024

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

Stand: January 14, 2025

1.3 C++ Threads ...

Example: Summation of an array with multiple threads

(☞ 01/sum.cpp)

```
#include <iostream>
#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.

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

1.3 C++ Threads ...

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.

1.3 C++ Threads ...

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.

1.3 C++ Threads ...

Example: simulating a monitor with C++ threads

(☞ `01/monitor.cpp`)

```
#include <thread>
#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 give 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.

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!

Parallel Processing

Winter Term 2024/25

2 Basics of Parallel Processing

2 Basics of Parallel Processing ...

Contents

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

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

2.1 Motivation ...

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: Frontier (ORNL, USA), 1206 PFlop/s (with 8699904 CPU cores!)

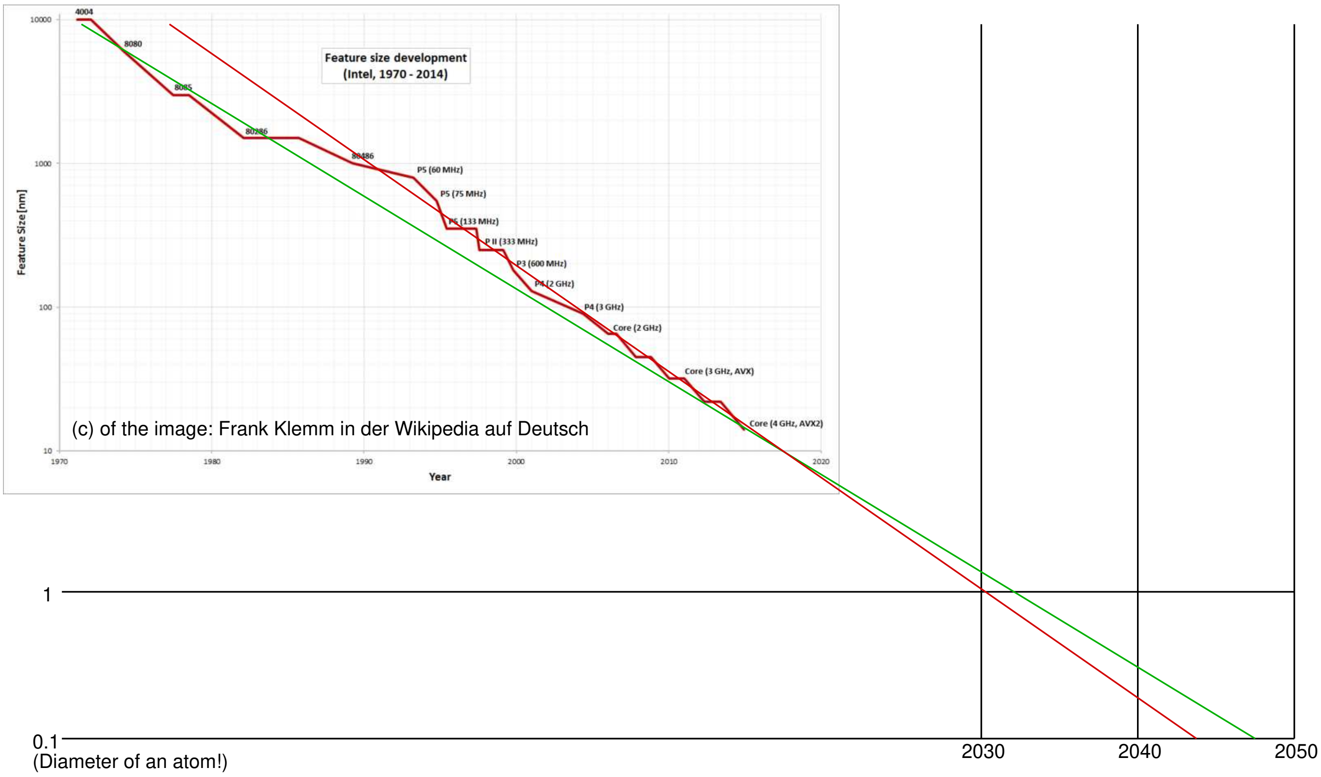
2.1 Motivation ...

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:

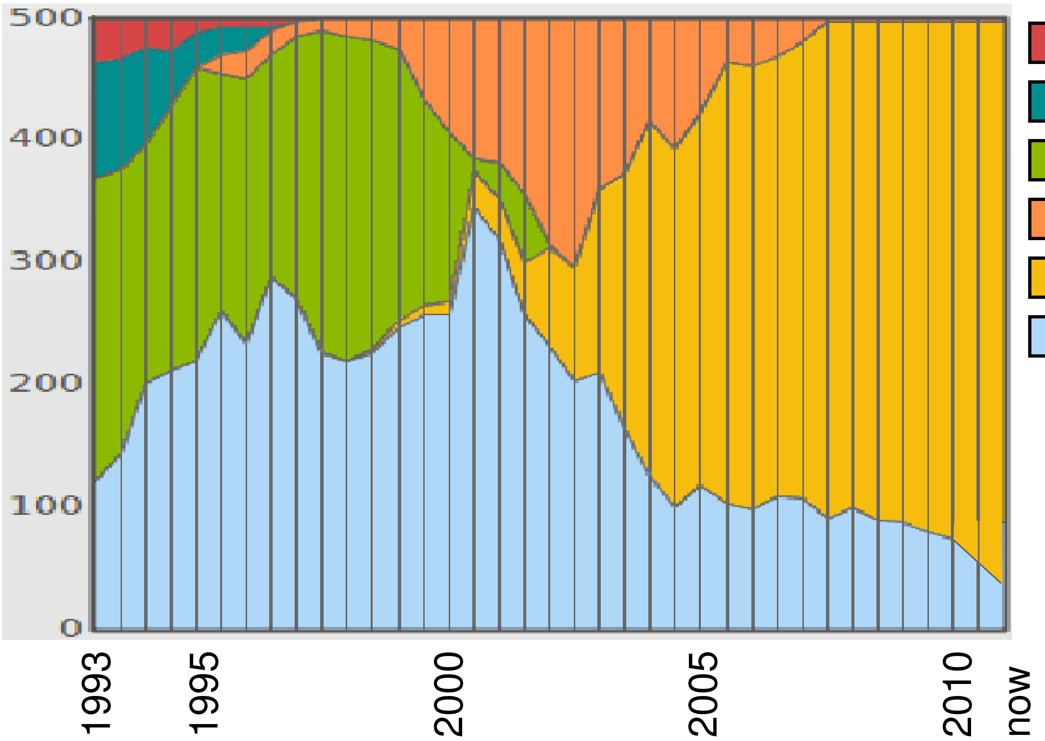
An estimation of the end of “Moore’s Law”:



(c) of the image: Frank Klemm in der Wikipedia auf Deutsch

2.1 Motivation ...

Architecture trend of high performance computers

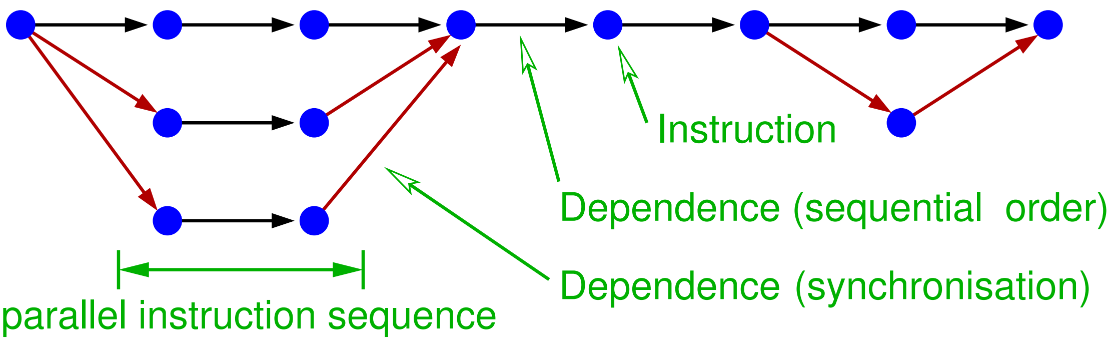


Source:
Top500 List
www.top500.org

2.2 Parallelism

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

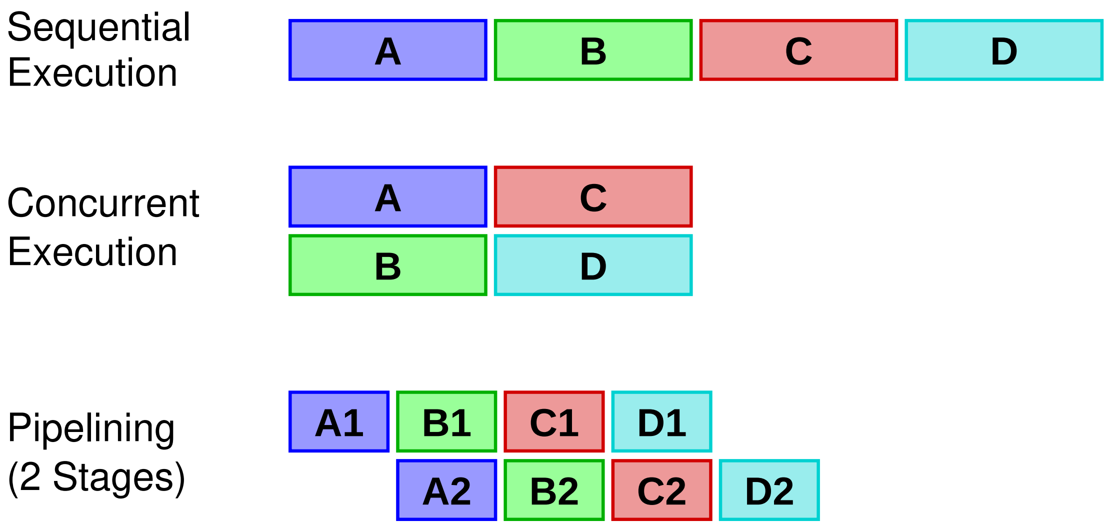

Instruction
Dependence (sequential order)
Dependence (synchronisation)
parallel instruction sequence

2.2 Parallelism ...

Concurrency vs. pipelining

- **Concurrency** (***Nebenläufigkeit***): instructions are executed simultaneously in different exceution 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.*”

2.2 Parallelism ...

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

2.2 Parallelism ...

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, ...

2.2 Parallelism ...

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

2.2 Parallelism ...

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

2.2 Parallelism ...

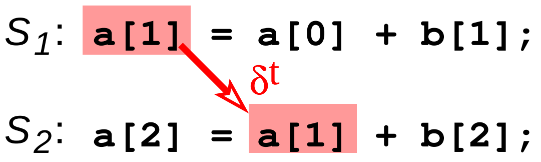
Granularity

- 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 = abs(x - y);`
  - but also in different iterations of a loop
- **True / flow dependence** (***echte Abhängigkeit***) $S_1 \xrightarrow{\delta^t} S_2$



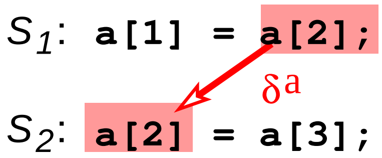
```
S1: a[1] = a[0] + b[1];
S2: a[2] = a[1] + b[2];
```

S1 (i=1) writes to a[1], which is later read by S2 (i=2)

2.3 Parallelisation and Data Dependences ...

(Animated slide)

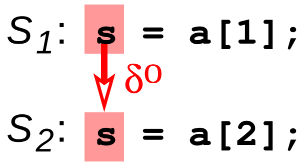
- **Anti dependence** (***Antiabhängigkeit***) $S_1 \xrightarrow{\delta^a} S_2$



```
S1: a[1] = a[2];
S2: a[2] = a[3];
```

S1 (i=1) read the value of a[2], which is overwritten by S2 (i=2)

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



```
S1: s = a[1];
S2: s = a[2];
```

S1 (i=1) writes a value to s, which is overwritten by S2 (i=2)

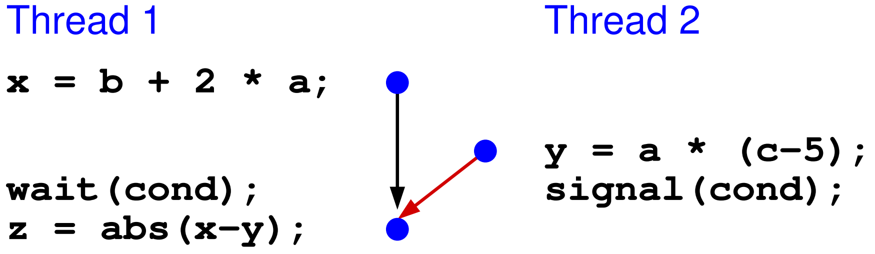
- **Anti** and **Output** dependences can always be removed by consistent renaming of variables

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);
```

Parallel Processing

Winter Term 2024/25

21.10.2024

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

Stand: January 14, 2025

2.4 Parallel Computer Architectures

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)?
- Thie 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

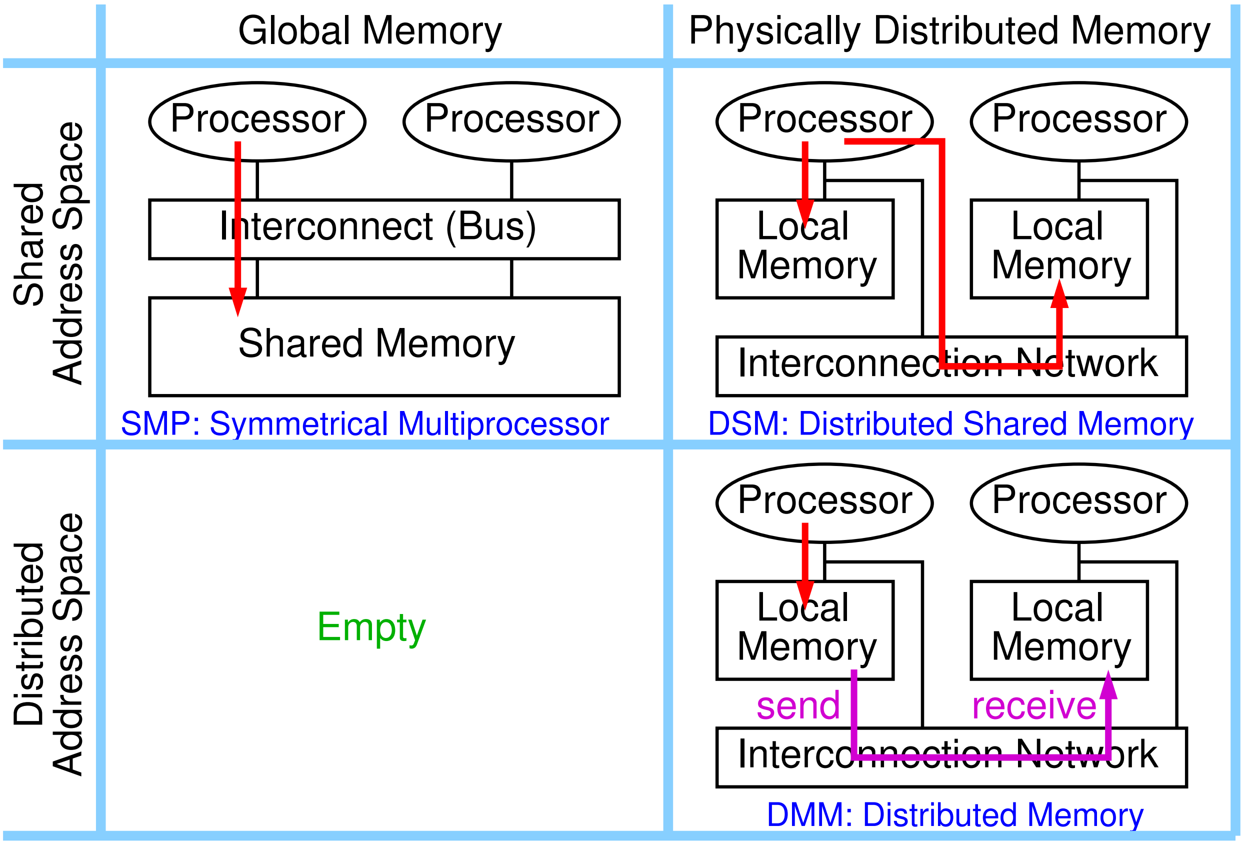
Classes of MIMD computers

- 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 ...

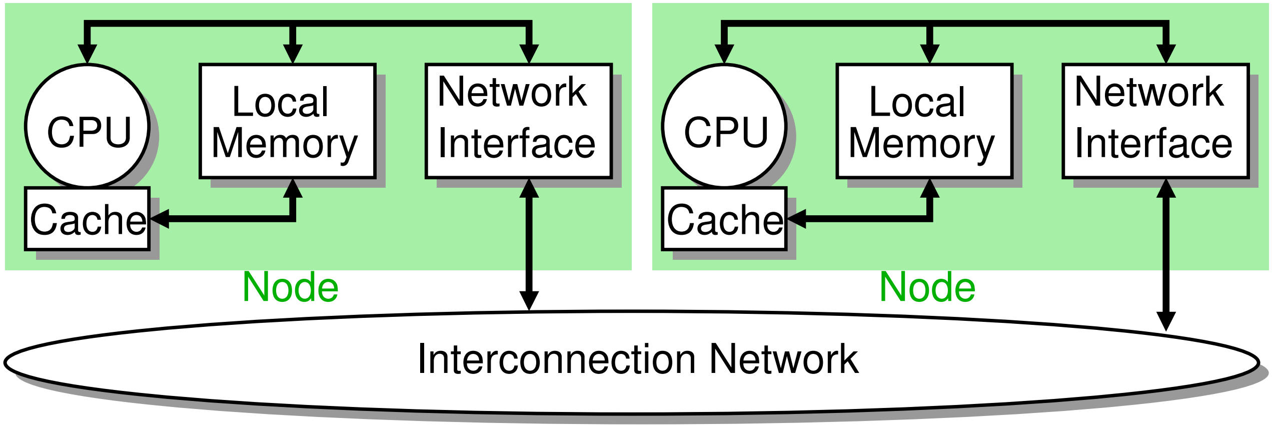
- **UMA**: **Uniform Memory Access**
  - global memory, shared address space
  - all processors access the memory in the same way
  - access time is equal for all processors
  - typical representative of this class:
    symmetrical multiprocessor (SMP), early multicore-CPUs
- **NUMA**: **Nonuniform Memory 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 derivates)

2.4.1 MIMD: Message Passing Systems ...

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. $\sim 20 - 100 ns$)
- In principle unlimited scalability
  - e.g. Frontier: 135936 nodes, (8699904 cores)

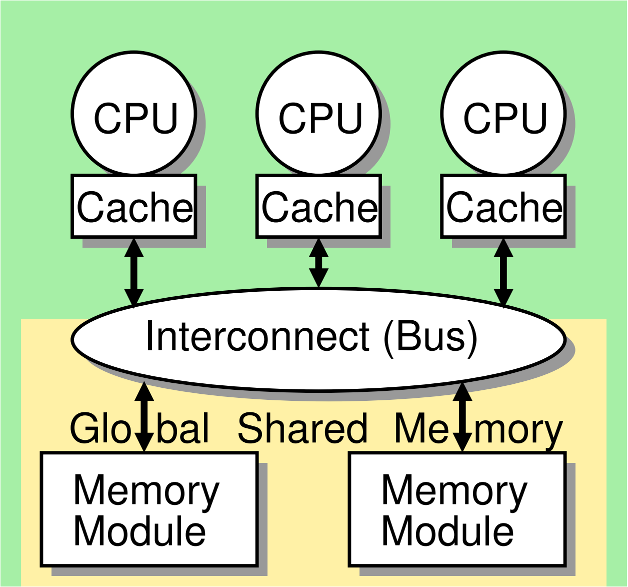
2.4.1 MIMD: Message Passing Systems ...

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 interactiv use

2.4.2 MIMD: Shared Memory Systems

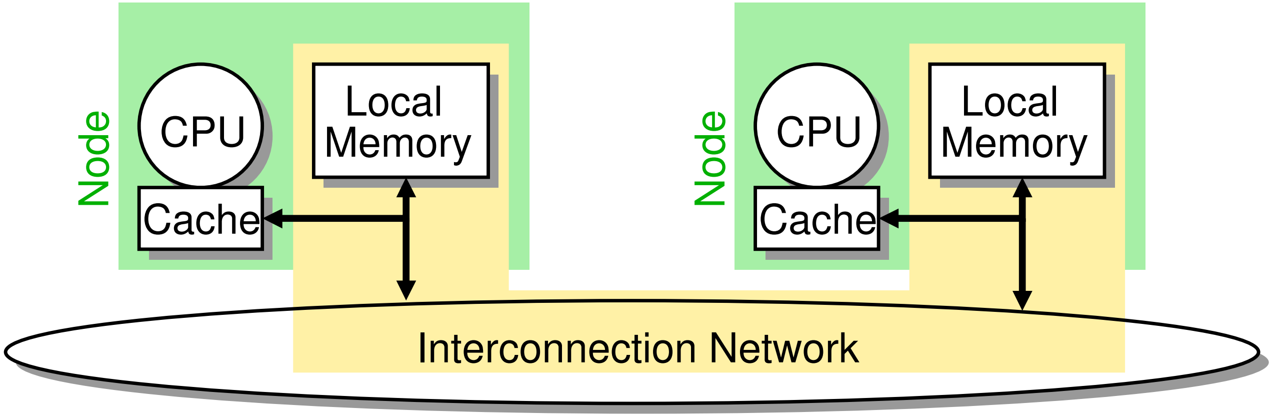
Symmetrical multiprocessors (SMP)



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

2.4.2 MIMD: Shared Memory Systems ...

Multiprocessor systems with distributed shared memory (DSM)



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

2.4.2 MIMD: Shared Memory Systems ...

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

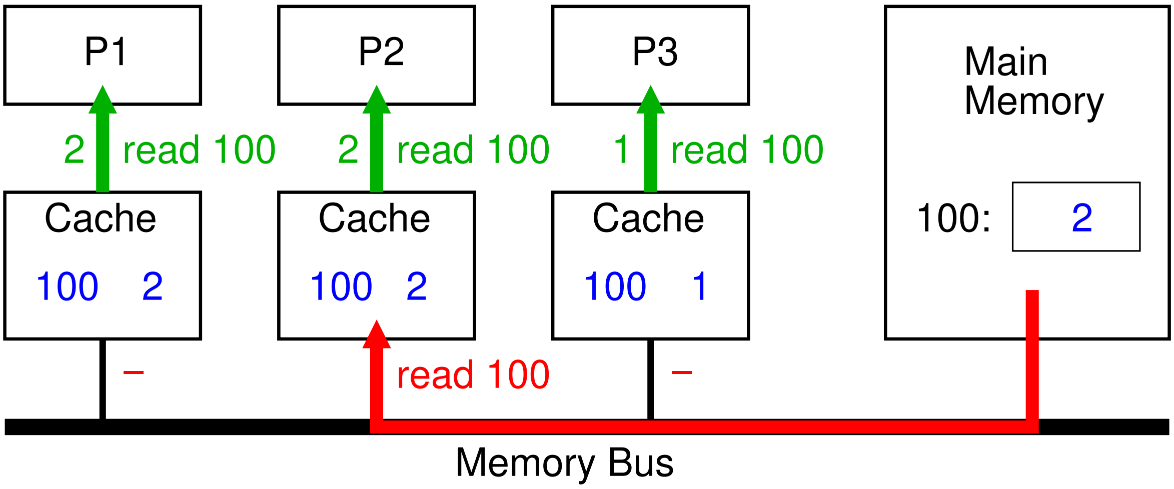
Caches in shared memory systems

- **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 existance of multiple copies of data cean lead to inconsistencies: **cache coherence problem** (☞ **BS-1**)

(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

2.4.2 MIMD: Shared Memory Systems ...

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?

2.4.2 MIMD: Shared Memory Systems ...

Memory consistency: a simple example

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

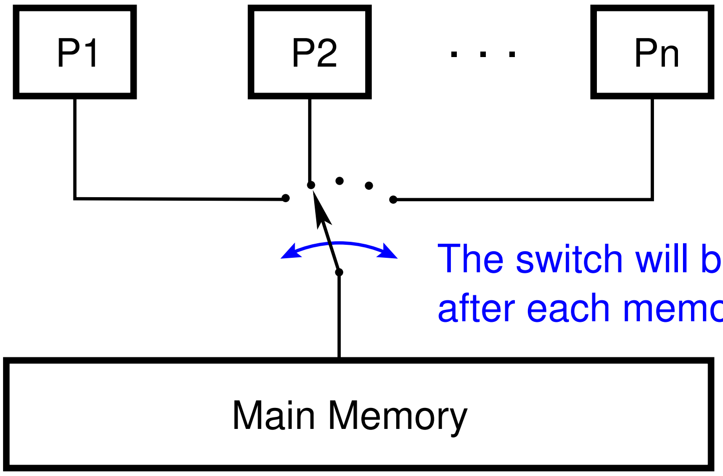
- Intuitive expectation: the output `"0 0"` can never occur
- But: with many SMPs/DSMs the output `"0 0"` is possible
  - (CPUs with dynamic instruction scheduling or write buffers)
- In spite of cache coherency: intuitively inconsistent view on the main memory:

T_1: `A=1, B=0` T_2: `A=0, B=1`

2.4.2 MIMD: Shared Memory Systems ...

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:



Processors execute memory operations in program order

The switch will be randomly switched after each memory access

2.4.2 MIMD: Shared Memory Systems ...

Interleavings (*Verzahnungen*) in the example

Some possible execution sequences using the abstract machine:

```
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
```

No sequential consistency:

```
A = 0
        B = 0
        B = 1
        print A
A = 1
print B

B=0  A=0
```

2.4.2 MIMD: Shared Memory Systems ...

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 `atomic` / `volatile`!

2.4.2 MIMD: Shared Memory Systems ...

Consequences of weak consistency: examples

- all variables are initially 0

		Possible results with sequential consistency	"unexpected" behavior with weak consistency:

`A=1;`
`print B;`
	`B=1;`
	`print A;`
		0,1
		1,0
		1,1
			due to swapping of the read and write accesses

`A=1;`
`valid=1;`
	`while (!valid);`
	`print A;`
		1	due to swapping of the write accesses to A and valid

2.4.2 MIMD: Shared Memory Systems ...

Weak consistency models ...

- Memory consistency can (and must!) be enforced as needed, using special instrcutions
  - *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:

`A=1;`
`fence;`
`print B;`
	`B=1;`
	`fence;`
	`print A;`
		Fence ensures that the write access is finished before reading

`A=1;`
`fence;`
`valid=1;`
	`while (!valid);`
	`fence;`
	`print A;`
		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.

2.4.3 SIMD

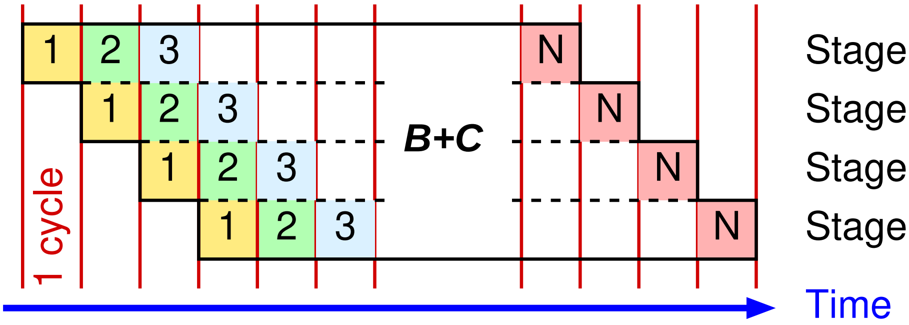
- Only a single instruction stream, however, the instrcutions have **vectors** as operands ⇒ data parallelism
- **Vector** = one-dimensional array of numbers
- Variants:
  - vector computers
    - pipelined arithmetic units (vector units) for the processing of vectors
  - SIMD extensions in processors (SSE, AVX)
    - Intel: 128 Bit registers with, e.g., four 32 Bit `float` 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 $j = 1, ..., N$
- 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:

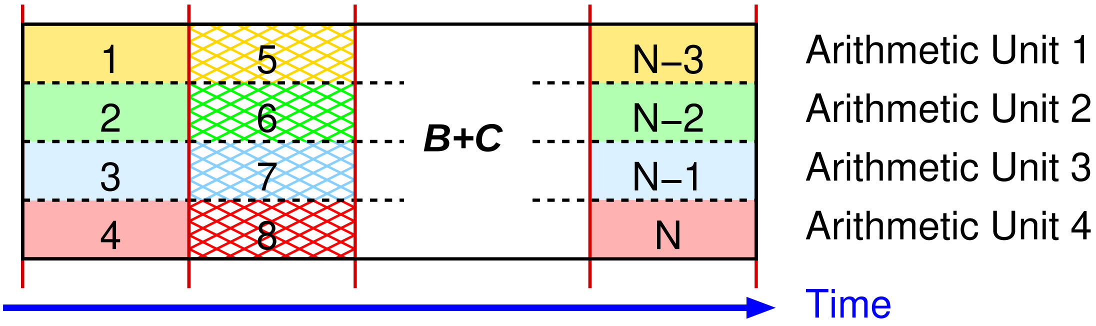


2.4.3 SIMD ...

(Animated slide)

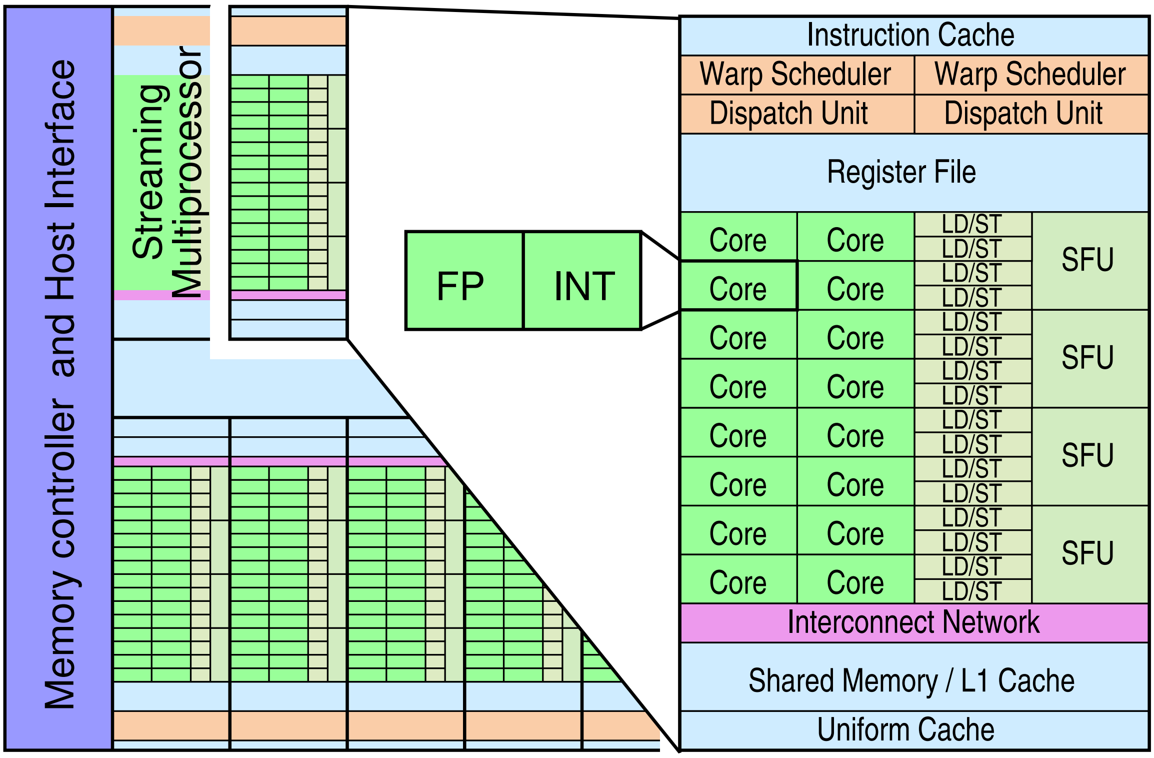
Example: addition of two vectors

- $A_j = B_j + C_j$, for all $j = 1, ..., N$
- SSE 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:



(Animated slide)

Architecture of a GPU (NVIDIA Fermi)



Notes for slide 113:

The current NVIDIA Ampere Architecture is similar, with some differences:

- it has a total of 8192 cores
- in addition to int32 and fp32 cores, each SM also has 8 fp64 cores and one tensor core.

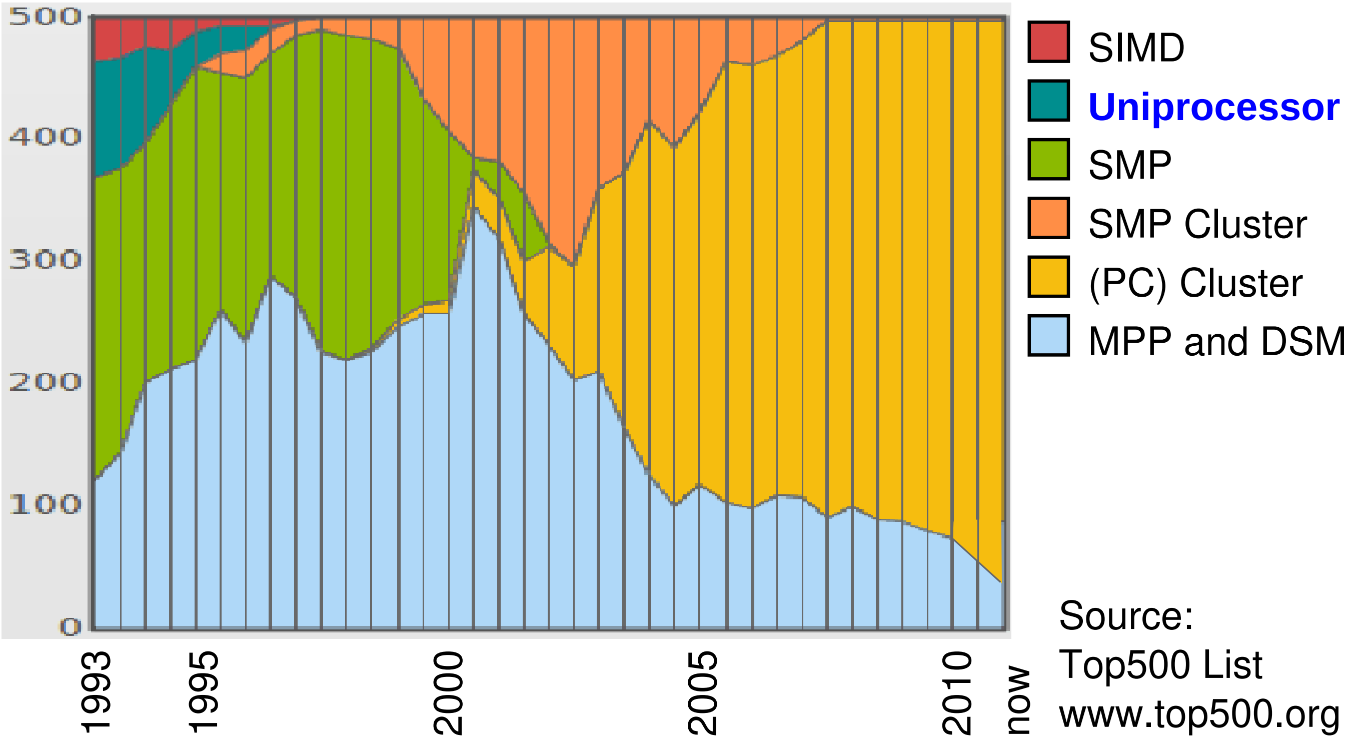
2.4.3 SIMD ...

Programming of GPUs (NVIDIA Fermi)

- Partitioning of the code in groups (*warps*) of 32 threads
- *Warps* are distributed to the streaming multiprocessors (SEs)
- Each of the two *warp schedulers* of an SE executes one instruction with 16 threads per clock cycle
  - in a SIMD manner, i.e., the cores all execute the same instruction (on different data) or none at all
  - e.g., with `if-then-else`:
    - first some cores execute the `then` branch,
    - then the other cores execute the `else` branch
- Threads of one warp should address subsequent memory locations
  - only in this case, memory accesses can be merged

2.4.4 High Performance Supercomputers

Trends



Source:
Top500 List
www.top500.org

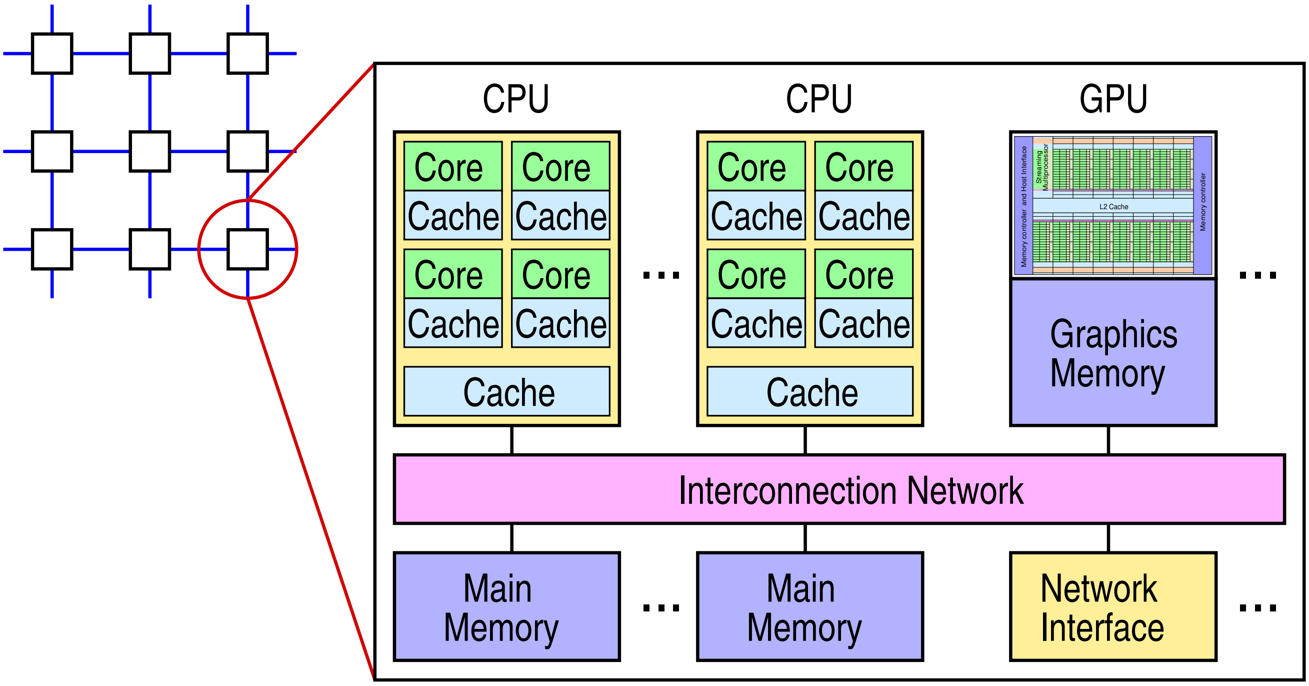
2.4.4 High Performance Supercomputers ...

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

2.4.4 High Performance Supercomputers ...

Typical architecture: ...



Parallel Processing

Winter Term 2024/25

22.10.2024

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

Stand: January 14, 2025

Lunch with the ET-I Profs

- **Wednesday, Oct. 23rd**
  12:00 - 13:00
  LEO Paul-Bonatz Campus
- Explicit offer for asking questions to ETI professors, e.g.:
  - can I write my Thesis abroad?
  - what kind of industry collaborations do you have?
  - hat sort of Erasmus partnerships do you have?
  - which lectures would you recommend me to take?
  - ...?

2.5 Parallel Programming Models

In the followig, 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 118:

- 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.

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 panalty)
    - *shared virtual memory* (SVM)
- Examples:
  - PThreads, Java Threads, C++ Threads
  - Intel Threading Building Blocks (TBB)
  - OpenMP (☞ **3.1**)

2.5.1 Shared Memory ...

Example for data exchange

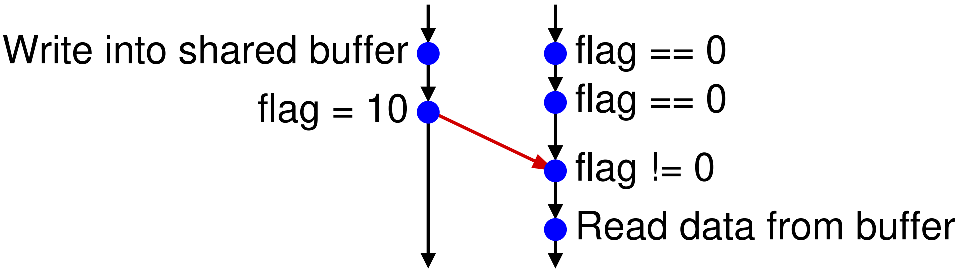
Producer Thread

```
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

2.5.2 Message Passing ...

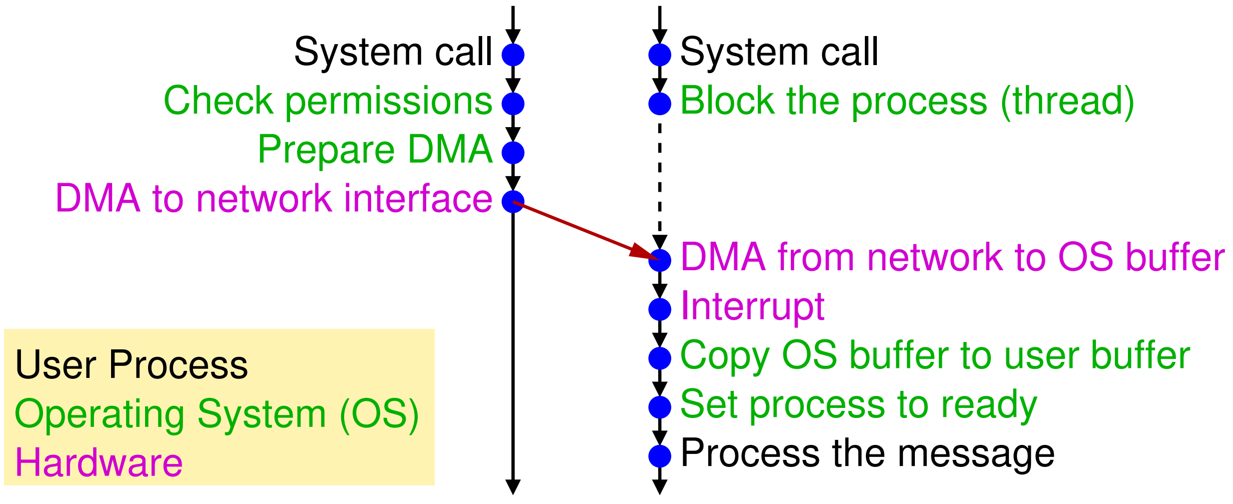
Example for data exchange

Producer Process

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

Consumer Process

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



System call
Check permissions
Prepare DMA
DMA to network interface

System call
Block the process (thread)
DMA from network to OS buffer
Interrupt
Copy OS buffer to user buffer
Set process to ready
Process the message

User Process
Operating System (OS)
Hardware

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

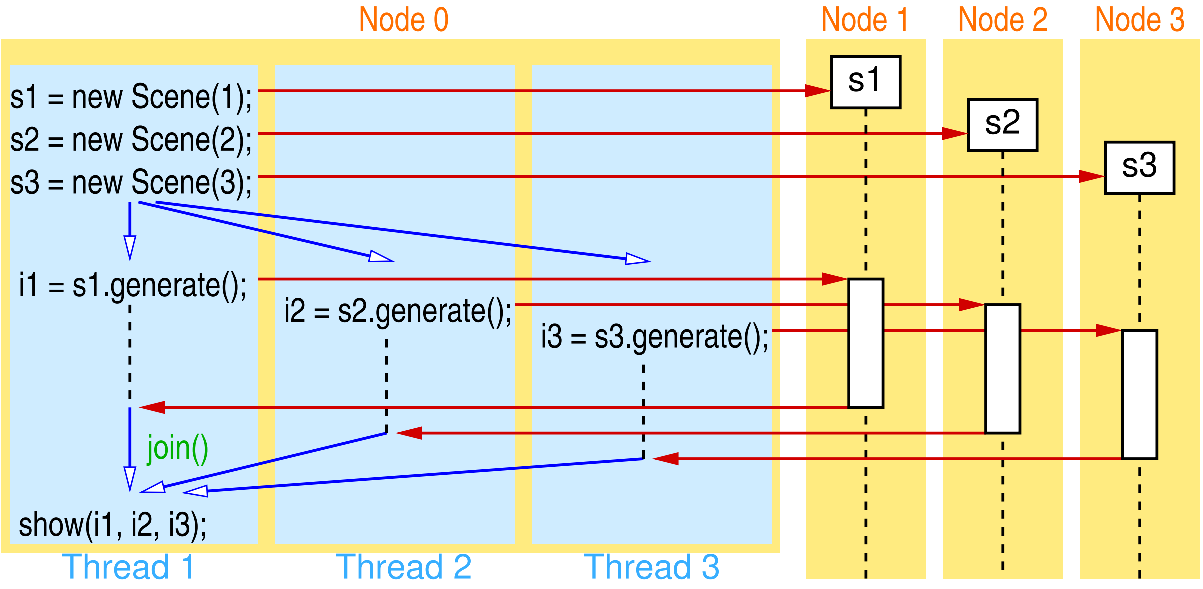
Notes for slide 123:

Example

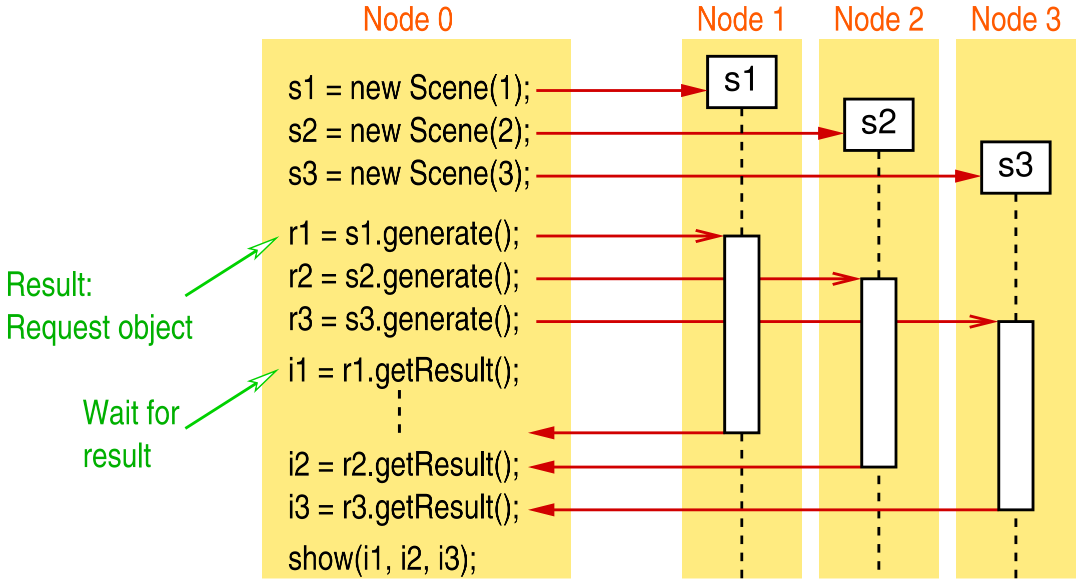
- Class `Scene` as description of a scene
  - constructor `Scene(int param)`
  - method `Image generate()` computes the image
- Computation of three images with different parameters (sequentialy):

```
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);
```

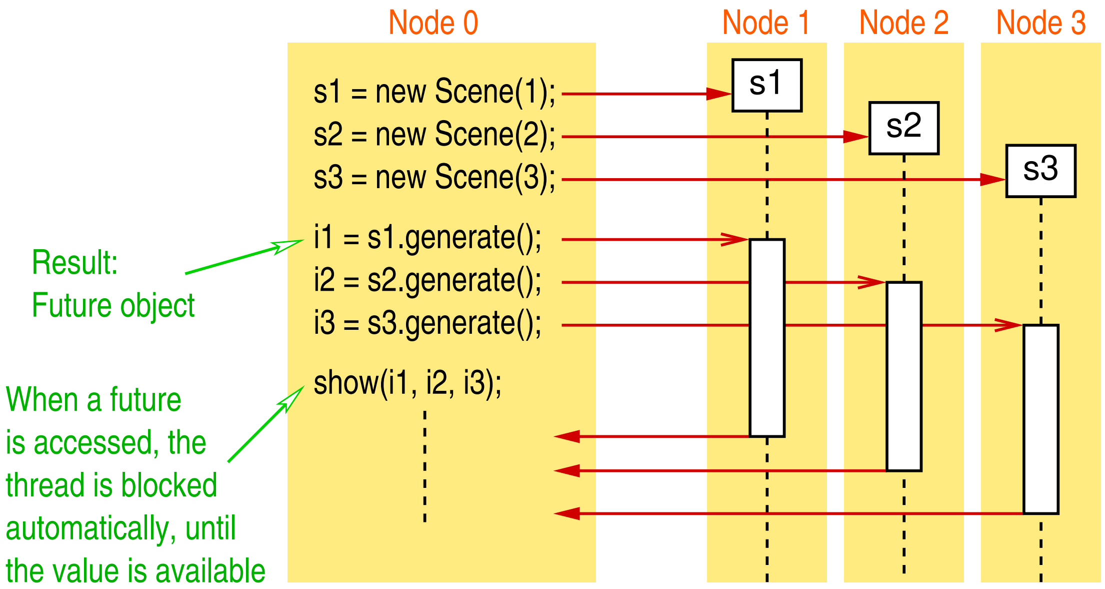
Parallel computing with threads



Parallel computing with asynchronous RPC



Parallel computing with futures



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

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

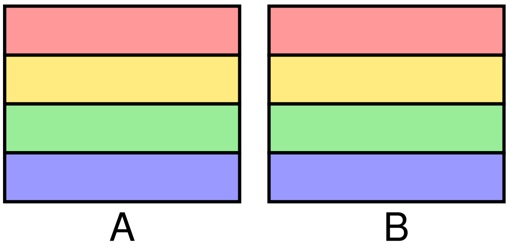
(Animated slide)

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 125:

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.

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

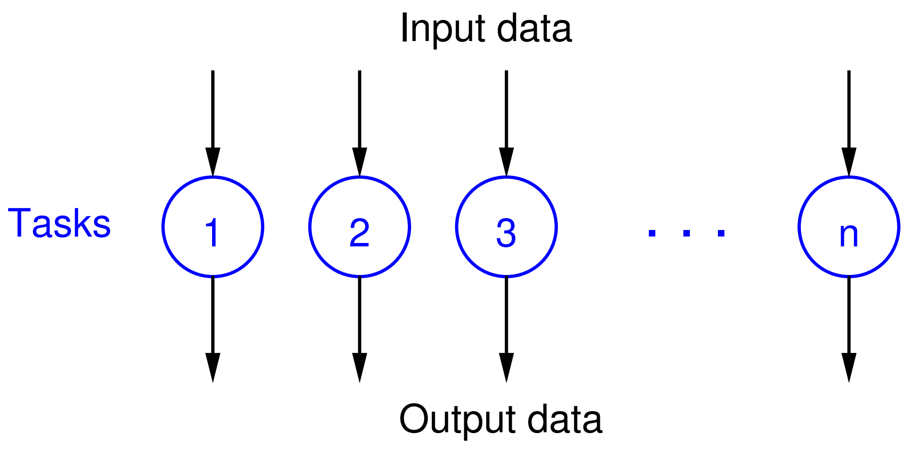
- Models / patterns for parallel programs

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


Input data

Tasks

1

2

3

. . .

n

Output data

2.7.1 Embarrassingly Parallel ...

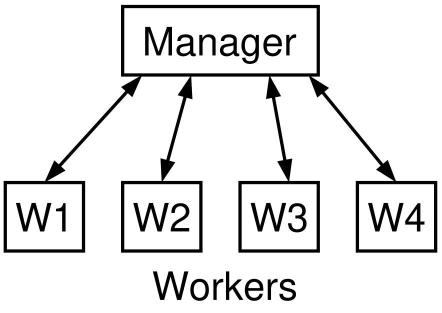
Examples for embarrassingly parallel problems

- 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 Organisation Forms for Parallel Programs ...

