Parallel Computational Mathematics

Fall 2021

Dr. Patrick Diehl

Contents

Forward

Introduction: C++ and the C++ standard template library

1. Introduction C++

This chapter introduces the core features of the C++ language. Specifically, we focus on introducing the C++ standard template library (see Section 3), which is essential for implementing mathematical equations and algorithms. Let us briefly look into this library so that we may implement the numerical examples in Part V. For more details we refer to

- Koenig Andrew. Accelerated C++: practical programming by example. Pearson Education India, 2000

since this book gives an excellent pragmatic overview with many examples. For even more C++ basics, we refer to

- Bjarne Stroustrup. Programming: principles and practice using C++. Pearson Education, 2014.

1.1 History of C and C++

The development of the programming language `C` started in 1972 as an improvement of the language `B`[1] language [82]. In 1978 the book by Dennis Ritchie and Brian Kernighan The C Programming Language [59] became known as the informal specification of the `C` language. In contrast to the simple and small standard library, `C` compilers varied widely and had no standards. The American National Standards Institute (ANSI) began writing the `C` standard based on the Unix implementation of the language. This later became the foundation of the 1988 POSIX[2] standard. One year later, the `C` standard was published as ANSI X3.159-1989 ”Programming Language C.” More common names for this version are ANSI C or C89. The International Organization of Standardization (ISO) adopted the ANSI C specification and published it as ISO/IEC 9899:1990, which is called C90. Note that C90 and C89 refer to the same standard. This standard was revised in the 1990s and published as ISO/IEC 9899:1999 in 1999 which is the C99 standard. In 2007 the C11 standard was published and in 2018 the C18 standard.

Concurrently, in 1979, Bajarne Stroustrup began developing ”C with classes”[3], which later became `C++`. Stroustrup added classes, derived classes, inlining, and default arguments

Listing 1.1: A simple C++ program, the so-called “Hello World” example.

```
1 // a small C++ program
2 #include <iostream>
3
4 int main()
5 {
6     std::cout << "Hello,␣world!" << std::endl;
7     return 0;
8 }
```

to the `C` compiler [97]. The name `C++` symbolizes the increment of `C` using the increment operator `++`. The `C++` Programming Language launched in 1985, though without an official standard [98]. Updated and standardized versions followed: In 1998 C++98 [62]; In 1999 C++ 2.0; In 2003 C++03 [20]; In 2011 C++11 [21]; In 2014 C++14 [22]. The latest standard is C++17 [93], and the upcoming one is C++20.

1.2 Getting started with C++

To begin with C++ programming, we look at a simple C++ program: the classic”Hello World” example. Listing 1.1 shows this program. The first line in green is a comment. Single-line comments start with `//`. Programmers often use them to explain the functionality of the program or the next lines of code. Once may also use multi-line comments[4] by enclosing the text within `/* */`. Comment early and often; comments are crucial for readability and clarity of the program, especially if the code is shared with other collaborators. Fore more details refer to [58].

The second line starts with a so-called include directive[5] `#include <iostream>`. This include directive incorporates functionality of the C++ standard library (see Chpater 3). In our case we include the `iostream` header so that we may print ”Hello World” to the terminal (see Line 6).

The fourth line `int main()` starts with the Main function[6], which is the entry point of the program. This means all subsequent lines are executed sequentially. Every C++ program which will be compiled to an executable file needs exactly one function called `main`, which has an integer `int` as its return type. On most operating systems a return value of zero means that the program executed successfully, and any other value (often 1 or -1) indicates a failure. The second-to-last line `return` is the return statement[7], which must match the return type in front of the `int main()`.

Once we have written the program, we have to compile the C++ code into an executable so that we can run the code and print “Hello world” to the terminal. There are a plethora of C++ compilers[8] available, but this book will use the GNU Compiler Collection (GCC) for all examples. Line 1 in Listing 1.2 shows how to compile the file `lecture1-1.cpp`, which contains the C++ code in Listing 1.1, to an executable. GCC provides the `g++` compiler to compile C++ code and the `gcc` compiler for C code. As the first argument to the `g++` compiler, we enter the C++ file name and add the `-o` option to specify the name of the executable. To run the generated executable, we type `./lecture-1-1` in the terminal. For basic usage of the Linux terminal refer to [73, 83].

Listing 1.2: Compilation and execution of the C++ program.

```
g++ lecture1-1.cpp -o lecture1-1
./lecture1-1
```

Exercise 1.1 Download the example program[9] from GitHub and compile it with your favorite C++ compiler. After running the example you can try to modify it–For example you could print a different text or add a second line to the output. ■

1.3 Fundamental data types

In this section we introduce the fundamental data types[10] provided by the C++ language. First, the numeric data types. To represent natural numbers $\mathbb{N} = \{0,1,2,\ldots\}$ the `unsigned int` data type is available. To represent integer numbers $\mathbb{Z} = \{\ldots,-2,-1,0,1,2,\ldots\}$ the `int` data type is available. For these data types we can apply the following sizes: `short`, `long`, and `long long`. In the `#include <climits>`[11] header the minimal and maximal value of all integer data types are defined. For example the minimal value of `int` data type is given by `INT_MIN` and the maximal value by `INT_MAX`, respectively. For more details about the binary numeral system, refer to [34].

To represent real numbers $\mathbb{R}$ the `float` data type and `double` data type are available. In the `#include <cfloat>`[12] header the minimal and maximal value of all floating point data types are defined. For example the minimal value of `double` data type is given by `DBL_MIN` and the maximal value by `DBL_MAX`, respectively.

Fore more details about the IEEE 474 standard for how floating point numbers are represented in the computer, refer to [35, 50]. Table 1.1 summarizes all the available numeric data types and their ranges. The next section shows how to get the range of the IEEE 474 standard for floating point numbers.

Data type	Size (Bytes)	Min	Max
		Natural numbers $\mathbb{N}$	
`unsigned short int`	2	0	65,535
`unsigned int`	4	0	4,294,967,295
`unsigned long int`	4	0	4,294,967,295
`unsigned long long int`	8	0	8,446,744,073,709,551,615
		Integer numbers $\mathbb{Z}$	
`short int`	2	-32,768	32,768
`int`	4	-2,147,483,648	2,147,483,648
`long long int`	8	-2^{63}	$2^{63}-1$
		Real numbers $\mathbb{R}$	
`float`	4		
`double`	8		

Table 1.1: Overview of the fundamental numeric data types.

To represent a boolean value we use the $\mathbf{B} = \{0,1\}$ the `bool` data type which has

Listing 1.3: Computation of the sum from 1 up to n using the for loop statement.

```
1  // Compute the sum using a for loop
2  #include <iostream>
3
4  int main()
5  {
6      unsigned int result = 0;
7      unsigned int n = 10;
8
9      for(size_t i = 0; i < n; i=i+1)
10         result = result + i;
11
12     std::cout << "Result=␣" << result << std::endl;
13
14     return 0;
15 }
```

exactly one of the two available values, `true` or `false`. Note that the C++ STL offers `std::complex`[13] for complex numbers $\mathbb{C}$, however, this one is not within the fundamental data types.

1.4 Statements and flow control

1.4.1 Iteration statements

For some applications, we have to repeat an instruction or group of instructions multiple times. The C++ language provides two iteration statements: the `for` loop and the `while` loop. Let us look how to compute the sum of the numbers from 1 up to n

$$r = \sum_{i=1}^{n} i. \tag{1.1}$$

The first solution uses a `for` loop statement[14], which is shown in Listing 1.3. Line 9 shows the `for` loop statement with its three arguments. First, the so–called loop variable `size_t i = 0` which is initialized to zero. Note that the loop variable is only defined within the loop's body (The part between the curly braces). Second, the condition statement `i < n`, which means that the loop body is repeated until the variable i is equal to or larger than n. The third statement manipulates the loop variable. In our case the loop variable is incremented by one after each execution of the loop body. Note that we use the `for` loop statement if we know in advance exactly how many times we want to repeat a block of code.

The second option employs a `while` loop statement[15], which is shown in Listing 1.4. Line 10 shows the `while` loop statement with its one argument. This is the condition statement `i < n`, which means that the loop body is repeated until the variable i is equal to or larger than n. Note in the previous example we had three arguments. In this case the loop variable is declared before the loop in Line 9, and the third statement appears in Line 13 where the loop variable is incremented by one in each iteration. Note that we use the `while` loop statement, if we do not know the number of iterations in advance.

Listing 1.4: Computation of the sum from 1 up to n using the while loop statement..

```cpp
// Compute the sum using a while loop
#include <iostream>

int main()
{
    unsigned int result = 0;
    unsigned int n = 10;

    size_t i = 0;
    while ( i < n)
    {
        result = result + i;
        i+= 1;
    }
    std::cout << "Result=␣" << result << std::endl;

    return 0;
}
```

This example demonstrates that we can write every `for` loop statement as a `while` loop statement. For more details we refer to [6, Chapter 2].

Exercise 1.2 Explain in your own words in which cases you should use a `for` loop statement and a `while` loop statement. ■

1.4.2 Selection statements

For some applications, different sections of code should run depending on certain conditions. Equation 1.2 shows how to compute the sum from 1 to n with different cases for even and odd numbers. If the number is even, it is added to the result, but if it is odd, its square is added to the result.

$$r = \sum_{i=1}^{n} f(i) \text{ with } f(i) = \begin{cases} i, & \text{if } i \text{ is even} \\ i^2, & \text{else} \end{cases} \tag{1.2}$$

Listing 1.5 shows the implementation of Equation 1.2 using a `for` loop. The skeleton of the `for` loop is identical to the one in Listing 1.3, but the `if` statement[16] in Line 8 is added to check whether the current number in the series is even or odd. The `if` statement takes exactly one argument, the condition statement. If the statement is evaluated as `true` the code between `if` and `else` runs. If the statement is evaluated as `false` the code line after `else` runs. It is also possible to use `else if` after the first `if`.

The second selection statement is the `switch` statement[17]. We use this statement to execute different code branches depending on a single variable. Listing 1.6 shows one example that writes the name of the color to the standard output. In this case we use an enumeration `enum`[18] to store the colors. The `switch` takes one argument and executes the code between the matching `case` and the following `break`. For more details we refer to [6, Chapter 2].

Listing 1.5: Computation of the sum from one up to n using the selection statement.

```cpp
// Example with if statement
#include <iostream>

int main()
{
    size_t result = 0;
    for(size_t i = 1; i != 5; i++){
            if( i % 1 == 0)
                    result = result + i;
            else
                    result = result + i * i;
    }

    std::cout << "Result=␣" << result << std::endl;

    return 0;
}
```

1.5 Operators

For the example in Listing 1.3 we have seen the operator `i<n` which is a so–called comparison operator. Next to the comparison operators, C++ language has following operators[19]:

- Comparison operators, see Table 1.2,
- Arithmetic operators, see Table 1.3,
- Logical operators, see Table 1.4, and
- Assignment operators, see Table 1.5,

logical operators, arithmetic, and assignment.

Exercise 1.3 Write a small C++ program using selection statements and operators to determine if a given year is a lap year. Following logical statements should be implemented:

- If year is divided by 4 but not by 100, then it is a leap year.
- If year is divided by both 100 and 400, then it is a leap year.
- If year is divided by 400, then it is a leap year.
- And in all other cases, it is not a leap year.

■

1.5.1 Operator overloading

The operators in the previous section are defined for the fundamental data types, see Section 1.3, and for the STL containers, see Section 3.2, if applicable. However, for own defined `struct` and `class` these are not defined and the programmer has to define them. With C++ 17 38 operators can be overloaded and two more operators were added since C++ 20[20]. Let us look into the `struct` for the mathematical vector. We refer to Section 1.6.2 for more details about `struct` and focus on the overloading of operators in this section.

Listing 1.7 shows the definition of the `struct` vector as a template `template<typename T>`. For more details, we refer to Section 1.8. To do some operation like to add two vectors

Operator	Name	Example
`==`	Equal to	`x==y`
`!=`	Not equal	`x!=y`
`>`	Greater than	`x > y`
`<`	Less than	`x < y`
`>=`	Greater than or equal	`x >= y`
`<=`	Less than or equal	`x <= y`

Table 1.2: Comparison operators

Operator	Name	Description	Example
`+`	Addition	Computes the sum of two values	$2+2=4$
`-`	Subtraction	Computes the difference of two values	$5-3=2$
`/`	Division	Divides two values	$6/2=3$
`*`	Multiplication	Multiplies two values	$2\times 2=4$
`%`	Modulo	Returns the division remainder	`2%1=0`
`++`	Increments	Add plus one to the value	`1++=2`
`--`	Decrements	Subtract one of the value	`1--=0`

Table 1.3: Arithmetic operators

Operator	Name	Description	Example
`&&`	Logical and	Returns `true` if both statements are true	`x > 5 && x < 10`
||	Logical or	Returns `true` if one statement is true	`x > 5` |y < 10|
`!`	Logical not	Inverse the statement	`!(x > 5 && x < 10)`

Table 1.4: Logical operators

Operator	Name	Example	Equivalent
`=`	Assignment	`x = 5`	`x = 5`
`+=`	Plus equal	`x+= 5`	`x = x + 5`
`-=`	Minus equal	`x-= 5`	`x = x - 5`
`*=`	Multiplication equal	`x*= 5`	`x= x * 5`
`/=`	Division equal	`x/= 5`	`x= x / 5`
`%=`	Modulo equal	`x%= 5`	`x = x % 5`

Table 1.5: Assignment operators

Listing 1.6: Computation of the sum from one up to n using the selection statement.

```cpp
// Example for a switch statement
#include <iostream>

int main()
{
    enum color {RED, GREEN, BLUE};
    switch(RED) {
        case RED:   std::cout << "red\n"; break;
        case GREEN: std::cout << "green\n"; break;
        case BLUE:  std::cout << "blue\n"; break;
    }

    return 0;
}
```

`vector<double>a;` and `vector<double>b;` we have to define the operator `+`. This is done in Line 7 and is similar to the definition of a function. The name of the function has to be `operator+` since we overload the plus operator. As the argument a second vector `rhs` in our example the vector `b` is provided. Since we return a new vector the return type of the function is `vector<T>`. The values of the new vector are the additions of the vector components. Because we overloaded the plus operator, we can write following expression `vector<double> c = a+b;`. However, for the expression `vector<double> c = a-b;` the C++ compiler would report following error "error: no match for 'operator-' (operand types are 'vector' and 'vector')" since the minus operator was not overloaded.

Exercise 1.4 Overload the minus and the multiplication operator for the `struct vector`. ■

In Line 11 the output parameter `<<` is overloaded to print the coordinate values to the standard output stream. For this operator two arguments are provided. The `ostream& os` and after this the vector to be printed. In Line 13 the vector coordinates are printed to the output stream with the predefined operator `<<`. The return type of the operator overload function is of the type `ostream&`. For this operator overload, the keyword `friend`[21] is needed. using the `friend` declarations allows a function or another class access to private and protected members of the struct.

Exercise 1.5 Overload the input operator >> for the `struct vector`. ■

1.6 Structuring source code

For large code bases, we like to organize the code and avoid to have one huge file with thousand of lines. There, C++ provides two fundamental ways to organize the code

1. To structure the code it self, we can use functions, e.g. `double norm()`, and `struct` or `class`.
2. To split the code into separated files to make all files shorter and separate the code by its functionality, we can use the s-called header files and source files.

For more details we refer to [6, Chapter 4]. Within the computer science, the research area software engineering deals with the aspect how to organize large code bases and make it

Listing 1.7: Example for the operator overload for the plus operator and the output operator.

```
template<typename T>
struct vector {
T x;
T y;
T z;
// Overload the addition operator
vector<T> operator+(const vector<T> rhs){
return vector<T>( x + rhs.x, y + rhs.y, z + rhs.z );
}
//Overload the output operator
friend ostream& operator
        <<(ostream& os, const vector<T>& vec)
{
    os << vec.x << " " << vec.y << " " <<  vec.z;
    return os;
}
};
```

Listing 1.8: Example for a function definition to compute the maximum of two numbers.

```
1 int max(int a, int b)
2 {
3 return a>b?a:b;
4 }
```

maintainable. For more details, we refer to [49, 94].

1.6.1 Functions

To use code again and do not have to repeat code blocks multiple times, one can use function definitions[22]. Listings 1.8 shows the definition of the function `max`. In Line 1 the return type `int` of the function is defined which means that this function will return one integer value. This is happening in Line 3 using the `return`[23] keyword. If the function has no return value, the keyword `void` is used. In Line 1 the name of the function[24] `max` is defined and in the parentheses the function arguments are provided separated by commas. In this example two integer values with the name `a` and `b` are provided. For the return value, a short form of the `if` statement. the so–called conditional operator[25], is provided which means if $a > b$ return a and else return b. The function is called as `double result = max(5,7.7)`.

Function are defined between `#include` and `int main (void)` in the source code file. Listing 1.9 shows the usage of a function definition for the example in Equation 1.2. For more details we refer to [6, Chapter 4].

1.6.2 Struct

In some case, we like to group data, for example to represent a vector $v = (x,y,z)^T \in \mathbb{R}^3$. Here, the `struct`[26] expression is provided. Note that the struct was introduced in the C language and its companion in the C++ language is the `class` expression. However, Listing 1.9: Example for a function definition to compute Equation 1.2.

```cpp
// Example: function defintion
#include <iostream>

size_t f(size_t i)
{
    if( i % 1 == 0)
            return  i;
    else
        return  i * i;
}

int main()
{
    size_t result = 0;
    for(size_t i = 1; i != 5; i++){
        result = f(i);
    }

    std::cout << "Result=␣" << result << std::endl;

    return 0;
}
```

to make the C language a subset of the C++ language, the `struct` is still available. Listing 1.10 shows the struct with the three variables for each direction of the vector space. To declare a vector, we just write `struct vector v` to have an vector with the name `v` and to initialize the vector with the unit vector `struct vector v = {1,1,1}`[27]. To access the x component of the vector, we write `v.x` and to assign a new value the expression `v.x=42` is used. For more details we refer to [6, Chapter 4].

Constructor

Each `struct` and `class` has a default constructor[28]. However, one can overload the constructor for example to initialize the vector a zero $v = \{0,0,0\}$. Line 11 shows the constructor to initialize an zero vector. The constructor is like a function without the `return` option with the same name as the `struct` and `class`. As the constructor arguments the three vector components are given. Note that we assign the value zero to all of them. In Line 12 we assign the argument's values to the variables within the struct by using `x(x)` which means that we assign the `double x` of the `struct` the value of the `x` in the parentheses. Now we can initialize the `struct` in two different ways. First, using `struct vector v;` will result in $v = \{0,0,0\}$ since we assign zero to all the values. Second, using `struct vector v = vector(1,2,3);` will result in $v = \{1,2,3\}$. For more details we refer to [6, Chapter 4].

Member function

A often used task is to compute the length of a vector $\sqrt{x^2+y^2+z^2}$, thus we want to add this function to the `struct vector` to call `norm()` to compute the norm, see Line 15. The syntax for member functions is the same as for functions. see Section 1.6.1. The main

Listing 1.10: Example for a structure for a three dimensional vector.

```
#include <cmath>

struct vector
{
// vector components
double x;
double y;
double z;

// constructor
vector(double x=0, double y=0, double z=0)
        x(x), y(y), z(z) {}

// member function to compute the vector's length
double norm(){
        return std::sqrt(x*x+y*y+z*z);
}
}
```

difference is that the function definition is between the parentheses of the `struct` definition. For more details we refer to [6, Chapter 4].

Exercise 1.6 Transform the `struct` in Listing 1.10 to a `class`. ■

1.6.3 Header and Source files

A header file[29] is a text file and a common naming convention is that header files end with .h or .hpp, e.g. average.h. To use the defined function in the header file, the file is included using the `#include`[30] expression for example `#include<average.h>`. Note that the header files if the C++ standard library and the C++ STL do not end with .h or .hpp. Before we look into the syntax of a header file, some remarks on good and bad practice are given.

Following things are considered as good practice:
- Each header file provides exactly one functionality
- Each header file includes all its dependencies

Following things should not be in header files and be considered as bad practice:

- built-in type definitions at namespace or global scope
- non-inline function definitions
- non-const variable definitions
- aggregate definitions
- unnamed namespaces
- using directives

Listing 1.11 shows an example for a header file for the median function. At the beginning and at the end of each header file, the so-called include guards avoid that functions or data structures have multiple definitions. In Line 1 we check if the definition `UTIL_H` is not defined by using the expression `ifndef`[31] and is closed in Line 15. The compiler checks if

Listing 1.11: Example for header file.

```
1  #ifndef UTIL_H  //include guard
2  #define UTIL_H
3
4  #include <vector>
5  #include <algorithm>
6
7  // Utilities for the vector container
8  namespace util {
9
10 double average(std::vector<double> vec){
11 return std::accumulate(vec.begin(), vec.end(), 0.0f)
12      / vec.size();
13 }
14 }
15 #endif
```

the definition `UTIL_H` was already seen and only if not, the source code is compiled. To let the compiler know that the code was compiled the expression `define`[32] in Line 2 is used. A short form is the `#pragma once`[33]. Next, all headers needed in this file are included.

In Line 8 the `namespace`[34] expression is used to avoid naming conflicts and structure in large projects. Because the function `average` is within `namespace util` defnied, the usage of this function is `double res = util::average(vector);`. With the namespaces one can structure the projects as computation, util, and IO for example. So by using the namespace it is more defined which functionality is provided. It is possible to nest namespaces to have more structure.

A common folder structure for a project with header files in shown in Listing 1.12. In the folder includes all header files (*.hpp) and the folder sources all source files (*.cpp) are collected. Listing 1.13 shows the usage of the average function defined in the file util.h. However, to compile the file main.cpp file, the compiler needs to know where the util.h is located. The compilation of the main.cpp is the same as before, but the path to the header files needs to be specified as `-I ../includes`, see Listing 1.14.

Listing 1.12: Folder structure for a project with header files.

```
sources/
    main.cpp
includes/
    util.h
```

Listing 1.13: Example for the main.cpp file using a header file.

```
#include<util.h>

int main(void){

std::vector<double> vec = {1,2,3};

double res = util::average(vec);
}
```

Listing 1.14: Compilation of the main.cpp file using a header file.

```
g++ -o main -I ../includes main.cpp
```

1.6.4 Classes

One important feature provided by the C++ language is the feature `class`[35]. Note the with C we had `struct` which are very similar to the `class`. However, one thing of the C++ language is the compatibility to the C language. Meaning that it is possible to compile C code using a C++ compiler. Therefore, the `struct` keyword is still available but not really needed since the keyword `class` is provided.

Listing 1.15 shows the definition of a `class` for a three dimensional vector. In Line 1 a `class` with the name `vector3` is defined. All source code within the `{ };` is in the scope of the `class`. Three so-called privacy[36] option are available for classes. The first option is shown in Line 3. The `private` option means that the `double` values are only accessible within the `class` itself. So these values are hidden and can not be changed without using any of the `public` methods below. The methods below Line 7 are declared as public. Which means the are accessible from outside the `class`. Let us make an example for the accessibility by creating `Vector3 vector;` an object. Since `double x` is declared as `private`, we can not call `vector.x;` since it is not accessible from outside the class. However, we can type `double len = vec.norm();` since this method is defined as `public`. The third option is the `friend`[37] option. The `friend` option allows a function or another class access to private members.