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

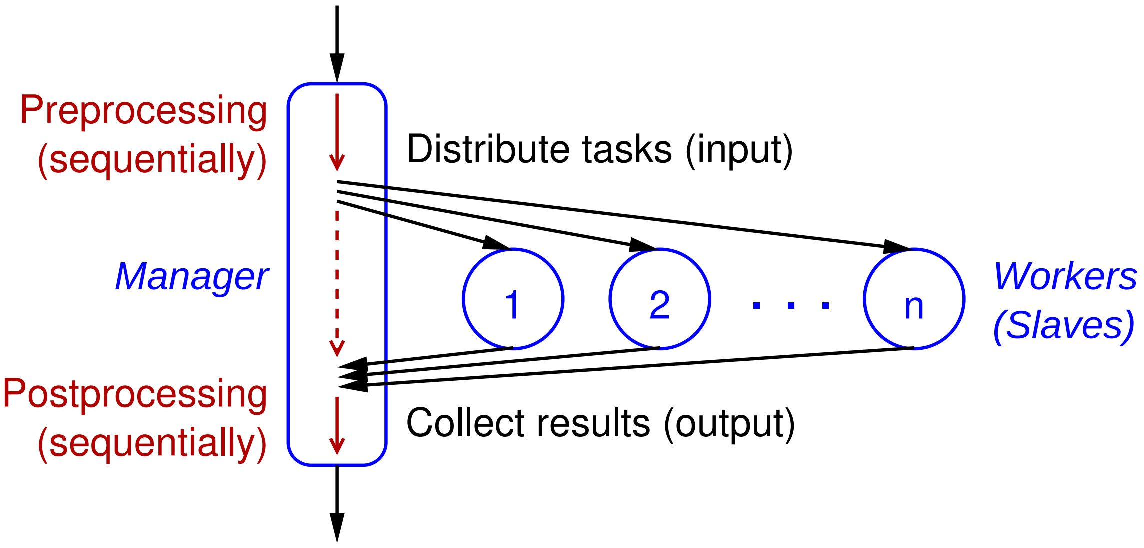


- The manager (or sometimes also the workers) can create additional tasks, while the workers are working
- The manager can become a bottleneck
- The manager should be able to receive the results asynchronously (non blocking)

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

Typical application

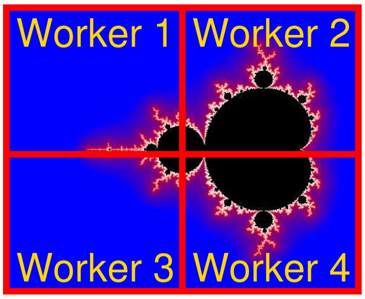
- Often only a part of a task can be parallelised in an optimal way
- In the easiest case, the following flow will result:



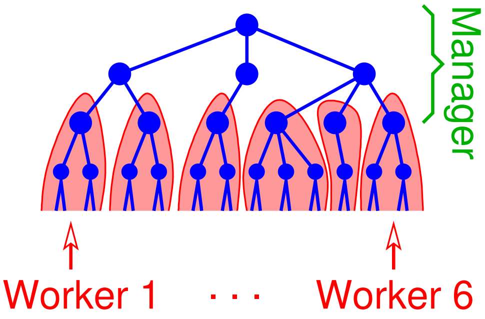
2.7.2 Manager/Worker Model (Master/Slave Model) ...

Examples

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



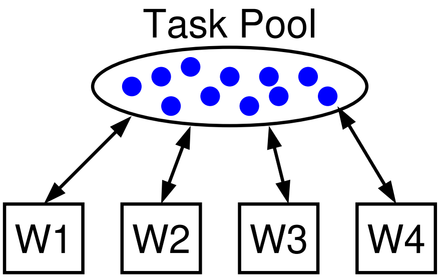
- Tree search
  - manager traverses the tree up to a predefined depth; the workers process the sub-trees



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



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

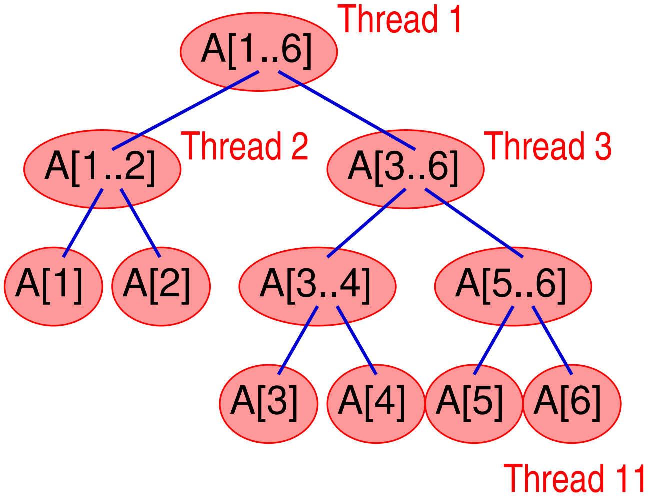
2.7.4 Divide and Conquer ...

(Animated slide)

Example: parallel quicksort

```
Qsort($A_{1\,..\,n}$)
    If $n = 1$: done.
    Else:
        Determine the pivot $S$.
        Reorder $A$ such that
        $A_i \le S$ for $i \in [1, k[$ and
        $A_i \ge S$ for $i \in [k, n]$.
        Execute Qsort($A_{1\,..\,k-1}$)
        and Qsort($A_{k\,..\,n}$)
        in parallel.
```

Example:



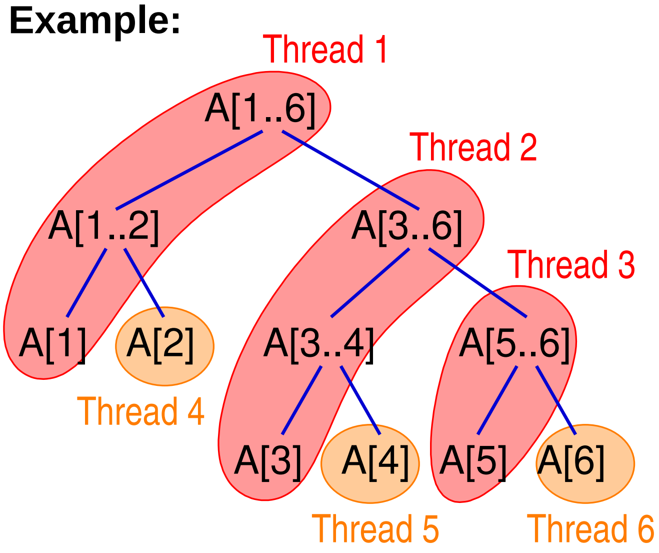
2.7.4 Divide and Conquer ...

(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

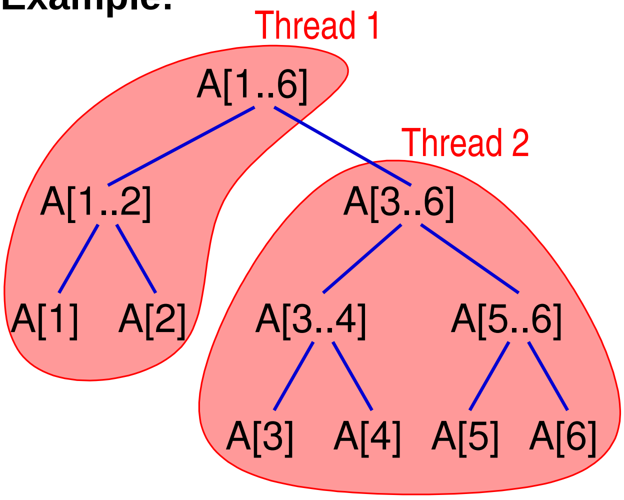
2.7.4 Divide and Conquer ...

(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

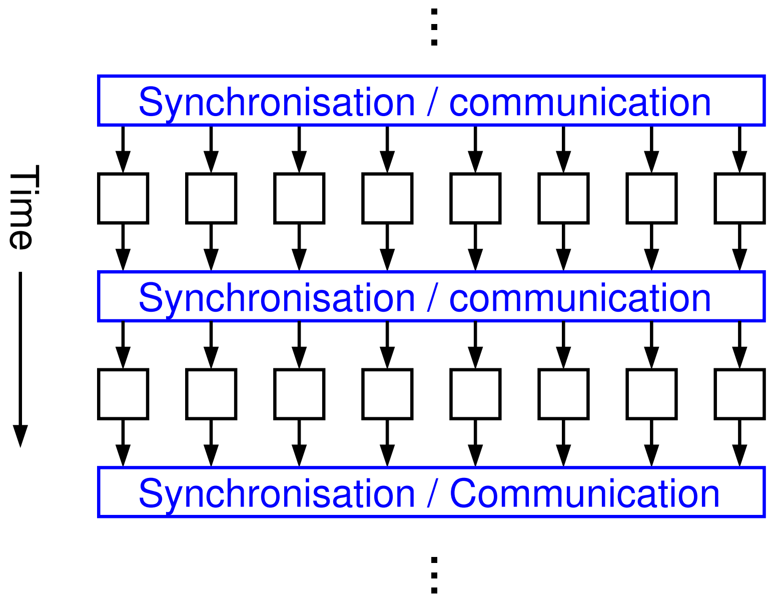
2.7 Organisation Forms for Parallel Programs ...

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

2.7.5 Data parallel Model: SPMD ...

Typical sequence



⋮

Synchronisation / communication

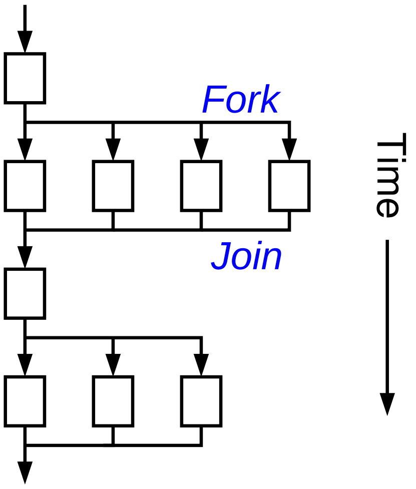
Synchronisation / communication

Synchronisation / Communication

⋮

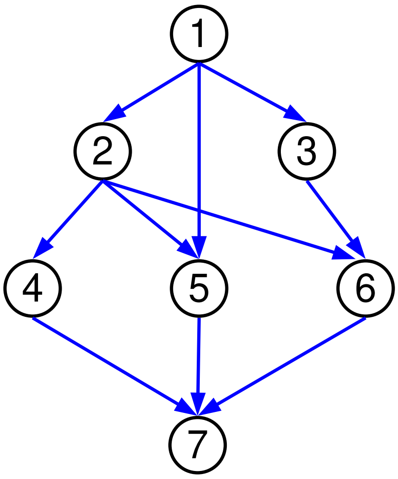
2.7.6 Fork/Join Model

- 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

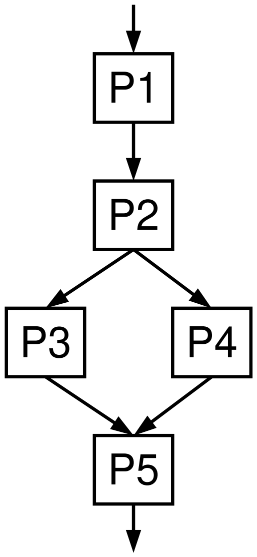
- Tasks and their dependences (data flow) are represented as a graph
- An edge in the graph denotes a data flow
  - e.g., task 1 produces data, task 2 starts execution, when this data is entirely available



- 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



Parallel Processing

Winter Term 2024/25

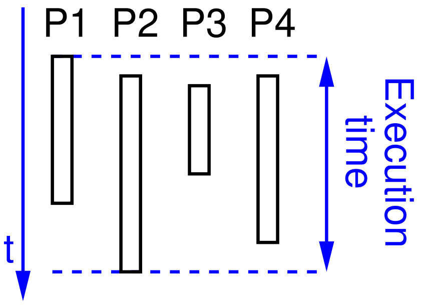
28.10.2024

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

Stand: January 14, 2025

2.8 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



2.8.1 Performance Metrics

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

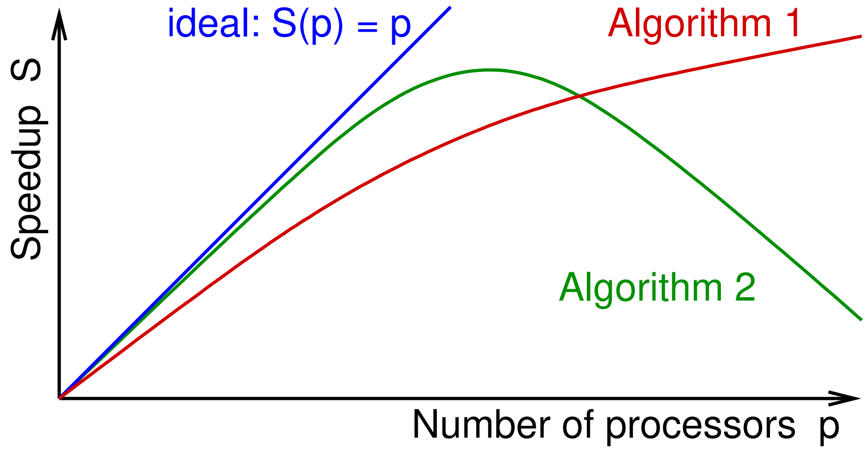
- Optimum: $S(p) = p$
- Often: with **fixed** problem size, $S(p)$ declines again, when p increases
  - more communication, less computing work per processor

Notes for slide 145:

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

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:

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

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

Notes for slide 147:

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

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

2.8.1 Performance Metrics ...

Superlinear speedup

- Sometimes we observe $S(p) > p$, although this should actually be impossible
- Causes:
  - 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
  - cache effects
    - with p processors, the amount of cache is p times higher that with one processor
    - thus, we also have higher cache hit rates

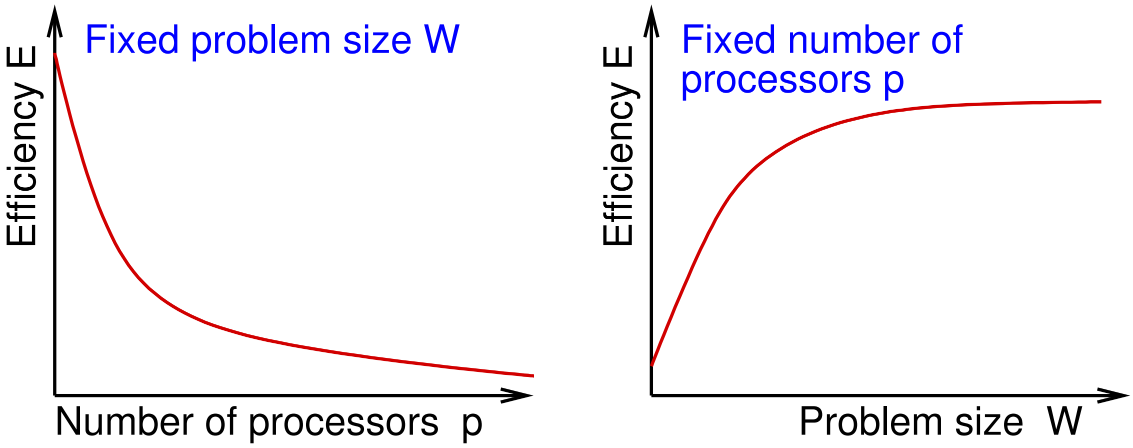
Efficiency

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

- 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 151:

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

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

2.8.1 Performance Metrics ...

Scalability ...

- Isoefficiency function $I(p)$
  - solution of the equation $W = K \cdot T_o(W, p)$ w.r.t. W
  - K = constant, depending on the required efficiency
- Good scalability: $I(p) = \mathcal{O}(p)$ or $I(p) = \mathcal{O}(p \log p)$
- Bad scalability: $I(p) = \mathcal{O}(p^k)$
- Computation of $T_o(W, p)$ by analysing the parallel algorithm
  - how much time is needed for communication / synchronisation and potentially additional computations?
  - more details and examples in chapter 2.8.5

2.8.2 Reasons for Performance Loss

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

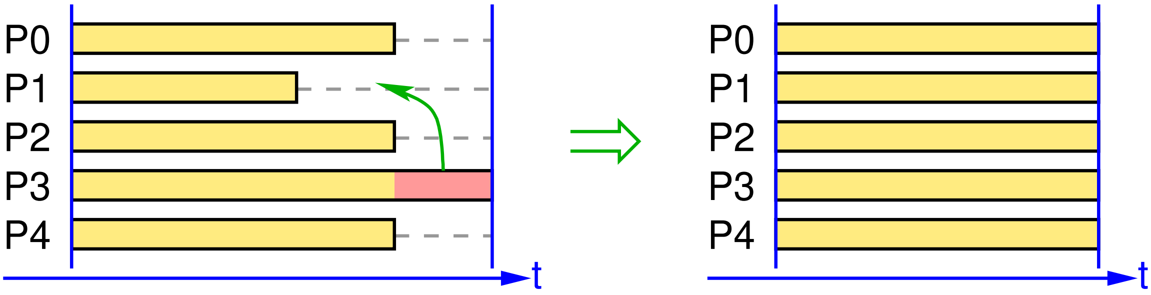
2.8.2 Reasons for Performance Loss ...

- **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

2.8.3 Load Balancing

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

2.8.3 Load Balancing ...

Reasons for load imbalance

- Unequal computational load of the tasks
  - e.g., atmospheric model: areas over land / water
- Heterogeneous execution plattform
  - e.g., processors with different speed

static

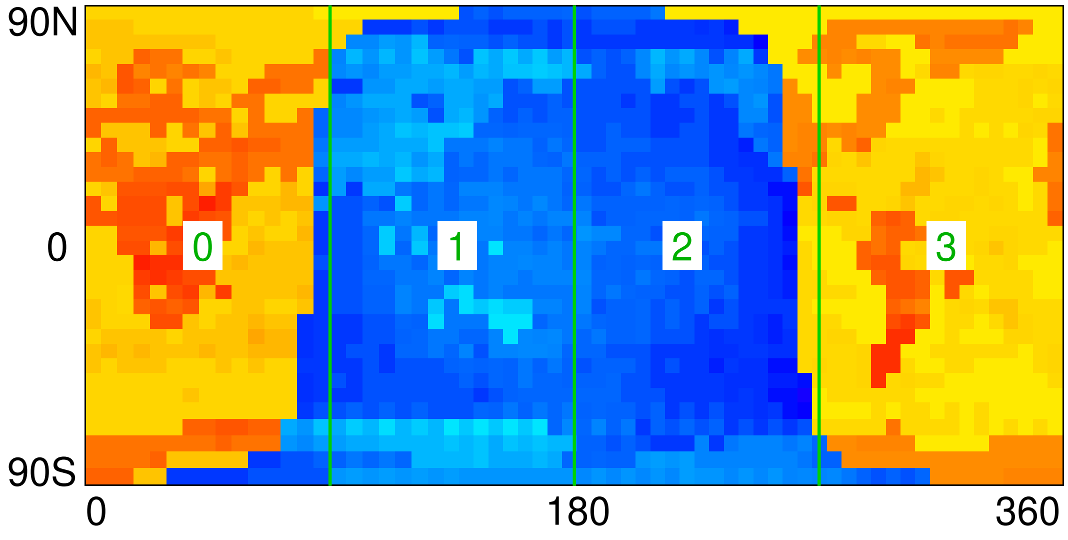
- Computational load of the tasks changes dynamically
  - e.g., in atmospheric model, depending on the simulated time of day (solar radiation)
- Background load on the processors
  - e.g., in a PC cluster

dynamic

2.8.3 Load Balancing ...

(Animated slide)

Example: atmospheric model



🟦 Night: low execution time per task

🟥 Day: high execution time per task

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

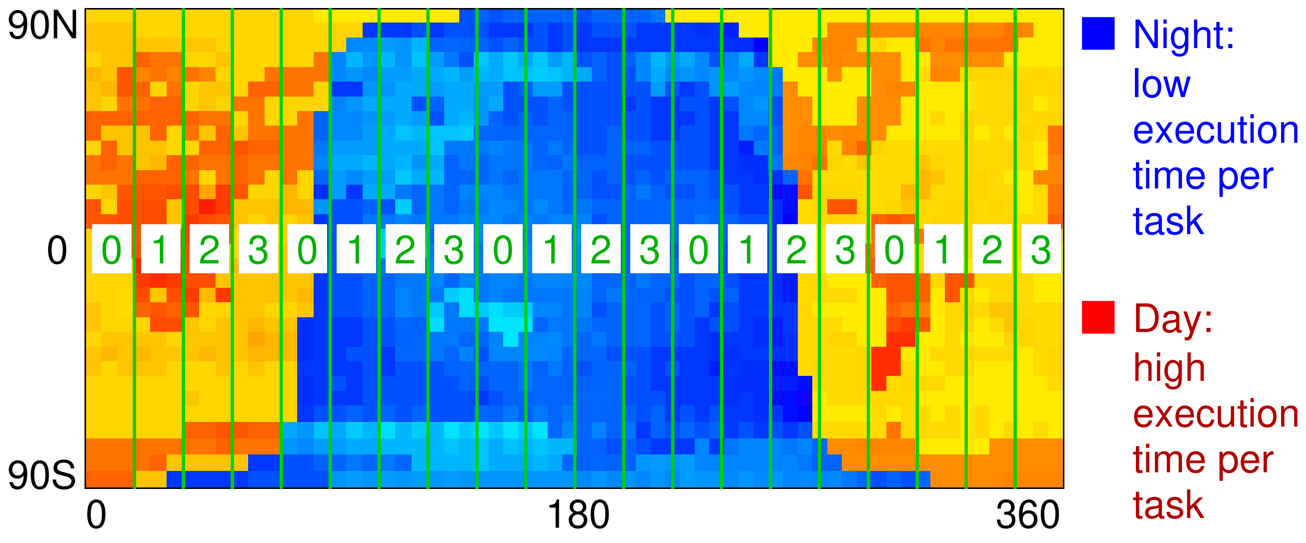
2.8.3 Load Balancing ...

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

2.8.3 Load Balancing ...

Example: atmospheric model, cyclic mapping



- Each processor has tasks with high and low computational load

2.8.3 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

2.8.4 Performance Analysis of Parallel Software

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

2.8.5 Analytical Performance Modelling

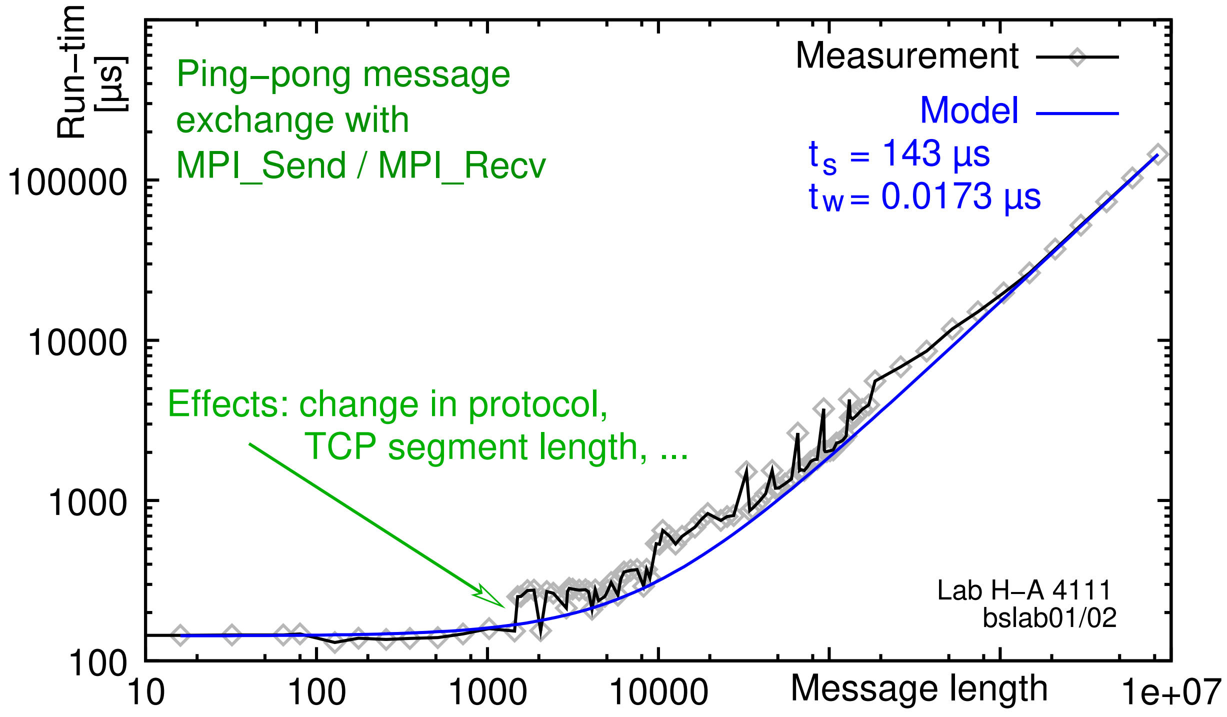
Models for communication time

- 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:

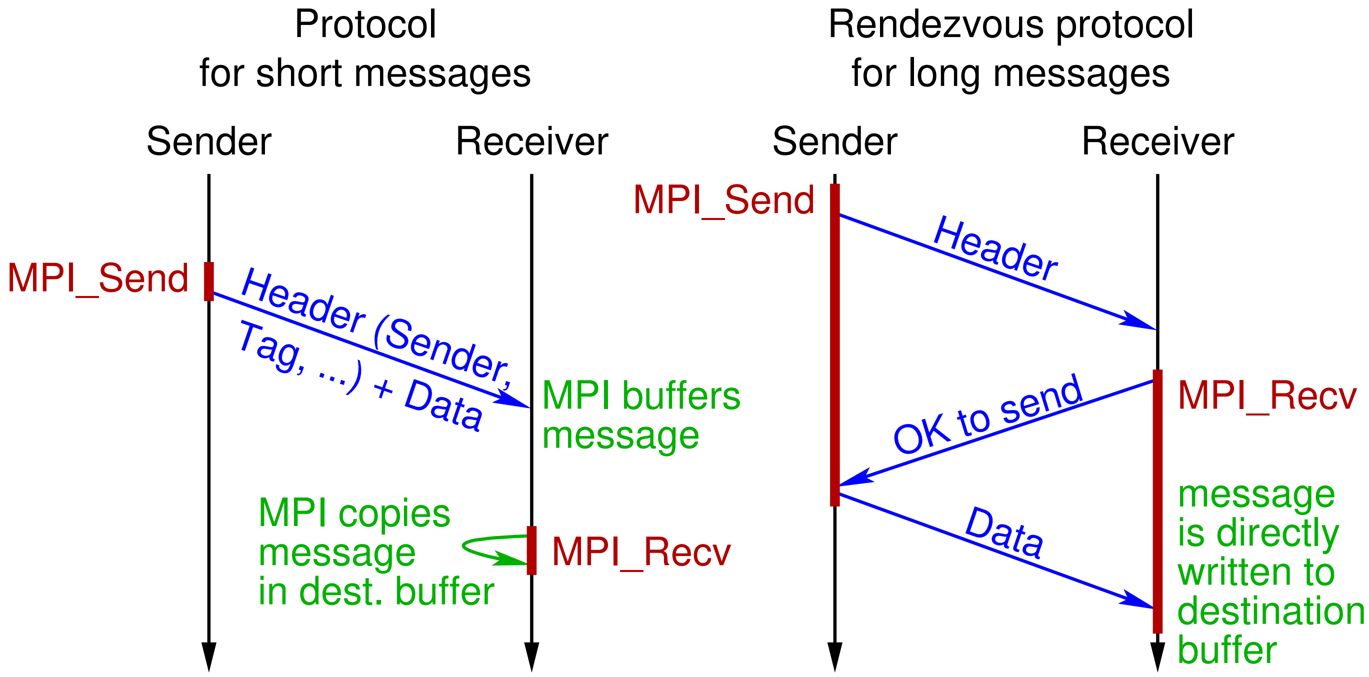
PC cluster H-A 4111:	$t_s = 71.5\ \mu s,$	$t_w = 8{,}6\ ns$
SMP cluster (remote):	$t_s = 25.6\ \mu s,$	$t_w = 8{,}5\ ns$
SMP cluster (local):	$t_s = 0{,}35\ \mu s,$	$t_w = 0{,}5\ ns$

2.8.5 Analytical Performance Modelling ...

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 $k = 1 ... \sqrt{p}$
- 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 $C_{ij} = \sum_{k=1}^{\sqrt{p}} A_{ik} \cdot B_{kj}$

Parallel Processing

Winter Term 2024/25

29.10.2024

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

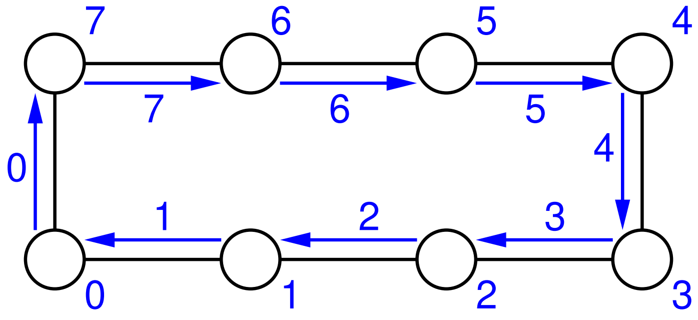
Stand: January 14, 2025

2.8.5 Analytical Performance Modelling ...

(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

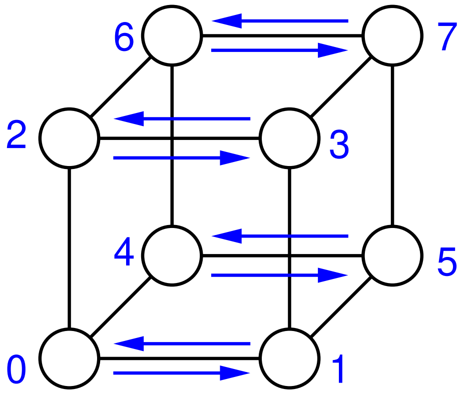
- Cost: $t_s(p-1) + t_w m(p-1)$ (m: data length)

2.8.5 Analytical Performance Modelling ...

(Animated slide)

All-to-all broadcast ...

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

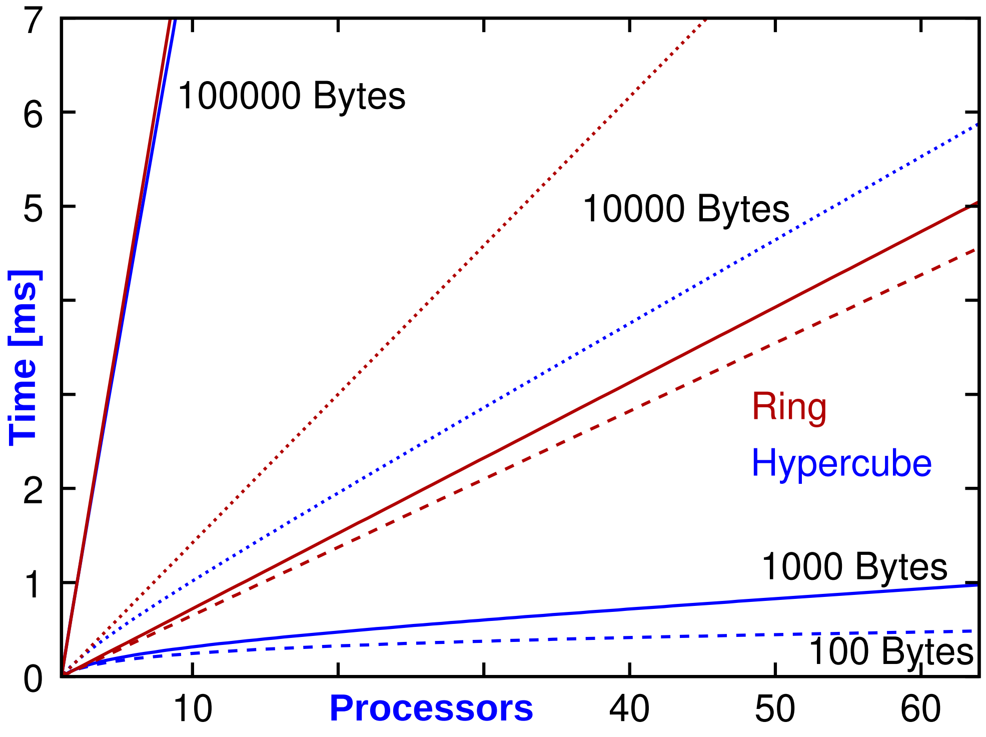


1. Pairwise exchange in x direction
2. Pairwise exchange in y direction
3. Pairwise exchange in z direction

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

2.8.5 Analytical Performance Modelling ...

All-to-all broadcast ...



t_s = 71,5 µs

t_w = 8,65 ns

(Data from Lab H–A 4111)

2.8.5 Analytical Performance Modelling ...

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 169:

When we compare the result for $T(p)$ with the formula $T(p) = \frac{W+T_o(W,p)}{p}$ slide from 151, 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:

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

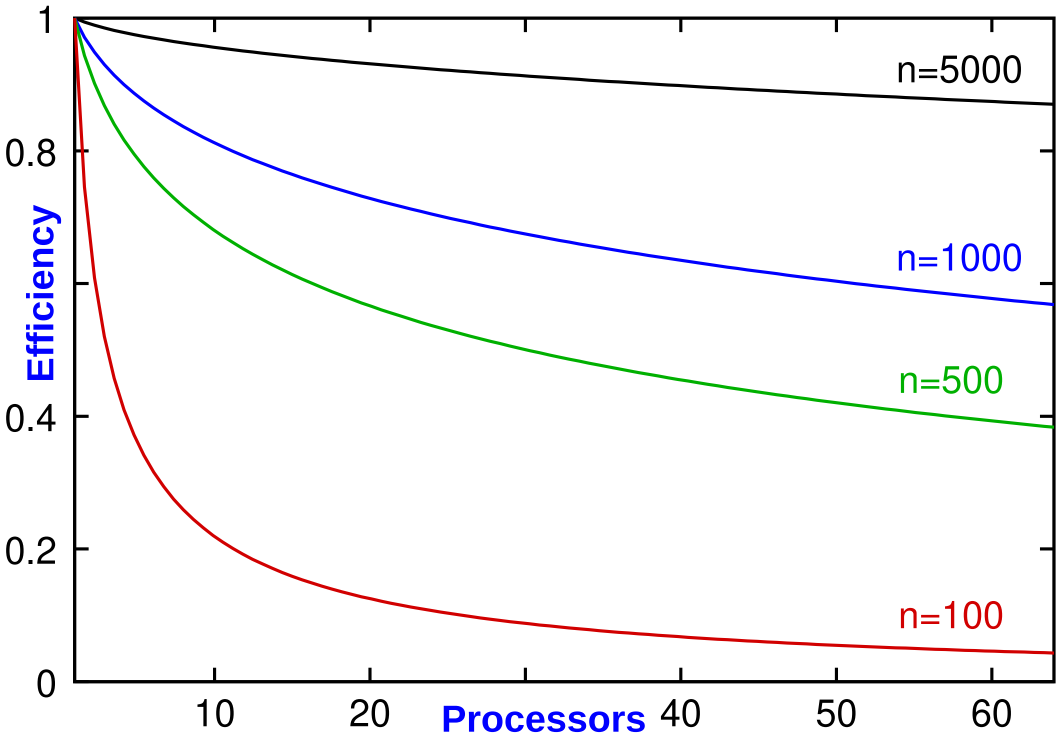
Or (by dividing both sizes by n and reordering):

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

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

2.8.5 Analytical Performance Modelling ...

Efficiency of matrix multiplication



Execution time per matrix element:

t_c = 1.3 ns

t_s = 71,5 µs

t_w = 69.2 ns (1 double value)

(Data from Lab H–A 4111)

2.8.6 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

2.8.6 Performance Analysis Tools ...

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, ...

2.8.6 Performance Analysis Tools ...

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

2.8.6 Performance Analysis Tools ...

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)

2.8.6 Performance Analysis Tools ...

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;
```

2.8.6 Performance Analysis Tools ...

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

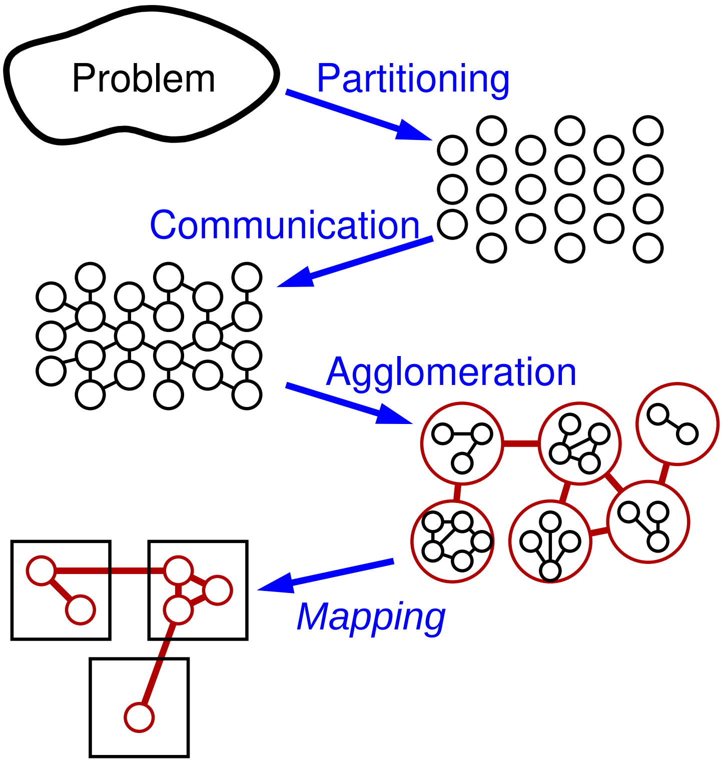
2.9 A Design Process for Parallel Programs

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)

2.9 A Design Process for Parallel Programs ...



Problem

Partitioning

Communication

Agglomeration

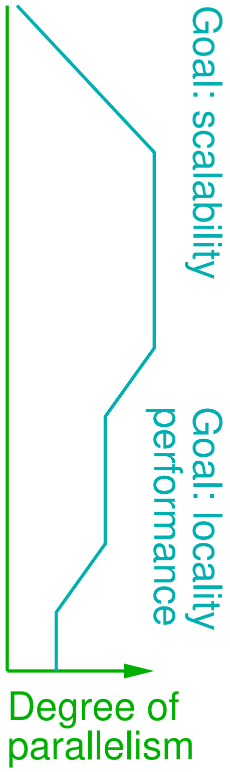
Mapping

Split the problem in as many small tasks as possible

Data exchange between tasks

Merging of Tasks

Mapping to Processors



2.9.1 Partitioning

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

Data partitioning (data parallelism)

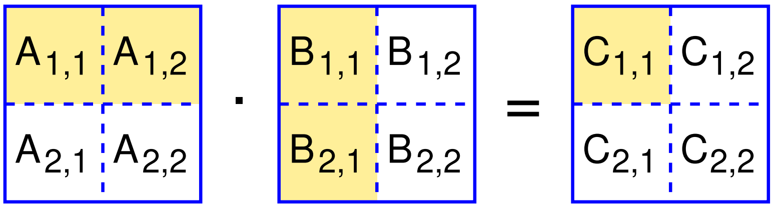
- Tasks specify **identical computaions** for a **part** of the data
- In general, high degree of parallelism is possible
- We can distribute:
  - input data
  - output data
  - intermediate data
- In some cases: recursive partitioning (*divide and conquer*)
- Special case: partitioning of search space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

2.9.1 Partitioning ...

Example: matrix multiplication

- Product $C = A \cdot B$ of two square matrices
  - $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$, 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:



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

2.9.1 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

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

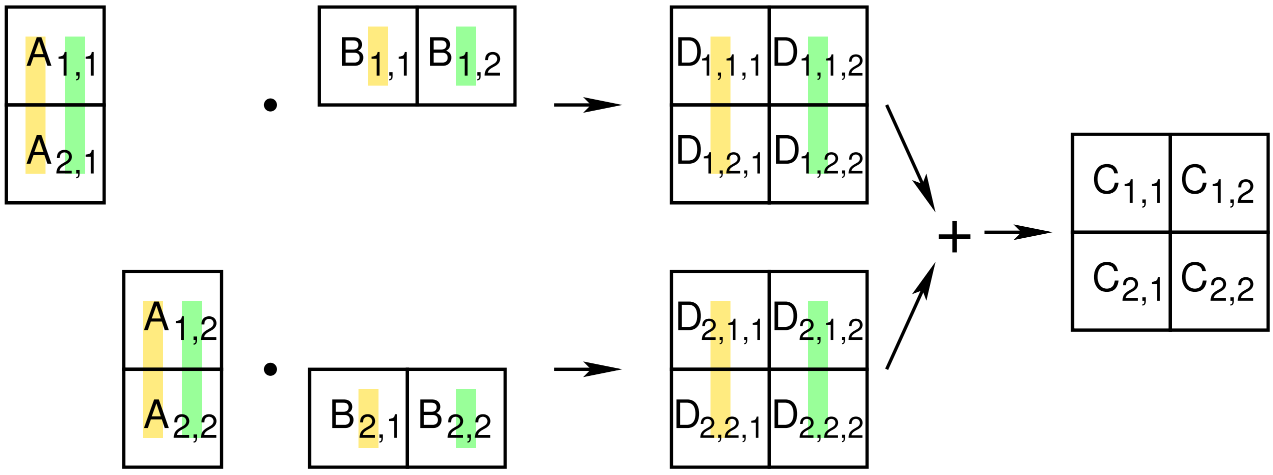
- Results in four independent tasks:

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

2.9.1 Partitioning ...

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

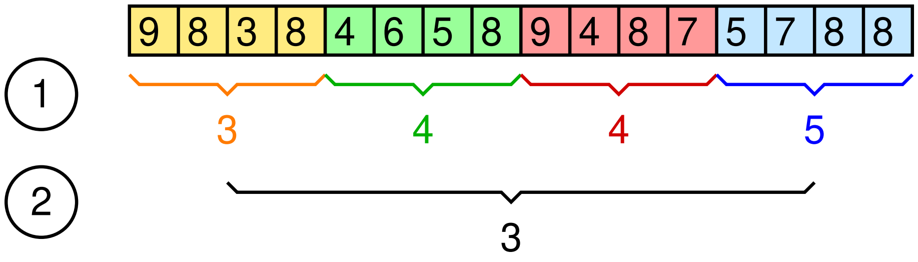
- Distribution of intermediate data (higher degree of parallelism)
  - here: 8 multiplications of sub-matrices



2.9.1 Partitioning ...

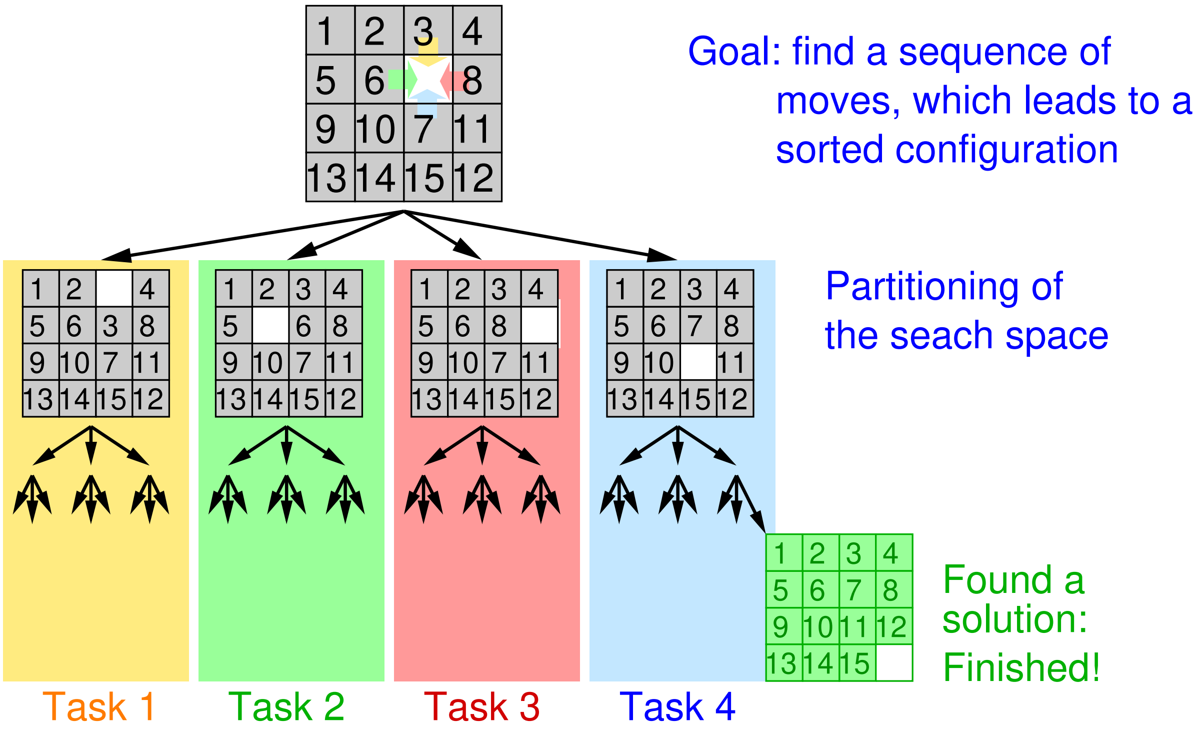
Example: minimum of an array

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



2.9.1 Partitioning ...

Example: sliding puzzle (partitioning of search space)



Goal: find a sequence of moves, which leads to a sorted configuration

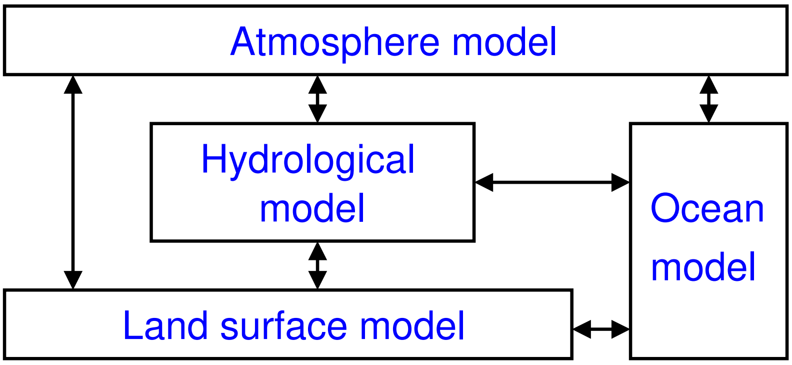
Partitioning of the seach space

Found a solution: Finished!

2.9.1 Partitioning ...

Task partitioning (task parallelism)

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


Atmosphere model

Hydrological model

Ocean model

Land surface model

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

2.9.2 Communication

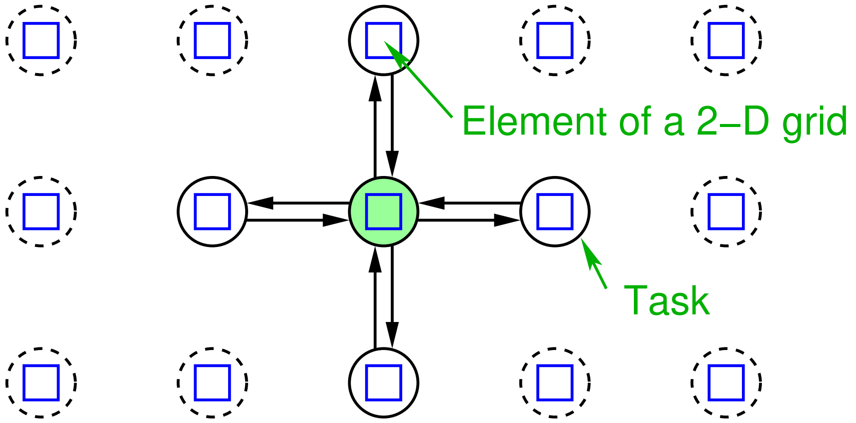
- Two step approach
  - definition of the communication structure
    - who must exchange data with whom?
    - sometimes complex when using data partitioning
    - often simple when using task partitioning
  - definition of the messages to be sent
    - which data must be exchanged when?
    - taking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

2.9.2 Communication ...

Different communication patterns:

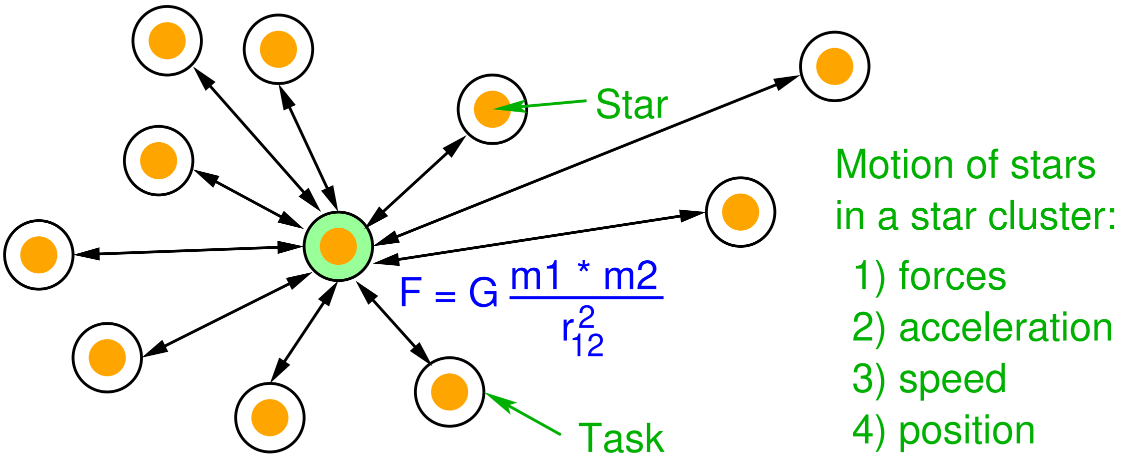
- Local vs. global communication
  - lokal: task communicates only with a small set of other tasks (its “neighbors”)
  - global: task communicates with many/all other tasks
- Structured vs. unstructured communication
  - structured: regular structure, e.g., grid, tree
- Static vs. dynamic communication
  - 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



$F = G \frac{m1 * m2}{r_{12}^2}$

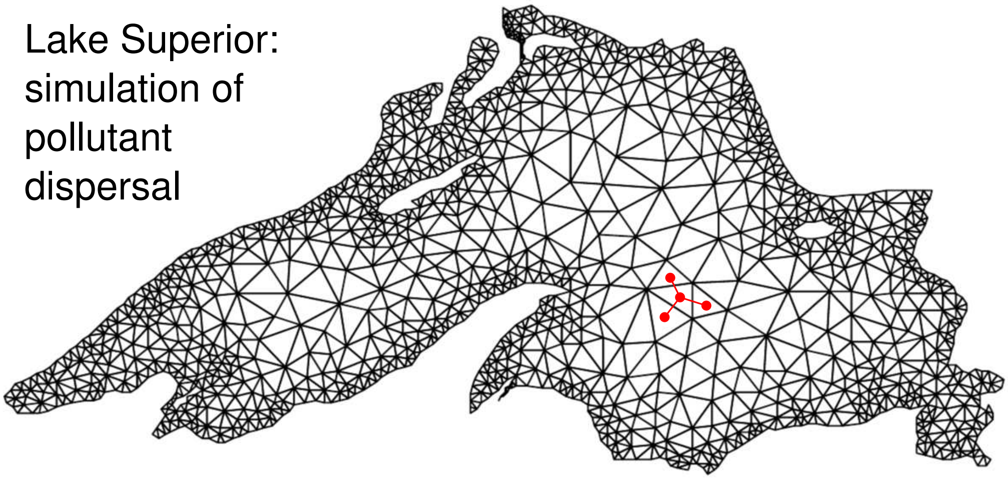
- The effective force on a star in a star cluster depends on the masses and locations of all other stars
  - possible approximation: restriction to relatively close stars
    - will, however, result in dynamic communication

2.9.2 Communication ...

Example for structured / unstructured communication

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

Lake Superior: simulation of pollutant dispersal



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

2.9.3 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
    - sufficently fine-grained parallelism for mapping phase

Parallel Processing

Winter Term 2024/25

04.11.2024

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

Stand: January 14, 2025

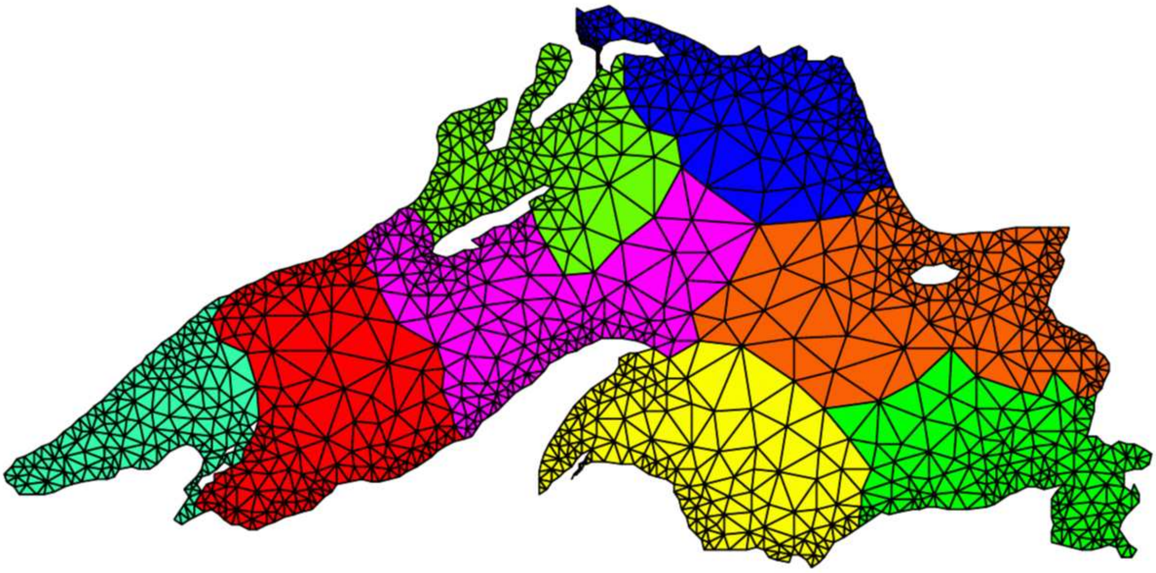
2.9.4 *Mapping*

- 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 2024/25

3 Parallel Programming with Shared Memory

3 Parallel Programming with Shared Memory ...

Contents

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

Literature

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

3 Parallel Programming with Shared Memory ...

Approaches to programming with threads

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

3.1 OpenMP Basics

Background

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

3.1 OpenMP Basics ...

Parallelization using directives

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

3.1 OpenMP Basics ...

Parallelization using directives ...

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

3.1 OpenMP Basics ...

Parallelization using directives: discussion

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

3.1 OpenMP Basics ...

Compiling and executing OpenMP programs

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

3.1.1 The `parallel` directive

(Animated slide)

An example (☞ `03/firstprog.cpp`)

Program

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

Compilation

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

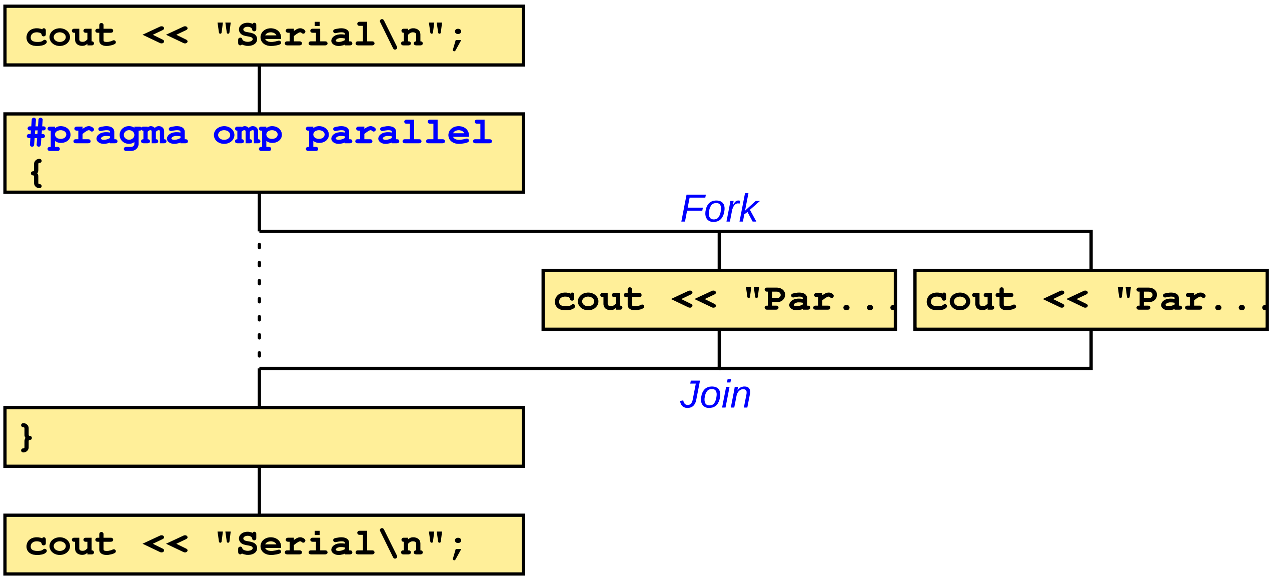
Execution

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

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

3.1.1 The `parallel` directive ...

Execution model: fork/join