A common practice is to have header files and class files to provide the functionality of a `class` to other classes. Listing 1.16 shows the header file extracted from the `class` definition in Listing 1.15. In the header file the attributes and the member functions of the `class` are defined. For example the function `double norm();` has no definition in this file and it is not define how the function is implemented. However, we know that the `class vector3` has this function. The implementation of the function is done in the corresponding source file, see Listing 1.17. Note that we have to include the corresponding header file in Line 1. In addition, in a class file, we have to add the name of the `class` to all functions, see Line 3, we have to add `vector3::` to the constructor and the function in Line 8. For more details we refer to [6, Chapter 9].

Listing 1.15: Example for a class definition.

```cpp
class vector3 {

private:

double x , y , z;

public:

vector3(double x = 0, double y=0, double z=0)
    : x(x) , y(y) ,z(z) {}

double norm(){ return std::sqrt(x*x+y*y+z*z);}
};
```

Listing 1.16: Corresponding header file to the class definition in Listing 1.15.

```cpp
class vector3 {

private:

double x , y , z;

public:
vector3(double x = 0, double y=0, double z=0);

double norm();
};
```

For the compilation, we have to first compile the source file using `g++ -c vector3.cpp` to compile the class file `vector3.cpp`. Note since we compile a file without a `int main()` function the option `-c` is needed. The last step is to compile the `main.cpp` file using `g++ main.cpp vector3.o -o main`. Note the file `vector3.o` was generated with the previous command. For more details about making compilation easier, we refer to Section 1.7.

1.7 Building with CMake

CMake[38] is a cross-platform free and open-source software tool for managing the build process of software using a compiler-independent method. It supports directory hierarchies and applications that depend on multiple libraries. It is used in conjunction with native build environments such as Make, Ninja, Apple's Xcode, and Microsoft Visual Studio. It has minimal dependencies, requiring only a C++ compiler on its own build system[39].

In the previous two section, we learned how to compile using header files and classes using the GNU compiler. However, for large code bases, one do not want to compile all files by hand or write a script to do so. CMake is a neat tool to generate the build recipe for us. First, we start to look into how to compile a single source file (`main.cpp`). Therefore, Listing 1.17: Corresponding class file to the class definition in Listing 1.15.

```cpp
#include "vector3.h"

vector3::vector2(double x, double y, double z)
{
    x = x; x = y; z = z;
}

double vector3::norm(){return std::sqrt(x*x+y*y+z*z)}
```

we generate a `CMakeLists.txt` file in the same folder as the source file is located. The content of the `CMakeLists.txt` is shown in Listing 1.18. In Line 1 the minimum required CMake version is specified. This is important because some features are only available in this version or are deprecated in any older version. In Line 2 the project's name is defined. In Line 3 we define that we want to compile the file `main.cpp` as an executable with the name out. This would be equivalent to `g++ main.cpp -o hello`. Listing 1.19 shows how to compile the `main.cpp` file using CMake. In line 1 a new folder with the name `build` is generated. The best practice is to have a build folder where the code is compiled. So we can easily delete the folder and have a clean build. In Line 2 we change to the build folder. In Line 3 we call `cmake ..` to generate the `Makefile`. Note that we have to use the two dots, because the `CMakeLists.txt` is located one folder above. In Line 4 we call `make` to compile the code and in Line 5 we execute the compiled program.

For a project with class and header files, a common folder structure is shown in Listing 1.21. A common practice is to have a folder `include` for the header files, a folder `src` for the source files, and the `CMakeLists.txt`. Listing 1.21 shows the corresponding `CMakeLists.txt` file. In Line 4 the include directory is added to the project which means `-I ../includes` is added as an argument to the compiler. In Line 6 the source files to compile are added manually by specifying their file names. This is feasible for small projects, however, for large amount of files it is too much work. Line 10 shows are more handy way to add all source code files in the folder `src`.The last step is to add all the sources to the executable in Line 12. Note that we only covered the minimal basics of CMake. For more details, we refer to [23].

Listing 1.18: Content of the CMakeLists.txt.

```
1 cmake_minimum_required(VERSION 3.10.1)
2 project(hello_world)
3 add_executable(hello main.cpp)
```

Listing 1.19: Build instructions for CMake.

```
1 mkdir build
2 cd build
3 cmake ..
4 make
5 ./hello
```

Listing 1.20: Structure of a CMake project.

```
.
|-- CMakeLists.txt
|-- build
|-- include
|   \-- vector2.h
\-- src
    |-- vector2.cpp
    \-- main.cpp
3 directories, 4 files
```

Listing 1.21: Build instructions for CMake.

```
project(directory_test)

#Include headers
include_directories(include
   )

#Adding all sources
#set(SOURCES src/main.cpp
   src/vector2.cpp)

#Adding sources easier
file(GLOB SOURCES "src/*.
   cpp")

add_executable(test ${
   SOURCES})
```

1.8 Generic programming

In some cases, we need to write the same function for different data types, e.g. `double` and `float`, see Listing 1.22. We would need to write the same function for all data types. Thus, we will produce the same computation multiple time and have too much redundant code.If there is an error in the computation, we would have to correct it for all of the functions. Function templates[40] are provided by the C++ language. Listing 1.22 shows starting at Line 11 how to combine the previous two function into one. In Line 11 the expression `typename` indicates that we define a function template and within the parentheses the `typename T` is defined which is a placeholder for the explicit data type. For the remaining function definition everything keeps the same and only the specific data type, e.g. `double` and `float`, is replaced by `T`. Now, the function is used as `add<double>` or `add<float>` or `add<int>` for the various data types without explicit implementing all of them. This is a neat feature to reduce the amount of code.

The same is possible for `struct` and `classes` by adding `template<typename T>`[41] above the definition and using the `T` instead of `double` as in Listing 1.10. Now, the function is used as `struct vector <double> v;` or `struct vector<float> v;` or `struct vector <int> v;` for the various data types without explicit implementing all of them. For the function `norm()` there is no need to use `template<typename T>` again and the return type `double` is replaced by `T`. Fore more details, we refer to [53]. For further watching, we recommend the C++ Lecture 2 - Template Programming 2[42] and C++ Lecture 4 - Template Meta Programming[43].

Exercise 1.7 Use the `struct` in Listing 1.10 and make it a generic one by adding the `template <typename T>` and replace all `double` by `T`. ■

1.9 Lambda function

In Section 1.6.1 function expression was introduced as `int compute(int a, int b);` . Here, the function has a name `compute` and this name is used to call the function.

Listing 1.22: Example for the usage function templates.

```cpp
// Definition of multiple functions
double add(double a, double b) {
        return a + b;
}

float add(float a, float b) {
        return a + b;
}

// Function template
template<typename T>
T add(T a, Tb){
        return a+b;
}
```

However, in some cases it can be neat to use a function exactly once, for example in the STL Algorithms, see Section 3.3. To use a function only one time the so-called lambda expression or lambda function[44] is shown in Listing 1.23. Within the [...] the capture clause of the parameters within the (...) are defined. Ans as for the function the code of the function is defined within {...}. Note that the `-> return-type`, e.g. `-> int`, is somehow an optimal parameter and in most cases this parameter is evaluated by the compiler and only in few special cases the return type needs to be defined. Following capture clauses[45] are available:

- `[&]` : capture all external variable by reference
- `[=]` : capture all external variable by value
- `[a, &b]` : capture a by value and b by reference

For more details about the capture classes, we refer to the next section.

Listing 1.24 sketches some practical example how to transform a function to a lambda expression. From Line 2–4 defines the function to print the element of the vector piece-wise to the standard output stream. In Line 5 the short form of a `for` loop is used to loop over all elements of the vector piece-wise. Note that `i` is not the index and it is the value of the vector at position `std::for_each` is handling. Since we use the function `void print(int i)` only once, a short form of this function is used in Line 8.

Exercise 1.8 Try to understand the transformation of the function `void print(int i)` to the corresponding lambda function. ■

Listing 1.24 also shows some example to find the first number greater than 4 in a vector using the `std::find_if`[46]. Many more algorithms are available in the `#include <algorithm>`[47].

1.10 Pointers

Imagine following conversation:

Person A: Would you teach a toddler how to eat with a butcher's knife?

Listing 1.23: Example for lambda functions.

```
[ capture clause ] (parameters) -> return-type
{
   //definition of method
}
```

Listing 1.24: Practical example for a lambda function.

```
// Print the values of the vector using a function
void print(int i){
std::cout << i << std::endl;
}
std::for_each(v.begin(), v.end(), print);

// Print the values of the vector using a function
std::for_each(v.begin(),v.end(),
        [](int i){std::cout<< i << std::endl;})

// Find the first number greater than 4 in a vector
std::vector<int>:: iterator p = std::find_if(
    v.begin(),
        v.end(),
        [](int i)
    {
        return i > 4;
    });
std::cout << "First␣number␣greater␣than␣4␣is␣:␣" << *p <<
   endl;
```

Listing 1.25: Introduction to pointers to objects.

```
// Initialize
int x = 42;

// Get the pointer to the object x
int* p = &x;

// Get the object the pointer is pointing to
int tmp = *p;

// Using pointers to manipulate objects
std::cout << x << std::endl;
*p = 43;
std::cout << x << std::endl;
```

Person B: No!

Person A: So stop mentioning pointers to people barely starting with C++.

Therefore, the book does not talk much about pointers, because in most cases, you do not need pointers to implement mathematical algorithms. If you need them, you should carefully check your implementation and see if you can avoid them. However, the introduce the basics so you know about pointers and can use them if you really need them.

A pointer p is a value that represents the address of an object. Every object x has a distinct unique address to a part of the computer's memory. Listing 1.25 gives some example. In Line 2 the object `int x` is generated in the computer's memory and the value 42 is stored. In Line 5 the address to the memory where the object `x` is stored is store in `int* p` by using the so–called `&` address operator. In Line 8 we get the value 42 stored at the address `p` by using the so-called deference operator. In Line 11 the value 42 of the object `x` is printed. In Line 12 we use the pointer `p` to the object `x` to set a new value 43. In Line 13, we print the object `x` again and we will see the new value 43 without accessing the object `x` directly.

In the first example, we used a pointer to a single object. In the second example, we will use a pointer to an array of objects, see Listing 1.26. In Line 1 a pointer to the `array` is obtained. Using the dereference operator on the pointer gives us access to the first element of the array, see Line 2. With the so-called pointer arithmetic we can access the second and third element of the error by adding one or two the pointer before we use the dereference operator. In Line 13 we compute the distance between two pointers which is the length of the array in this case. Note that `ptrdiff_t`[48] is a signed type because the distance can be negative.

In the last example, we look into pointers to function, the so-called function pointers. In Listing 1.27 shows how to generate function pointers to the function `square`. In Line 7 the first possibility to generate a function pointer to the `square` function. The first `int` stands for the return type of the function and the second `int` for the function's argument. In Line 2 the left-hand side is the same, but on the right-hand side we use the address

Listing 1.26: Introduction to pointers to range of objects.

```cpp
int* array = new int[3];
*array = 1;
*(array + 1) = 2;
*(array + 2) = 3;

// Accessing the first element
int first = *array;

// Accessing the second element
int second = *(array + 1);

// Getting the distance between two pointers
ptrdiff_t dist = array+2 - array;
```

Listing 1.27: Example for function pointers.

```cpp
1  int square(int a)
2  {
3  return a * a;
4  }
5
6  // Generating a function pointer
7  int (*fp)(int) = square; //We need the (int) for
8  int (*fp2)(int) = &square; // the return type
9
10 // Calling the function using its pointer
11 std::cout << (*fp)(5);
12 std::cout << fp2(5);
```

operator. In Lines 112–12 the function is called using its function pointer. Note that each of two lines to get the pointer or call the function are equivalent.

1.10.1 Memory management

From the Spider-Man comics and the movies, we all know the sentence

With great power there must also come great responsibility

and this can be referenced for the usage of pointers as well. In C++ we have two kind of memory management:

1. Automatic memory management
   This is what happens using the C++ standard library and the C++ STL. The system is allocation the memory for use, e.g. if we generate some array `double int[8]` or one of the containers. If the array goes out of scope which means it is not used anymore, the system deallocates the used memory.
2. Dynamic memory management
   If we use a pointer, the user has to allocate and clear the memory for each generated

Listing 1.28: Example for dynamic memory management.

```
1  // Allocate the memory for one single integer value
2  int* p = new int(42);
3
4  // Deallocate the memory
5  delete p;
6
7  // Allocate the memory for five integer values
8  int* p = new int[5];
9
10 // Deallocate the memory
11 delete[] p;
```

object. The programmer allocates the memory with the `new`[49] keyword and deallocates the memory with the the `delete`[50] keyword.

Listing 1.28 shows some examples for dynamic memory management. In Line 2 the memory for one single integer value is allocated. In Line 5 the memory is deallocated which means the memory at this address is free again. In Line 8 the memory for five integer values is allocated. Note that here we have to add `[]` to the `delete` keyword.

1.11 Moving data

In some cases, if we pass a value to some function, we like to avoid to copy the data and instead we like to `std::move`[51] the data. Let us look into the example in Listing 1.29 to explain what we mean by moving a value. In Line 4 we add the string `hello` to the vector using the `push_back` method. However, by passing the string `hello` a copy of the string is passed to the function. Depending on the object size, the copying takes some time. However, if we print the content of the string `hello` by using the copy, the value of the string will be `"Hello"`. If, we want to avoid the copying, one can use the `std::move` function in Line 9. However, if we print the content of the sting `hello`, the empty string will be printed. This happens since we moved the data (in that case the content `"Hello"`) to the `std::vector<std::string>`. So, if we print the content of `v[i]`, we will see again the content `"Hello"` again, since we moved the content. Note that you have to be aware of undefined states after moving. For example `v.clear()` is a valid state since there is no precondition. However, `v.back()` could result in a undefined behavior, since the size of the string is zero.

1.11.1 Smart pointer

The so-called smart pointers are defined in the header `#include <memory>`[52]. In the previous section, we looked at so-called raw pointers and these should be used only in small code blocks of limited scope or where performance is a major issue. Using a raw pointer you are responsible to manage the memory and deallocate the memory if the object is not needed anymore. Using a smart pointer there is no need to call the `delete` explicitly. The first smart pointer is the `std::unique_ptr`[53], see Listing 1.30. The unique pointer points to exactly one object in the memory and no other pointer can point to this object. In Line 1 we initialize a smart unique pointer containing a array `double []` by using `std::unique_ptr<double[]>a`. We use the `new` operator to allocate a array of size two.

Listing 1.29: Example for the usage of `std::move` to move data.

```
1  std::string hello = "Hello";
2  std::vector<std::string>v;
3
4  // Add the string hello to the end of the vector
5  v.push_back(hello);
6  std::cout << "After␣copying␣the␣string,␣its␣content␣is:␣" <<
      hello << std::endl;
7
8  //Move the data and avoid the copying
9  v.push_back(std::move(hello));
10 std::cout << "After␣moving␣the␣string,␣its␣content␣is:␣" <<
      hello << std::endl;
11
12 //Printing the moved content
13 std::cout << "After␣moving␣the␣string,␣its␣content␣is:␣" << v
      [1] << std::endl;
```

Listing 1.30: Using the smart unique pointer.

```
1  // Generate a unique pointer of a double array
2  std::unique_ptr<double[]>a(new double[2]);
3
4  // Initialize the values
5  a[0] = 1;
6  a[1] = 2;
7
8  // Generate a copy of the array a
9  //std::unique_ptr<double[]>b(a);
10
11 // Generate a copy of the array a
12 std::unique_ptr<double[]>b(std::move(a));
```

For more details about the `new` keyword, we refer to Section 1.10.1. In the Lines 5–6 the values of the array are initialized. Note that the Line+9 is commented out on purpose, since this line of code will not compile. Since we use a `std::unique_ptr` for the array `a`, we can not use a second smart pointer `b` pointing to `a`. However, moving the pointer `a` to the unique pointer `b` will work, since we move the control from `a` to `b`. Fore more details about `std:move`, we refer to Section 1.11.

The second smart pointer is the so–called share pointer `std::shared_ptr`[54]. The shared pointer allows that pointers can point to the same object and a reference counter is used. Listing 1.31 shows the usage of smart pointers. In Line 2 a smart pointer of a double array is generated and we allocate a array of size two. Now, since we use a shared pointer the pointer `a` can be passed to the new share pointer `b`, since multiple pointer can point to the same object. In addition, we can use the function `use_count()` to check the pointers pointing to the object the pointer `a` is pointing to.

Listing 1.31: Using the smart unique pointer.

```cpp
// Generate a unique pointer of a double array
std::shared_ptr<double[]>a(new double[2]);

// Initialize the values
a[0] = 1;
a[1] = 2;

// Generate a copy of the array a
//std::unique_ptr<double[]>b(a);

std::cout << a.use_count) << std::endl;
```

Notes

[1]https://en.wikipedia.org/wiki/B_(programming_language)
[2]https://en.wikipedia.org/wiki/POSIX
[3]http://www.stroustrup.com/bs_faq.html#invention
[4]https://en.cppreference.com/w/cpp/comment
[5]https://en.cppreference.com/w/cpp/preprocessor/include
[6]https://en.cppreference.com/w/cpp/language/main_function
[7]https://en.cppreference.com/w/cpp/language/return
[8]https://en.wikipedia.org/wiki/List_of_compilers#C++_compilers
[9]https://github.com/diehlpkteaching/ParallelComputationMathExamples
[10]https://en.cppreference.com/w/cpp/language/types
[11]https://en.cppreference.com/w/cpp/header/climits
[12]https://en.cppreference.com/w/cpp/header/cfloat
[13]https://en.cppreference.com/w/cpp/numeric/complex
[14]https://en.cppreference.com/w/cpp/language/for
[15]https://en.cppreference.com/w/cpp/language/while
[16]https://en.cppreference.com/w/cpp/language/if
[17]https://en.cppreference.com/w/cpp/language/switch
[18]https://en.cppreference.com/w/cpp/language/enum
[19]https://en.cppreference.com/w/cpp/language/operator_precedence
[20]https://en.cppreference.com/w/cpp/language/operators
[21]https://en.cppreference.com/w/cpp/language/friend
[22]https://en.cppreference.com/w/c/language/function_definition
[23]https://en.cppreference.com/w/cpp/language/return
[24]https://en.cppreference.com/w/cpp/language/functions
[25]https://en.cppreference.com/w/cpp/language/operator_other#Conditional_operator
[26]https://en.cppreference.com/w/c/language/struct
[27]https://en.cppreference.com/w/c/language/struct_initialization
[28]https://en.cppreference.com/w/cpp/language/default_constructor
[29]https://docs.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=vs-2019
[30]https://en.cppreference.com/w/cpp/preprocessor/include
[31]https://en.cppreference.com/w/cpp/preprocessor/conditional
[32]https://en.cppreference.com/w/cpp/preprocessor/replace
[33]https://en.cppreference.com/w/cpp/preprocessor/impl
[34]https://en.cppreference.com/w/cpp/language/namespace
[35]https://en.cppreference.com/w/cpp/language/classes
[36]https://en.cppreference.com/w/cpp/language/access
[37]https://en.cppreference.com/w/cpp/language/friend
[38]https://cmake.org/
[39]https://en.wikipedia.org/wiki/CMake
[40]https://en.cppreference.com/w/cpp/language/function_template
[41]https://en.cppreference.com/w/cpp/language/templates

[42]https://www.youtube.com/watch?v=iU3wsiJ5mts
[43]https://www.youtube.com/watch?v=6PWUByLZO0g
[44]https://en.cppreference.com/w/cpp/language/lambda
[45]https://en.cppreference.com/w/cpp/language/lambda#Lambda_capture
[46]https://en.cppreference.com/w/cpp/algorithm/find
[47]https://en.cppreference.com/w/cpp/algorithm
[48]https://en.cppreference.com/w/cpp/types/ptrdiff_t
[49]https://en.cppreference.com/w/cpp/language/new
[50]https://en.cppreference.com/w/cpp/language/delete
[51]https://en.cppreference.com/w/cpp/utility/move
[52]https://en.cppreference.com/w/cpp/header/memory
[53]https://en.cppreference.com/w/cpp/memory/unique_ptr
[54]https://en.cppreference.com/w/cpp/memory/shared_ptr

2. The C standard library

The ANSI C standard[55] is the specification for the C standard library (libc). The C standard library provides following functionality

- Handling set of characters in the `#include <cstring>` header,
- handling times and dates in the `#include <ctime>` header,
- Support of complex numbers in the `#include <ccomplex>`,
- Mathematical functions in the `#include <cmath>` header,
- Limits of integer types in the `#include <climits>` header,

and many more features. However, these are the features we will use most in this course. For more details, we refer to [54].

2.1 Strings

The STL provides the class `string`[56] to store sequences of characters. For the usage of this class the header `#include <string>` has to be added to the cpp file to make `std::string` available. Listing 2.1 shows how to use the string class to write a set of characters to standard output stream[57] `std::cout` and read them from standard input stream[58] `std::cin`. To use these functionality the `#include <iostream>` header is needed.

In Line 7 the set of characters `"Please␣enter␣your␣name:␣"` is written to the standard output stream using the operator `<<`. In Line 9 a string object with the identifier `name` is declared. All variables have a name `name` and a type `std::string`. Since the variable is declared but not initialized yet, the variable is empty or a null string. The assignment operator `=` is used to initialize the variable with a set of characters `std::string name = "Mike"`. In Line 10 the variable is initialized with the content provided by the standard input stream `std::cin` and the `>>` operator. In Line 12 the content of the variable is written to the standard output stream. Note that you can concatenate strings using the `>>` operator multiple times. To generate a line break the statement `std::endl` is used. Note that we only handled the basis features here, since these are necessary for the purpose of this course. For more details we refer to [6, Chapter 1].

Listing 2.1: Example reading and writing strings.

```cpp
// Read person's name and greet the person
#include <iostream>
#include <string>

int main()
{
    std::cout << "Please␣enter␣your␣name:␣";
    // Read the name
    std::string name;
    std::cin >> name;
    // Writing the name
    std::cout << "Hi,␣" << name << "!" << std::endl;
    return 0;
}
```

2.2 Random number generation

For some applications, e.g. Monte Carlo methods, see Chapter 10, random numbers are essential. The trivial way to generate a integer random number in the range of zero and `RAND_MAX` is to use `std::rand`[59] provided by the `#include <cstdlib>` header. Listing 2.2 shows a small example to generate a random number. Note that one has to provide a seed to the random number generator to get a different random numbers each time the program is executed. One way to do so, is to use the current time `std::time(0)`[60] provided by the `#include <ctime>` header. Line 10 shows how to use the current time passed as an argument `std::srand(std::time(0))` as a seed for the random number generator. Line 12 shows how to get one random number. Note that the seed has to be set only once, but always before any random number is drawn.

For more advanced usage of random number generators the `#include<random>` header is provided. More advanced means that not only integer random number can be drawn and range can be provided. Listing 2.3 shows how to generate uniform distributed random numbers. Line 8 generates a random number device `std::random_device rd` [61]. Next, the engine for the random number generation is chosen. In this case the `mersenne_twister_engine` [70] is used by providing the random device as an argument `std::mt19937 gen(rd())`[62]. Next the uniform distribution has to be specified by `std::uniform_int_distribution` for integer values and `std::uniform_real_distribution` for floating point numbers. In Line 12 the interval from 1 to 6 for integer numbers and in Line 14 for double numbers is specified. Line 15 shows how to get a random number by using the distribution by passing the engine as an argument `dis(gen)`.