```
cout << "Serial\n";
#pragma omp parallel
{
cout << "Par..
cout << "Par..
}
cout << "Serial\n";
```

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)

3.1.1 The `parallel` directive ...

Shared and private variables

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

3.1.1 The `parallel` directive ...

Shared and private variables ...

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

Notes for slide 208:

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

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

3.1.1 The `parallel` directive ...

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

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

Each thread has an initialized copy of j

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

h is private

Accesses to k usually should be synchronized!

Output (with 2 threads)**:**

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

3.1.2 Library routines

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

3.1.2 Library routines ...

Example using library routines (☞ `03/threads.cpp`)

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

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

3.2 Loop parallelization

Motivation

- Implementation of data parallelism
  - threads perform identical computations on different parts of the data
- Two possible approaches:
  - primarily look at the data and distribute them
    - distribution of computations follows from that
    - e.g., with HPF or MPI
  - primarily look at the computations and distribute them
    - computations virtually always take place in loops (⇒ 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 213:

- The option `nowait` is not accepted in a `#pragma omp parallel for` (as at the end of a parallel region, there always is a global synchronisation)
- Besides the option `nowait`, the following additional options can be specified in the `<clause_list>` of a `for` directive:
  - `private`, `firstprivate`, `lastprivate`, `shared`: see slides 208 and 218 (These options are only accepted in a `#pragma omp parallel for`, not in a `#pragma omp for` inside a parallel region)
  - `schedule`: see slide 215
  - `ordered`: see slide 251
  - `reduction`: see slide 249
  - `collapse(<num>)`: this option tells the compiler that the next `<num>` (perfectly) nested loops should be collapsed into a single loop, whose iterations will then be distributed.

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

Example: vector addition

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

Short form for

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

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

Scheduling of loop iterations

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

Scheduling example(☞ 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

Scheduling example: results

- Runtime with 4 threads on the lab computers:

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

- Load imbalance when using `"static"`
  - thread 1: i=0..9999, thread 4: i=30000..39999
- `"static,1"` and `"dynamic"` use a block size of 1
  - each thread executes every 4th iteration of the `i` loop
  - can be very inefficient due to caches (*false sharing*, ☞ **5.1**)
    - remedy: use larger block size (e.g.: `"dynamic,100"`)
- `"guided"` often is a good compromize between load balancing and locality (cache usage)

Shared and private variables in loops

- The `parallel for` directive can be supplemented with the options `private`, `shared` and `firstprivate` (see slide 207 ff.)
- In addition, there is an option `lastprivate`
  - private variable
  - after the loop, the master thread has the value of the last iteration
- Example:

```
int i = 0;
#pragma omp parallel for lastprivate(i)
for (i=0; i<100; i++) {
    ...
}
std::cout << "i=" << i << std::endl;    // prints the value 100
```

3.2.2 Parallelization of Loops

(Animated slide)

When can a loop be parallelized?

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

No dependence

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

True dependence

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

Anti dependence

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

3.2.2 Parallelization of Loops ...

(Animated slide)

Handling of data dependences in loops

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

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

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

3.2.2 Parallelization of Loops ...

Handling of data dependences in loops ...

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

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

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

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

3.2.3 Simple Examples

(Animated slide)

Matrix addition

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

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

No dependences in ’j’ loop:

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

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

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

Inner loop can be executed in parallel

3.2.3 Simple Examples ...

(Animated slide)

Matrix addition

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

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

No dependences in 'i' loop:

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

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

Outer loop can be executed in parallel

Advantage: less overhead!

3.2.3 Simple Examples ...

Matrix multiplication

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

True dependece in the 'k' loop

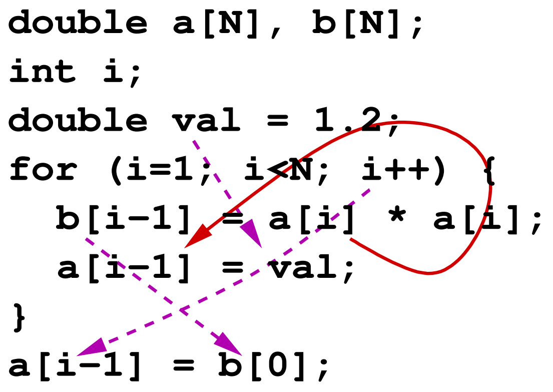
No dependences in the 'i' and 'j' loops

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

3.2.3 Simple Examples ...

(Animated slide)

Removing dependences



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

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

Anti depend. between iterations ⟶ Renaming + barrier

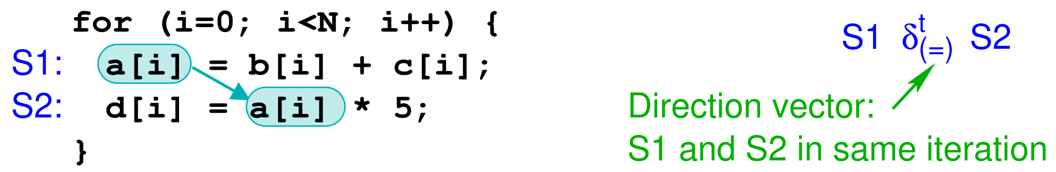
True dependece between loop and environment ⟶ lastprivate(i) + barriers

3.2.4 Dependence Analysis in Loops

(Animated slide)

Direction vectors

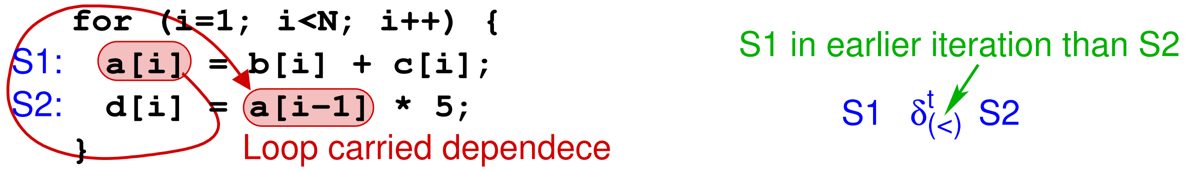
- Is there a dependence within a single iteration or between different iterations?



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

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

Direction vector:
S1 and S2 in same iteration

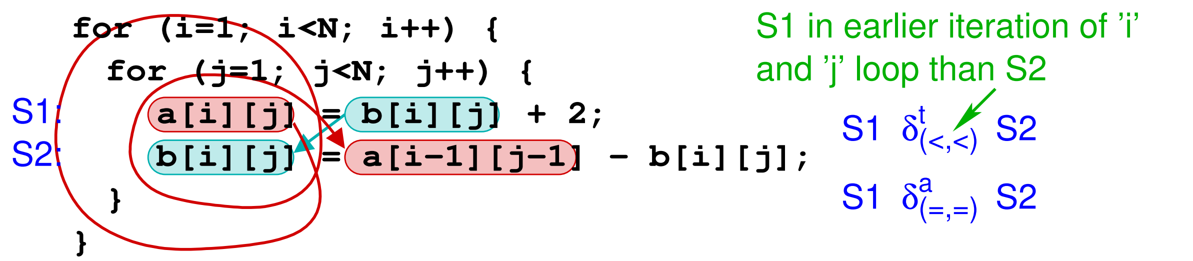


```
    for (i=1; i<N; i++) {
S1:   a[i] = b[i] + c[i];
S2:   d[i] = a[i-1] * 5;
    }
```

Loop carried dependece

S1 in earlier iteration than S2

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



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

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

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

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

3.2.4 Dependence Analysis in Loops ...

Formal computation of dependences

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

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

Equation system:

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

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

3.2.4 Dependence Analysis in Loops ...

Usage: applicability of code transformations

- Permissibility of code transformations depends on the (possibly) present data dependences
- E.g.: parallel execution of a loop is possible, if
  - this loop does not *carry* any data dependence
  - i.e., all direction vectors have the form $(..., =, ...)$ or $(..., \neq, ..., *, ...)$ [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 $(..., <, >, ...)$

Notes for slide 228:

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

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

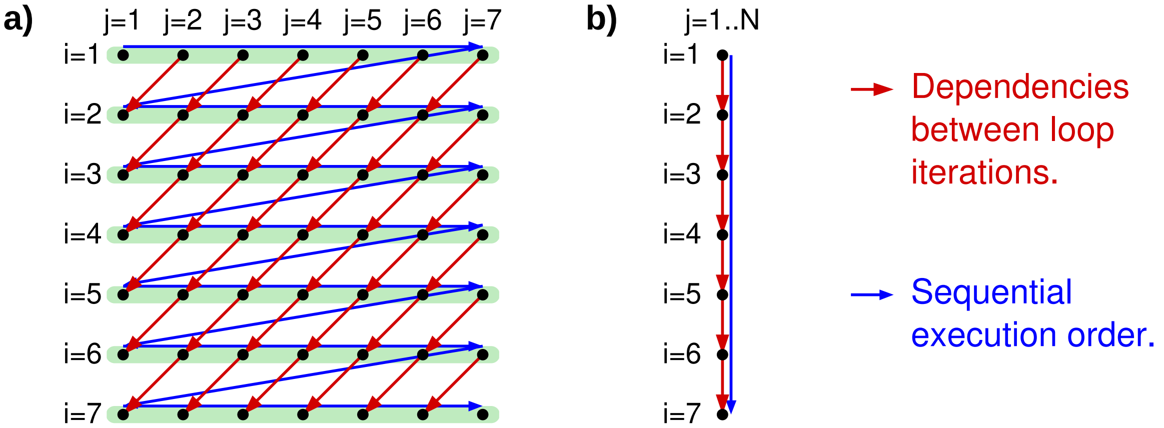
There is an anti-dependence from S2 to S1 (consider e.g. a[3][3]: it is read in iteration i=2, j=4 and is written later in iteration i=3, j=3.

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

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

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

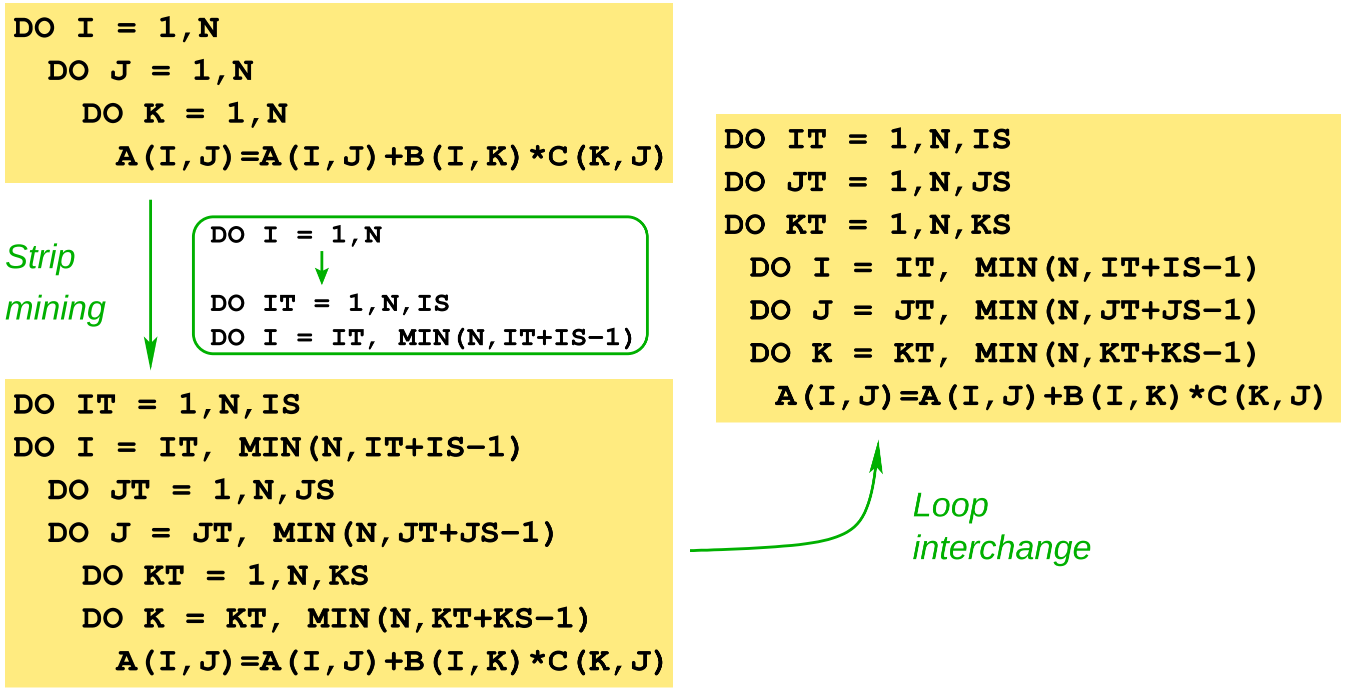
Note that when looking at the outer i-loop, we have to consider its complete body as one statement (i.e., we have to look at the union of all iterations of the inner j-loop), so we end up with the picture in figure b). We immediately see that this is a sequential loop.



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

3.2.4 Dependence Analysis in Loops ...

Example: block algorithm for matrix multiplication



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

Strip mining

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

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

Loop interchange

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

3.2.4 Dependence Analysis in Loops ...

Example: loop splitting

- Consider the following loop:

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

- We have $S1\ \delta^t_{(<)}\ S2$, which prevents parallelization of the loop without synchronization

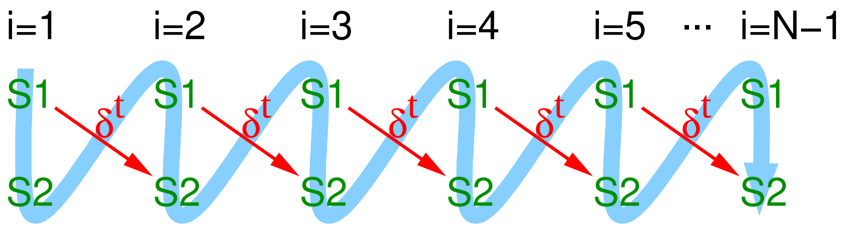
- However, since we do *not* have any dependence $S2\ \delta_{(<)}\ S1$, loop splitting is permitted, which results in:

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

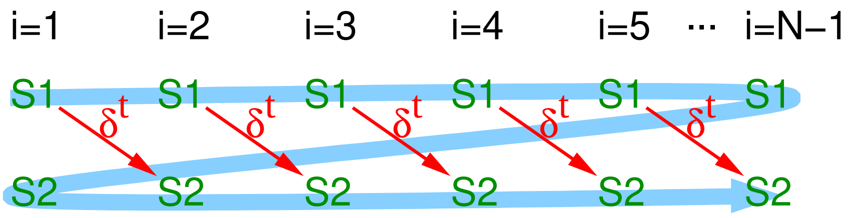
3.2.4 Dependence Analysis in Loops ...

Example: loop splitting ...

- Execution of the original loop:



- Execution of the transformed loop:



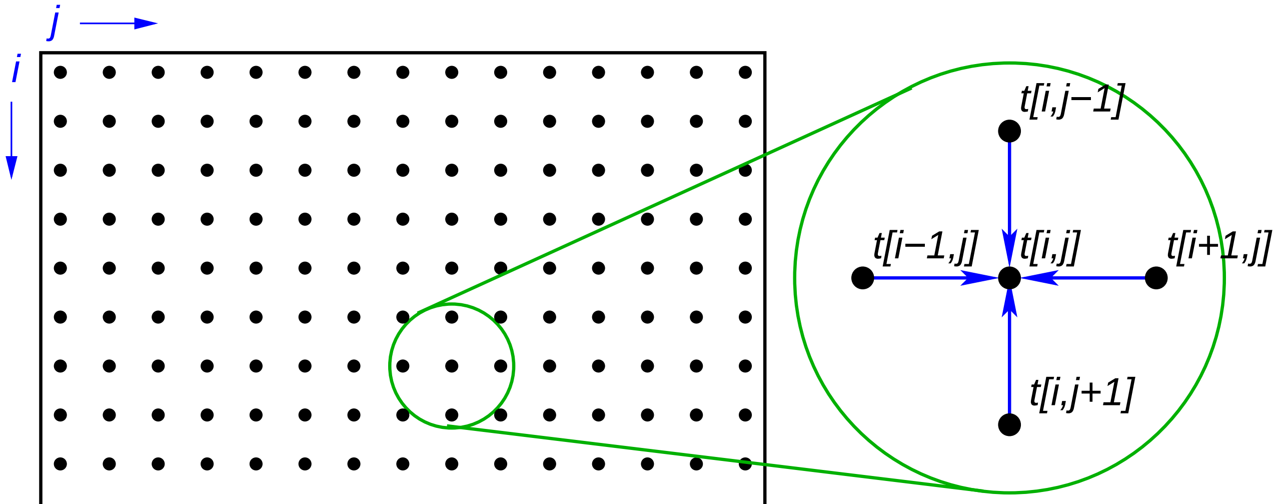
3.3 Exercise: The Jacobi and Gauss/Seidel Methods

Numerical solution of the equations for thermal conduction

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

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

Numerical solution of the equations for thermal conduction ...



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

Metal plate

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

Variants of the method

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

Parallel Processing

Winter Term 2024/25

18.11.2024

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

Stand: January 14, 2025

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

Variants of the method ...

Jacobi

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

Gauss/Seidel

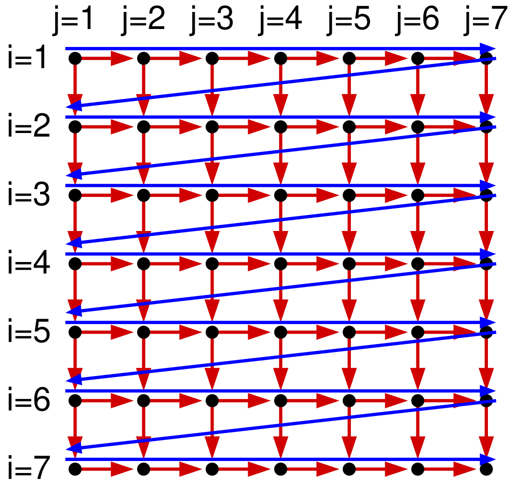
```
do {
  for(i=1;i<N-1;i++) {
    for(j=1;j<N-1;j++) {
      a[i][j] = 0.25 *
        (a[i-1][j] + ...);
    }
  }
} until (converged);
```

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

(Animated slide)

Dependences in Jacobi and Gauss/Seidel

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



Sequential execution order

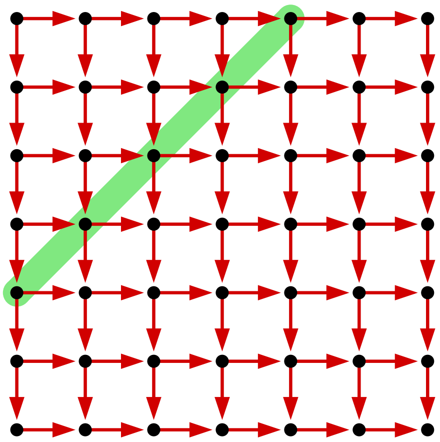
The figure shows the loop iterations, not the matrix elements!

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

(Animated slide)

Parallelisation of the Gauss/Seidel method

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



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

Loop restructuring in the Gauss/Seidel method

- Row-wise traversal of the matrix:

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

- Diagonal traversal of the matrix (☞ `03/diagonal.cpp`):

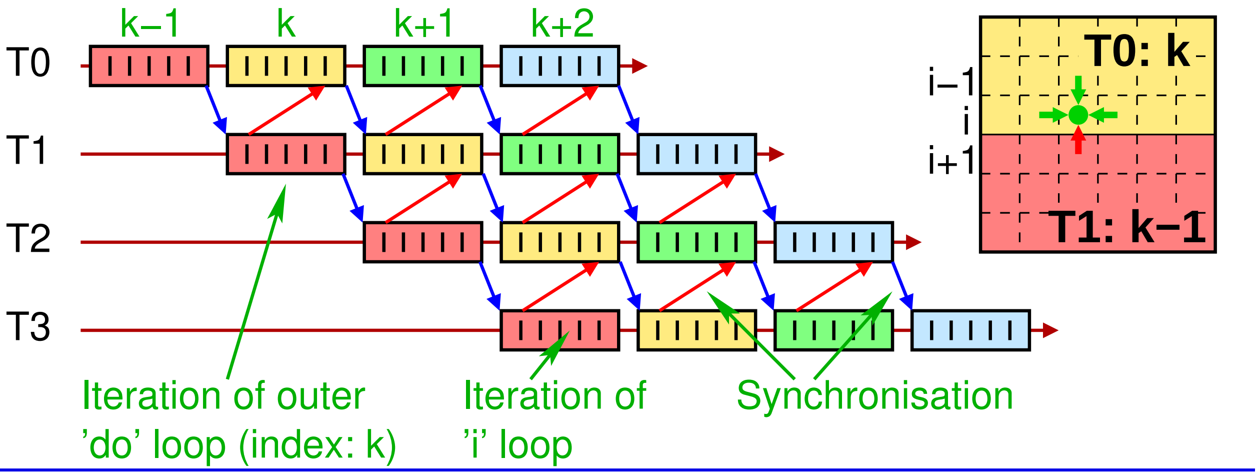
```
for (ij=1; ij<2*n-4; ij++) {
   int ja = (ij <= n-2) ? 1 : ij-(n-3);
   int je = (ij <= n-2) ? ij : n-2;
   for (j=ja; j<=je; j++) {
      i = ij-j+1;
      a[i][j] = ...;
```

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

(Animated slide)

Alternative parallelization of the Gauss/Seidel method

- Requirement: number of iterations is known in advance
  - (or: we are allowed to execute a few more iterations after convergence)
- Then we can use a pipeline-style parallelization
  - synchronisation via `ordered` (☞ **3.4.4**)



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

Results

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

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

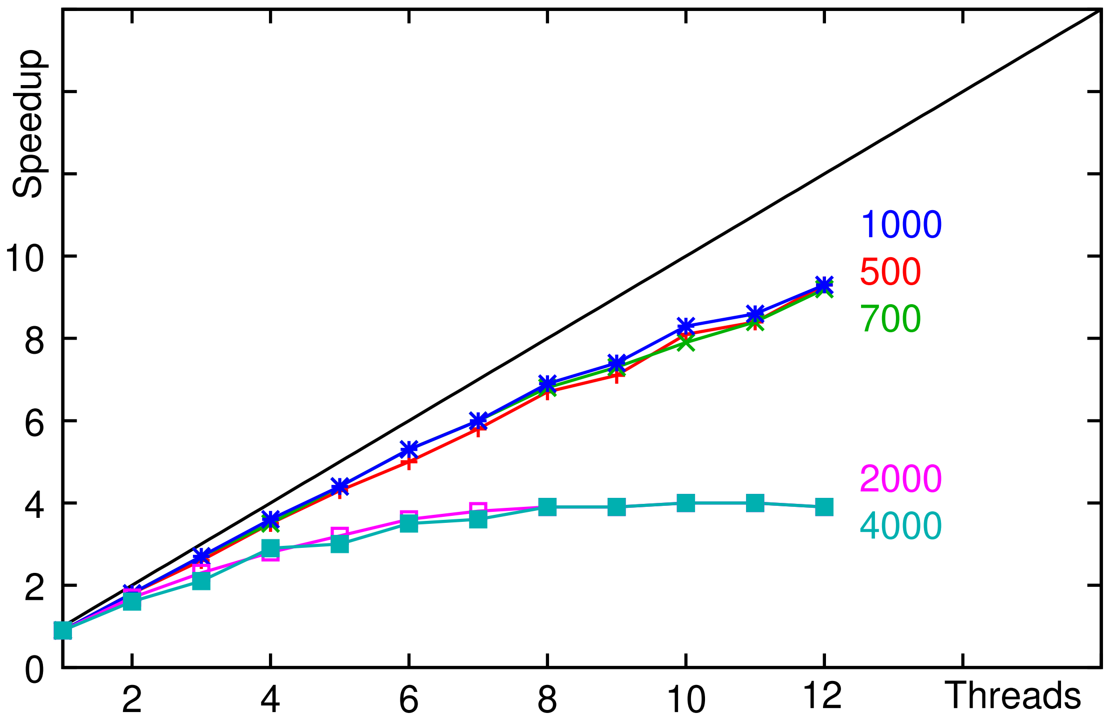
- Slight performance loss due to compilation with OpenMP
- Diagonal traversal in Gauss/Seidel improves performance
- High speedup with Gauss/Seidel at a matrix size of 700
  - data size: $\sim$ 8MB, cache size: 4MB per dual core CPU

Notes for slide 240:

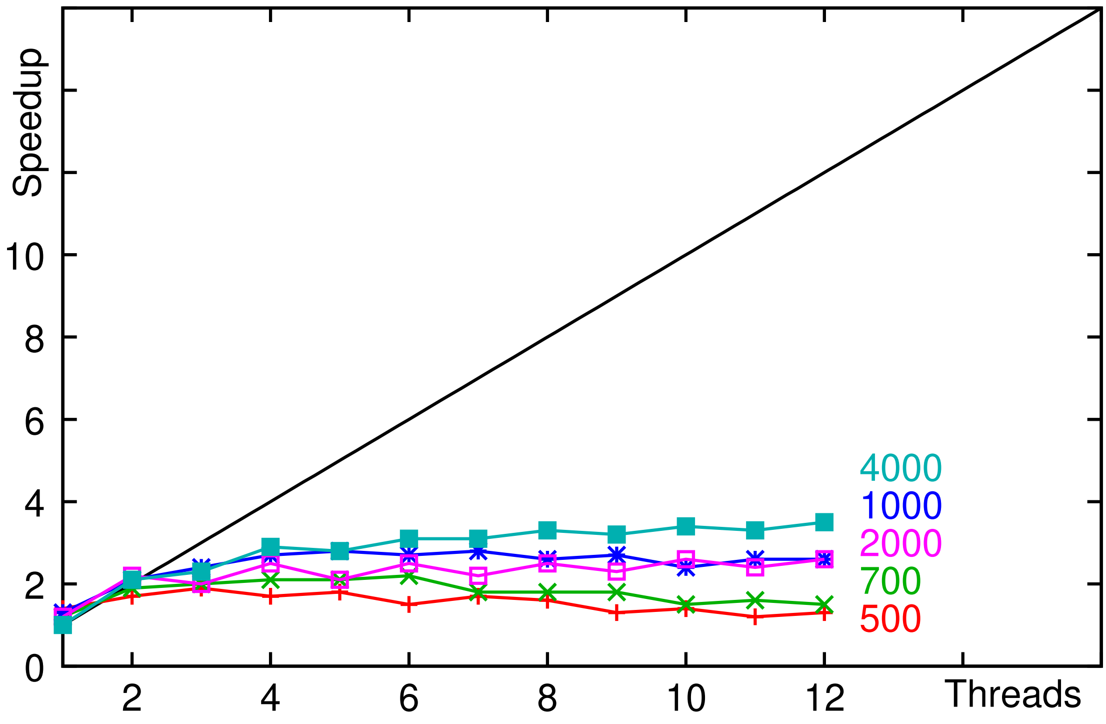
Results of the pipelined parallelization of the Gauss/Seidel method (`g++ -O`, bslab10, eps=0.001):

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

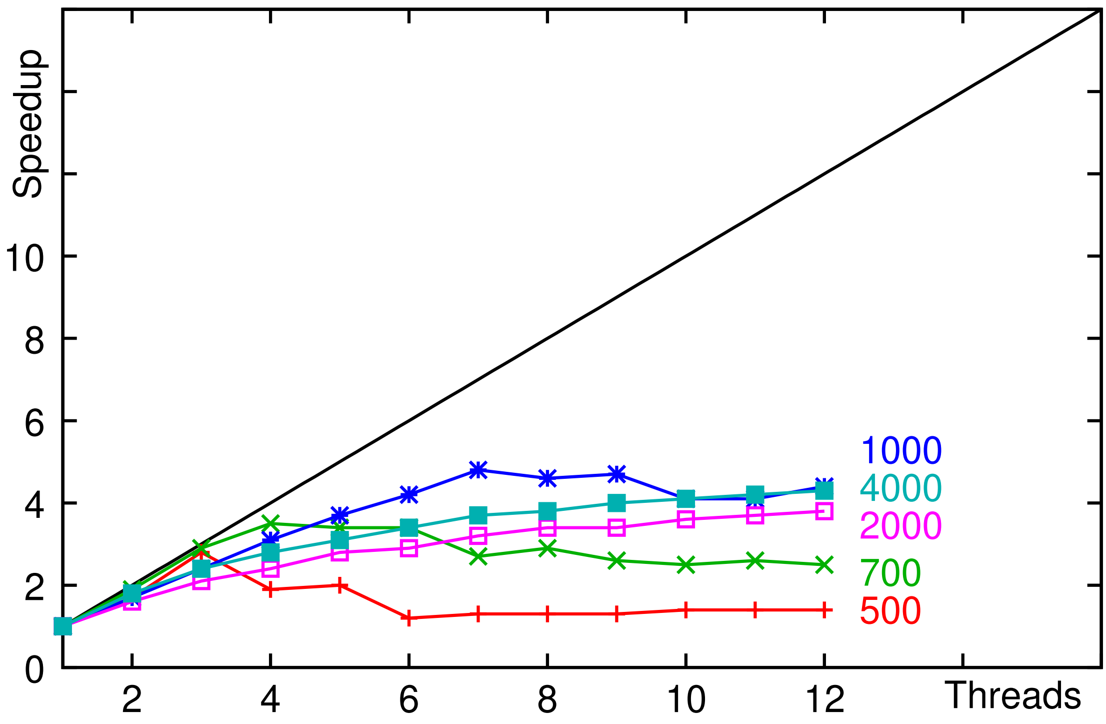
Speedup on the HorUS cluster: Jacobi



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



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



3.4 OpenMP Synchronization

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