2.3 Numerical limits

Since the limits of the numerical data types depend on the various things, the `#include <limits>` header[63] is available to access this information. For the integer data types, the function `std::numeric_limits<unsigned int>::min()` is provided to receive the smallest finite value and the function `std::numeric_limits<unsigned int>::max()` the largest finite value of the `unsigned int` data type.

Listing 2.2: Example using the trivial random number generator.

```
// Include for using rand
#include <cstdlib>
#include <iostream>
//Include for getting the current time
#include <ctime>

int main()
{
    // Use the current time as random seed
    std::srand(std::time(0));
    // Get one random number
    int random_variable = std::rand();
    std::cout
        << "Random␣value␣on␣[0␣" << RAND_MAX << "]:␣"
        << random_variable << '\n';
}
```

Listing 2.3: Example using the trivial random number generator.

```
// Include for advanced random numbers
#include <random>
#include <iostream>

int main()
{
    //Generate a random number device
    std::random_device rd;
    //Set the standard mersenne_twister_engine
    std::mt19937 gen(rd());
    //Specify the interval [1,6]
    std::uniform_int_distribution<size_t> dis(1, 6);
    //Specifiy the interval [1.0,6.0]
    std::uniform_real_distribution<double> disd(1,6);
    std::cout << dis(gen) << "␣" << disd(gen) << '\n';
}
```

Listing 2.4: Example accessing the numerical limits of floating point types.

```cpp
1  #include <limits>
2  #include <iostream>
3
4  int main()
5  {
6    std::cout << "type\tround()\teps\tmin()\t\tmax()\n";
7    std::cout << "double\t"
8      << std::numeric_limits<double>::round_error() <<'\t'
9      << std::numeric_limits<double>::epsilon() <<'\t'
10     << std::numeric_limits<double>::min() <<'\t'
11     << std::numeric_limits<double>::max() <<'\n';
12 }
```

For floating point numbers, two additional values are accessible, see Listing 2.4. In Line 8 the rounding error `std::numeric_limits<double>::round_error()`[64] which returns the maximum rounding error of the given floating-point type is shown. In Line 9 the value epsilon `std::numeric_limits<double>::epsilon()`[65] which is the difference between 1.0 and the next representable value of the given floating-point type is obtained. Fore more details about the IEEE 474 standard how floating point numbers are represented in the computer we refer to [35, 50]. The next two lines of code show how to access the minimal and maximal value.

2.4 Reading and writing files

For numerical simulations, it is essential to read files, e.g. configuration files, and store their values or write the simulation results to permanent storage. First, we look into how to read the content of a file line by line. To do so, the `ifstream`[66] provided by the `#include <fstream>`[67] header. Listing 2.5 shows how to read the file's content `"example.txt"` line by line. In Line 7 a `ifstream` with the name `myfile` is declared. With the parentheses its constructor is called and the parameter is the file name of the file we want to open. Note that we assume that the file is located next to the cpp file. In Line 8 we check if the file could be opened successful. In that case the function `is_open()`[68] will return `true`. In line 10 the function `getline`[69] is called to access the each line of the file. The first argument is the `ifstream` and the second argument is a `std::string` where the line of the file is stored. Each time the function is called there is new content in the argument `line`. If there is no next line. the function returns `false` and the `while` loop stops. In Line 16 the `ifstream` is closed by calling the `close()`[70] function.

Exercise 2.1 Instead of printing the file content to the standard output device, store each line of the file in a `std::vector<string>`. ■

Second, we look into how to write the text `"Writing␣this␣to␣a␣file"` into the file `"example.txt"`, see Listing 2.6. In Line 6 the `std::ofstream`[71] is declared. In Line 7 the function `open()`[72] is called. The first argument is the file name of the file to create. The second argument is the file mode `std::ios::out`[73]. In Line 8 the operator `<<` is used to

Listing 2.5: Example for reading the content of file "example.txt" line by line.

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4
5  int main () {
6    std::string line;
7    std::ifstream myfile ("example.txt");
8    if (myfile.is_open())
9    {
10     while ( getline (myfile,line) )
11     {
12         std::cout << line << '\n';
13     }
14     myfile.close();
15   }
16   return 0;
17 }
```

Listing 2.6: Example for writing to the file "example.txt".

```
1  // basic file operations
2  #include <iostream>
3  #include <fstream>
4
5  int main () {
6      std::ofstream myfile;
7      myfile.open ("example.txt", std::ios::out);
8      myfile << "Writing␣this␣to␣a␣file.\n";
9      myfile.close();
10     return 0;
11 }
```

write the string to the file. By using "n" we indicate a line break and all content after will be in a new line of the file. In Line 9 the file is closed by calling the `close()`[74] method.

Exercise 2.2 Instead writing one string to the file, write all string in a `std::vector<string>` to the file with each string in a new line. ■

.

Notes

[55]http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf
[56]http://www.cplusplus.com/reference/string/string/
[57]http://www.cplusplus.com/reference/iostream/cout/?kw=cout
[58]http://www.cplusplus.com/reference/iostream/cin/?kw=cin
[59]http://www.cplusplus.com/reference/cstdlib/rand/
[60]http://www.cplusplus.com/reference/ctime/time/?kw=time
[61]http://www.cplusplus.com/reference/random/random_device/
[62]http://www.cplusplus.com/reference/random/mersenne_twister_engine/
[63]https://en.cppreference.com/w/cpp/types/numeric_limits
[64]https://en.cppreference.com/w/cpp/types/numeric_limits/round_error
[65]https://en.cppreference.com/w/cpp/types/numeric_limits/epsilon
[66]https://en.cppreference.com/w/cpp/io/basic_ifstream
[67]https://en.cppreference.com/w/cpp/header/fstream
[68]https://en.cppreference.com/w/cpp/io/basic_fstream/is_open
[69]https://en.cppreference.com/w/cpp/string/basic_string/getline
[70]https://en.cppreference.com/w/cpp/io/basic_ifstream/close
[71]https://en.cppreference.com/w/cpp/io/basic_ostream
[72]https://en.cppreference.com/w/cpp/io/basic_ofstream/open
[73]https://en.cppreference.com/w/cpp/io/ios_base/openmode
[74]https://en.cppreference.com/w/cpp/io/basic_ofstream/close

3. The C++ Standard Template Library (STL)

3.1 Overview of the STL

Figure 3.1 shows the four components of the C++ Standard Template Library (STL). The main focus in this course is on the algorithm component, container component, and iterators component. The functions component provides the so–called `Functors`[75]. A functor is an object, which is treated a function or a function pointer. The component iterators[76] provides six iterators for working upon a sequence of values, e.g. containers. The usage of iterators will be discussed in Section 3.2.4. For the Algorithms component[77] following algorithm classes:

- Sorting[78] - Ordering elements in a container with respect to their order,
- Searching[79] - Searching for elements in a sorted array, and
- STL algorithms - Provides algorithms, like finding the largest element (max[80]) in an container or compute the sum[81] of all elements;

will be reviewed. All of these algorithm classes will be showcased on the container `std::vector` in Section 3.2. For more details on the STL we refer to [75, 95], but remember learning C++ is like learning a new sportive activity, practicing (writing code) is essential to improve your skills. For further watching, we recommend the C++ Lecture 1 - The Standard Template Library[82].

Most important take away of this section is:
- Never implement your own algorithm or container, if you can find it within the STL.
- If you can not find it within the STL, think if you really need this feature.

3.2 Containers

Before we look into the containers, we start with an example to showcase the need of containers. Let us assume we want to compute the average

$$a = \frac{1}{n}\sum_{i=1}^{n} i \tag{3.1}$$

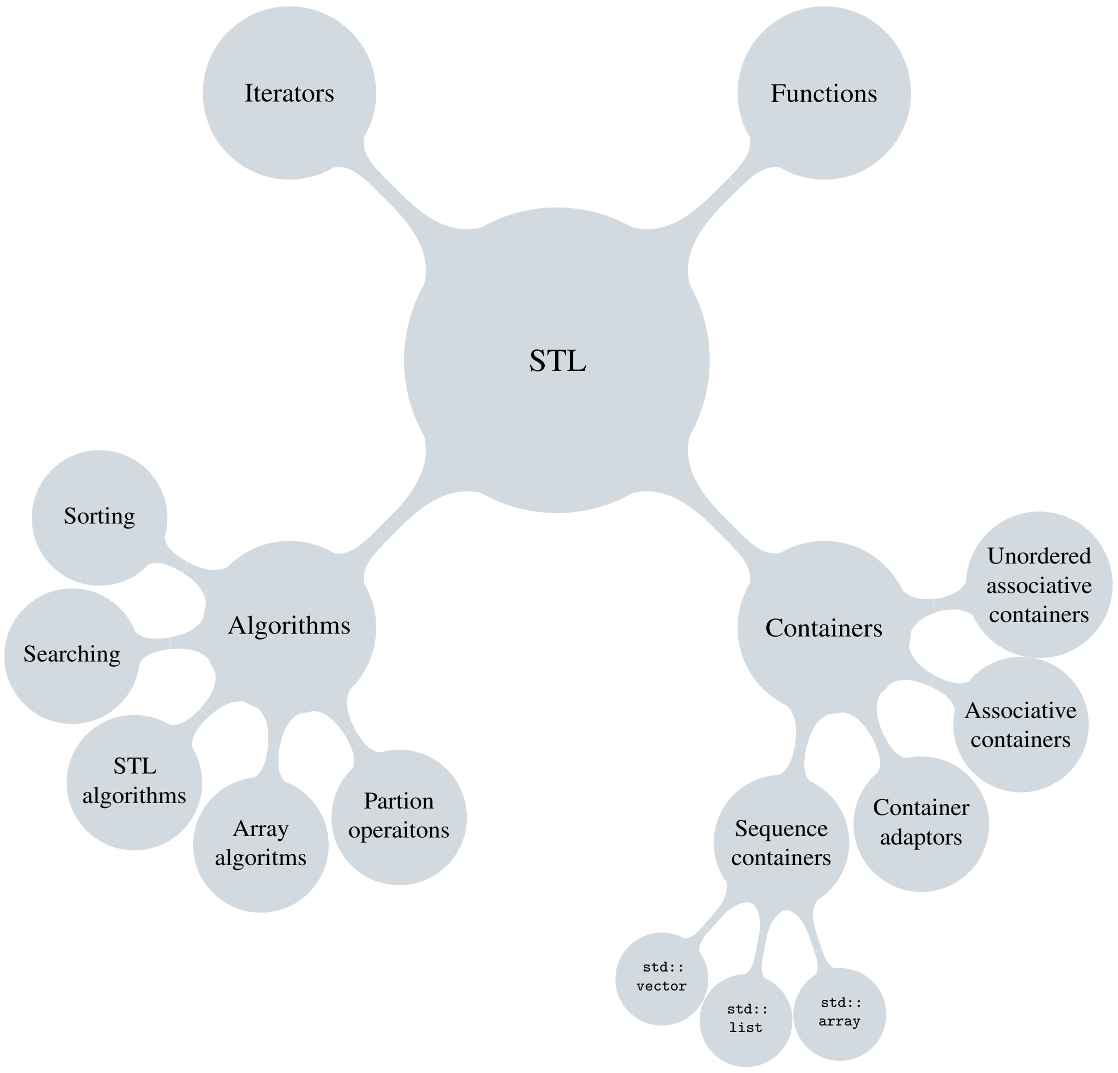


Figure 3.1: Overview of the C++ Standard Template Library (STL): Algorithms, Containers, Iterators, and Functions. This course will mainly focus on the Algorithms and Container components.

of the number from one to *n*. Listing 3.1 sketches how to compute the average using the ingredients of the previous chapter. Only one new feature `std::setprecision`[83] is a new feature provided by `#include <iomanip>` header and you should be able to understand this code. If you have any issues, we highly recommend to go back to the previous chapter and read one more time the section about loop statements, see Section 1.4.1. With `std::setprecision(3)` it is specified that only three digits of the following floating point number are printed. For example if one wants to print `const long double pi = std::acos(-1.L);` and uses `std::setprecision(3)` only 3.14 is printed. Thus, depending on the application the accuracy can be varied.

In this example multiple values are read from the standard input using `while (std::cin >> x)` in Line 9. The `while` statement reads a new value from the standard input device, stores it in the variable `x`, until the users types `\n`, which corresponds to a line break, since the loop condition is `false`. However, if we want to compute the median of a list of elements, we need to store the elements, process them, and print the average. To store these elements, we will look into the `std::vector` container and the `#include<algorithm>`

Listing 3.1: Computation of the average of the numbers from one to n.

```cpp
#include <iostream>
#include <iomanip>

int main()
{
double sum = 0;
size_t count = 0;
double x = 0;
    while (std::cin >> x)
    {
        sum += x;
        ++count;
    }
std::cout << "Average:␣"
    << std::setprecision(3)
    << sum / count << std::endl;
}
```

header. In Section 3.3 an example to compute the average is provided, since we have all the needed ingredients studied. For more details we refer to [6, Chapter 3].

3.2.1 Vector

The container `std::vector` represents an object to store an arbitrary amount of the same data types. From the mathematical point of view the `std::vector` is comparable to a vector

$$\mathbf{v} = \{v_i \,|\, i = 1, \ldots, n\} \text{ with } \mathbf{v}[i] = v_i \text{ and } |v| = n. \tag{3.2}$$

Note in C++ the elements in a vector start with index zero and the index of the last element is $n-1$ with a vector length of n. To initialize an empty vector with the name `values` the expression `std::vector<double> values;` is used. Between the parenthesis the data type of all elements of the vector is specified. In this case only `double` values can be stored in the vector. In this case the length of the vector `values.size()` will return zero and `values.empty()` will return `true` since the vector is empty with the meaning that there are not elements stored. In addition, a vector can be filled with values during its definition using `std::vector<double> v = 1, 2.5;`. In this case the length of the vector `values.size()` will return two and `values.empty()` will return `false`.

Let us write the computation of the average again using the `std::vector`. Listing 3.2 shows the new implantation of the computation of the average (Listing 3.1). In Line 7 the `std::vector` with the name `values` for storing `double` values is declared. In Line 11 with `values.push_backx` the value of `x` is inserted at the end of the vector. To replace the third element of the vector by the value 1.5 the expression `values[3]=1.4` is used. To replace the last element with zero the expression `values[values.size()-1]=0` is used. To access the elements on the i-th index the expression `values[i]` is used. The first element is accessed using `values.first()` and the last element using `values.last()`. More details about iterators are discussed in Section 3.2.4. The last element is deleted by

Listing 3.2: Computation of the average of the numbers from one to *n* using containers.

```
1  #include <iostream>
2  #include <vector>
3  #include <numeric>
4
5  int main()
6  {
7  std::vector<double> values;
8  double x;
9      while (std::cin >> x)
10     {
11         values.push_back(x);
12     }
13 double sum =
14 std::accumulate(values.begin(), values.end(), 0.0f);
15 std::cout << "Average: "
16     << sum / values.size() << std::endl;
17 }
```

using `values.pop_back()` and the *i*-th element by `values.erase(values.start()+i`.

In Line 14 the sum of all elements in the vector is computed by using `std::accumulate` from the Algorithms component. The first argument `values.begin()` and the second argument `values.end()` defines the range of the vector. Here, it is the full vector, but for example to keep out the first element of sum, one can use `values.begin()+1`. The third argument is the initial value of the sum. More details about the Algorithms will be studied in Section 3.3.

Compared to other containers, e.g. `std::list`, the `std::vector` is designed for

1. Are sufficient for small amount of elements. A good estimate is around 7000 elements,
2. Are optimized to access elements arbitrary, and
3. Performs well adding one element by the time to the end of the vector.

For example the complexity for inserting or removing an element in a vector is $\mathcal{O}(n^2)$ and for the container `std::list` the complexity is $\mathcal{O}(n)$ [61, 71].

3.2.2 List

Depending on the use case next to the `std::vector` container, the `std::list` container is available. The `std::list` container is provided by the `#include <list>` header. The usage of this container is similar to the `std::vector` and one can just replace `std::vector` by `std::list` in the code. Therefore, we will not provide any source code example here, since you can just look on them in the previous section. Compared to other containers, e.g. `std::vector`, the `std::list` is designed for

1. Are slower for small amount of elements, and
2. Are optimized to insert and delete elements anywhere.

For example the complexity for inserting or removing an element in a vector is $\mathcal{O}(n^2)$ and for the container `std::list` the complexity is $\mathcal{O}(n)$ [61, 71].

Listing 3.3: Usage of arrays using the language keyword.

```
1  //Define the length
2  size_t size = 6;
3
4  //Generate a double array of size 6
5  double array[size];
6
7  //Initializing
8  double array = {1,2,3,4,5};
9
10 //Access all elements
11 for(size_t i = 0; i < size ; i++){
12         array[i] = i*2;
13         std::cout << array[i] << std::endl;
14     }
15
16 //Access the first element
17 *array = 42;
18 std::cout << array[0] << std::endl;
```

3.2.3 Array

Another container is the `std::array` and it is provided by the `#include <array>` header[84]. Note that the array keyword is also available as a language feature[85]. The major difference is that the number of elements must be known at compile time and can not grow or shrink dynamically. Listing 3.3 shows the usage of the array as the language feature. In Line 2 the size of the array is defined using `size_t` type since the size of the array is always positive. In Line 5 the array with the name `array` is defined using `[size]` to specify its size and we use the keyword `double` to specify the type of the elements. In this case we have a array of five `double` values which are not initialized. In Line 8 the values of the array are initialized from one to five using `{1,2,3,4,5}`. Lines 11–14 show how use a `for` loop to overwrite the values assigned in Line 8 and print them to the standard output stream. In Line 17 we use the dereference operator `*` to access the first element of the array which is equivalent to `array[0]` to put the value 42 at the first position.

After looking into the usage of the array provided by as language feature, we look into the container version. The basic concepts are similar, but the container version can be used within the algorithms of the STL library which is sometimes a neat feature. Listing 3.4 shows the usage of the container version. In Line 7 the array is initialized very similar as for the language version, but since the `std::array` is provided by the C++ Standard Template Library the template specialization `<int,3>` is needed, where the first argument defines the data type and the second one the length of the array. In Line 11 the array is sorted using the `sort` method which we used to sort a `std::vector` or `std::list`. Note that is is only possible with the container version. Another nice feature is the range-based `for` loop in Lines 14–15.

Listing 3.4: Usage of arrays as containers.

```cpp
#include <algorithm>
#include <array>

int main()
{

// generate and initialize the array
std::array<int, 3> array = {1, 2, 3};

// Sort the array
std::sort(array.begin(), array.end());

// Use the range-based loop to print the elements
for(const auto& s: array)
        std::cout << s << ' ';

return 0;
}
```

3.2.4 Iterators

Iterators provided by the `\#include<iterator>`[86] header are pointing to some specific element, e.g. `std::array` or `std::vector`, and provides some fast way to iterator over all elements in the range. As the example, we use the a vector `std::vector<int> v = {1,2,3,4,5};` and to access the first element `v.begin()`[87] and to access the last element `v.end()`[88] is used. For the algorithms in the next Section, we use these for example to sort `std::sort(v.begin(),s.end(),std::greater<int>()`[89] from the largest to the lowest number. We can also use `v.next()`[90] to get the next element and `v.prev()`[91] to get the previous element.

Using iterators, we can do advanced iterating over vectors, see Listing 3.5. In Line 9 a constant iterator `std::vector<int>::const_iter` and assign the first element of the vector to it. For the `for` loop in Section 1.4.1 this would be equivalent to loop variable `size_t i = 0`. In Line 11 we use the not equal operator `iter != values.end()` as the condition statement. The equivalent for the `for` loop would be `i < vector.size()`. In 12 the manipulation statement `++iter` is used and for the `for` loop we would use `i++`.To get the content of the vector, we use the deference operator `*iter`. Note for the `for` loop we would use `values[i]`.

With the iterators erasing elements gets easier, since we can use the expression `values.erase(iter)`[92] instead of `vlaues.erase(values.begin()+i)`. Note that the `erase` function returns the iterator of the element the iterator is pointing to after the deletion `iter = vlaues.erase(iter)` which is useful for some algorithms.

Listing 3.5: Printing a vector using iterators.

```
#include <iostream>
#include <vector>
#include <iterator>

int main()
{
 std::vector<int> values = {1,2,3,4,5};
 for(
    std::vector<int>::const_iterator iter =
    values.begin();
    iter != values.end();
    ++iter
    )
    {
        std::cout << *iter << std::endl;
    }

}
```

3.3 Algorithms

In this section some of the algorithms provided by the STL are studied. For a complete list of all available algorithms we refer to[93]. The median for a sorted list of numbers $\mathbf{v} = \{v_i | i = 1,\ldots,n\}$ is given as

$$median = \begin{cases} v[\frac{n}{2}] \text{ if } n \text{ is even} \\ \frac{1}{2}\left(v[\frac{n}{2}] + v[\frac{n}{2} - 1]\right) \text{else} \end{cases} . \tag{3.3}$$

To compute the median of a `std::vector`, we have to sort the vector first. The STL provides the `std::sort` algorithm in the `#include <algorithm>` header. Listings 3.6 shows the computation of the median using the STL. In Line 6 a new feature `typedef`[94] to shorten long lines of codes is introduced. In that case we do not want to type each time `std::vector<double>:: size_type` to get the data type of the vector size and want to use `vec_sz` instead. Each time the compiler recognizes `vec_sz` it will replace it by the long form. This is a neat feature to make the code more readable.

Line 13 shows how to use sort the values stored in the `std::vector` in Line 9–12. one has to provide the range of the vector to the sort function. Note that the current values in the vector will be replaced by the sorted ones. To keep the unsorted valued, a copy of the vector can be obtained by the `std::copy`[95] algorithm.

Another example is to compute the sum of all elements of a `std::vector` using a `for` loop or using the `std::accumulate`[96] provided by the `#include <numerics>`[97] header. Listing 3.7 shows how to compute the sum and some neat algorithms. To fill a vector with the values one to ten, the function `std::ito`[98] in Line 8 is used instead of writing a for loop. In Line 12–13 the sum is computed using the loop and in Line 17 the sum computed using the STL. One can easily see that the code in Line 17 is shorter and easier

Listing 3.6: Computation of the median using the sorting algorithm provided by the STL.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main(){
typedef std::vector<double>::size_type vec_sz;
std::vector<double> values;
double x;
    while (std::cin >> x)
    {
        values.push_back(x);
    }
std::sort(values.begin(),values.end());
vec_sz mid = values.size() / 2;
double median = values.size() % 2 == 0 ?
    0.5*(values[mid]+values[mid-1]) : values[mid];
std::cout << "Median:␣"
    << median << std::endl;
}
```

to understand. Therefore, it is recommended to use the STL were possible. In line 25–29 the values of the vector are printed to the standard output stream using a `for` loop. In Line 32 instead of using the `for` loop, the expression `std::for_each`[99] provided by the `#include <algorithm>` header is used. This lien of code iterates over all elements in the vector and call the function `print` and passes each element to the function. Note that the function can have only one argument and its type has to match the type of the vector.