```
int j = 0;

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

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

3.4 OpenMP Synchronization ...

Synchronization in OpenMP

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

3.4 OpenMP Synchronization ...

3.4.1 Critical sections

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

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

3.4.2 Atomic operations

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

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

Notes for slide 247:

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

3.4.2 Atomic operations ...

Examples

- Atomic adding:

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

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

- Atomic *fetch-and-add*:

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

- Instead of +, all other binary operators are possible, too
- With OpenMP 4, an atomic *compare-and-swap* can not yet be implemented
  - use builtin functions of the compiler, if necessary
  - (OpenMP 5.1 introduces a `compare` clause)

Notes for slide 248:

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

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

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

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

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

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

3.4 OpenMP Synchronization ...

3.4.3 Reduction operations

- Often loops aggregate values, e.g.:

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

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

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

3.4.3 Reduction operations ...

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

```
int a[N];
int sum = 0;
#pragma omp parallel
{
   int lsum = 0; // local partial sum
#  pragma omp for nowait  ←——  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
}                         to the global sum
printf("sum=%d\n",sum);
```

3.4 OpenMP Synchronization ...

3.4.4 Execution in program order

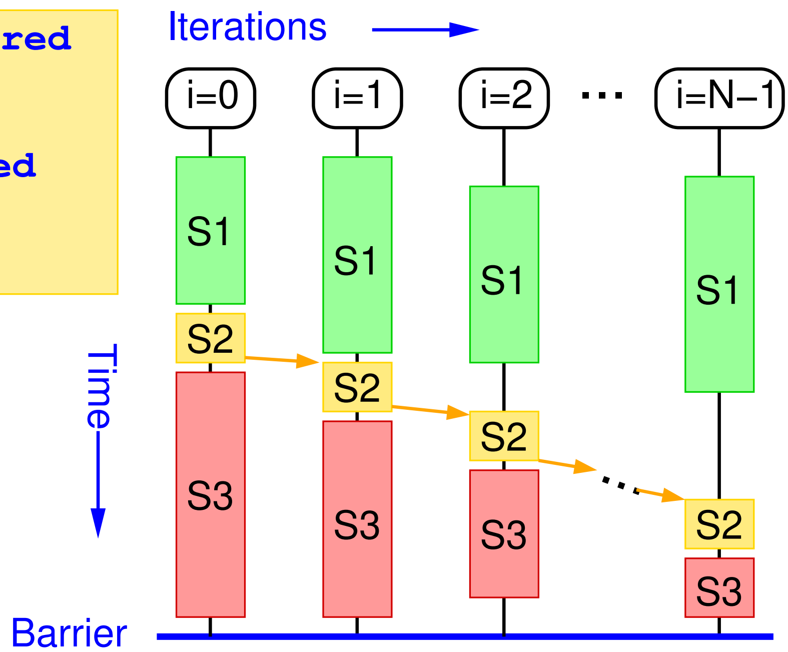
```
#pragma omp for ordered
for(...) {
   ...
   #pragma omp ordered
   Statement / Block
   ...
}
```

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

3.4.4 Execution in program order ...

Execution with `ordered`

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



3.4.4 Execution in program order ...

Execution with `ordered` ...

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

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

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

Notes for slide 253:

Example for a nested loop with dependencies:

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

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

3.4 OpenMP Synchronization ...

3.4.5 Barrier

```
#pragma omp barrier
```

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

3.4.5 Barrier ...

Example (☞ 03/barrier.cpp)

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

#define N 10000

float a[N][N];

main() {
  int i, j;

#pragma omp parallel
  {
    int thread = omp_get_thread_num();
    cout << "Thread " << thread << ": start loop 1\n";
```

3.4.5 Barrier ...

```
#pragma omp for private(i,j)   // add nowait, as the case may be
    for (i=0; i<N; i++) {
      for (j=0; j<i; j++) {
        a[i][j] = sqrt(i) * sin(j*j);
      }
    }

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

3.4.5 Barrier ...

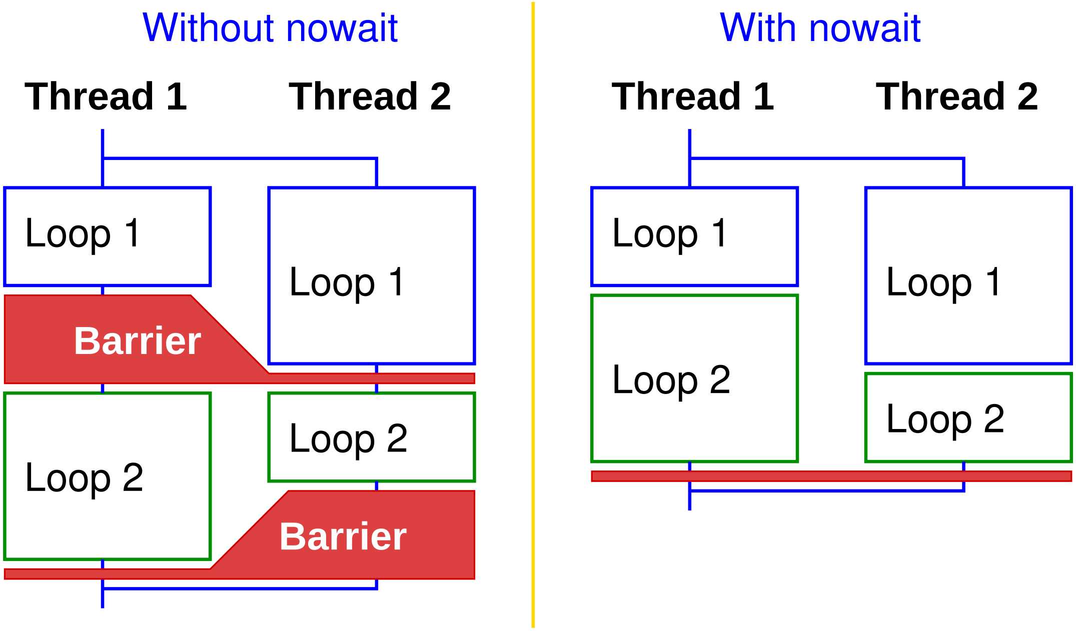
Example ...

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

3.4.5 Barrier ...

Example ...

- Executions of the program:



3.4 OpenMP Synchronization ...

3.4.6 Execution using a single thread

`#pragma omp single`
Statement / Block

`#pragma omp master`
Statement / Block

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

Notes for slide 259:

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

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

Parallel Processing

Winter Term 2024/25

25.11.2024

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

Stand: January 14, 2025

3.5 Task Parallelism with OpenMP

3.5.1 The `sections` Directive: Parallel Code Regions

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

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

3.5.1 The sections directive ...

Example: independent code parts

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

Important!!

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

3.5.1 The sections directive ...

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

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

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

3.5.1 The `sections` directive ...

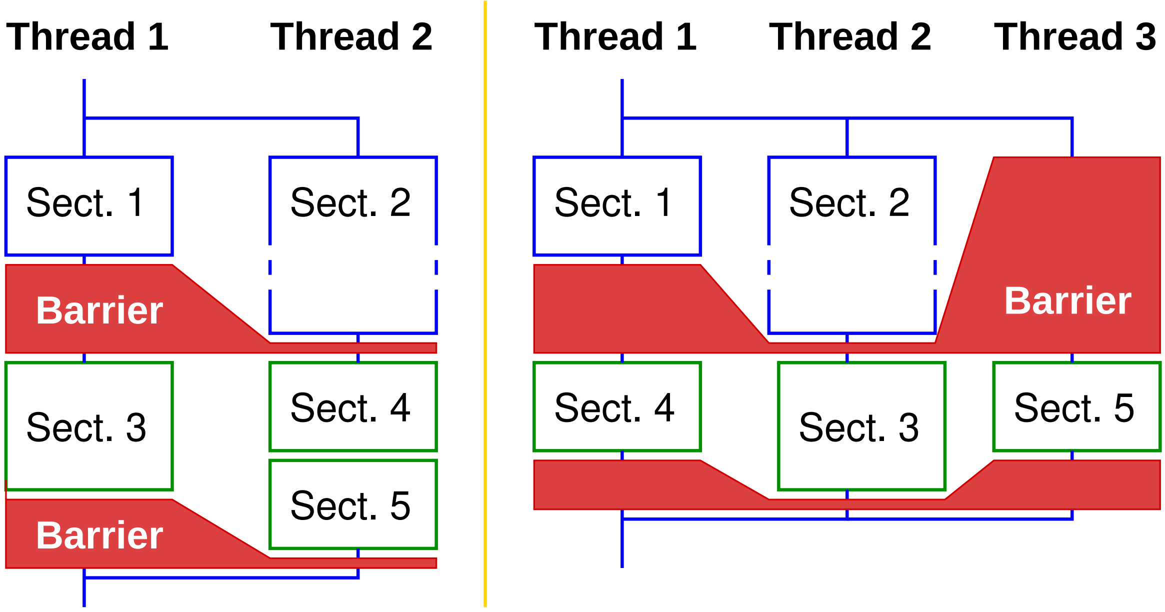
Example: scheduling / influence of `nowait` ...

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

3.5.1 The `sections` directive ...

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

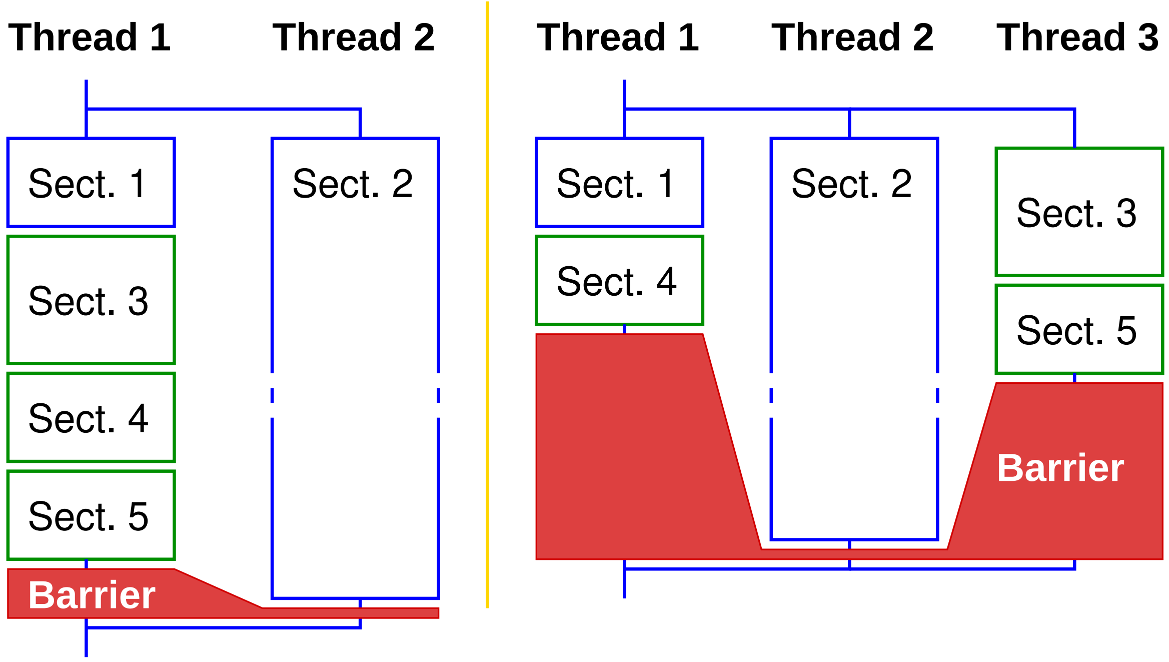
- Executions of the program **without** `nowait` option:



3.5.1 The `sections` directive ...

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

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



3.5 Task Parallelism with OpenMP ...

3.5.2 The `task` Directive: Explicit Tasks

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

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

3.5.2 The `task` Directive ...

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

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

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

Notes for slide 267:

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

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

3.5.2 The `task` Directive ...

Task synchronization

```
#pragma omp taskwait
```

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

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

3.5.2 The `task` Directive ...

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

- Imagine the following change when calling quicksort:

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

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

3.5.2 The `task` Directive ...

Example: parallel quicksort ...

- Solution 1:

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

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

Notes for slide 270:

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

3.5.2 The task Directive ...

Example: parallel quicksort ...

- Solution 2:

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

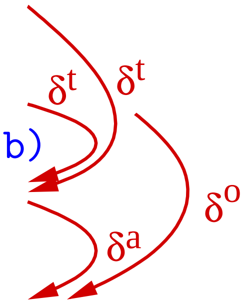
  - advantage: only wait at one single place
  - disadvantage: semantics of quicksort() must be very well documented

3.5.2 The task Directive ...

Dependences between tasks (☞ 03/tasks.cpp)

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

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



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

Notes for slide 272:

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

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

Array sections can be specified using the notation:

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

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

3.6 Tutorial: Tools for OpenMP

3.6.1 Debugging

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

3.6.1 Debugging ...

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

3.6.1 Debugging ...

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

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

3.6.1 Debugging ...

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

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

- All commands can also be abbreviated in `gdb`

3.6.1 Debugging ...

Sample session with `gdb` (sequential)

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

3.6.1 Debugging ...

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

3.6.1 Debugging ...

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

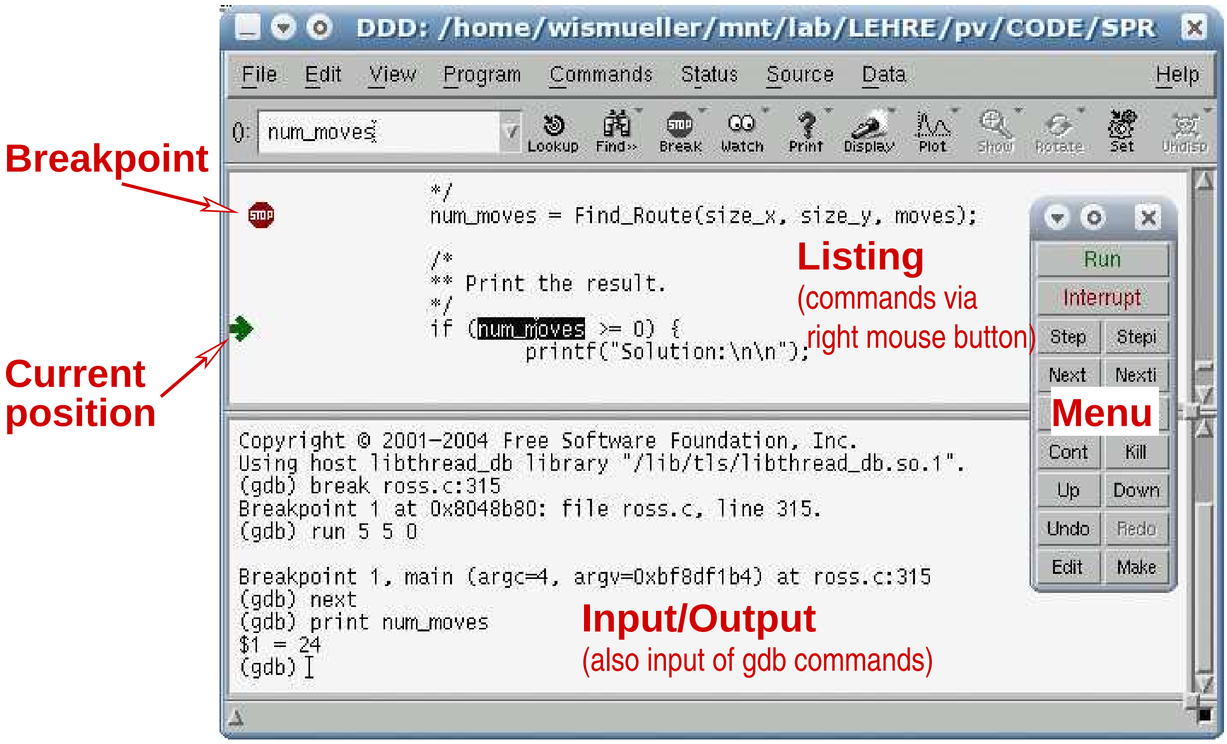
3.6.1 Debugging ...

Sample session with `gdb` (OpenMP)

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

3.6.1 Debugging ...

Sample session with `ddd`


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

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

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

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

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

3.6.2 Performance Analysis

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

Notes for slide 282:

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

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

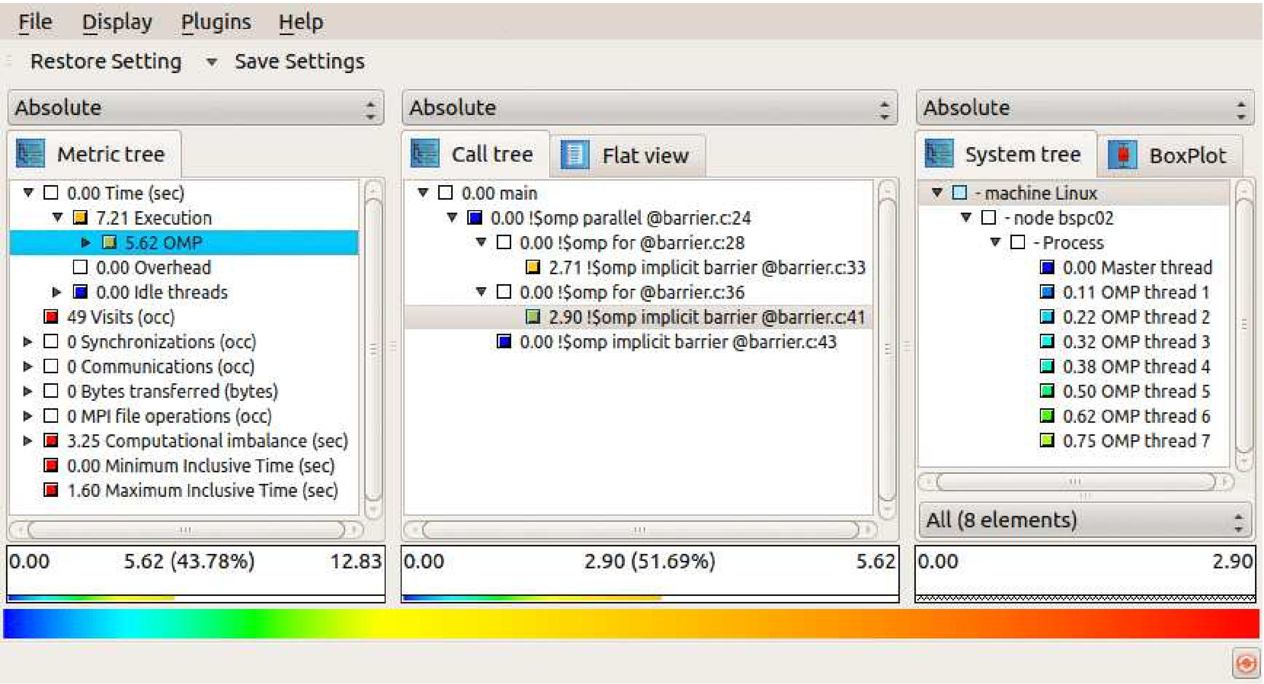
3.6.2 Performance Analysis ...

Performance analysis using Scalasca

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

3.6.2 Performance Analysis ...

Performance analysis using Scalasca: Example from slide 255



```
▼ □ 0.00 Time (sec)
  ▼ 7.21 Execution
    ▶ 5.62 OMP
    □ 0.00 Overhead
  ▶ 0.00 Idle threads
  49 Visits (occ)
▶ □ 0 Synchronizations (occ)
▶ □ 0 Communications (occ)
▶ □ 0 Bytes transferred (bytes)
▶ □ 0 MPI file operations (occ)
▶ 3.25 Computational imbalance (sec)
  0.00 Minimum Inclusive Time (sec)
  1.60 Maximum Inclusive Time (sec)
```

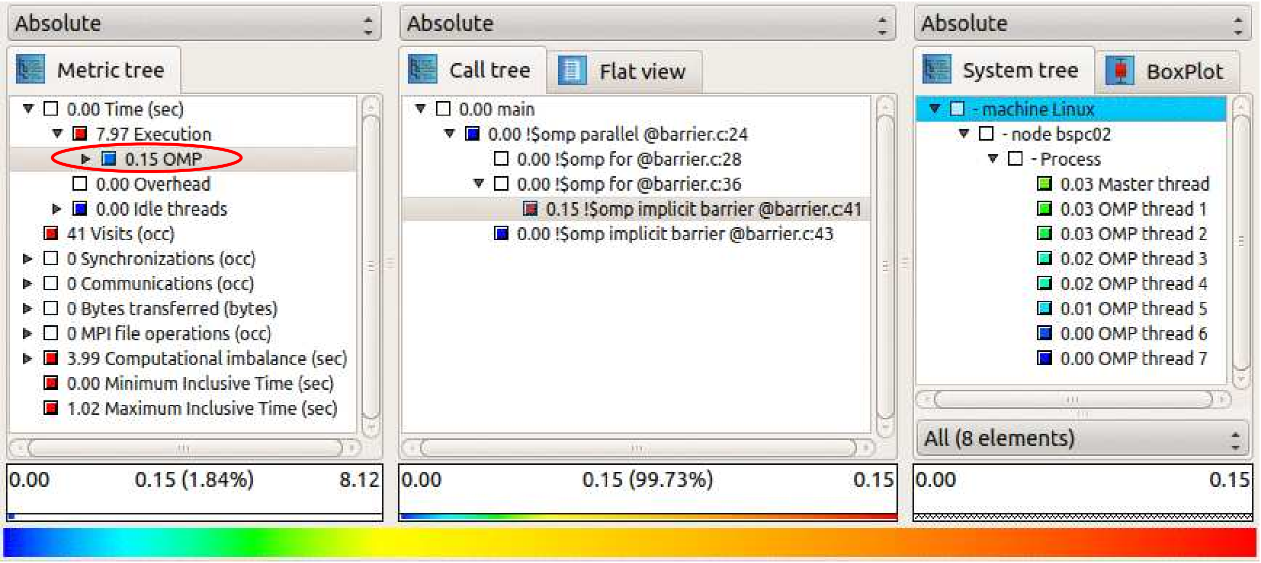
```
▼ □ 0.00 main
  ▼ 0.00 !$omp parallel @barrier.c:24
    ▼ □ 0.00 !$omp for @barrier.c:28
        2.71 !$omp implicit barrier @barrier.c:33
    ▼ □ 0.00 !$omp for @barrier.c:36
        2.90 !$omp implicit barrier @barrier.c:41
      0.00 !$omp implicit barrier @barrier.c:43
```

```
▼ □ - machine Linux
  ▼ □ - node bspc02
    ▼ □ - Process
        0.00 Master thread
        0.11 OMP thread 1
        0.22 OMP thread 2
        0.32 OMP thread 3
        0.38 OMP thread 4
        0.50 OMP thread 5
        0.62 OMP thread 6
        0.75 OMP thread 7
```

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

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

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



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

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

Notes for slide 285:

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

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

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

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

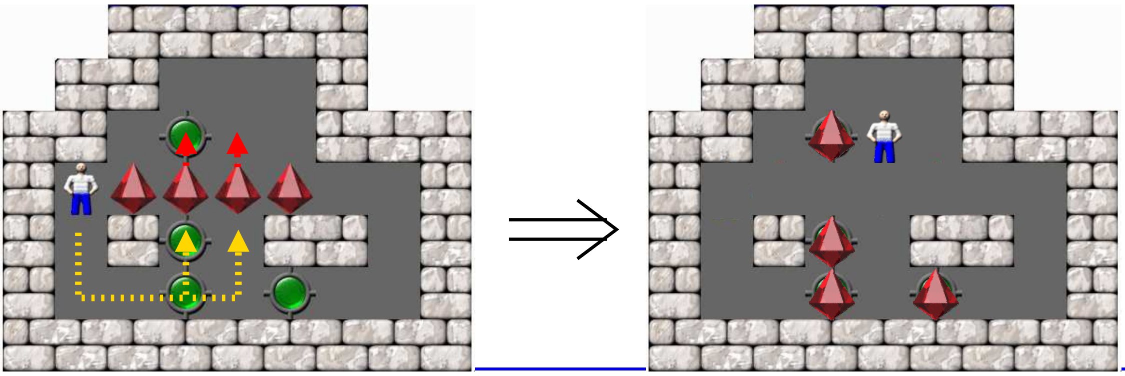
- Scalasca still shows a load imbalance (*Computational imbalance*), since, e.g., thread 7 still calculates much more in the first loop than thread 1. Scalasca is not able to recognize that this imbalance exactly cancels the corresponding imbalance in the second loop.