There are many more algorithms in the STL as shown here. These algorithms will be introduced in the reaming parts of the book, especially with the numerical examples in Chapter V. We recommend to have a look in the algorithms to write more efficient and less confusing code. For more details we refer to [6, Chapter 6].

3.4 Parallel Algorithms

Since the C++17 standard the parallel algorithm are specified. Currently, only the GNU compiler collection 9 and the MS Visual Studio compiler 19.14[100] implement this as an experimental feature. 69 of the algorithms from the `#include <algorithm>`, `#include <numeric>`, and `#include <memory>` are available[101]. Note that this is an experimental feature and following compiler flags have to be added `-std=c++1z` to use the experimental features and `-lttb` to use the Threading Building Blocks (TTB) library[102] for the parallel execution. Listing 3.8 shows one example how to compute the sum over a vector in sequential and parallel.

In Line 4 the `#include <chrono>` header[103] which is needed for time measurements. In Line 14 a timer `t1` is generated by using the expression `std::chrono::high_resolution_clock::now();`[104]. After this line of code is executed the current time is stored in the timer `t1`. In Line 16 after the line of code, we wanted to measure the execution time, a second

Listing 3.7: Example for a function definition to compute the maximum of two numbers.

```
#include <vector>
#include <iostream>
#include <numerics>
#include <algorithm>

void print(double v){
        std::cout << v << "␣";
}

int main(){

std::vector<double> values (10);
std::iota(values.begin(), values.end(), 1);

//Compute the sum using a for loop
double sum = 0;
for( auto& v : values)
        sum += v;
std::cout << "Sum:" << sum << std::endl;

//Compute the sum using STL
sum = std::accumulate((values.begin(), values.end(),0);
std::cout << "Sum:" << sum << std::endl;

//Check the result by printing the vector using a for loop
for( size_t i = 0 ; i < values.size(); i++)
        std::cout << values[i] << "␣";
        std::cout << std::endl;

}

//Check the result by printing the vector using STL
std::for_each(values.begin(), values.end(), print);
```

timer `t2` is generated with the current time after Line 15 was executed. In Line 17 the difference between the two timers is computed by the expression `std::chrono::duration <double, std::milli> ms = t2 - t1;`[105]. Wit the second argument the unit is specified and in that case `std::milli`[106] return the time difference in milliseconds. In Line 18 the expression `std::fixed`[107] restricts the number of decimal points printed.

Note that in Line 15 the expression `std::accumulate` is used to compute the sum of the elements of vector `nums` in a sequential manner. Meaning only one element is added up each time. For a large amount of elements this can be very time consuming. To make the computation of the sum more efficient, the parallel version of the algorithm can be used. Note that in the parallel algorithms the name of the algorithm is `std::reduce` [108] and not `std::accumulate`. For the parallel algorithms the function parameters are identical, however, there is one additional parameter, the so-called execution policy, which is placed in front to the function parameters. In our case the `std::execution::par`[109] execution policy. With this execution policy the code is executed using all threads of the hardware. Currently, the feature to specify the amount of threads is currently not implemented. To execute the same lien of code in a sequential manner the execution policy `std::execution::seq` is used. The header `#include <execution>` is necessary. Following execution policies are available:

- `std::execution::seq`
  The algorithm is executed sequential, like `std::accumulate` in the previous example and using only once thread.
- `std::execution::par`
  The algorithm is executed in parallel and used multiple threads.
- `std::execution::par_unseq`
  The algorithm is executed in parallel and vectorization is used.

Fore more details, we refer to the talk "The C++17 Parallel Algorithms Library and Beyond"[110] at CppCon 2016. Listing 3.9 shows how to compile the code using the experimental feature and some time measurements.

Listing 3.8: Computation of the median using the sorting algorithm provided by the STL.

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <chrono>
#include <execution>
#include <numeric>

int main(){


    std::vector<double> nums(900000000,1);

    {
    auto t1 = std::chrono::high_resolution_clock::now();
    auto result = std::accumulate(nums.begin(), nums.end(),
       0.0);
    auto t2 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms = t2 - t1;
    std::cout << "std::accumulate␣result␣" << result
              << "␣took␣" << std::fixed <<  ms.count() << "␣
                 ms\n";
    }

    {
    auto t1 = std::chrono::high_resolution_clock::now();
    auto result = std::reduce(
                      std::execution::par,
                      nums.begin(), nums.end());
    auto t2 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms = t2 - t1;
    std::cout << "std::reduce␣result␣" <<
              std::scientific  << result << "␣took␣"  << std::
                 fixed << ms.count() << "␣ms\n";
    }

    return 0;
}
```

Listing 3.9: Compilation of the parallel algorithm example.

```
g++ -std=c++1z -ltbb lecture6-loops.cpp
./a.out
std::accumulate result 9e+08 took 10370.689498 ms
std::reduce result 9.000000e+08 took 612.173647 ms
```

Notes

[75]https://www.geeksforgeeks.org/functors-in-cpp/
[76]https://en.cppreference.com/w/cpp/iterator
[77]https://en.cppreference.com/w/cpp/algorithm
[78]https://en.cppreference.com/w/cpp/algorithm/sort
[79]https://en.cppreference.com/w/cpp/algorithm/search
[80]https://en.cppreference.com/w/cpp/algorithm/max
[81]https://en.cppreference.com/w/cpp/algorithm/accumulate
[82]https://www.youtube.com/watch?v=asGZTCR53KY&list=PL7vEgTL3FalY2eBxud1wsfz8OKvE9sd_z
[83]https://en.cppreference.com/w/cpp/io/manip/setprecision
[84]https://en.cppreference.com/w/cpp/container/array
[85]https://en.cppreference.com/w/cpp/language/array
[86]https://en.cppreference.com/w/cpp/header/iterator
[87]https://www.cplusplus.com/reference/iterator/begin/
[88]https://www.cplusplus.com/reference/iterator/end/
[89]https://en.cppreference.com/w/cpp/utility/functional/greater
[90]http://www.cplusplus.com/reference/iterator/next/
[91]http://www.cplusplus.com/reference/iterator/prev/
[92]https://en.cppreference.com/w/cpp/string/basic_string/erase
[93]https://en.cppreference.com/w/cpp/algorithm
[94]https://en.cppreference.com/w/cpp/language/typedef
[95]https://en.cppreference.com/w/cpp/algorithm/copy
[96]https://en.cppreference.com/w/cpp/algorithm/accumulate
[97]https://en.cppreference.com/w/cpp/header/numeric
[98]https://en.cppreference.com/w/cpp/algorithm/iota
[99]https://en.cppreference.com/w/cpp/algorithm/for_each
[100]https://en.cppreference.com/w/cpp/compiler_support
[101]https://en.cppreference.com/w/cpp/experimental/parallelism
[102]https://github.com/oneapi-src/oneTBB
[103]https://en.cppreference.com/w/cpp/chrono
[104]https://en.cppreference.com/w/cpp/chrono/high_resolution_clock
[105]https://en.cppreference.com/w/cpp/chrono/duration
[106]https://en.cppreference.com/w/cpp/numeric/ratio/ratio
[107]https://en.cppreference.com/w/cpp/io/manip/fixed
[108]https://en.cppreference.com/w/cpp/experimental/reduce
[109]https://en.cppreference.com/w/cpp/experimental/execution_policy_tag
[110]https://www.youtube.com/watch?v=Vck6kzWjY88

II

HPX

4. Introduction to HPX

HPX (High Performance ParalleX) is a general purpose C++ runtime system for parallel and distributed applications of any scale. It strives to provide a unified programming model which transparently utilizes the available resources to achieve unprecedented levels of scalability. This library strictly adheres to the C++14 Standard and leverages the Boost C++ Libraries which makes HPX easy to use, highly optimized, and very portable. These are the most notable features of HPX:

- HPX exposes a uniform, standards-oriented API for ease of programming parallel and distributed applications.
- HPX provides unified syntax and semantics for local and remote operations.
- HPX exposes a uniform, flexible, and extendable performance counter framework [39, 40] which can enable runtime adaptivity
- HPX has been designed and developed for systems of any scale, from hand-held devices to very large scale systems (Raspberry Pi, Android, Server, up to super computers [25, 47]).

For a brief overview of HPX, we refer to [46, 57] and for a detailed overview, we refer to [45]. For more details about asynchronous many-task systems (AMT), we refer to [100].

4.0.1 Using HPX

Let us look into HPX's hello world example. We have to ways to initialize the HPX runtime system. First way is to include the header `#include <hpx/hpx_main.hpp>`, see Listing 4.1. In that case, the only thing we have to add is the new header file. Note that this header file should be the first one to be included. Before we can call the first HPX function, the HPX runtime system needs to be initialized. Second way is to include the header `#include <hpx/hpx_init.hpp>`, see Listing 4.2. In that case, the `hpx_main` function is defined in Line 4 and we place the code as we like to have in the `main` function there and use `hpx::finalize()` as the return value to make sure the HPX runtime system is stopped. To initialize the HPX runtime system, the function `hpx::init(argc, argv)`

Listing 4.1: Initializing the HPX runtime system (I).

```cpp
#include <hpx/hpx_main.hpp>
#include <iostream>

int main()
{
    std::cout << "Hello␣World!\n" << std::endl;
    return 0;
}
```

Listing 4.2: Initializing the HPX runtime system (II).

```cpp
#include <hpx/hpx_init.hpp>
#include <iostream>

int hpx_main(int, char**)
{
    // Say hello to the world!
    std::cout << "Hello␣World!\n" << std::endl;
    return hpx::finalize();
}

int main(int argc, char* argv[])
{
    return hpx::init(argc, argv);
}
```

has to be called. Note that this header file should be the first one to be included. All HPX functions have to be called within the `hpx_main` function to make sure the HPX runtime system is initialized.

Assuming that HPX is installed on the system, we need to provide some compiler and linker flags to compile the HPX application. Note that on Fedora one can install HPX by using `sudo dnf install hpx-devel` or using this tutorial[111]. Listing 4.3 shows a example CMakeLists.txt file to compile the programs shown in Listing 4.1 or Listing 4.2. For more details about CMake, we refer to Section 1.7. Listing 4.4 shows how to compile the program and run it. Note that the command line option `--hpx:threads` specifies how many CPUs HPX is allowed yo use.

Listing 4.3: Content of the CMakeLists.txt to build HPX applications.

```
cmake_minimum_required(
   VERSION 3.3.2)
project(my_hpx_project CXX)
find_package(HPX REQUIRED)
add_hpx_executable(
   my_hpx_program
    SOURCES main.cpp
)
```

Listing 4.4: Build instructions for CMake.

```
cmake .
make
./my_hpx_program --hpx:
   threads=4
```

4.1 Parallel algorithms

In Section 3.4 we looked at the experimental parallel algorithms provided by the C++ STL. HPX provides the parallel algorithms as well and the API is identical and we just need to replace the `std` name space with `hpx` name space. Recall the example in Listing 3.8 and now we implement the same example using HPX's parallel algorithms. Listing 4.5 shows how to compute the sum of the elements in the vector `values` parallel. Note that solely had to replace `std::execution::par` by HPX's name space which is a little bit different and reads as `hpx::execution::par`. The same for `std::reduce` and this name space reads as `hpx::ranges::reduce`[112]. Until now the API is equal to the one of the C++ STL. Now, we look into the additional features provided by HPX. First, we look into the additional features for execution policies. In Line 16 we specify a dynamic chunk size `dynamic_chunk_size` and pass this execution policy to the execution policy using `.with(scs)`. Following execution parameters are provided:

- `hpx::execution::static_chunk_size`[113]
  Loop iterations are divided into pieces of a given size and then assigned to threads.
- `hpx::execution::auto_chunk_size`[114]
  Pieces are determined based on the first 1% of the total loop iterations.
- `hpx::execution::dynamic_chunk_size`[115]
  Dynamically scheduled among the cores and if one core finished it gets dynamically assigned a new chunk.

For more details, we refer to [38]. Another possibility is to use machine learning techniques for choosing the chunk size. For more details, we refer to [65]. Second, in HPX once can obtain a future from a parallel for loop and us it for synchronization. In Line 23 of Listing 4.5 shows how to obtain a future with the result of the reduce operation by adding the expression `hpx::execution::task` as an argument to the execution policy. Now we can use the parallel for loops and combined them with the future for asynchronous programming. Note that currently these features are only available yet in HPX. Third, HPX provides range-based for loops[116] which is neat for iteration over the elements of a vector using the index and not the vector element itself. Listing 4.6 shows how to use a range-based parallel for loop to print the vector's element to the standard output stream. The second function argument is the first value of the vector, the third one the vector's length, and the fourth argument is a Lambda function, see Section 1.9. The first argument of the Lambda function is the index of the the vector to be processed in the range of `0` and `values.size()`.

Listing 4.5: Parallel algorithms (reduce) using HPX.

```
#include <hpx/hpx_init.hpp>
#include<hpx/include/parallel_reduce.hpp>

int main()
{

std::vector<double> values = {1,2,3,4,5,6,7,8,9};

// HPX parallel algorithms
std::cout<< hpx::ranges::reduce(hpx::execution::par,
        values.begin(),
        values.end(),
        0);

// HPX parallel algorithms using execution policies
hpx::execution::dynamic_chunk_size scs(10);
std::cout<< hpx::ranges::reduce(hpx::execution::par.with(cs),
        values.begin(),
        values.end(),
        0);

// HPX parallel algorithms returning a future
auto f = hpx::ranges::reduce(
        hpx::execution::par(hpx::execution::task),
        values.begin(),
        values.end(),
        0);

std::cout<< f.get();

return EXIT_SUCCESS;
}
```

Listing 4.6: Parallel range-based for loops using HPX.

```cpp
#include <hpx/hpx_init.hpp>
#include<vector>
#include<iostream>
#include<hpx/include/parallel_for_loop.hpp>

int main()
{

std::vector<double> values = {1,2,3,4,5,6,7,8,9};

hpx::for_loop(
        hpx::execution::par,
        0,
        values.size();
        [](boost::uint64_t i)
                {
                std::cout<< values[i] << std::endl;
                }
        );

return EXIT_SUCCESS;
}
```

4.2 Asynchronous programming

HPX provides the same features as the C++ language for asynchronous programming, see Chapter 6 for more details. In this section, we show how to use HPX's function instead of `std::future` and `std::async`, since HPX provides more flexibility here. As a disclaimer this is really easy, since we can use the code of the previous example and just replace the name space `std` with the name space `hpx`. Listing 4.7 shows an example of the example for computing the square number of a asynchronously. In Line 2 the header `#include <hpx /incldue/lcos.hpp>` is needed to use `hpx::future` and `hpx::async`[117]. In Line 12 the function `square` is called asynchronously using `hpx::async(square,10)`. Note that the first argument is the name of the function and the second one the function argument. The function call return a `hpx::future<int>` since the return type of the function is `int`. To access the result of the function, if the computation has finished the function `.get()` is used. Note that the only difference here is not to include the header `#include <future>` and use `hpx::future` instead of `std::future` and same for `hpx:async` instead of `std::async`. Thus, it is really easy to switch between HPX and C++ for asynchronous programming.

Exercise 4.1 Write the program in Listing 6.3 using `hpx::future` and `hpx::async`. ■

The benefit of using HPX is that more features for the synchronization of future is provided. In Listing 4.8 some of these functionality is shown. In Line 1 a `std::vector` holding the `hpx::future<int>` is declared. In Lines 2–3 two futures of the two asynchronous function class are pushed to the vector. In Line 6 the expression `hpx::when_all` is used to make a barrier which waits until all computations of the asynchronous launched functions are ready. By calling `.then()` we specify what is done if all futures are

Listing 4.7: Asynchronous computation of the square number using HPX.

```
#include <hpx/hpx_init.hpp>
#include <hpx/incldue/lcos.hpp>
#include <iostream>

int square(int a)
{
    return a*a;
}

int main()
{
    hpx::future<int> f1 = hpx::async(square,10);

    std::cout << f1.get() << std::endl;

    return EXIT_SUCCESS;
}
```

ready. To do so, we provide a lambda function, see Section 1.9, which has a future with the `std::vector` of futures as its argument. In Line 7 we use the function `.get()` and this future to get the `std::vector` of futures. In line 7 and Line 8, we print the results as usual. Following synchronization options[118] are available:

- `hpx::when_all`
  It AND-composes all the given futures and returns a new future containing all the given futures.
- `hpx::when_any`
  It OR-composes all the given futures and returns a new future containing all the given futures.
- `hpx::when_each`
  It AND-composes all the given futures and returns a new future containing all futures being ready.
- `hpx::when_some`
  It AND-composes all the given futures and returns a new future object representing the same list of futures after n of them finished.

4.2.1 Advanced asynchronous programming

HPX provides additional features for asynchronous programming which are not yet in the C++ standard. In this section, we look into these features on some small examples, In Section 13.2 all of them are combined to have the asynchronous implementation of one-dimensional heat equation. First, we look into one feature which will not be used for the one-dimensional heat equation, however, it is still useful to combine the parallel algorithms in Section 4.1 with asynchronous programming. This feature is shown in Line 22 of Listing 4.5. Second, we will look into the features which we will use for the asynchronous implementation of the heat equation. In some cases, for example if we initialize values at the beginning of simulation, we need a future to synchronize with the actual computation but this future is already ready since no computation is needed. Listing 13.4 shows the

Listing 4.8: Advanced synchronization of futures using HPX.

```cpp
std::vector<hpx::future<int>> futures;

futures.push_back(hpx::async(square,10);
futures.push_back(hpx::async(square,100);

hpx::when_all(futures).then([](auto&& f){
 auto futures = f.get();
 std::cout << futures[0].get()
        << "␣and␣" << futures[1].get();
});
```

Listing 4.9: Use a ready future to initialize the computational data.

```cpp
1 auto f = hpx::make_ready_future(1);
2 /*
3  * Since the future is ready the output will happen
4  * and there will be no barrier.
5  */
6 std::cout << f.get() << std::endl;
```

usage of `hpx::make_ready_future` to generate a future filled with the initial value of one. Since we used a so-called ready future the code in Line 6 will be immediately executed, since there will no barrier because the future is ready and the data is available when we call `.get()`.

HPX provides additional features for continuation of the work flow. We will look into to different ways to attach some new task once the depending futures are ready. Listing 4.10 show the first approach were the future return by `hpx::when_all` is used to specify the next depending task. In Line 2 and Line 3 the futures of the two asynchronous function calls are stored in the vector `futures` and in Line 7, we use `hpx::when_all` for synchronization as before. However, this time we use the fact that `hpx::when_all` returns a future and we can call the `.then()` function of the returned future. We pass a lambda function, see Section 1.9, to this function which contains the code which is executed once the two futures are ready. The first and only argument is the `std::vector<hpx::lcos::future<int>> futures` inside a `hpx::lcos::future<std::vector<hpx::lcos::future<int>>>`. Therefore, we have to call `f.get()` in Line 10 to access the `std::vector`. In the `for` loop, we iterate over the two futures and gather the results which will be printed in Line 14.

A more efficient way were is no need to wrap the `std::vector` into some additional future. Listing 4.11 shows the usage of `hpx::dataflow` to do exactly the same what is shown in Listing 4.10. The first argument indicates if the lambda function, see Section 1.9, will be executed synchronously `hpx::launch::sync` or asynchronously `hpx::launch::async` returning a future. As the second element the lambda function which is executed after the futures are ready is given. In the `for` loop the results are gather and finally printed.

Another important feature is the unwrapping the futures to pass their content to some function directly without calling `.get()` for all of the futures. Look at Listing 4.12 shows the function `sum` taking two integers as its arguments and print their sum on the

Listing 4.10: Usage of `hpx::when_all.then()` for the continuation of futures.

```cpp
std::vector<hpx::lcos::future<int>> futures;
futures.push_back(hpx::async(square,10));
futures.push_back(hpx::async(square,100));

// When all returns a future containing the vector
// of futures
hpx::when_all(futures).then([](auto&& f){
    // We need to unwrap this future to get
    // the content of it
    auto futures = f.get();
    int result = 0;
    for(size_t i = 0; i < futures.size();i++)
        result += futures[i].get();
    std::cout << result << std::endl;
});
```

Listing 4.11: Usage of `hpx::dataflow` for the continuation of futures.

```cpp
hpx::dataflow(hpx::launch::sync,[](auto f){
    int result = 0;
    for(size_t i = 0; i < f.size();i++)
        result += f[i].get();
    std::cout << result << std::endl;
},futures);
```

Listing 4.12: Unwrapping a function to pass futures without calling `.get()`.

```
1  void sum(int first, int second){
2
3  std:: cout << first + second << std::endl;
4
5  }
6
7  auto f1 = hpx::async(square,10);
8  auto f2 = hpx::async(square,100);
9
10 // We have to call .get() to pass
11 // the values of the future
12 sum(f1.get(),f2.get());
13
14 // We can unwrapp the function
15 auto fp = hpx::util::unwrapping(sum);
16
17 // After unwrapping, we can pass the future
18 // directly to the function
19 hpx::dataflow(hpx::launch::sync,fp,f1,f2);
```

standard output stream in Line 1. In Line 7 and Line 8 we call the function `square` asynchronously, which is not shown here and just computes the square of the argument. In Line 12 the function `sum` is called and we need to call `.get()` twice to access the content of the futures. Doing this for two futures is doable, but no really convenient. HPX provides the unwrapping of the function `sum` so the `.get()` will be called internally and we can pass the futures directly to the function. In Line 15 we use `hpx::util::unwrapping` to unwrap the function `sum` and we get some function pointer `fp` back which points to the unwrapped function. In Line 19 we can now use `hpx::dataflow` to launch the function pointer `fp` synchronously and passing the futures directly without calling the `.get()`.

4.3 Semaphores

In Section 5.1 the `std::mutex`, which is tied to one thread and only one thread can lock or unlock the mutex. Now the look into a semaphore and here any thread can access the ownership on a semaphore. Note that the C++ standard does not define semaphores and they are only available suing HPX. The concept of semaphores was introduced by E. Dijkstra [27] and more details are available here [28]. Before we look into the source code, we will focus on one example.

Imagine a public library lending books with no late fee. The library has 5 copies of the Hitckhiker's Guide to the Galaxy [4]. So the first five people can borrow these copies and keep them for an infinite amount of time, since there are no late fees. Now, if person number six wants to borrow one copy, this person has to wait until one of the five borrowers return one copy. So the library assigns one of the copies to this person, but if none is waiting the copy just goes back to the shelf until one asks for it.

This example can be explained in C++ using a semaphore. A semaphore has two

Listing 4.13: HPX's semaphores.

```
1 // Generate a semaphore with maximal count nd
2 hpx::lcos::local::sliding_semaphore sem(nd);
3
4 // Release ownership for t
5 sem.signal(t);
6
7 // Obtain ownership for t
8 sem.wait(t);
```

variables. First, a `maximum count` which is from the example the total amount of copies. Second, a `current count` which relates to the amount of currently borrowed copies. Now, we have the the so-called P-Operation and V-Operation. The P-Operation is done using the `wait` function. Here the variable `current count` is decreased. If the count is $\geq$ zero then the decrement just happens and the function will return. If the count is zero the function will wait until one other thread called the `signal` function. This is refereed to as P-Operation. If the `signal` function is called, the current count is increased. If the count was zero before you called `signal` function and another thread was blocked in `wait` then that thread will be executed. If multiple threads are waiting, only one will be executed and the reaming ones have to wait for another increment of the counter. This is refereed to as V-Operation. Listing 4.13 shows the usage of the semaphore in HPX. In Line 2 the semaphore is generated an the maximal count is passed as argument `nd`. In Line 5 the ownership of thread `t` is released using the P-Operation (`signal` function). In Line 8 the thread `t` obtains the ownership using the V-Operation (`wait` function).

4.4 Distributed programming

4.4.1 Serialization

In shared memory parallelism the allocated data resits in the memory on the node, however, in the distributed memory parallelism each of the physical nodes has its own memory. If one uses `std::vector<double>` or `double[]` this is a so–called unflatten data structure representation in C++. However, this data structure can not be wrapped in a parcel and send over the network to another physical node. Before the data structure can be wrapped in a parcel, the data needs to be flatten to a one-dimensional stream of bits. For the serialized stream of bits there is human-readable (text) and non-human-readable (binary) format possible. The advantage of the text variant is that the message is readable, but is larger. For the binary variant the message is might smaller, but can not be analyzed for debugging.

Figure 4.1 shows the protocol to send data over the network from locality 1 to locality 2. On locality 1, first the data is allocated in the local memory, for example one could allocate the vector `std::vector<double> vec = 0.0,0.5,1.0;` in the local memory of locality 1. Second, the `std::vector<double>` is serialized which means the `std::vector` is transformed in a stream of bits containing the data of the vector and some additional information, e.g. the size of elements. Third, the flattened bit of streams is wrapped into a parcel which is send over the network to locality 2. For more details for sending parcels over the network, we refer to Section 7. On the receiving locality 2, first the parcel is

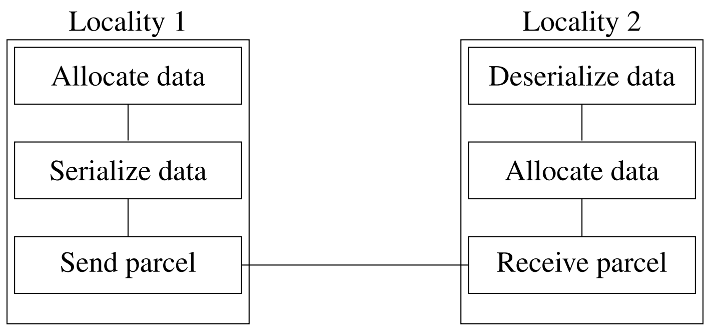


Figure 4.1: The communication between the localities (nodes) is handled by the so-called parcel port [55]. HPX uses MPI or libfrabric for communication between nodes.

Listing 4.14: Serialization in HPX.

```
1  // Allocation of the data
2  size_t size = 5;
3  double* data = new double[size];
4
5  // Serialization
6  using hpx::serialization::serialize_buffer;
7
8  serialize_buffer<double> serializable_data(
9       data, size,
10      serialize_buffer<double>::init_mode::reference);
11
12 // Deserialization
13 double* copied_data = serializable_data.data();
```

received and unpacked. Second, the data for the content of the parcel is allocated in the local memory of locality 2. Third, the flattened data from the received parcel is deserialized and stored in the local memory of locality 2.

Before, we looked into the general concept of serialization and now we look on the implementation within HPX. In Listing 4.14 the data is allocated in the first three lines. To serialize the `double* data` array, first a `hpx::serialization::serialize_buffer` us used in Line 6 is defined. In Line 8 the buffer `serialize_buffer<double>` with `double` as its template argument is used, since we intend to serialize the `double* data` array. As the arguments of the constructor, we pass the pointer to the data and the size of the data. For now we ignore the third argument and just use this mode as the default mode. This is the part of the serialization which happens on locality 1. The deserialization which would happen on locality 2 is shown in Line 12, assuming we received the `serializable_data` object on locality 2. On locality 2 a pointer `data* copied_data` is used to store the deseralized data obtained by the function `.data()`. For sending and receiving parcels, we will look into components and action, in Section 4.4.2.

4.4.2 Components and Actions

For distributed computations within HPX, we need to look following features:

Listing 4.15: Plain actions in HPX.

```
1 static void square(double a){
2
3         std::cout << a * a << std::endl;
4 }
5
6 // Register the plain action
7 HPX_PLAIN_ACTION(&square, square_action)
```

1. Components:
   The server represents the global data and is a so-called HPX component which allows to create and access the data remotely through the global address space (AGAS)[56].
2. Client:
   The client represents the local and remote access to the component's data on all local or remote localities.
3. Component action:
   Each function of the component (server) needs to be wrapped into a component action to be remotely and locally available.
4. Plain actions:
   Allows to wrap global (`static` functions in an action. So we can call this function remotely and locally.

Action

Plain actions

A plain action allows to call a `static` function locally and remotely. For a plain action, a `static` function `square` is defined, see Listing 4.15. Note that actions can have a `return` expression, but we can not change data within the action. In Line 6 the function `square` is registered as a action with the name `square_action` using the expression `HPX_PLAIN_ACTION`[119].

4.4.3 Receiving topology information

Following functions are available to receive topology information:

- `hpx::find_here`[120]
  Get the global address of the locality the function is called on.
- `hpx::find_all_localities`[121]
  Get the global addresses of all available localities.
- `hpx::find_remote_localities`[122]
  Get the global addresses of all available remote localities.
- `hpx::get_num_localities`[123]
  Get the number of all available localities.
- `hpx::find_locality`[124]
  Get the global address of any locality hosting the component.
- `hpx::get_colocation_id`[125]
  Get the locality hosting the object with the given address.

4.5 Overview of HPX headers

This section recaps some of the HPX headers and the functionality they provide. For a overview of all HPX headers, we refer to HPX's documentation [126].

- `#include <hpx/hpx_main.hpp>`
  This header includes the HPX run time systems and has to be always the first HPX header to be included. This header provides a way to initialize the HPX runtime system, see Listing 4.1. For more details, we refer to Section 4.0.1.
- `#include <hpx/hpx_init.hpp>`
  This header includes the HPX run time systems and has to be always the first HPX header to be included. This header provides a different way to initialize the HPX runtime system, see Listing 4.2. For more details, we refer to Section 4.0.1.
- `#include <hpx/include/locs.hpp>`
  This header provides for example `hpx::future` (`#include <hpx/future.hpp>`) and `hpx::async` (`#include <hpx/include/future.hpp>`) functionality. Fore more details, we refer to Section 4.2. In addition, the advanced synchronization features, see Section 4.2.1, are included in this header as well.
- `#include <hpx/algorithm.hpp>`
  This header provides the functionality of the parallel algorithms and compares to `#include <algorithm>`.
  - `#include <hpx/include/parallel_for_loop.hpp>` This header includes the method `hpx::for_loop` functionality, see Listing 4.6. Note if you intend to use multiple parallel algorithms, you could use `#include <hpx/algorithm.hpp>` which compares to `#include <algorithm>`.
  - `#include <hpx/include/parallel_reduce.hpp>`
    This header includes the method `hpx::ranges::reduce` functionality which is comparable to the `std::reduce`, see Listing 4.5. Note if you intend to use multiple parallel algorithms, you could use `#include <hpx/algorithm.hpp>` which compares to `#include <algorithm>`.
- `#include <hpx/modules/synchronization.hpp>`
  This header provides the `hpx::lcos::local::sliding_semaphore`, see Listing 4.13. Fore more details, we refer to Section 4.3.
- `#include <hpx/include/actions.hpp>`
  This header provides the functionality for actions which we need for distributed programming, see Section 4.4.2.
- `#incldue <hpx/include/components.hpp>`
  Provides the he functionality for the components which we need for the distributed programming, see Section 4.4.2.
- `#include <hpx/include/dataflow.hpp>`
  Provides `hpx::dataflow::dataflow`, see for example Listing 13.4.

Notes

[111]https://www.diehlpk.de/blog/hpx-fedora/
[112]https://hpx-docs.stellar-group.org/latest/html/libs/algorithms/api.html?highlight=reduce#_CPPv3N3hpx8parallel2v16reduceERR8ExPolicy8FwdIterB8FwdIterE1TRR1F
[113]https://hpx-docs.stellar-group.org/latest/html/libs/execution/api.html?highlight=static_chunk_size#_CPPv3N3hpx8parallel9execution17static_chunk_sizeE
[114]https://hpx-docs.stellar-group.org/latest/html/libs/execution/api.html?highlight=auto_chunk_size#_CPPv3N3hpx8parallel9execution15auto_chunk_sizeE
[115]https://hpx-docs.stellar-group.org/latest/html/libs/execution/api.html?highlight=dynamic_chunk_size#_CPPv3N3hpx8parallel9execution18dynamic_chunk_sizeE
[116]https://hpx-docs.stellar-group.org/latest/html/manual/writing_single_node_hpx_applications.html?highlight=parallel_for_loop
[117]https://stellar-group.github.io/hpx/docs/sphinx/latest/html/examples/fibonacci_local.html?highlight=async
[118]https://stellar-group.github.io/hpx/docs/sphinx/latest/html/terminology.html#term-lco
[119]https://hpx-docs.stellar-group.org/latest/html/libs/actions_base/api.html?highlight=plain_action#c.HPX_PLAIN_ACTION
[120]https://hpx-docs.stellar-group.org/latest/html/api/full_api.html?highlight=find_here#_CPPv4N3hpx9find_hereER10error_code
[121]https://hpx-docs.stellar-group.org/latest/html/api/full_api.html?highlight=find_all_localities#_CPPv4N3hpx19find_all_localitiesER10error_code
[122]https://hpx-docs.stellar-group.org/latest/html/api/full_api.html?highlight=find_remote_localities#_CPPv4N3hpx22find_remote_localitiesER10error_code
[123]https://hpx-docs.stellar-group.org/latest/html/libs/runtime_local/api.html?highlight=get_num_localities#_CPPv4N3hpx18get_num_localitiesEv
[124]https://hpx-docs.stellar-group.org/latest/html/api/full_api.html?highlight=find_locality#_CPPv4N3hpx13find_localityEN10components14component_typeER10error_code
[125]https://hpx-docs.stellar-group.org/latest/html/api/full_api.html?highlight=hpx%20get_colocation_id#_CPPv4N3hpx17get_colocation_idERKN6naming7id_typeE
[126]https://hpx-docs.stellar-group.org/latest/html/libs/include/api.html

III

Parallel and distributed computing

5. Parallel computing

In this Chapter, a brief overview of the technical aspects of parallel computing is given. Note that this course focuses on the implementation details, like asynchronous programming, see Chapter 6; parallel algorithms, see Section 3.4; and the C++ standard library for parallelism and concurrency (HPX), see Chapter II. Note that another option for parallel programming or multi-threaded programming is Open Multi-Processing[127] (OpenMP) and some more recent ones Rust[128], Go[129], and Julia language[130]. However, we provide some details and further references for the technical aspects and hardware details. For a general overview, we refer to [64]. Another option are acceleration cards like NVIDIA® or AMD® GPUs.

Let us begin with a definition of parallelism: 1) we need multiple resources which can operate at the same, 2) we have to have more than one task that can be performed at the same time, 3) we have to do multiple tasks on multiple resources the same time. First, we have to have multiple resources, e.g. multiple threads of a computation node at the same time. However, with current hardware architecture this is not an issue. Second, this part is more interesting, since we need some code which is independent of each other and can be executed concurrent. Third, here we want to have overlapping computations and communication on multiple resources. For more details about parallel computing, we refer to [36, 101].

For the second part of the definition, Amdahl's law [5] or strong scaling is important. Amdahl's law is given as

$$S = \frac{1}{(1-P)+\frac{P}{N}} \tag{5.1}$$

where S is the speed up, P the proportion of parallel code, and N the numbers of threads. Figure 5.1 plots Amdahl's law for different ratios of parallel code. Obviously, for zero percent parallel code, there is no speedup. If the portion to parallel code increases, the speedup increases up to a certain amount of threads. Therefore, the parallel computing

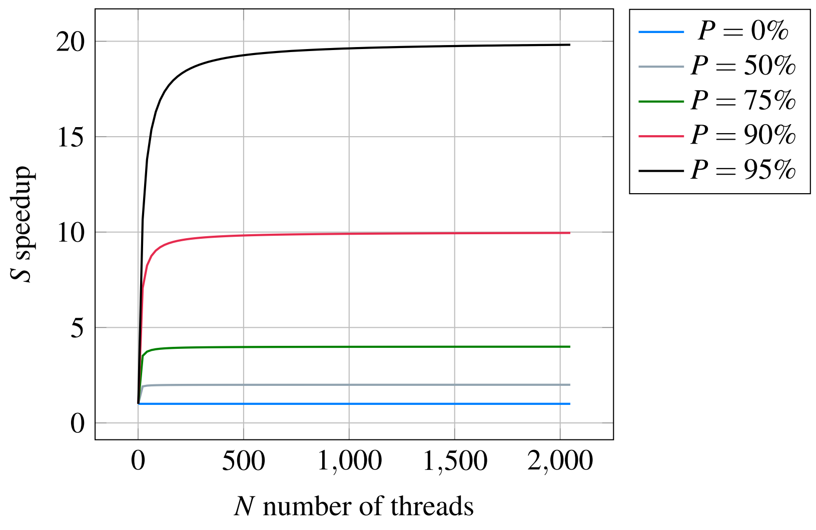


Figure 5.1: Plot of Amdahl's law for different parallel portions of the code.

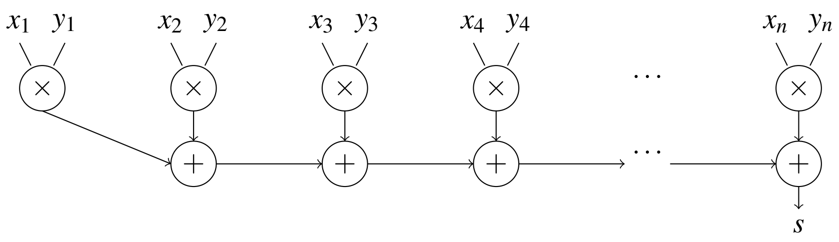
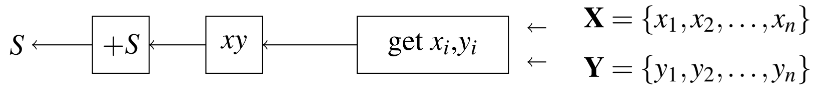


Figure 5.2: Flow chart of the sequential evaluation of the dot product of two vectors.

with many threads is only beneficial for highly parallelism in our program. For example if our code took 20 hours using a single thread to complete and there in a part of one hour which can not be executed in parallel. Thus, only 19 hours of execution time can be parallized ($p = 0.95$) and independent of the amount of threads we use the theoretical speedup is limited to $S = {}^{1}/_{(1-p)} = 20$.

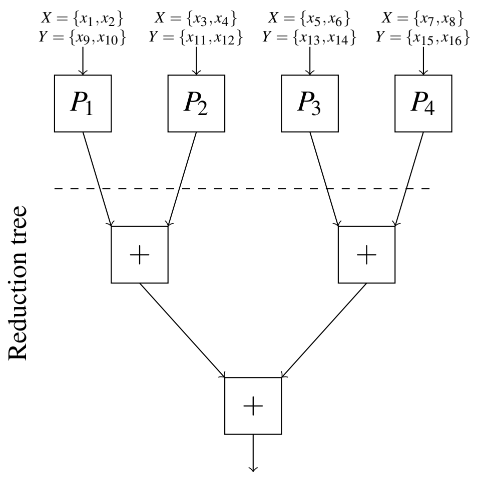
Before we look into different parallelism approaches, we look into the example how to compute the dot product $S = \mathbf{X} \cdot \mathbf{V} = \sum_i^N x_i y_i$ of two vectors $\mathbf{X} = \{x_1, x_2, \dots, x_n\}$ and $\mathbf{Y} = \{y_1, y_2, \dots, y_n\}$ in a sequential manner and extend this example to the various parallelism approaches. So we have to compute $S = (x_1 y_1) + (x_2 y_2) + \dots + (x_n y_n)$ as shown in the flow chart in Figure 5.2. In the sequential processing, the first to elements of each vector are multiplied $x_1 \times y_1$ and added to the temporal result. After that the second elements are multiplied and added to the temporal result, and so on.

The first parallelism approach is the pipeline parallelism [80]. The pipeline parallelism is used in vector processors and in execution pipelines in all general microprocessors. Let us look into some example of from the automotive industry. First, the body of the car is assembled. Second, workers assemble the chassis. Third, workers add the engine into the chassis. Next, the steering wheel is added and many more steps until the car is finally assembled. TO make this process efficient, the workers assembling the chassis do not wait until the last step is finalized before they start working on the next chassis. Side note this is similar to the assembly line introduced bu Henry Ford to enable mass production of cars [104].



S
$+S$
xy
get x_i,y_i
$\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$
$\mathbf{Y} = \{y_1, y_2, \ldots, y_n\}$

Figure 5.3: Flow chart for the pipeline processing for the dot product.



$X = \{x_1, x_2\}$ $Y = \{x_9, x_{10}\}$
$X = \{x_3, x_4\}$ $Y = \{x_{11}, x_{12}\}$
$X = \{x_5, x_6\}$ $Y = \{x_{13}, x_{14}\}$
$X = \{x_7, x_8\}$ $Y = \{x_{15}, x_{16}\}$
P_1
P_2
P_3
P_4
Reduction tree
$+$
$+$
$+$

Figure 5.4: Reduction tree for the dot product using single instructions and multiple data.

Figure 5.3 shows the data flow chart for the pipeline parallelism. In the first step, the values x_1 and y_1 are read from memory. In the second step the values are multiplied. In the last step the result of the multiplication is added to the variable S. However, the other threads do not idle until the result is computed and do a previous step if possible. Meaning if the multiplication at stage two is happening, another thread starts to get the next values. For more details, we refer to [79].

The second parallelism approach is the Single instructions and multiple data (SIMD). SIMD is part of Flynn's taxonomy, a classification of computer architectures, proposed by Michael J. Flynn in 1966 [29, 33]. Following aspects are relevant

- All perform same operation at the same time
- But may perform different operations at different times
- Each operates on separate data
- Used in accelerators on microprocessors
- Scales as long as data scales.

Figure 5.4 shows the reduction tree for the dot product computation. For this parallelism approach all threads perform the same operation at the same time. In our case all available threads multiply two values at the first level. Second one of these threads add the partial results. Until not all elements are read from the vector these steps are repeated. The last step is to accumulate all partial results and the final result is available. For example previous CUDA architectures were designed this way and introducing branching had some effect on the performance. Newer CUDA architectures perform better here and these things are explained in following talk[131].

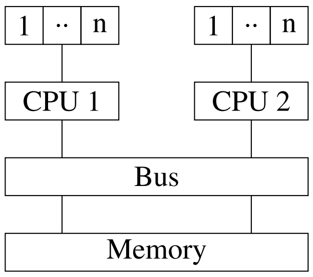


Figure 5.5: Uniform memory access (UMA)

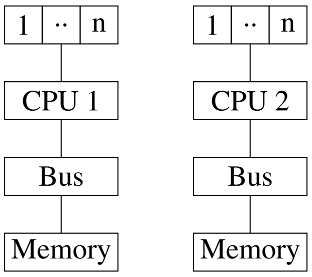


Figure 5.6: Non-uniform memory access (NUMA)

Memory access

For parallel computing, the memory access scheme is important to understand performance behavior. If we initialize for example the two vectors in the dot product example, some space in the memory is reserved and filled with the values. For the computation of the dot product these elements have to be read from memory and the CPU is doing the computation. In a layman's view the CPU is connected to the memory via a so-called bus. Depending on the bus's architecture the access time differs and may have effects on the performance if there is a switch from one CPU to the second CPU.

The first memory access scheme is uniform memory access (UMA), see Figure ref-fig:memory:uma, where all memory is attached to one bus and all CPU are attached to the same bus. Therefore, the memory access times are the same for all CPU. So we do not see any effect if we switch from one two two CPU. The second memory access scheme is non-uniform memory access (NUMA), see Figure 5.6. Here, the access time to the memory depends on the memory location relative to the CPU. Thus, local memory access is fast and non-local memory access has some overhead. For more details about memory access, we refer to [44, 81].

5.1 Caution: Data races and dead locks

Remember with great power comes great responsibility! Meaning with shared memory parallelism you add an additional source of error to your code. When using parallel execution policy, it is the programmer's responsibility to avoid

- data races
- race conditions
- deadlocks.

Let us look into some code examples for these kind of errors. A data race exists when multi-threaded (or otherwise parallel) code that would access a shared resource could do so in such a way as to cause unexpected results. Listing 5.1 shows an example for a data race for the variable `sum`. Since the parallel execution policy is used, multiple threads can access the variable `sum` at the same time which means that not all threads can write to the variable. Thus, the result is might not correct. There are two solutions to avoid the data race. First, the atomic library[132]. The atomic library[1] provides components for fine-grained atomic operations allowing for lockless concurrent programming. Each atomic operation is indivisible with regards to any other atomic operation that involves the same object. Atomic objects are free of data races. Listing 5.2 shows the solution

[1] https://en.cppreference.com/w/cpp/atomic

Listing 5.1: Example code and Solution for a data race.

```cpp
//Compute the sum of the array a in parallel
int a[] = {0,1,2,3,4};
int sum = 0;
std::for_each(std::execution::par,
              std::begin(a),
              std::end(a), [&](int i) {
  sum += a[i]; // Error: Data race
});
```

Listing 5.2: Solution to avoid the data race using `std::atomic`.

```cpp
//Compute the sum of the array a in parallel
int a[] = {0,1};
std::atomic<int> sum{0};
std::for_each(std::execution::par,
              std::begin(a),
              std::end(a), [&](int i) {
  sum += a[i];
});
```

using `std::atomic:<int>`[133]. The second solution is shown in Listing 5.3. Here, the `std::mutex` class is used to avoid the data race. The mutex class[134] is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads. In Line 4 a `std::mutex m;` is generated. In Line 8 the lock of the code is started by using `m.lock();` and in Line 10 the lock is released by using `m.unlock();`.

Exercise 5.1 Give a definition for `std::atomic` and `std::mutex` in your own words. ■

Another source of error is the race condition where a check of a shared variable within a parallel execution and another thread could change this variable before it is used. Listing 5.4 shows the solution to avoid the race condition. Imagine the code without the `std::mutex` and the implication to get a wrong result. In the code there is a check if `x` is equal to 5 and a special treatment of the computation in this case. Now in Line 4 it was true that `x` was equal to five and the thread enters the `if` branch. However, in between another thread could change the value of `x` and not `y = 5 *2` is computed. By using the mutex this situation is avoided.

Exercise 5.2 Explain a data race in your own words and explain why a `std::mutex` avoids the data race. ■

A deadlock describes a situation where two or more threads are blocked forever and waiting for each others. Following example taken from[135] explains a deadlock nicely.

Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time.

Listing 5.3: Solution to avoid the data race using `std::mutex`.

```cpp
//Compute the sum of the array a in parallel
int a[] = {0,1};
int sum = 0;
std::mutex m;
std::for_each(std::execution::par,
              std::begin(a),
              std::end(a), [&](int i) {
  m.lock();
  sum += a[i];
  m.unlock();
});
```

Listing 5.4: Example for the race condition.