3.7 Exercise: A Solver for the Sokoban Game

(Animated slide)

Background

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

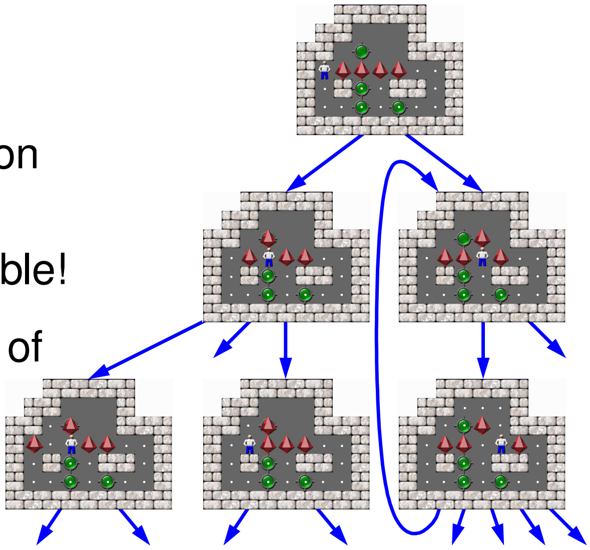


3.7 Exercise: A Solver for the Sokoban Game ...

(Animated slide)

How to find the sequence of moves?

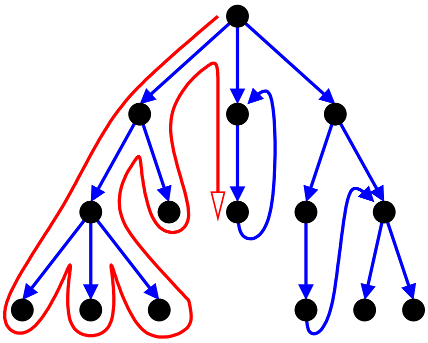
- Configuration: state of the play field
  - positions of the boxes
  - position of the player (connected component)
- Each configuration has a set of successor configurations
- Configurations with successor relation build a directed graph
  - not a tree, since cycles are possible!
- Wanted: shortest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



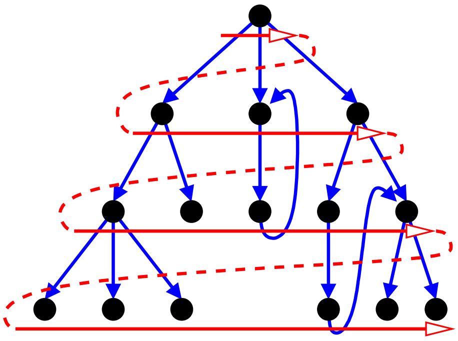
3.7 Exercise: A Solver for the Sokoban Game ...

How to find the sequence of moves? ...

- Two alternatives:
  - depth first search



  - problems:
    - cycles
    - handling paths with different lengths
  - breadth first search



  - problems:
    - reconstruction of the path to a node
    - memory requirements

Parallel Processing

Winter Term 2024/25

02.12.2024

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

Stand: January 14, 2025

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

Backtracking algorithm for depth first search:

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

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

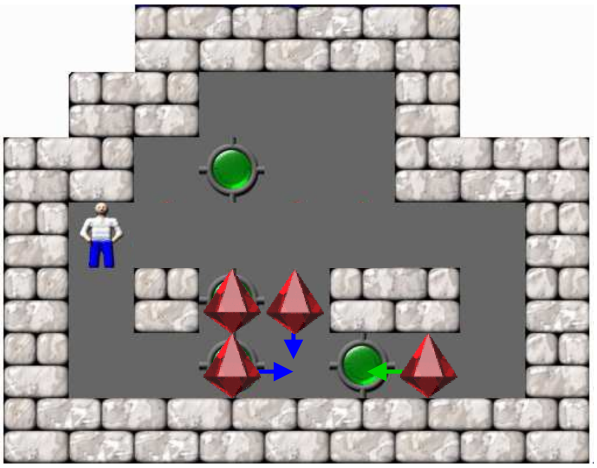
Algorithm for breadth first search:

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

(Animated slide)

Example for the *backtracking* algorithm

Configuration with possible moves



← Possible move

← Chosen move

3.8 Excursion: *Lock-Free* Data Structures

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

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

Example: appending to an array (at the end)

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

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

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

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

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

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

Element* firstNode = NULL;       // Pointer to first element

void add_first(Element* node) {
   Element* tmp;
   do {
     tmp = firstNode;
     node->next = tmp;
   } while (!compare_and_swap(&firstNode, tmp, node));
}
```

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

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

Parallel Processing

Winter Term 2024/25

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
- Tutorial: Working with MPI
- Complex data types in messages
- Communicators
- Collective operations
- Exercise: Jacobi and Gauss/Seidel with MPI
- Further concepts

Parallel Processing

Winter Term 2024/25

09.12.2024

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

Stand: January 14, 2025

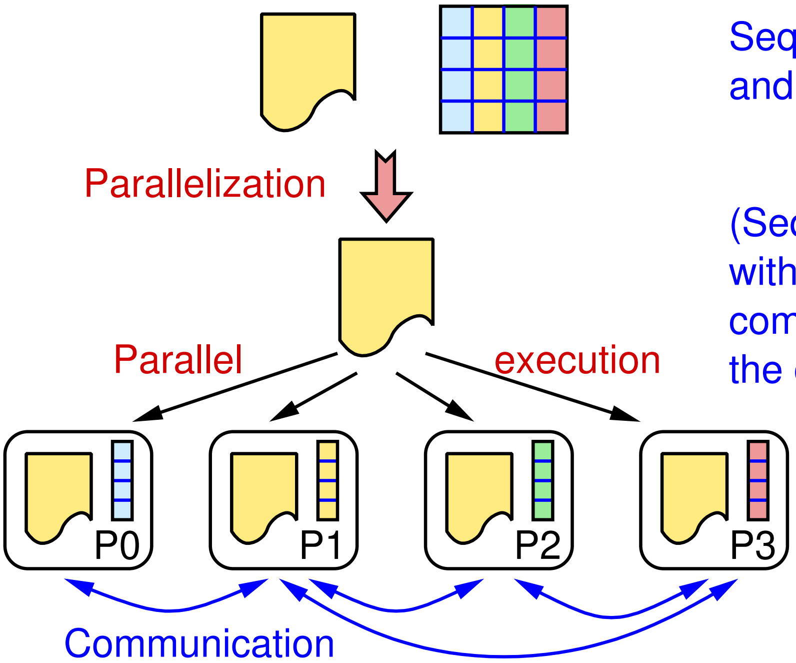
Organisation ...

Evaluation

- You all have got an invitation link for the evaluation of this lecture
- Please fill the questionaire **right now**!
  - only evaluate the lecture, the lab is evaluated separately

4.1 Typical approach

Data partitioning with SPMD model



Parallelization

Parallel execution

P0

P1

P2

P3

Communication

Sequential program and data partitioning

(Sequential) node program with message passing performs computations for a part of the data

Identical copies of the program, executed in parallel using multiple processes

4.1 Typical approach ...

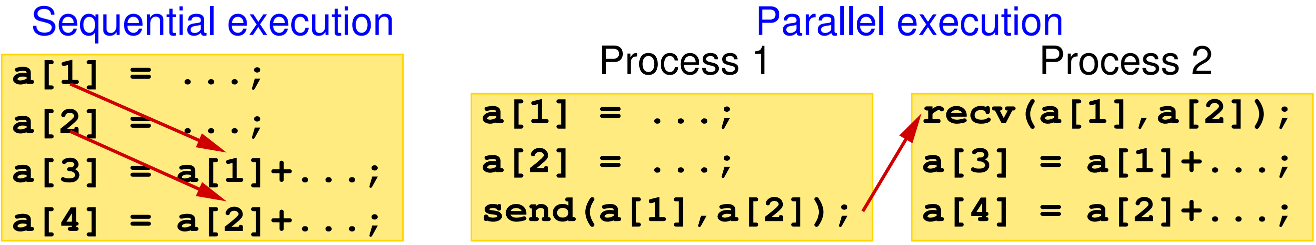
Activities when creating a node program

- Adjustment of array declarations
  - node program stores only a part of the data
  - (assumption: data are stored in arrays)
- Index transformation
  - global index ↔ (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 (☞ **4.11**)
- 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]);
```

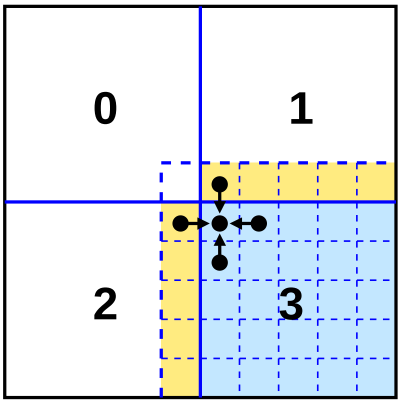
Process 2

```
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

4.2 MPI (*Message Passing Interface*)

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, ...

4.2 MPI (*Message Passing Interface*) ...

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

4.3 MPI Core routines

- 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

4.3 MPI Core routines ...

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

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
- After that, no other MPI routines must be used
  - in particular, no further `MPI_Init`
- `MPI_Finalize` does **not** terminate the process!

4.3 MPI Core routines ...

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

4.3 MPI Core routines ...

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 (☞ **4.7**)
- 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 (☞ **4.7**)

4.3 MPI Core routines ...

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

4.3 MPI Core routines ...

MPI_Send ...

- `MPI_Send` blocks the calling process, 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 process 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 323)

4.3 MPI Core routines ...

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)

- Process is blocked until the message has been completely received and stored in the receive buffer

4.3 MPI Core routines ...

MPI_Recv ...

- `MPI_Recv` only receives a message where
  - sender,
  - message tag, and
  - communicator

  match 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

4.3 MPI Core routines ...

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)`

4.3 MPI Core routines ...

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*

*☞ 4.10

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 plattform
- 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) {      /* Proccess 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 (on the XEON cluster):

  -
    ```
    mpiexec -n 2  ... ./pingpong 1000 1
    Time for one message: 50.094485 us
    Bandwidth: 0.019038 MB/s
    ```

  -
    ```
    mpiexec -n 2 ... ./pingpong 1000 100
    Time for one message: 50.076485 us
    Bandwidth: 1.904435 MB/s
    ```

  -
    ```
    mpiexec -n 2 ... ./pingpong 100 1000000
    Time for one message: 9018.934965 us
    Bandwidth: 105.741345 MB/s
    ```
- (Only) with large messages the bandwidth of the interconnection network is reached
  - XEON cluster: 1 GBit/s Ethernet (≙ 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()`

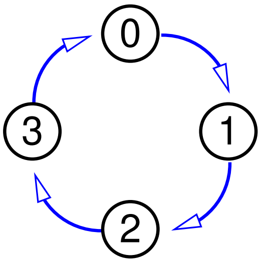
Result	Current wall clock time in seconds

- for timing measurements
- in MPICH2: 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

4.5 Point-to-point communication ...

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 bot 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 326:

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

- 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

- 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()`**

- 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.

4.6 Tutorial: Working with MPI (MPICH2/OpenMPI)

Available MPI implementations

- e.g., MPICH2 (Linux), OpenMPI 1.10.3
- Portable implementations of the MPI-2 standard

Compiling MPI programs: `mpic++`

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

4.6 Tutorial: Working with MPI (MPICH2/OpenMPI) ...

Running MPI programs: `mpiexec`

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

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

4.6 Tutorial: Working with MPI (MPICH2/OpenMPI) ...

Debugging

- MPICH2 and OpenMPI support `gdb` and `totalview`
- Using `gdb`:
  - `mpiexec -enable-x -n ...  xterm -e gdb myProg`
    - instead of `xterm`, you may (have to) use other console programs, e.g., `konsole` or `mate-terminal`
  - for each process, a `gdb` starts in its own console window
  - in `gdb`, start the process with `run` *`args...`*
- Prerequisite: compilation with debugging information
  - `mpic++ -g -o myProg myProg.cpp`

4.6 Tutorial: Working with MPI (MPICH2/OpenMPI) ...

Performance Analysis using Scalasca

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

Parallel Processing

Winter Term 2024/25

16.12.2024

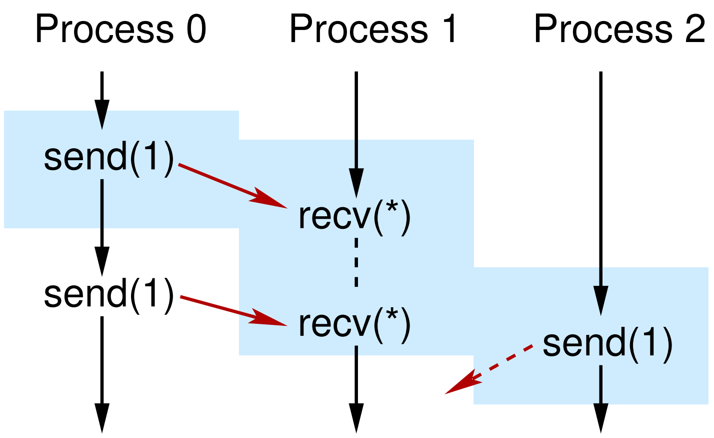
Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: January 14, 2025

4.7 Communicators

(Animated slide)

Motivation: problem of earlier communication libraries



🟦: Code in a parallel (e.g. numerical) library

If process 2 is ‘late’ for some reason: communication fails!

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

4.7 Communicators ...

- 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

4.7 Communicators ...

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 (☞ 4.8)
  - 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

4.7 Communicators ...

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

4.8 Collective operations

- Collective operations in MPI
  - must be executed concurrently by all processes of a process group (a communicator)
  - are blocking
  - do not neccessarily 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 335:

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 synchonization 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)

4.8 Collective operations ...

MPI_Barrier

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

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

4.8 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

4.8 Collective operations ...

Example: summing up an array

Sequential

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

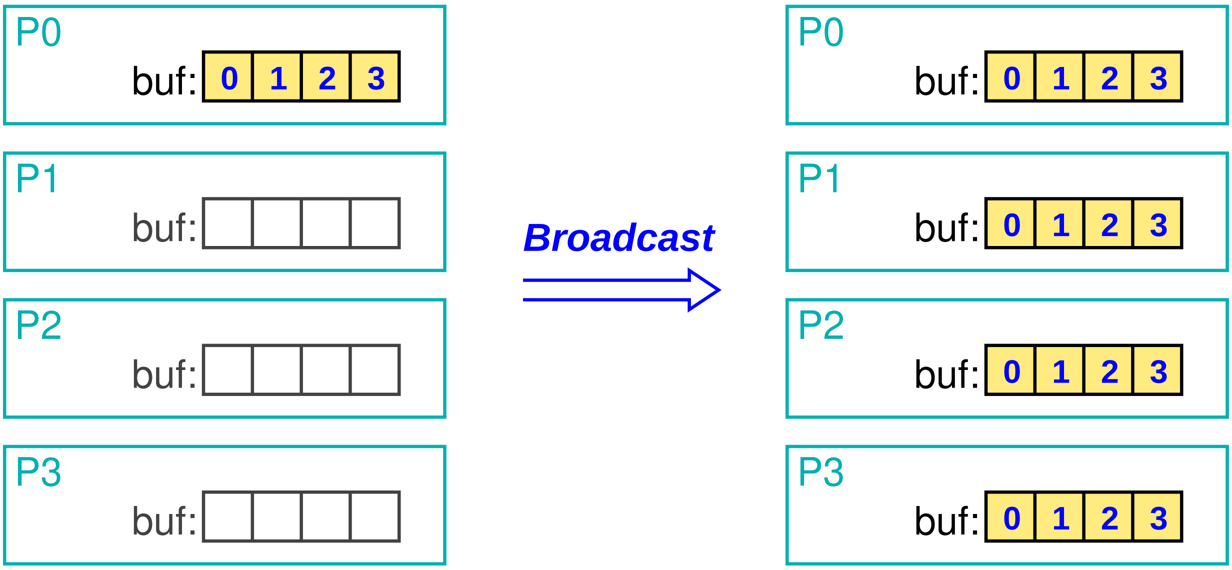
Parallel

```
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);
```

4.8 Collective operations ...

Collective communication: broadcast



P0 buf: 0 1 2 3

P1 buf:

P2 buf:

P3 buf:

Broadcast

P0 buf: 0 1 2 3

P1 buf: 0 1 2 3

P2 buf: 0 1 2 3

P3 buf: 0 1 2 3

4.8 Collective operations ...

MPI_Bcast

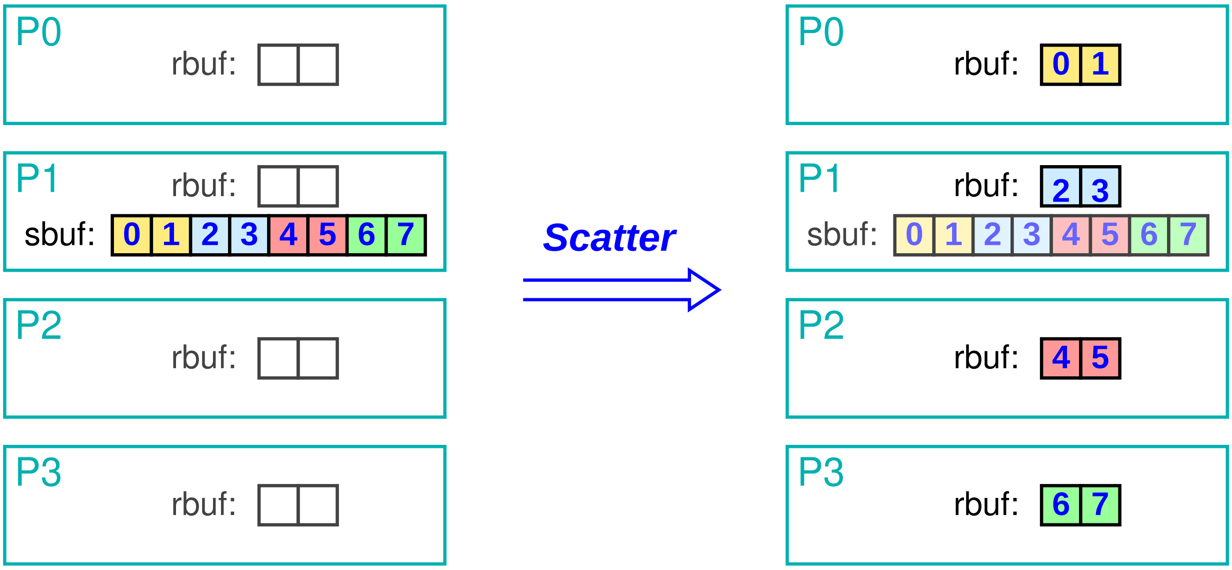
```
int MPI_Bcast(void *buf, int count, MPI_Datatype dtype,
                  int root, MPI_Comm comm)
```

IN `root` Rank of the sending process

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

4.8 Collective operations ...

Collective communication: scatter



P0 rbuf: □□

P1 rbuf: □□ sbuf: 0 1 2 3 4 5 6 7

P2 rbuf: □□

P3 rbuf: □□

Scatter

P0 rbuf: 0 1

P1 rbuf: 2 3 sbuf: 0 1 2 3 4 5 6 7

P2 rbuf: 4 5

P3 rbuf: 6 7

4.8 Collective operations ...

MPI_Scatter

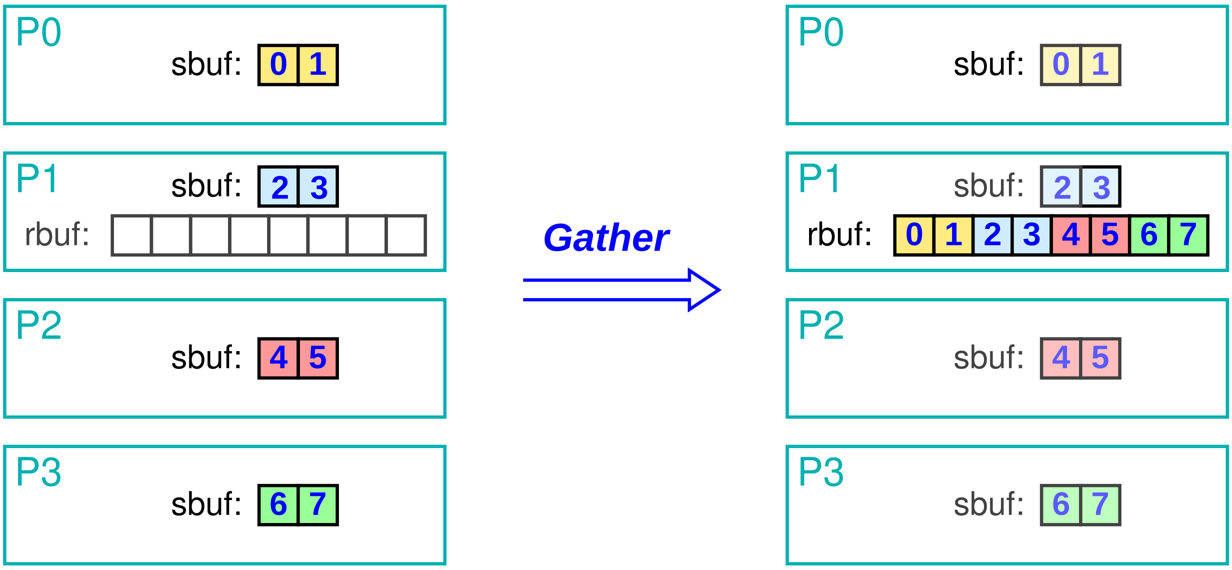
```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf, int recvcount,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

- 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 342:

- A problem that may arise when using `MPI_Scatter` is that the 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



P0
sbuf: 0 1

P1
sbuf: 2 3
rbuf:

P2
sbuf: 4 5

P3
sbuf: 6 7

Gather

P0
sbuf: 0 1

P1
sbuf: 2 3
rbuf: 0 1 2 3 4 5 6 7

P2
sbuf: 4 5

P3
sbuf: 6 7

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 sent the same amount of data
- `root` stores the data from process `i` starting at position `i * recvcount` in `recvbuf`
- `recvcount`: data length for each process (not the total length!)
- Alternative `MPI_Gatherv`: analogous to `MPI_Scatterv`

4.8 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)

4.8 Collective operations ...

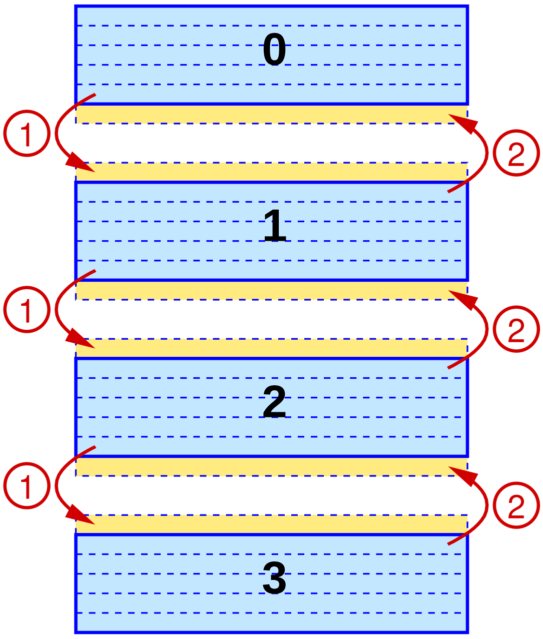
More collective communication operations

- MPI_Alltoall: all-to-all broadcast (☞ **2.8.5**)
- 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

4.9 Exercise: Jacobi and Gauss/Seidel with MPI

(Animated slide)

Gerneral approach



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

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

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

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

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

Gerneral approach ...

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

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

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

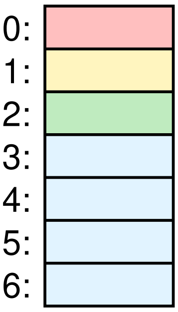
4.9 Exercise: Jacobi and Gauss/Seidel with MPI ...

Distribution of data

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

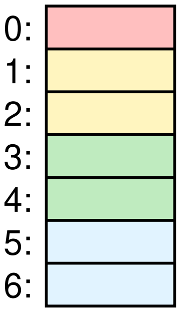
Notes for slide 349:

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



7 ÷ 4 = 1

Processes 0 – 2 each get one element, process 3 gets the rest.



7 + p ÷ 4 = 1 | 2 | 2 | 2

Process 0 gets one element, processes 1 – 3 get two elements each.

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.

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

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 350:

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;
}
```