```cpp
std::mutex m;

m.lock();
if (x == 5)  // Checking x
{
   // Different thread could change x

   y = x * 2; // Using x
}
m.unlock();
// Now it is sure that y will be 10
```

Exercise 5.3 The implementation of this examples is available on GitHub[136]. Play around with the example and try to understand why the code results in a deadlock. ■

6. Asynchronous programming

A different concept for shared memory parallelism is asynchronous programming [105]. Before we look into asynchronous programming, we look again into the concept of serial programming. Figure 6.1 shows the dependency graph for one computation and one can see that we can compute P and X independent and only H depends on both of them. Listing 6.1 shows the serial computation of the dependency graph. Each line of code is executed line by line Each time a function is called the code waits until the function finishes. Thus, we can not compute P and X independently, even if the data is independent.

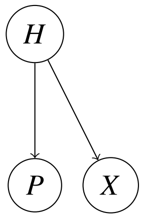


Figure 6.1: Example dependency graph

Listing 6.1: Synchronous execution of the dependency graph.

```cpp
auto P = compute();
auto X = compute();
auto H = compute(P,X);
```

To executed lines asynchronously the C++ language provides the `std::async`[137] expression provided by the `#include <future>`. Listing 6.2 shows the asynchronous implementation of the dependency graph in Figure 6.1. Line 2 shows the usage of `std::async` for the function `compute`. The first argument is the name of the function or a lambda expression, see Section 1.9. Because we used `std::async` this line of code is executed in the background on a different thread and the next line of code is executed, even if the result of the computation is not ready yet. Therefore, `std::async` return a `std::future<int>`[138] object provided by the `#include <future>` header which is a template and contains the return type of the function which is in this example the `int` data type. In Line 4, the computation of X is started on another thread. Such that both computations happens at the same time. In Line 7–9 the results of the asynchronous function call are gathered, since these are needed to compute H. With the `.get()` function a barrier is introduced and the line of codes waits until the computation is ready. In our case, we can wait since

Listing 6.2: Asynchronous execution of the dependency graph.

```
1  // Compute P
2  std::future<int> f1 = std::async(compute);
3  // Compute X
4  auto f2 = std::async(compute);
5
6  // Gather the results
7  int P = f1.get();
8  int X = f2.get();
9
10 // Compute the dependent result H
11 std::cout << compute(P,X) << std::endl;
```

we need the two results to compute the last one. Meaning that Line 8 is only executed if the computation in Line 2 has finished. Following synchronization features are available:

- `.get()` returns the result of the functions and wait until the computation finished
- `.wait()`[139] waits until the computation finished
- `.wait_for(std::chrono::seconds(1))`[140] returns if it is not available for the specified timeout duration
- `.wait_until(std::chrono::seconds(1))`[141] waits for a result to become available. It blocks until specified timeout time has been reached or the result becomes available, whichever comes first.

Example

Let us look into one example to show the parallelism using asynchronous programming for the Taylor series. The approximation of the sin function is given as

$$\sin(x) = \sum_{n=0}^{N} (-1)^{n-1} \frac{x^{2n}}{(2n)!} \tag{6.1}$$

One approach to parallize the above function using two threads is:

1. Split n into slices, e.g. 2 times $n/2$ for two threads
2. Start two times `std::async` where each thread computes $n/2$
3. Use the two futures to synchronize the results
4. Combine the two futures to obtain the result.

To distribute n into slices, we need to write the sum in Equation (6.1) as

$$\sum_{n=begin}^{end} (-1)^{n-1} \frac{x^{2n}}{(2n)!}. \tag{6.2}$$

Listing 6.3 shows how to implement the function to splice the computation of the Taylor series, see Line 5–14. In Line 18–19 the two splices $n/2$ are launched from 0 up to 49 on the first thread and from 50 up to 99 on the second thread. In Line 22 the result is gathered and finally the accumulated result is evaluated. For more details, we refer to following talk[142].

To compile the code using asynchronous programming, we need to add `-pthread` to our compiler to use the POSIX threads to launch the functions asynchronous (`std::async`). More details about POSIX threads [16, 60].

Listing 6.3: Asynchronous computation of the sin function using a Taylor series.

```cpp
#include <future>
#include <iostream>

// Function to compute portion of the Taylor series
double taylor(size_t begin, size_t end,
double x,size_t n){
double res = 0;

        for( size_t i = begin ; i < end ; i++)
        {
          res += pow(-1,i-1) * pow(x,2*n) / factorial(2*n);
        }
return res;
}

int main(){
        // Asynchronous computation using two slices
        auto f1 = std::async(taylor,0,49,2,100);
        auto f2 = std::async(taylor,50,99,2,100);

        // Gather the result
        double result = f1.get() + f2.get();

        // Print the result
        std::cout << "sin(2)= " res << std::endl;

        return 0;
}
```

7. Distributed Programming

Previously, we considered shared memory parallelism which means we only considered one physical computational node. Now, we will look into distributed programming because the memory or the computational resources of one physical computational node are not sufficient. A good definition for distributed computing is given in [102]: "A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another from any system". Fore more details about distributed systems, we refer to [102]. Figure 7.1 sketches the components of a distributed system. We have multiple computational nodes which are connected to a router or switch and they send messages to each other over the network. In that specific example, we have two nodes connected to one router. For the network connection Ethernet or more efficient Mellanox® InfiBand™ is used. A common standard to send and receive messages is the Message Passing Interface (MPI) standard. Here, we have the similar definition that the MPI standard specifies the API and several implementations, e.g. OpenMPI[143] or mpich2[144], are available. Next to these open source implementations there are commercial implementations, e.g. IntelMPI and IBM Spectrum-MPI available. For more details about MPI programming, we refer to [37]. Listing 7.1 shows some small example for sending messages and receiving messages. In Line 11 the MPI environment is initialized. In Line 12 the rank or if of the node where the code is executed is determined.

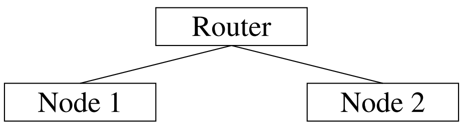


Figure 7.1: Sketch of the components of a distributed system. We have multiple computational nodes which are connected to a router or switch and they send messages to each other over the network. In that specific example, we have two nodes connected to one router. For the network connection Ethernet or more efficient Mellanox® InfiBand™ is used. A common standard to send and receive messages is the Message Passing Interface (MPI) standard.

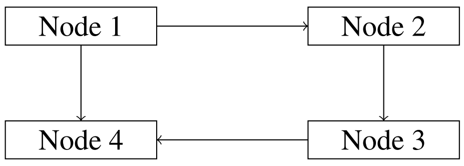


Figure 7.2: Example for some circular dependency which might result in some deadlock depending on how the messages are send and received.

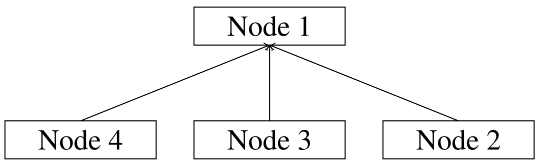


Figure 7.3: A common principle is to use supervisor node the node with rank zero and this node is used to control messages with all nodes.

Some common convention is that node with rank zero is the head node and does the synchronization. In Line 15 the head node waits for receiving some message and in Line 20 the other node is sending some message to the head node. Note that the MPI library has only a C interface and not C++ interface is available yet. The intention of this example was to show you how to use the low-level API of the MPI library and later we will see in Section 4.4 that HPX provides some abstraction layer to send and receive messages. One common term in high performance computing is `MPI+X` which means that the MPI is used to send and receive messages over the network and `X` is used for the parallelism on each of the computational nodes. For example OpenMP is used for shared memory parallelism for CPUs and CUDA™ or HIP™[145] are used for NVIDIA™ or AMD™ acceleration cards, respectively. Fore more details about CUDA™[146] programming, we refer to [85]. However, for the `MPI+X` approach the programmer has to deal with different API for the distributed and shared memory approach. Furthermore, for heterogeneous systems the programmer has to deal with a third API for the acceleration cards. Sometimes the different APIs result in duplicated code since one would need to implement the same piece of code using OpenMP and CUDA for example. Fore more details on how to overcome these issue, we refer to [8]. One attempt to provide a unified API for various shared parallelism options is kokkos [18]. Some alternative framework is libfabric [41] which was integrated within HPX. We could show that using synchronous communication suing MPI was up to a factor of three slower than asynchronous communication using libfabric [25].

7.1 Caution: Deadlocks

Same as for shared memory parallelism, we have to be careful with dead locks where two or more nodes exchanging messages and are blocked forever and waiting for each other. Some potential conditions for deadlocks are mutual exclusion, hold and wait, or circular wait while sending and receiving messages. To avoid deadlocks there should no be some cycles in the resource dependency graph, see Figure 7.2. A common thing is to use the computational node with rank zero as the head node and use it for synchronization and avoid some circular dependency, see Figure 7.3. For some good definitions of deadlocks, we refer to [14].

Listing 7.1: Small Message Passing Interface example to send and receive messages.

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int myrank, message_size=50, tag=42;
    char message[message_size];
    MPI_Status status;

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

    if (myrank == 0) {
        MPI_Recv(message, message_size, MPI_CHAR, 1, tag,
           MPI_COMM_WORLD, &status);
        printf("received␣\"%s\"\n", message);
    }
    else {
        strcpy(message, "Hello,␣there");
        MPI_Send(message, strlen(message)+1, MPI_CHAR, 0, tag
            , MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Notes

[127]https://www.openmp.org/
[128]https://www.rust-lang.org/
[129]https://golang.org/
[130]https://julialang.org/
[131]https://youtu.be/5vr7ItjyIH8
[132]https://en.cppreference.com/w/cpp/atomic/atomic
[133]https://en.cppreference.com/w/cpp/atomic/atomic
[134]https://en.cppreference.com/w/cpp/thread/mutex
[135]https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html
[136]https://github.com/diehlpkteaching/ParallelComputationMathExamples/blob/master/chapter10/lecture6-deadlock.cpp.ipynb
[137]https://en.cppreference.com/w/cpp/thread/async
[138]https://en.cppreference.com/w/cpp/thread/future
[139]https://en.cppreference.com/w/cpp/thread/future/wait
[140]https://en.cppreference.com/w/cpp/thread/future/wait_for
[141]https://en.cppreference.com/w/cpp/thread/future/wait_until
[142]https://www.youtube.com/watch?v=js-e8xAMd1s
[143]https://www.open-mpi.org/
[144]https://www.mpich.org/
[145]https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Compiler-SDK.html
[146]https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

IV

Linear algebra and solvers

8. Linear algebra

For the topic of linear algebra, we refer to [**hefferonlinear**, 87]. We focus on vector and matrices and some operations on them, as we need them for example in the finite element method. Note that we look from the computer science perspective on matrices and vectors and focus how efficiently use existing libraries in our code. Several highly optimized C++ linear algebra libraries [42, 84, 86, 103] are available. However, the look into the Blaze library since this library has a HPX backend for parallel computations.

8.1 Blaze library

Blaze is an open-source, high-performance C++ math library for dense and sparse arithmetic. With its state-of-the-art Smart Expression Template implementation Blaze combines the elegance and ease of use of a domain-specific language with HPC-grade performance, making it one of the most intuitive and fastest C++ math libraries available. More details about the implementation details [51, 52].

8.1.1 Vectors

A n dimensional vector space (or linear space) $\mathbf{u}$ is defined as $\mathbf{u} = (u_1, u_2, \ldots, u_{n-1}, u_{n-2}) \in \mathbb{R}^n$. In Blaze, a three dimensional vector[147] is defined as `blaze::DynamicVector<int> c (3 UL);`[148]. Note the Blaze is a template based library as the STL and we have to provide the data type of the vector in the parenthesizes `<int>` and in the second parenthesizes `(3UL)` the size of the vector. as for the `std::vector` we can get the size of the vector by the expression `c.size()` and to access a value, we use `auto value = c[i];`. Listing 8.1 shows how to iterate over a Blaze vector using the access operator `c[i]` and iterators `it-value`. For more details on iterators we refer to Section 3.2.4.

For the three dimensional vector space, we look into so some common operations which are often needed in simulations, e.g. in N-body simulation (Section 11) or peridyanmic simulation (Section 12). For a vector $\mathbf{u} = (x, y, z) \in \mathbb{R}^3$ the norm or length of the vector

Listing 8.1: Iterating over a Blaze vector using a for loop with iterators.

```cpp
#include <blaze/Math.h>

int main()
{
blaze::StaticVector<int,3UL> c{ 4, -2, 5 };

// Loop over the vector
for( size_t i=0UL; i< c.size(); ++i )
   std::cout << c[i] << std::endl;

// Iterate over a vector
blaze::CompressedVector<int> d{ 0, 2, 0, 0};
for( CompressedVector<int>::Iterator it=d.begin();
  it!=d.end(); ++it )
    std::cout << it->value() << std::endl;

}
```

reads as

$$|\mathbf{u}| = \sqrt{x^2+y^2+z^2} \tag{8.1}$$

and its direction is given as $\mathbf{u}/|\mathbf{u}|$. The norm of the vector $|c|$ is computed in Blaze using the expression `const double norm = norm( b );`[149]. The inner product $\bullet$ reads as

$$\mathbf{u}_1 \bullet \mathbf{u}_2 = x_1x_2 + y_1y_2 + z_1z_2. \tag{8.2}$$

Figure 8.1a shows the angle Θ between the two vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ defined using the inner product $\bullet$. The inner product $\mathbf{u}_1 \bullet \mathbf{u}_2$ is computed in Blaze using the expression `int result2 = inner(v1,v2);`[150]. The cross product $\times$ is defined by

$$\mathbf{u}_1 \times \mathbf{u}_2 = |\mathbf{u}_1||\mathbf{u}_2|sin(\theta)\mathbf{n} \tag{8.3}$$

and its geometric interpretation is sketches in Figure 8.1b. The cross product of the two vector is the orthogonal vector on the two vectors. In addition, the norm of the cross product $|\mathbf{u}_1 \times \mathbf{u}_2|$ is the are spanned by the two vectors. A more accessible form is

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} y_2z_2 - z_1x_2 \\ z_1x_2 - x_1z_2 \\ x_1y_2 - y_1x_2 \end{pmatrix}. \tag{8.4}$$

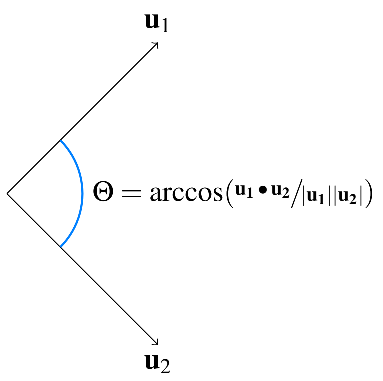
The inner product $\mathbf{u}_1 \times \mathbf{u}_2$ is computed in Blaze using the expression `cross(u1,u2);`.

8.1.2 Matrices

A matrix $\mathbf{A} \in \mathbb{R}^{n,m}$ has n rows and m columns

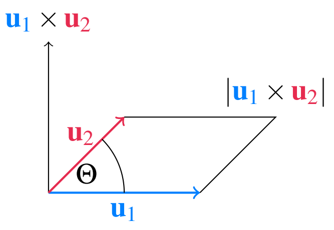
$$\mathbf{A} = \begin{pmatrix} a_{1,1} & \dots & a_{1,m} \\ \vdots & \dots & \vdots \\ a_{n,1} & \dots & a_{n,m} \end{pmatrix} \tag{8.5}$$

and following matrix operations are defined:



$\mathbf{u}_1$
$\Theta = \arccos(\mathbf{u}_1 \bullet \mathbf{u}_2 / |\mathbf{u}_1||\mathbf{u}_2|)$
$\mathbf{u}_2$

(a) The angle Θ between the two vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ defined using the inner product $\bullet$.



$\mathbf{u}_1 \times \mathbf{u}_2$
$|\mathbf{u}_1 \times \mathbf{u}_2|$
$\mathbf{u}_2$
Θ
$\mathbf{u}_1$

(b) Visualization of the inner product $|\mathbf{u}_1 \times \mathbf{u}_2|$ which is the orthogonal vector on the two others.

Figure 8.1: Geometric interpretation of the inner product $\bullet$ (a) and the cross product $\times$ (b).

- Scaling:

$$2\mathbf{A} = \begin{pmatrix} 2a_{1,1} & \ldots & 2a_{1,m} \\ \vdots & \ldots & \vdots \\ 2a_{n,1} & \ldots & 2a_{n,m} \end{pmatrix} \tag{8.6}$$

- Addition:

$$\mathbf{A}+\mathbf{B} = \begin{pmatrix} a_{1,1}+b_{1,1} & \ldots & a_{1,m}+b_{1,m} \\ \vdots & \ldots & \vdots \\ a_{n,1}+b_{n,1} & \ldots & a_{n,m}+b_{n,m} \end{pmatrix} \tag{8.7}$$

- Matrix vector multiplication

$$\mathbf{Av} = \begin{pmatrix} a_{1,1} * b_1 + & \ldots & +a_{1,m} * b_n \\ \vdots & \ldots & \vdots \\ a_{n,1} * b_1 + & \ldots & +a_{n,m} * b_n \end{pmatrix}. \tag{8.8}$$

let us look what kind of matrices are provided by Blaze and how to use them for calculations. Let us start with Blaze's matrix types[151], see Listing 8.2. The first type is the `DynanmicMatrix<T>`[152] which is a arbitrary sized matrix with dynamically allocated elements of arbitrary type `T`. Note that Blaze is a template-based library and the template type is provided within the first braces. For more details for C++ templates, we refer to 1.8. In the second pair of braces, the dimension of the n and m are given. Note that the values are not initialized of this matrix which means that the values can have any value. For large matrices the `DynanmicMatrix<T>` matrix is the best option, especially if the dimensions are not known at compile time. If the matrix is small and the dimensions are known at compile time, a `blaze::StaticMatrix`[153] matrix should be used. In Line 7 we define the 3×4 matrix, but do not allocate the memory yet and the matrix has zero rows and columns. Only after calling the constructor the memory is allocated. Note that the dimensions of the matrix are provided as template arguments in that case. All matrices are default row-major matrices and to switch to column-major matrices, the template argument `blaze::columnMajor` is available. The last matrix type is the `blaze::CompressedMatrix` which is used for sparse matrices[154] with only few non-zero entries.

Listing 8.2: Blaze matrix types.

```
// Definition of a 3x4 matrix
// Values are not initialized
blaze::DynamicMatrix<int> A( 3UL, 4UL );

// Definition of a 3x4 matrix
// with 0 rows and columns
blaze::StaticMatrix<int,3UL,4UL> A;

// Definition of column-major matrix
// with 0 rows and columns
blaze::DynamicMatrix<double,blaze::columnMajor> C;

// Definition of a 3x4 integral row-major matrix
blaze::CompressedMatrix<int> A( 3UL, 4UL );

// Definition of a 3x3 identity matrix
blaze::IdentityMatrix<int> A( 3UL );

// Definition of a 3x5 zero matrix
blaze::ZeroMatrix<int> A( 3UL, 5UL );
```

These are the main types of matrices provided by the Blaze library. However, there are some special purpose matrices which are often needed available. One is the identity matrix `blaze::IdentityMatrix` with has ones on all diagonal entries and is zero everywhere else. To have a matrix with zero valued elements, the `blaze::ZeroMatrix` is used.

For all matrices the size of the matrix `size( A );` returns the total amount of elements ($n \times m$). The number of rows are obtained by `M2.rows();` and the number of columns are obtained by `M2.columns();`. All matrix operations[155] are applied as `abs( A );` which means the absolute value of all matrix elements is computed. Note that the elements of a Blaze matrix are accessed using different kind of parentheses `A(0,0) = 1;` sets the first element of the matrix to one.

One often used task in linear algebra is decomposition of matrices. Blaze implements following decomposition methods: Cholesky [19], QR/RQ, and QL/LQ. Listing 8.3 shows how to use LU [15] decomposition method. Note that we do not cover this methods in this course, but it is an important feature you should know. For more details we refer for example to [78].

Another important feature is the computation of Eigenvalues and Eigenvectors which is shown in Listing 8.4. Note that we do not cover this methods in this course, but it is an important feature you should know. For more details we refer for example to [78].

Application

One application of matrices is communication between a group of people $P_1,\ldots,P_4$. Figure 8.2 shows the communication network of these four people as a directed graph. For example P_1 communications with P_2 and P_4. One question one can ask, is how long does it take to transfer a message from P_3 to P_2. To obtain this information, we can use a adjacency matrix [11] as in Equation (8.9) where a matrix element $a_{1,2} = 1$ means that

Listing 8.3: Matrix decomposition methods in Blaze

```cpp
blaze::DynamicMatrix<double,blaze::rowMajor> A;
// ... Resizing and initialization

blaze::DynamicMatrix<double,blaze::rowMajor> L, U, P;

// LU decomposition of a row-major matrix
lu( A, L, U, P );

assert( A == L * U * P );
```

Listing 8.4: Matrix decomposition methods in Blaze

```cpp

// The symmetric matrix A
SymmetricMatrix< DynamicMatrix<double,rowMajor>>
    A( 5UL, 5UL );
// ... Initialization

// The vector for the real eigenvalues
DynamicVector<double,columnVector> w( 5UL );
// The matrix for the left eigenvectors
DynamicMatrix<double,rowMajor>     V( 5UL, 5UL );

eigen( A, w, V );
```

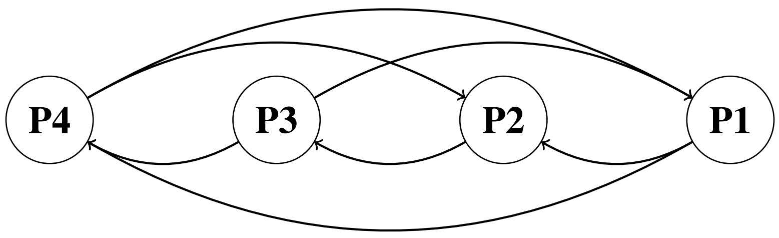


Figure 8.2: Graph of the communication network.

there is an edge in the graph from P_1 to P_2. By doing this for all people in our group, we will get this matrix. This matrix will tell us that P_1 has contact with P_2 and P_4, P_2 with P_3 and so on.

$$\mathbf{M} = \begin{Bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{Bmatrix} \tag{8.9}$$

To compute how knows the message after four cycles, we define

$$\mathbf{M}^4 = \mathbf{M} \cdot \mathbf{M} \cdot \mathbf{M} \cdot \mathbf{M},$$

which means for $\mathbf{M}^n$, we have to do n multiplications of $\mathbf{M}$. After the multiplications, we get following result

$$M^2 = \begin{Bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 2 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{Bmatrix}$$

and see that person P_3 can send some message to Person P_2 in two cycles. For more applications, we refer to [87].

Exercise 8.1 Transfer the matrix in Equation 8.9 into a Blaze matrix and try to reproduce the resulting matrix by multiplying the matrix four times. ■

8.2 Compiling code using Blaze

To use Blaze, we have to first install the library on our system[156]. Listing 8.5 shows how to install Blaze using CMake and Listing 8.6 how to install Blaze manually. Note that you should check if there is a newer version of Blaze available.

Listing 8.5: Installing Blaze using CMake.

```
tar -xvf blaze-3.6.tar.gz
cd blaze-3.6
cmake -DCMAKE_INSTALL_PREFIX=/home/patrick/blaze .
make install
```

Listing 8.6: Installing Blaze manually.

```
tar -xvf blaze-3.6.tar.gz
cd blaze-3.6
cp -r ./blaze /home/patrick/blaze
```

After installing Blaze, we can use preferable CMake, see Listing 8.7, or compile the code by hand, see Listing 8.8. For more details about CMake, we refer to Section 1.7. Note that we have already installed Blaze on the server and there is no need to install Blaze on your own device.

Listing 8.7: Compilation using CMake.

```
find_package( blaze )
if( blaze_FOUND )
   add_library(
      blaze_target
      INTERFACE )
   target_link_libraries(
      blaze_target
        INTERFACE blaze::
           blaze )
endif()
```

Listing 8.8: Manually compilation.

```
g++ -I/home/diehlpk/blaze
   BlazeTest.cpp
```

Currently, we only have compiled Blaze for serial execution. To compile Blaze with C++ 11 threads, we have to add following arguments `-std=c++11 -DBLAZE_USE_CPP_THREADS` to the compiler and export following environment variable `export BLAZE_NUM_THREADS =4 // Unix systems`. For HPX parallelism, we have to add following arguments `-DBLAZE_USE_HPX_THREADS` to the compiler and run `./a.out --hpx:threads=4` to use four threads. Fore more details, we refer to [157].

9. Solvers

Another important task in applied mathematics is to solve linear equations systems. Before we dig into the numerical and implementation details, we look into one example we know from our school lessons in mathematics. Figure 9.1 plots the two functions $f_1(x_1) = -{}^{3}\!/_{2}x_1 + 1$ and $f_2(x_1) = -{}^{2}\!/_{6}x_1 - {}^{8}\!/_{6}$. From a visual perspective one can see that the intersection of these two functions is at $(2, -2)$. However, for more complex functions or more degree of freedoms the visual approach can get cumbersome. Another approach is to formulate the corresponding linear equations systems and solve it to get the intersections. For the linear equation system, we want to have the both functions in the form $3x_1 + 2x_2 = 2$ and $2X_1 + 6x_2 = -8$ which are just a different way to write $f_1(x_1)$ and $f_2(x_1)$. Now we want to define a matrix $\mathbf{M}$ and the right-hand side $\mathbf{b}$ to find the solution $\mathbf{x}$ as $\mathbf{Mx} = \mathbf{b}$. Using the second form the function representation, we get

$$\mathbf{Mx} = \mathbf{b} \tag{9.1}$$

$$\begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ -8 \end{pmatrix}. \tag{9.2}$$

We know from school how to solve the matrix using Gaussian elimination [13]. In Equation (9.3) the first line is multiplied by two and the second line by three to get the same factor in the first column. In Equation (9.4) we can subtract the first line from the second line to get a zero in the second line. In Equation (9.5) we now can get the value for x_2 because we know $2x_2 = -28 \to x_2 = -2$. In Equation (9.6) the first line is multiplied by seven and the second line by two to get the same factor in the second column. In Equation (9.7) the second row is subtracted from the first one. In Equation (9.8) we now can get the value for x_1 because we know $42x_1 = 84 \to x_2 = 2$. Which is the same solution as visual obtained in Figure 9.1.

Note that one can implement the Gauss elimination, but the theoretical complexity of this algorithm is $\mathcal{O}(n^3)$ where n is the number of unknowns [32]. So this algorithm is feasible for thousands of unknown, but might not scale for millions of unknowns. Fore

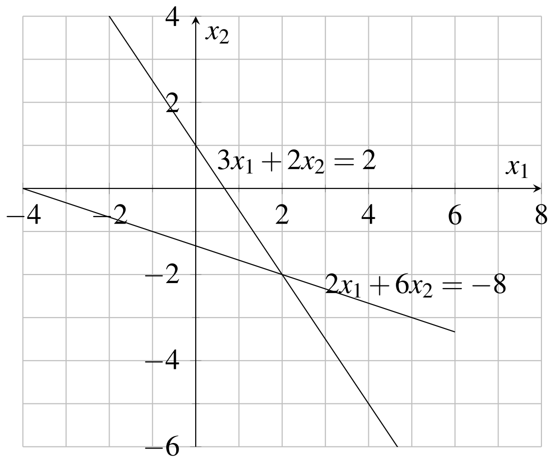


Figure 9.1: Plots of the function f_1 and f_2 to visually obtain the intersection of the the two lines.

more details about the complexity, we refer to [31]. In that case the so-called iterative methods are used. We will look into the Conjugate Gradient method in the next section. For more details about iterative methods we refer to [12, 76].

$$\begin{pmatrix}3 & 2\\2 & 6\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix}=\begin{pmatrix}2\\-8\end{pmatrix}\quad\left|\begin{matrix}\cdot 2\\\cdot 3\end{matrix}\right. \tag{9.3}$$

$$\begin{pmatrix}6 & 4\\6 & 18\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix}=\begin{pmatrix}4\\-24\end{pmatrix}\quad\left|\begin{matrix}\\-R1\end{matrix}\right. \tag{9.4}$$

$$\begin{pmatrix}6 & 4\\0 & 14\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix}=\begin{pmatrix}4\\-28\end{pmatrix}\quad\left|\begin{matrix}\\\to x_2=-2\end{matrix}\right. \tag{9.5}$$

$$\begin{pmatrix}6 & 4\\0 & 14\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix}=\begin{pmatrix}4\\-28\end{pmatrix}\quad\left|\begin{matrix}\cdot 7\\\cdot 2\end{matrix}\right. \tag{9.6}$$

$$\begin{pmatrix}42 & 28\\0 & 28\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix}=\begin{pmatrix}28\\-56\end{pmatrix}\quad\left|\begin{matrix}-R2\\\ \end{matrix}\right. \tag{9.7}$$

$$\begin{pmatrix}42 & 0\\0 & 28\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix}=\begin{pmatrix}42\\-56\end{pmatrix}\quad\left|\begin{matrix}\to x_1=2\\\ \end{matrix}\right. \tag{9.8}$$

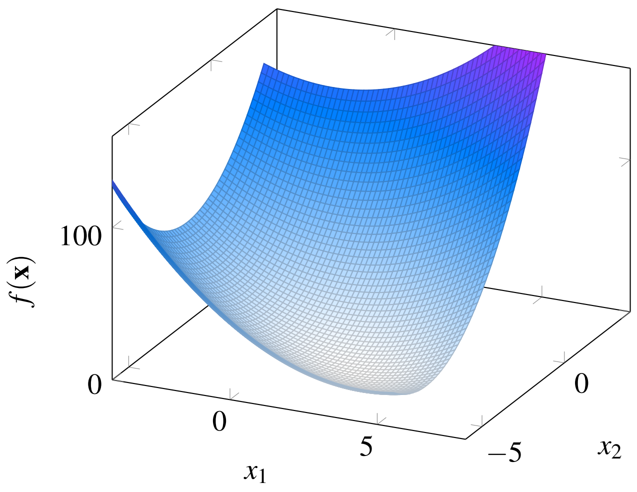
9.1 Conjugate Gradient solver

For one examples of iterative solvers, we look into the most popular iterative method for solving large systems of linear equations. More details about iterative methods [12, 76]. The Conjugate Gradient (CG) solver which was developed by Hestenes and Stiefel in 1952 [48]. The method solves linear equation systems with the form $\mathbf{A}\mathbf{x}=\mathbf{b}$. The matrix $\mathbf{A}$ has to be symmetric $\mathbf{A}^T=\mathbf{A}$ and positive-definite $\mathbf{x}^T\mathbf{A}\mathbf{x}>0,\forall\mathbf{x}>0$.

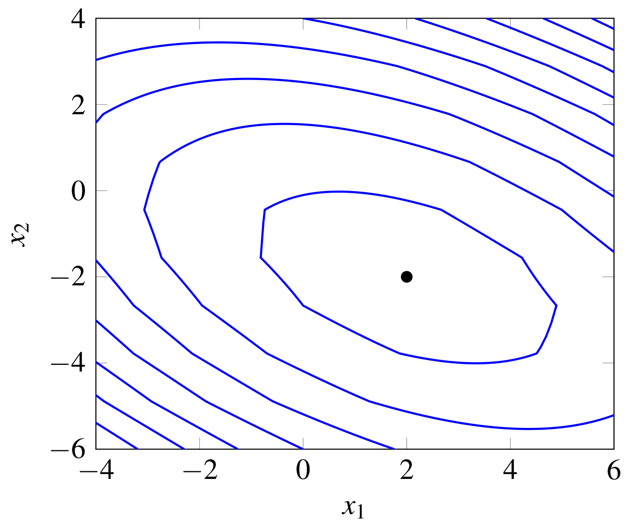
We use the problem showed in Figure 9.1 and define the system $\mathbf{A}\mathbf{x}=\mathbf{b}$ as

$$\mathbf{A}=\begin{pmatrix}3 & 2\\2 & 6\end{pmatrix},\mathbf{x}=\begin{pmatrix}x_1\\x_2\end{pmatrix},\text{ and }\mathbf{b}=\begin{pmatrix}2\\-8\end{pmatrix}.$$

Since we know that solving that solving the matrix form can be expensive for large amounts



(a) Plot of the quadratic form $f(\mathbf{x})$



(b) Contour plot of the quadratic form $f(\mathbf{x})$

Figure 9.2: Plot of the quadratic form $f(\mathbf{x})$ (a) and contour plot of the quadratic form $f(\mathbf{x})$ (b).

of unknowns, the quadratic form, which is a function of the vector $\mathbf{x}$

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - b^T\mathbf{x} + c \tag{9.9}$$

can be minimized to find the solution for $\mathbf{x}$. Figure 9.2a shows the plot of the quadratic form and Figure 9.2b shows the contour plot of the quadratic form with the solution $(2,-2)$. To exemplify the minimization to find the solution, we can place a golf ball at any position of the bowlish from of the quadratic form and the golf ball will roll to the solution, since the solution is the minimum of the quadratic form.

Now we want to mimic the metaphor of the rolling golf ball in a numerical sense. Therefore, we chose an arbitrary point $\mathbf{x}_0$ on the quadratic form and intend to slide down to the bottom of the bowl shape of the quadratic form to find the solution. Thus, we need to find the direction which decreases most. To find these, we use the gradient of the

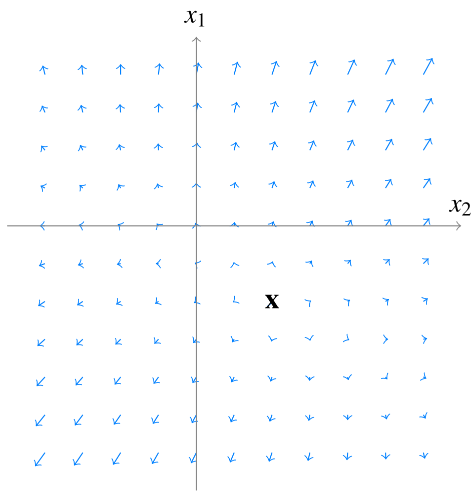


Figure 9.3: Plot of the gradient field $f'(x)$ with the solution $\mathbf{x}$.

quadratic form

$$f'(\mathbf{x}) = \begin{pmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \frac{\partial}{\partial x_2} f(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(\mathbf{x}) \end{pmatrix}. \tag{9.10}$$

Using the quadratic form, the gradient and applying some mathematics, one can show that the gradient reads as

$$f'(\mathbf{x}) = \frac{1}{2}\mathbf{A}^T\mathbf{x} + \frac{1}{2}\mathbf{A}\mathbf{x} - \mathbf{b} \tag{9.11}$$

and for a symmetric matrix $\mathbf{A}$ the gradient reads as

$$f'(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}. \tag{9.12}$$

Note that this course focus on the computer science and implementation aspects of the CG solver and for more details about the mathematics, we refer to [88] where the very nice example for the introduction of the conjugate gradient method was adapted from. Figure 9.3 plots the gradient field and one can see that the gradient at the solution $\mathbf{x}$ is zero. Thus, we can minimize the initial guess $\mathbf{x}_0$ such that the gradient $f'(x) = 0$.

Exercise 9.1 To gain better understanding, we encourage you to follow the calculations in [88] or even better try to do the altercations on your own and check them later in the reference. ■

To implement the metaphor of the rolling golf ball, we use the method of the steepest decent. In this method, we chose an random initial guess $\mathbf{x}_0$ and slide down to the bottom of the quadratic form $f(\mathbf{x})$ by taking a series of steps $\mathbf{x}_1, \mathbf{x}_2, \ldots$. For each step we go to the direction which f decreases most which is the opposite of $f'(\mathbf{x}_i)$ which is $-f'(\mathbf{x}_i) = \mathbf{b} - \mathbf{A}\mathbf{x}_i$.

Method of the steepest decent

Before, we look into the method, we have to define two terms, see Figure 9.4. Assume we know the solution $\mathbf{x}$ and we have our initial guess $\mathbf{x}_0$ we can define the error $\mathbf{e} = \mathbf{x}_0 - \mathbf{x}$ or more

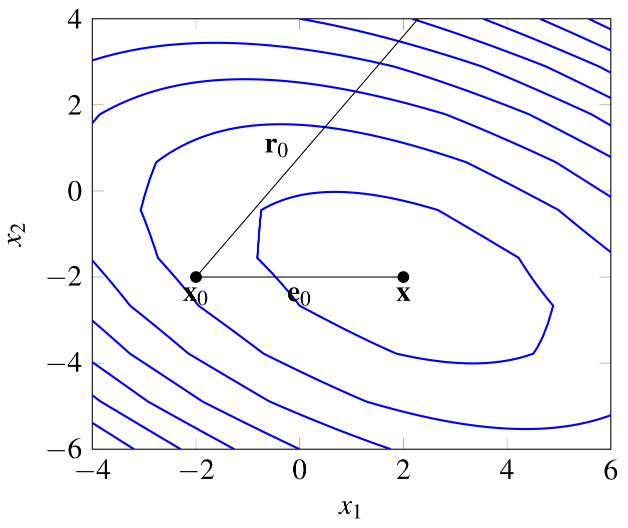


Figure 9.4: Visualization of the error $\mathbf{e}$ and the residual $\mathbf{r}$. We have to determine how long we should go along the residual line within one iteration.

general for the i-th iteration as $\mathbf{e}_i = \mathbf{x}_i - \mathbf{x}$. The residual $\mathbf{r}_i = \mathbf{b} - \mathbf{A}\mathbf{x}_i = -f'(\mathbf{x_i})$ which defines how far we are form the correct value for the right-hand side $\mathbf{b}$. Remember the metaphor of the golf ball sliding down to the bottom of the bowlish shape. This means for us now that we have to go along the line $\mathbf{r}_0$ to get closer to the bottom. Since we do not have gravity guiding us to the bottom. we have to decide on how long we want to go along the line $\mathbf{r}_0$.

So we have to find α for how long we go in the direction of the steepest decent $\mathbf{x}_1 = \mathbf{x}_0 + \alpha\mathbf{r}_0$. Figure 9.6a visualized the plane defined by $\mathbf{x}_1 = \mathbf{x}_0 + \alpha\mathbf{r}_0$ and the quadratic form $f(\mathbf{x})$. Next, we look at the intersection of the two surfaces which is some parabola, see Figure 9.6b. Now, we have to find the minimum of the parabola $\frac{d}{d\alpha}f(\mathbf{x}_0 + \alpha\mathbf{r}_0) = 0$ to determine the optimal value for α. Applying the chain rule results in $\frac{d}{d\alpha}f(\mathbf{x}_0 + \alpha\mathbf{r}_0) = f'(\mathbf{x}_0 + \alpha\mathbf{r}_0)^T\mathbf{r}_0$. This expression is zero if and only if the two vectors are orthogonal. We can do some calculations

$$\mathbf{r}_1^T\mathbf{r}_0 = 0 \tag{9.13}$$
$$(-\mathbf{A}\mathbf{x}_1)^T\mathbf{r}_0 = 0 \tag{9.14}$$
$$(-\mathbf{A}(\mathbf{x}_0 + \alpha\mathbf{r}_0))^T\mathbf{r}_0 = 0 \tag{9.15}$$
$$(\mathbf{b} - \mathbf{A}\mathbf{x}_0)^T\mathbf{r}_0 - \alpha(\mathbf{A}\mathbf{r}_0)^T\mathbf{r}_0 = 0 \tag{9.16}$$
$$(\mathbf{b} - \mathbf{A}\mathbf{x}_0)^T\mathbf{r}_0 = \alpha(\mathbf{A}\mathbf{r}_0)^T\mathbf{r}_0 \tag{9.17}$$
$$\mathbf{r}_0^T\mathbf{r}_0 = \alpha\mathbf{r}_0^T(\mathbf{A}\mathbf{r}_0) \tag{9.18}$$
$$\alpha = \frac{\mathbf{r}_0^T\mathbf{r}_0}{\mathbf{r}_0^T\mathbf{A}\mathbf{r}_0} \tag{9.19}$$

to compute α. Note that this course focus on the computer science and implementation aspects of the CG solver and for more details about the mathematics, we refer to [88] where the very nice example for the introduction of the conjugate gradient method was adapted from. Figure 9.5 shows the first iteration and how long to go along the the residual line to the next guess of the solution $\mathbf{x}_1$ and we clearly see that the gradient is to $\mathbf{x}_0$.

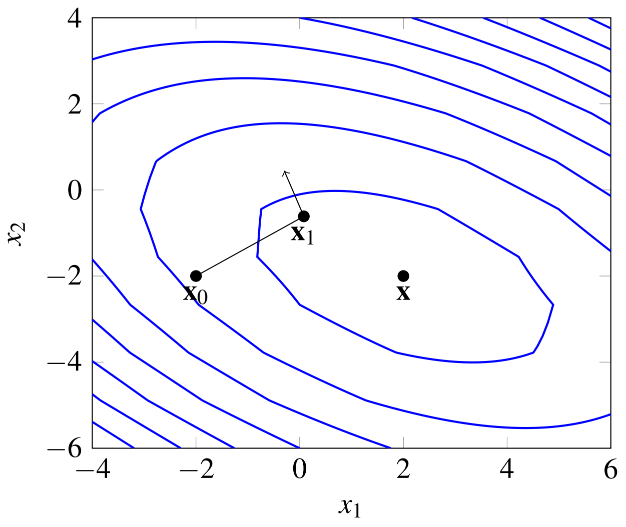
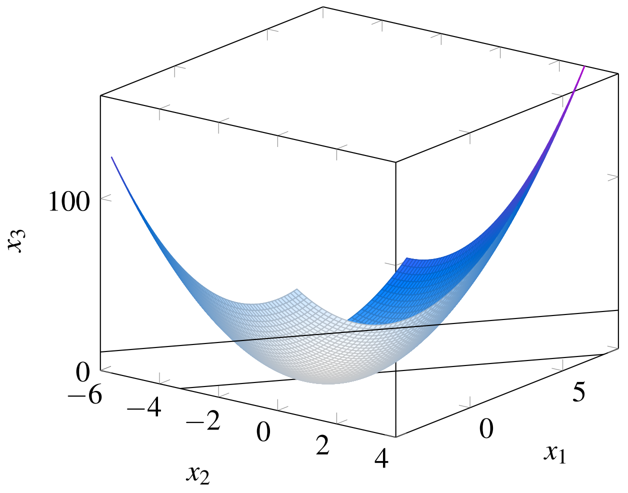


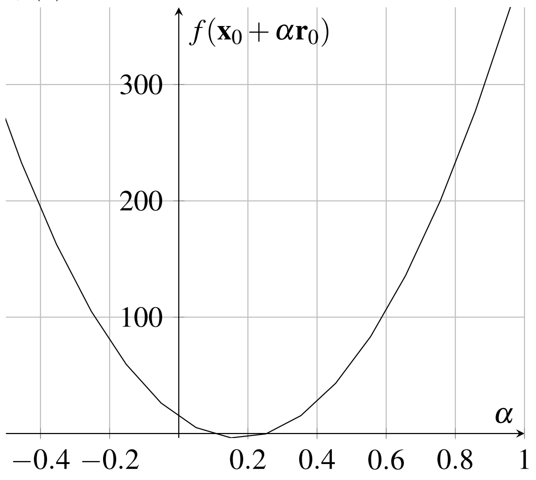
Figure 9.5: first iteration and how long to go along the the residual line to the next guess of the solution $\mathbf{x}_1$ and we clearly see that the gradient is to $\mathbf{x}_0$.

Exercise 9.2 To gain better understanding, we encourage you to follow the calculations in [88] or even better try to do the altercations on your own and check them later in the reference. ■

Now, we extend this example to the algorithm to iterate to the solution of the linear equation system. Figure 9.7 shows the flow chart of the conjugate gradient solver. First, we have to check if the residual is close to the tolerance ε, if so, we guessed $\mathbf{x}_0$ close enough to the solution. If not, the residual $\mathbf{r}_i$ is evaluated. Next, we compute α_i and the new guess $\mathbf{x}_{i+1}$. Now, we check if the residual with respect to $\mathbf{x}_{i+1}$ is close enough to the tolerance, if so, we return the solution. If not, we proceed with the next iteration. Algorithm 1 shows the pseudo code for the CG method. Here more implementation details as in the flow chart are provided. Figure 9.8 shows the first five iterations of the CG algorithm with the initial guess $\mathbf{x_0} = (-2,-2)^T$ and the solution $\mathbf{x}_5 = (1.93832964,-2.)^T$. Fore more details on iterative solver, we refer to [10].



(a) Plot of the quadratic form $f(\mathbf{x})$ and the area of the line search $\mathbf{x}_1 = \mathbf{x}_0 + \alpha\mathbf{r}_0$



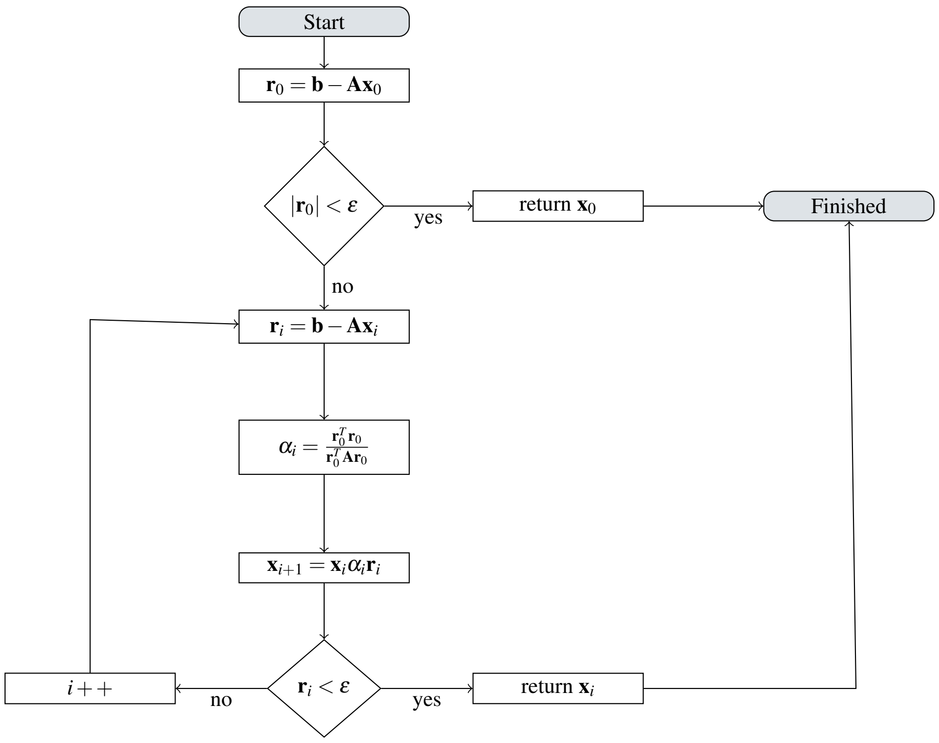
(b) Contour plot of the quadratic form $f(\mathbf{x})$

Figure 9.6: Plot of the two surfaces (a) and resulting parabola of the intersection of these two surfaces (b).

Exercise 9.3 Implement the conjugate gradient algorithm using Blaze library. This code produces a matrix $\mathbf{A}$ and a vector $\mathbf{b}$, such that the vector $\mathbf{x}$ is the solution for $\mathbf{Ax} = \mathbf{b}$