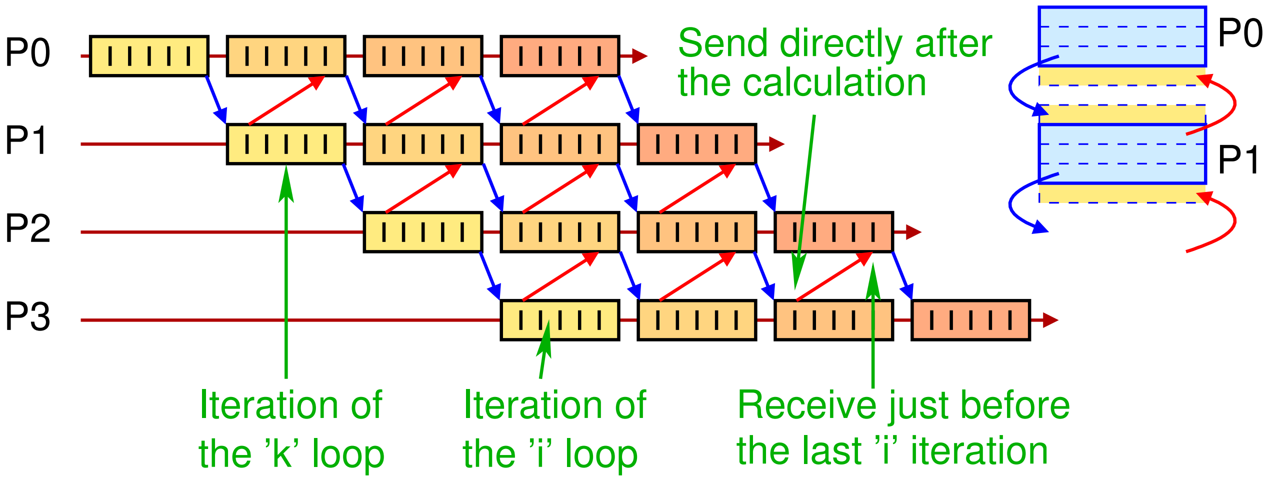
becomes

```
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;
}
```

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

On the parallelization of the Gauss/Seidel method

- ➥ Similar to the pipelined parallelization with OpenMP (☞ **3.3**)



Send directly after the calculation

Iteration of the 'k' loop

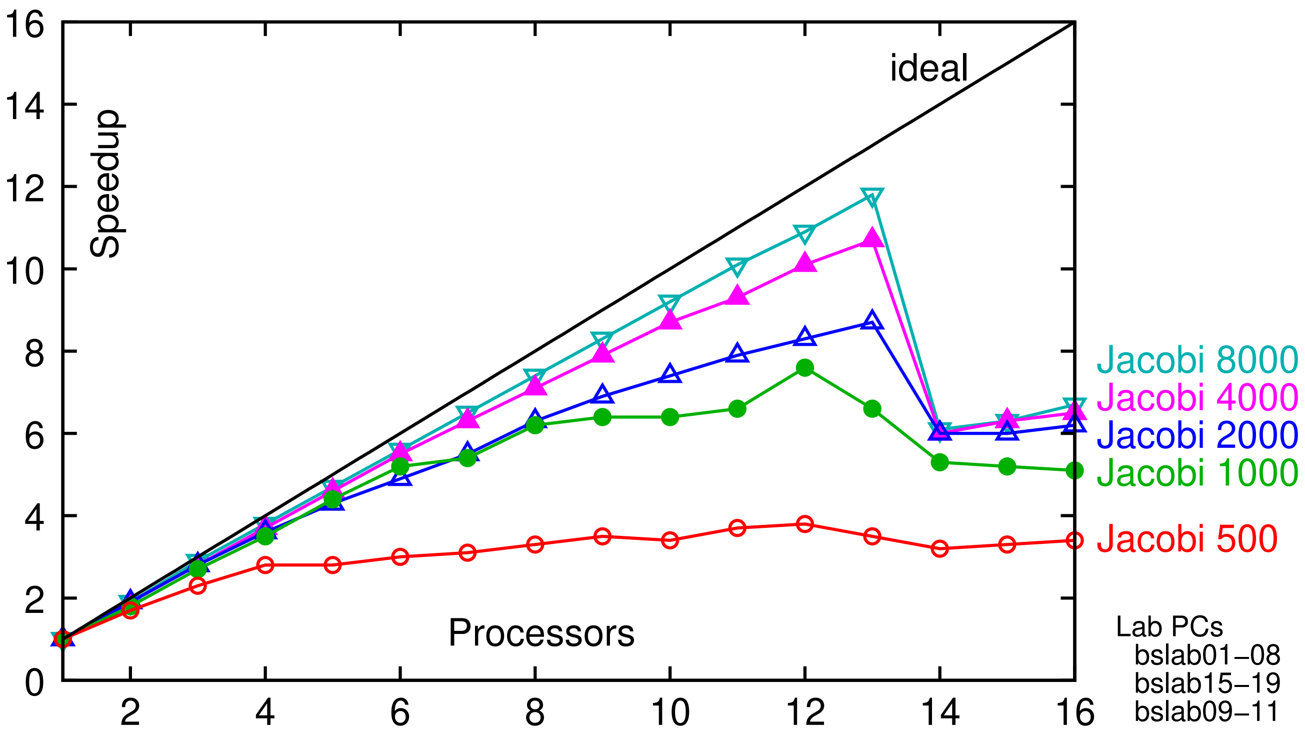
Iteration of the 'i' loop

Receive just before the last 'i' iteration

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

(Animated slide)

Obtained speedup for different matrix sizes



4.10 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 353:

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 parge 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);
```

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.

4.10 Complex data types in messages ...

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)

4.10 Complex data types in messages ...

MPI_Type_vector: non-contiguous arrays

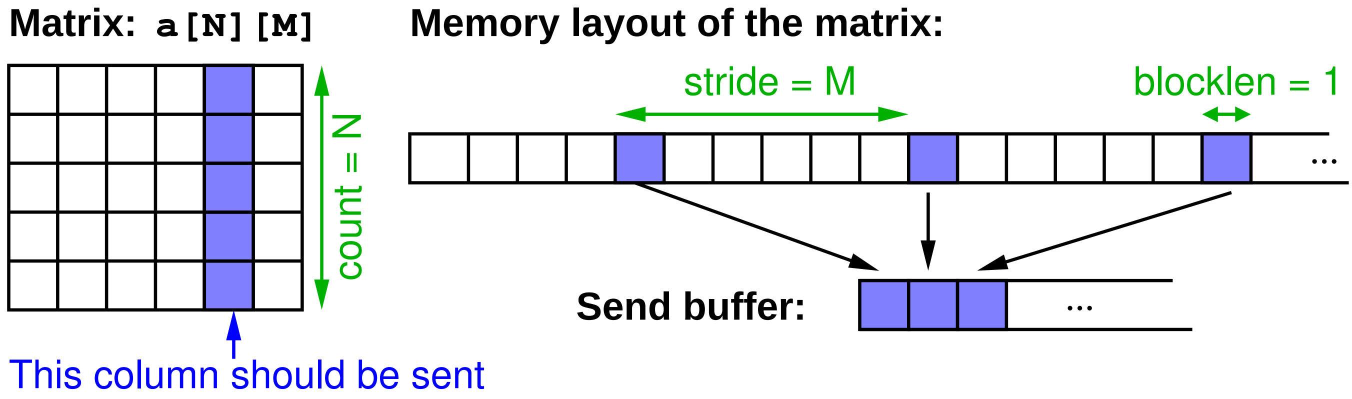
```
int MPI_Type_vector(int count, int blocklen, int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

IN	`count`	Number of data blocks
IN	`blocklen`	Length of the individual data blocks
IN	`stride`	Distance between successive data blocks
IN	`oldtype`	Type of the elements in the data blocks
OUT	`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**

4.10 Complex data types in messages ...

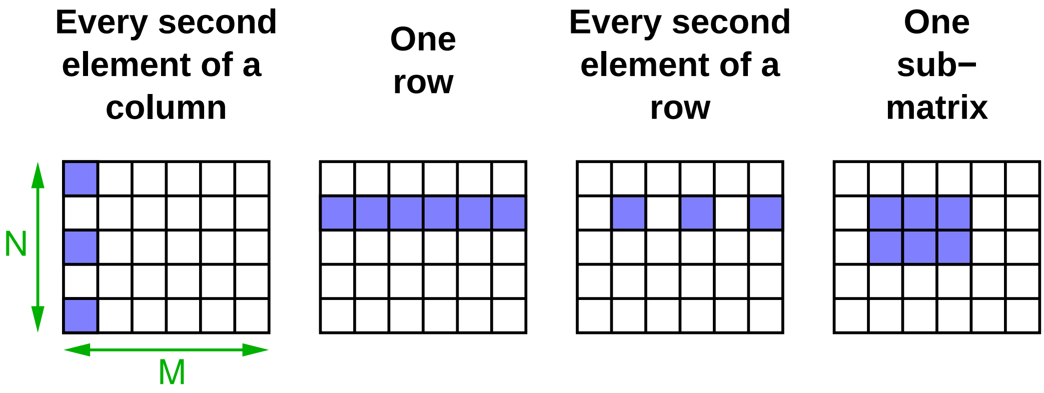
Example: transferring a column of a matrix



```
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 356:

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	M / 2	2
blocklen	1	M	1	1	3
stride	2 * M	x	1	2	M

4.10 Complex data types in messages ...

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
  - reciever 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 357:

Strided arrays that have been created using `MPI_Type_vector` can usually transmitted as efficient 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.

4.10 Complex data types in messages ...

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`

Parallel Processing

Winter Term 2024/25

13.01.2025

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

Stand: January 14, 2025

4.11 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*

4.11 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
    - specifiec by an MPI data type
  - file operations: individual / collective, private / shared file pointer, blocking / non-blocking

4.12 Summary

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

Parallel Processing

Winter Term 2024/25

5 Optimization Techniques

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	Run time: 80ms	(bspc02, g++ –O3)
N=8193:	Run time: 140 ms	Run time: 80ms	

- 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

5.1 Cache Optimization ...

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!

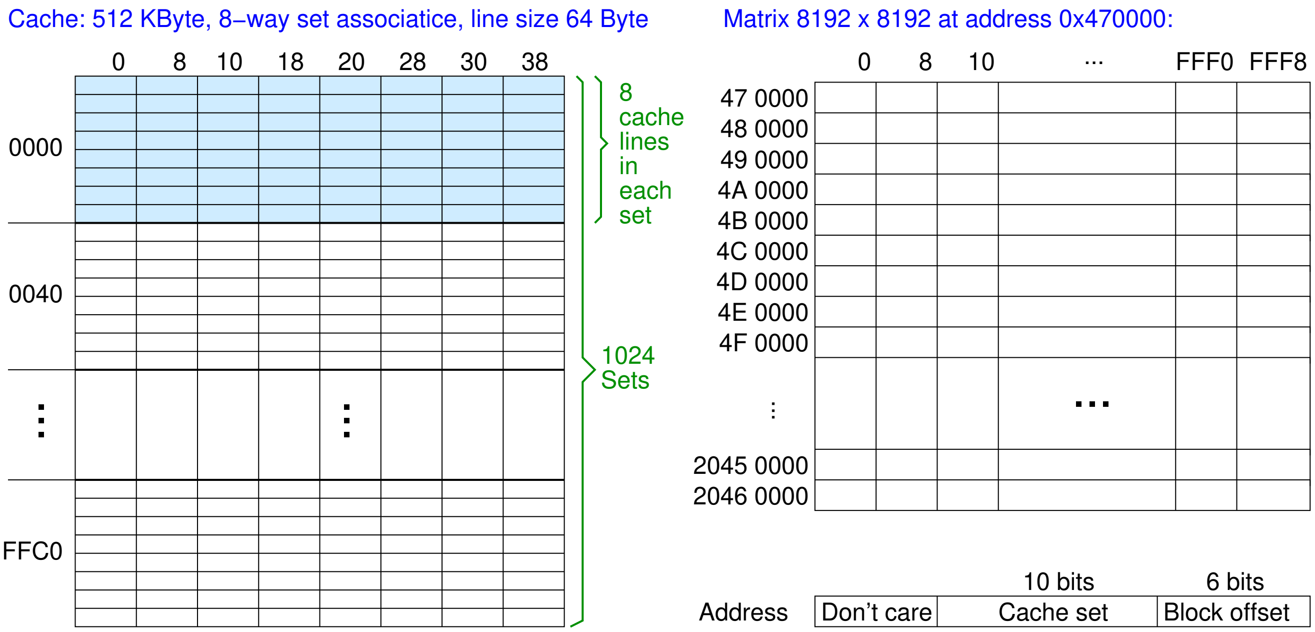
5.1 Cache Optimization ...

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 ...



Notes for slide 367:

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.

5.1 Cache Optimization ...

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 matrices)

5.1 Cache Optimization ...

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:

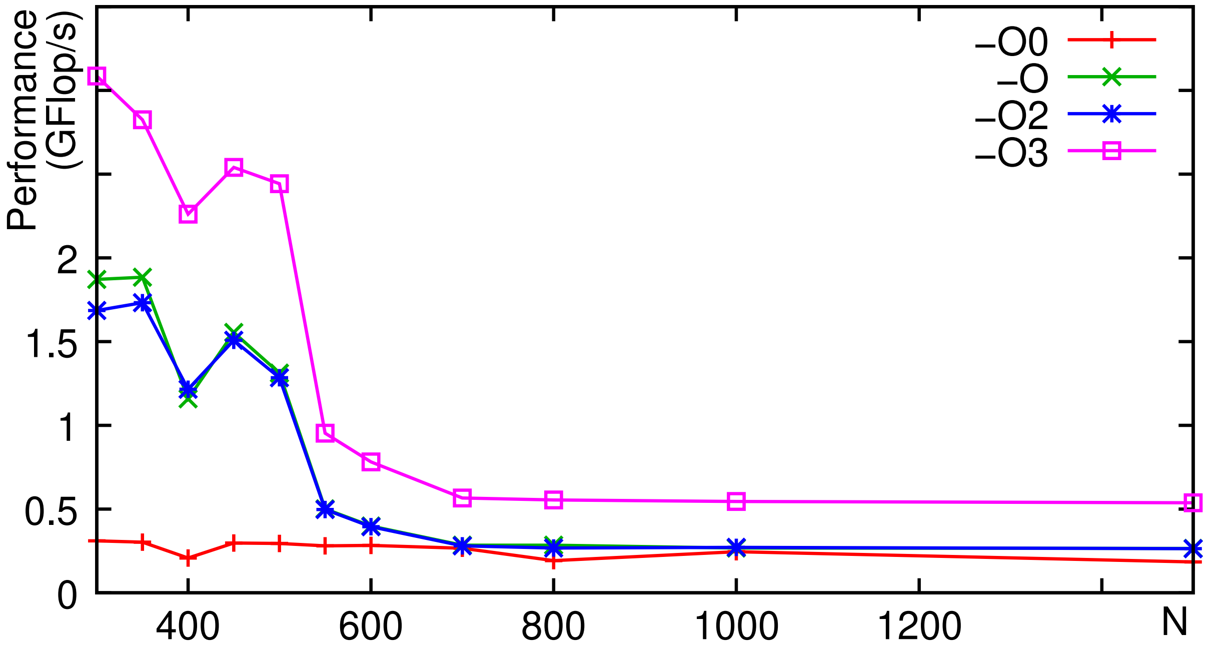
```
double a[N][N], b[N][N], ...
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      c[i][j] += a[i][k] * b[k][j];
```

- Performance with different compiler optimization levels: (N=500, g++ 4.6.3, Intel Core i7 2.8 GHz (bspc02))
  - -O0: 0.3 GFlop/s
  - -O: 1.3 GFlop/s
  - -O2: 1.3 GFlop/s
  - -O3: 2.4 GFlop/s (SIMD vectorization!)

5.1 Cache Optimization ...

Example: Matrix multiply ...

- Scalability of the performance for different matrix sizes:



5.1 Cache Optimization ...

Example: Matrix multiply ...

- Optimized order of the loops:

```
double a[N][N], b[N][N], ...
for (i=0; i<N; i++)
  for (k=0; k<N; k++)
    for (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=500, -O3: 4.2 GFlop/s instead of 2.4 GFlop/s
  - considerably better scalability

Notes for slide 372:

The statement `c[i][j] += a[i][k] + b[k][j]` has a true dependence, an anti dependence and an output dependence bewteen different iterations of the `k`-loop. Thus, the dependence vector for all these dependences is $(=,=,<)$.

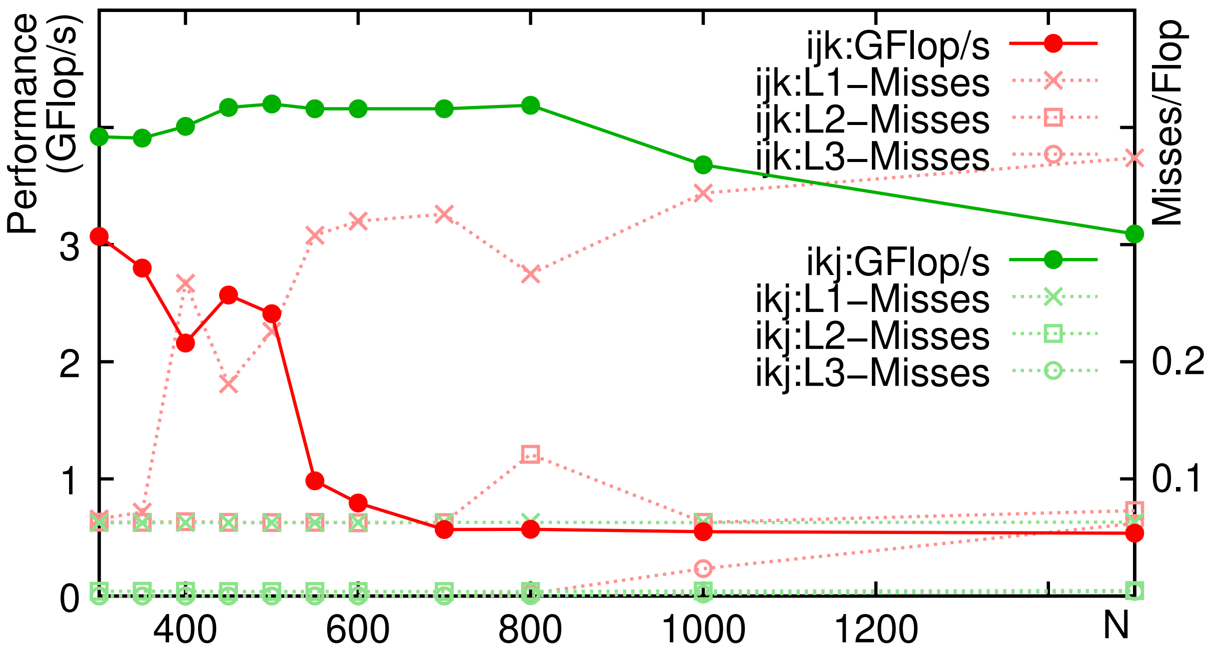
So according to slide 228, interchanging the `j`- and `k` is permitted, since the loops are perfectly nested, the loop bounds are independent, and there is no dependence with a direction vector of $(*,<,>)$.

5.1 Cache Optimization ...

(Animated slide)

Example: Matrix multiply ...

➥ Comparison of both loop orders:



Notes for slide 373:

The decrease in performance of the ’ijk’ loop order between N=500 and N=550 is due to a large increase in the L3 misses from $4.4 \cdot 10^{-5}$ to $1.4 \cdot 10^{-4}$ (which is not visible in the figure due to the scaling) and the increase in L1 misses.

The decrease in performance of the ’ikj’ loop order between N=800 and N=1000 is also caused by an increase in L3 misses (from $1.4 \cdot 10^{-4}$ to $1.6 \cdot 10^{-3}$).

5.1 Cache Optimization ...

Example: Matrix multiply ...

- Block algorithm (tiling) with array padding:

```
double a[N][N+1], b[N][N+1], ...
for (ii=0; ii<N; ii+=4)
 for (kk=0; kk<N; kk+=4)
  for (jj=0; jj<N; jj+=4)
   for (i=0; i<4; i++)
    for (k=0; k<4; k++)
     for (j=0; j<4; j++)
      c[i+ii][j+jj] += a[i+ii][k+kk] * b[k+kk][j+jj];
```

- Matrix is viewed as a matrix of 4x4 sub-matrices
  - multiplication of sub-matrices fits into the L1 cache
- Acheives a performance of 4 GFlop/s even with N=2048

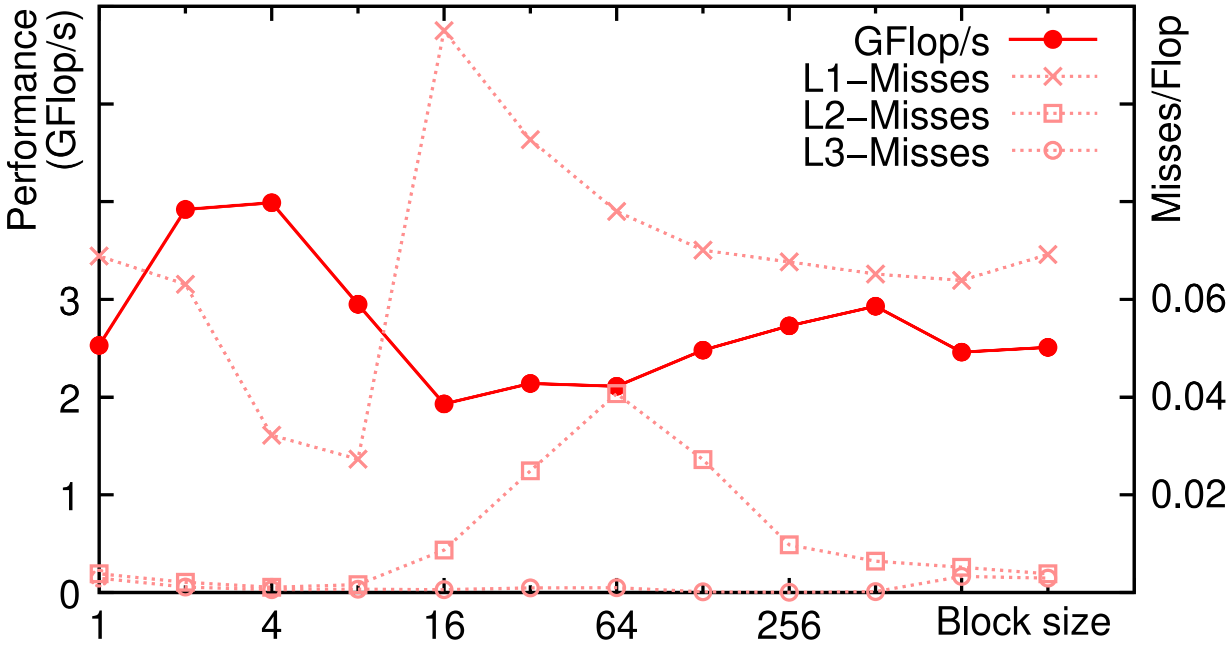
Notes for slide 374:

See slide 229 for the details of the code transformation (strip mining followed by loop interchange) used to create this version of the code.

5.1 Cache Optimization ...

Example: Matrix multiply ...

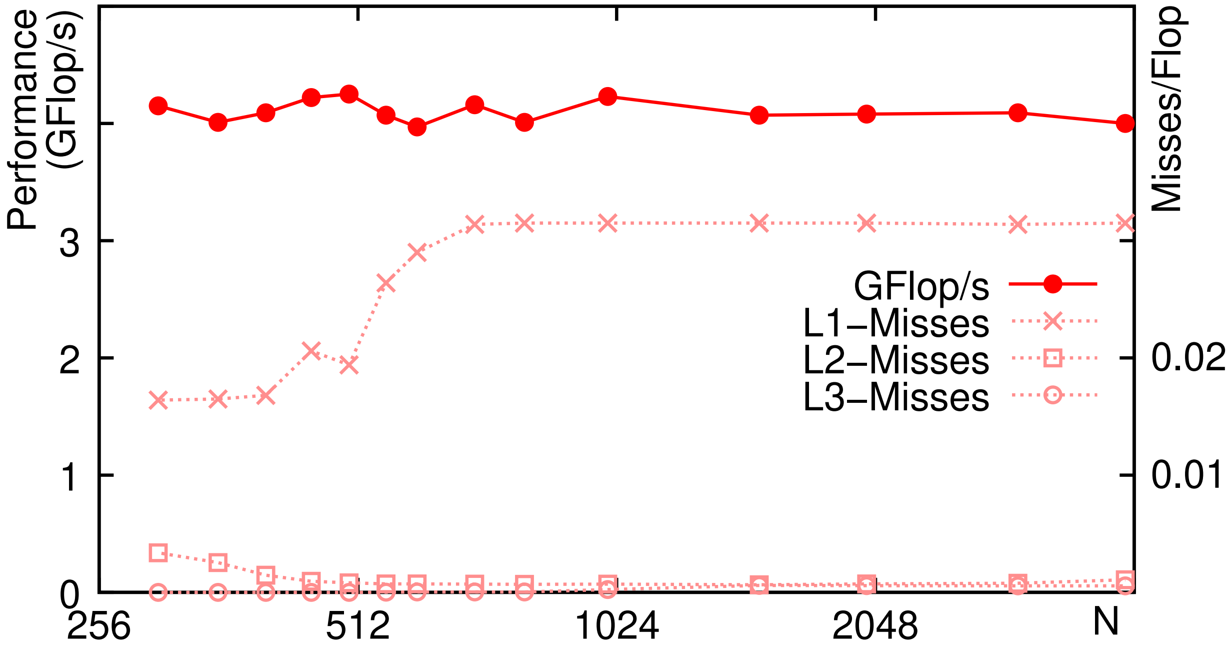
➥ Performance as a function of block size (N=2048):



5.1 Cache Optimization ...

Example: Matrix multiply ...

- Scalability of performance for different matrix sizes:



5.1 Cache Optimization ...

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 378:

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.

5.2 Optimization of Communication

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 $\mathbf{32 \cdot 145 = 4640 \mu s}$
  - one message with 1024 Byte needs only $\mathbf{159 \mu s}$
- 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;
}                          }
```

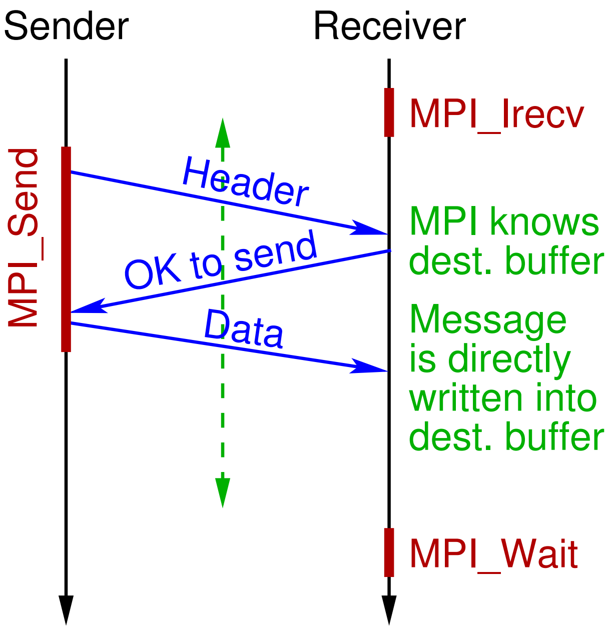
```
for (i=0; i<N; i++) {      recv(b, N, P1);
  b[i] = f(..., i);        for (i=0; i<N; i++) {
}                            a[i] = a[i] + b[i];
send(b, N, P2);            }
```

- 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)

5.2 Optimization of Communication ...

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 oder 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 2024/25

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**)
- Design process (classes of partitioning, communication, mapping)
- **Organisation forms** (manager/worker, task pool, divide and conquer, SPMD, fork/join, ...)
- **Performance** (speedup, efficiency, performance modeling)

6 Summary / Important Topics ...

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

6 Summary / Important Topics ...

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