```
for(int i=0; i<N; ++i) {
        A(i,i) = 2.0;
        b[i] = 1.0*(1+i);
        x[i] += x[i-1];
    }
```

You can use the matrix $\mathbf{A}$ and the vector $\mathbf{b}$ as the input of your CG implementation and compare your solution with the vector $\mathbf{x}$ to validate your code. You should not use this vector as the input of the CG algorithm, since your code might stop at step (2) already. ■



Start
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$|\mathbf{r}_0| < \varepsilon$
yes
return $\mathbf{x}_0$
Finished
no
$\mathbf{r}_i = \mathbf{b} - \mathbf{A}\mathbf{x}_i$
$\alpha_i = \frac{\mathbf{r}_0^T \mathbf{r}_0}{\mathbf{r}_0^T \mathbf{A}\mathbf{r}_0}$
$\mathbf{x}_{i+1} = \mathbf{x}_i \alpha_i \mathbf{r}_i$
$i{+}{+}$
no
$\mathbf{r}_i < \varepsilon$
yes
return $\mathbf{x}_i$

Figure 9.7: Flow chart for the Conjugate Gradient method to solve $\mathbf{M}\mathbf{x} = \mathbf{b}$.

Notes

[147]`https://bitbucket.org/blaze-lib/blaze/wiki/Vectors`
[148]`https://bitbucket.org/blaze-lib/blaze/wiki/Vector%20Types#!dynamicvector`
[149]`https://bitbucket.org/blaze-lib/blaze/wiki/Vector%20Operations#!norms`
[150]`https://bitbucket.org/blaze-lib/blaze/wiki/Vector-Vector%20Multiplication#!inner-product-scalar-product-d`
[151]`https://bitbucket.org/blaze-lib/blaze/wiki/Matrix%20Types`
[152]`https://bitbucket.org/blaze-lib/blaze/wiki/Matrix%20Types#!dynamicmatrix`
[153]`https://bitbucket.org/blaze-lib/blaze/wiki/Matrix%20Types#!staticmatrix`
[154]`https://bitbucket.org/blaze-lib/blaze/wiki/Matrix%20Types#!sparse-matrices`
[155]`https://bitbucket.org/blaze-lib/blaze/wiki/Matrix%20Operations`
[156]`https://bitbucket.org/blaze-lib/blaze/wiki/Configuration%20and%20Installation`
[157]`https://bitbucket.org/blaze-lib/blaze/wiki/Shared%20Memory%20Parallelization`

Algorithm 1 Implementation of the Conjugate Gradient method with some additions where the factor β is computed.

```
$\mathbf{r_0} = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
**if** $\mathbf{r}_0 < \varepsilon$ **then return** $\mathbf{x}_0$
**end if**
$\mathbf{p}_0 = \mathbf{r}_0$
$k = 0$
**while** true **do**
    $\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k}$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{p}_k$
    **if** $\mathbf{r}_{k+1} < \varepsilon$ **then**
        exit loop
    **end if**
    $\beta_k = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$
    $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
    $k = k + 1$
**end while**
return $\mathbf{x}_{k+1}$
```

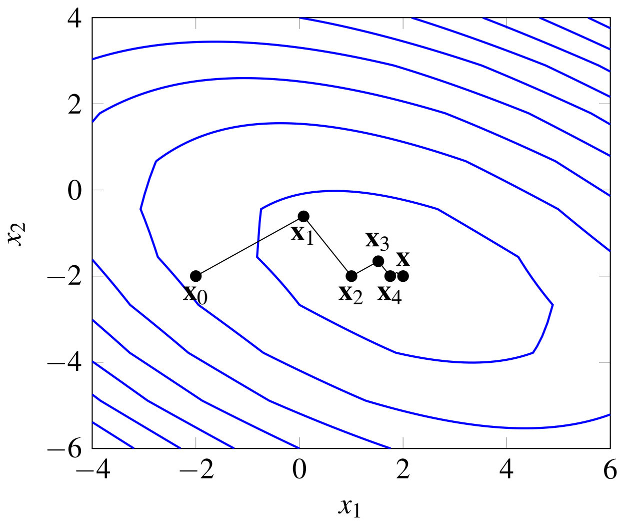


Figure 9.8: Visualization of the line search for the first five iterations of the Conjugate Gradient algorithm with the solution $\mathbf{x}_5 = (1.93832964, -2.)^T$.

V Numerical examples

10. Monte-Carlo methods

Monte Carlo methods are computational algorithms which rely on repeated random sampling to obtain numerical results. The principle is to use randomness to solve the problem because it is difficult or impossible to use other approaches. When this method was developed in the 1940s by Ulam and von Neumann they called the method ”Monte Carlo” in reference to the Monte Carlo Casino in Monaco where Ulam’s uncle gambled. Today Monte Carlo methods are widely used in the following three problem classes:

- Optimization,
- Numerical integration, and
- Probability distributions.

For the importance of the method we refer to [63], and for more details about Monte Carlo Methods we refer to [89].

Let us look into the computational aspects of the Monte Carlo methods. Independent of the problem class, a general pattern is observed:

1. Define the input parameters,
2. Randomly chose input parameters,
3. Do deterministic computations on the inputs,
4. Aggregate the results.

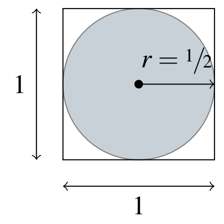


Figure 10.1: Sketch of the geometry used within the Monte Carlo method to estimate the value of π.

To understand these four steps, we will compute the value of π using a Monte Carlo method. Figure 10.1 sketches the first two ingredients: a unit square and a circle. First, a unit square is defined as 1×1, which means it has a side length of 1. The area A_s is therefore also 1. Second, the circle with radius $r = 1/2$ is drawn at the center of the unit square. The area of the circle is $A_c = \pi r^2$. Using the radius $r = 1/2$ the area is $A_c = \pi(1/2)^2 = \pi/4$. Now, since we have defined the area of the circle and the square, we can use them to estimate the value of π:

$$\begin{aligned} A_c &= \pi/4 \\ \pi &= 4A_c \\ \pi &= 4A_c/A_s. \end{aligned} \tag{10.1}$$

Note that the operation on the first equation is a multiplication by four. Going from the second line to the third line, we use the fact that the area of the square is one.

Now, we can estimate π by the general pattern described above.

- Define the input parameters:
  A coordinate $(x,y) \in \mathbb{R}$ in the domain of the unit square $[0,1] \times [0,1]$
- Randomly chose input parameters:
  We randomly draw N values for x and y in the range of $[0,1]$
- Do deterministic computations on the inputs:
  We must validate if the coordinate (x,y) is inside the circle or not with the inequality $x^2 + y^2 \leq 1$. If the coordinate is within the circle, we increment N_C.
- Aggregate the results:
  We compute $\pi \approx 4N_c/N$

Figure 10.2 shows the flow chart of the algorithm for estimating π using the Monte Carlo method. First, the decision if the current draw of the random number is less than the desired total amount of random numbers N. If we have not yet drawn enough random numbers, we have to guess two random numbers x and y (see Section 2.2 for how to generate random numbers in C++). Next, we have to check if the drawn coordinate (x,y) is within the circle. If so, we increment the count of the number of points that have landed within the circle N_c. We have to repeat these steps until $i > N$. Once we have drawn enough random numbers, we can compute $\pi \approx 4N_c/N$ and finish the program.

Next,let us ask the question, "What is a good choice for N to get a good approximation of *pi*?" Figure 10.3 shows the distribution of the point inside the circle (red) and outside of the circle (blue) for $N = 10$, $N = 100$, and $N = 1000$ random numbers. One can see that a certain amount of random numbers is needed to have enough samples inside and outside of the circle. Figure 10.4 shows the absolute error in percent for various amounts of random numbers. One can see that with a thousand random numbers the accuracy is quite reasonable.

Exercise 10.1 Make a list of which C++ features we need to implement the flow chart in Figure 10.2. ■

Exercise 10.2 Implement the Algorithm in Figure 10.2 using the random numbers in Section 2.2. ■

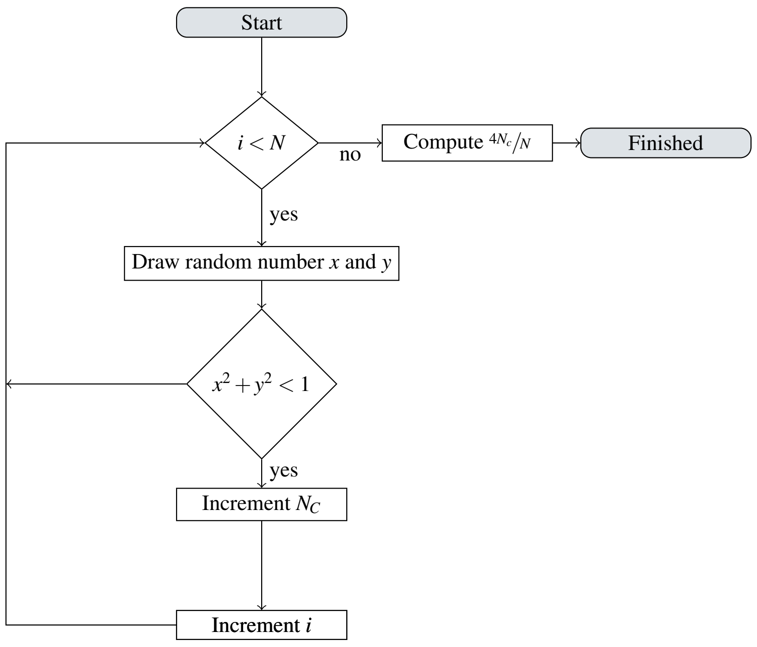


Figure 10.2: Flow chart for the Monte Carlo method to estimate π.

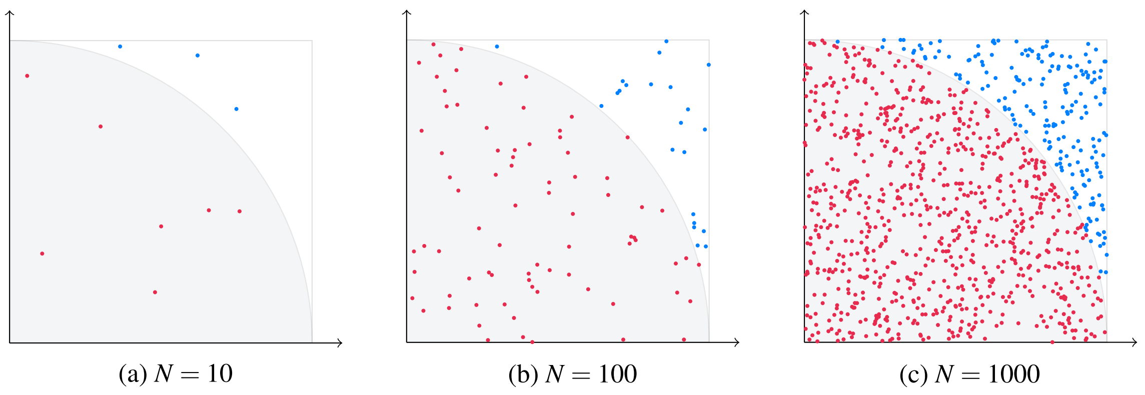


Figure 10.3: Distribution of the points inside the circle (red) and outside of the circle (blue) for $N = 10$, $N = 100$, and $N = 1000$ random numbers.

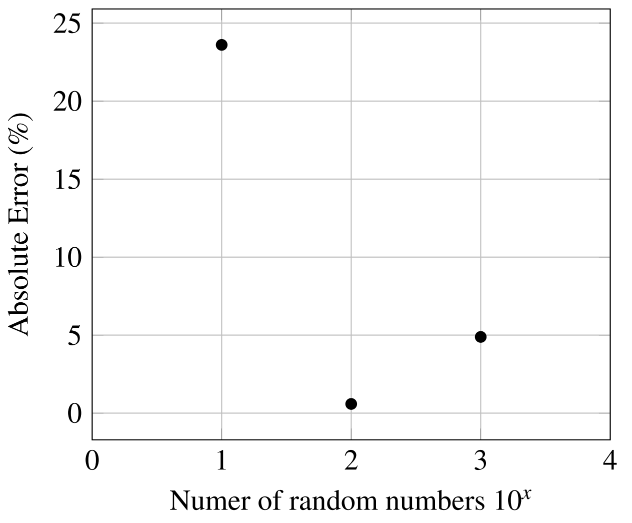


Figure 10.4: The absolute error for various amounts of random numbers. One can see that with a thousand random numbers the accuracy is quite reasonable.

11. N-body problems

The N-body problem is the physics problem of predicting the individual motions of members of a group of celestial objects interacting with each other gravitational. We want to predict the interactive forces and the motion of all celestial bodies at all future times. We assume that we know their orbital properties, e.g. the initial positions, velocity, and time.

Before we look into the N-body problem, let us step back and look at the two-body problem. Let us look at two gravitational bodies with the masses m_i and m_j and the positions $\mathbf{r}_i, \mathbf{r}_j \in \mathbb{R}^3$. To define the equation of motion, we refer to the following definitions:

1. The Law of Gravitation:
   The force of m_i acting on m_j is
   $$\mathbf{F}_{ij} = Gm_im_j\frac{\mathbf{r}_j - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_i|^3}(SeeFigure\ 11.1) \tag{11.1}$$
   The universal constant of gravitation G was estimated as $6.67408 \cdot 10^{-11} m^3 kg^{-1} s^{-2}$ in 2014 [72].
2. Velocity and acceleration:
   (a) The velocity of m_i is
   $$\mathbf{v}_i = \frac{d\mathbf{r}_i}{dt} \tag{11.2}$$
   (b) The acceleration of m_i is
   $$\mathbf{a}_i = \frac{d\mathbf{v}_i}{dt} \tag{11.3}$$
   For more details about vectors and basic vector operations, we refer to Section 8.1.1.
3. The second Law of Mechanics: (Force is equal to mass times acceleration)
   $$\mathbf{F} = m\mathbf{a} \tag{11.4}$$

m_1
$\mathbf{F}_1$
$\mathbf{F}_2$
m_2

Figure 11.1: Sketch of the two celestial bodies with the masses m_1 and m_2 and the gravitational interaction forces $\mathbf{F}_1$ and $\mathbf{F}_1$. Equation 11.8 shows the equation of motion for the two-body system.

With these three definitions, we can derive the equation of motion for the first body as follows:

$$\mathbf{F}_{ij} = Gm_im_j\frac{\mathbf{r}_j-\mathbf{r}_i}{|\mathbf{r}_j-\mathbf{r}_i|^3} \tag{11.5}$$

$$m_i\mathbf{a}_i = Gm_im_j\frac{\mathbf{r}_j-\mathbf{r}_i}{|\mathbf{r}_i-\mathbf{r}_j|^3} \tag{11.6}$$

$$\frac{d\mathbf{v}_i}{dt} = Gm_j\frac{\mathbf{r}_j-\mathbf{r}_i}{|\mathbf{r}_j-\mathbf{r}_i|^3} \tag{11.7}$$

$$\frac{d^2\mathbf{r}_i}{dt^2} = Gm_j\frac{\mathbf{r}_j-\mathbf{r}_i}{|\mathbf{r}_j-\mathbf{r}_i|^3} \tag{11.8}$$

To get from Equation (11.5) to Equation (11.6), we substitute $\mathbf{F}_{ij}$ with $m_i\mathbf{a}_i$ using Equation 11.4. From Equation (11.6) to Equation (11.7), we divide by m_i and replace $\mathbf{a}_i$ according to Equation 11.3. From Equation (11.7) to Equation (11.8), we substitute Equation 11.2. Note that we used Newton's law of universal gravitation [74].

Now we formulate the problem for n bodies, assuming that the force at one body is equal to the sum over all bodies, excepting itself.

$$\mathbf{F}_i = \sum_{j=1,i\neq j}^{n}\mathbf{F}_{ij} = \sum_{j=1,,i\neq j}^{n} Gm_j\frac{\mathbf{r}_j-\mathbf{r}_i}{|\mathbf{r}_j-\mathbf{r}_i|^3}. \tag{11.9}$$

These are the laws of conservation for the N-body problem:

1. Linear Momentum: $\sum_{i=1}^{n} m_i\mathbf{v}_i = M_0$
2. Center of Mass: $\sum_{i=1}^{n} m_i\mathbf{r}_i = M_0t + M_1$
3. Angular Momentum: $\sum_{i=1}^{n} m_i(\mathbf{r}_i\times\mathbf{v}_i) = \mathbf{c}$
4. Energy: T-U=h with
   $T = \frac{1}{2}\sum_{i=1}^{n} m_i\mathbf{v}_i\circ\mathbf{v}_i, U = \sum_{i=1}^{n}\sum_{j=1}^{n} G\frac{m_im_j}{|\mathbf{r}_i-\mathbf{r}_j|}$

These laws are just shown for completeness. For more details about the theory and the derivations, we refer to [1, 2]. This text focuses on the implementation details of the N-body problem with C++.

11.1 Algorithm

Figure 11.2 shows the three steps for the N-body simulation. In this section we focus on the implementation details of the first two steps. Equation 11.9 shows how to compute the force for one celestial object. Recall that the $\sum$ translates to a `for` loop as we discussed in Section 1.4.1. To compute the forces of all bodies, the so-called nested `for` loop or direct sum is used. Listing 11.1 shows the concept of the direct sum which is robust, accurate, and completely general. The computational costs per body are $\mathcal{O}(n)$ and the computational

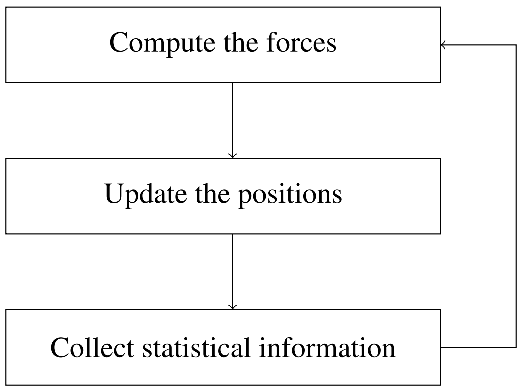


Figure 11.2: The three steps of the algorithm for the N-body simulation. First, the forces for all objects are computed using Equation 11.9. Second, the updated positions are computed using Equation 11.13 and Equation 11.14. Third, the statistical information is evaluated.

Listing 11.1: Example for the direct sum.

```cpp
for(size_t i = 0; i < bodies.size(); i++)
        for(size_t j = 0; j < bodies.size(); j++)
                if ( i != j )
                        //Compute forces
```

costs for all bodies are $\mathcal{O}(n^2)$. The symbol $\mathcal{O}$ is the so-called "Big O" notation, which we use to describe algorithm run time or space requirement growth as the input size grows. In our case the computational cost per body increases linearly, since we have to compute the force $n-1$ times for all particles. The Big O notation $\mathcal{O}(n)$ means that the total computational cost for n computations is less than or equal to n. These symbols are defined in the Bachmann–Landau notation [7, 61, 66]. For all bodies the computational cost increases to the power of two since we have to compute the forces $n-1$ times for all n bodies. The direct sum is feasible for a small number of celestial objects, but for larger numbers the tree-based codes or the Barnes-Hut method [9] reduce the computational costs to $\mathcal{O}(n\log(n))$.

For the second step of the algorithm, we need to update the positions for the evolution of the system over the time T. For the discretization in time, we define the following quantities:

- Δt the uniform time step size
- t_0 the beginning of the evolution
- T the final time of the evolution
- k the time steps such that $k\Delta t = T$.

Next, we need to compute the derivatives to obtain the velocity and the acceleration of each celestial object. One numerical method to approximate the derivation is given by

$$u'(x) \approx \frac{u(x+h)-u(x)}{h} \tag{11.10}$$

which is the finite difference method. Figure 11.3 sketches the principle of the finite difference method. For a sufficiently small h, we can approximate the derivation at the coordinate x. For example: Choosing $h=1$ and $x=3$, we get $u'(x) = {}^{(4-3)}/_{1} = 1$ which aligns

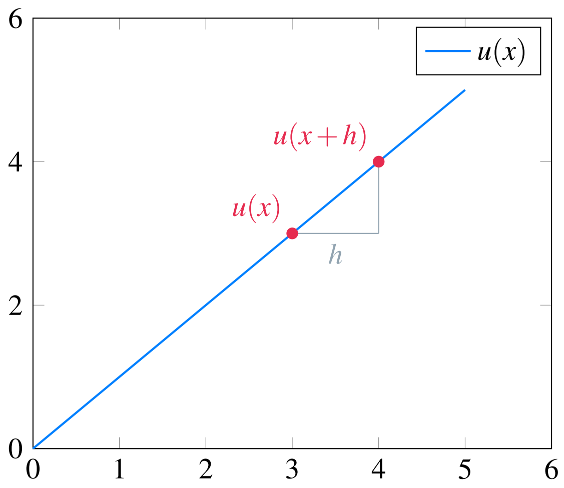


Figure 11.3: The principle of the finite difference method. For a sufficiently small h, we can approximate the derivation at the coordinate x. For example: Choosing $h = 1$ and $x = 3$, we get $u'(x) = {}^{(4-3)}/_{1} = 1$ which aligns with $u'(x) = 1$ using the analytic derivation of $u(x)$. Now we can use the Euler method to compute the updated positions at time t_{k+1}.

with $u'(x) = 1$ using the analytic derivation of $u(x)$. Now we can use the Euler method to compute the updated positions at time t_{k+1}. First we approximate the velocity using the finite difference scheme

$$\mathbf{v}_i(t_k) = \frac{d\mathbf{r}_i}{dt} \approx \frac{\mathbf{r}_i(t_{k+1}) - \mathbf{r}_i(t_k)}{\Delta t}. \tag{11.11}$$

We do the same for the acceleration

$$\mathbf{a}_i(t_k) = \frac{d\mathbf{v}_i}{dt} \approx \frac{\mathbf{v}_i(t_k) - \mathbf{v}_i(t_k - 1)}{\Delta t} = \frac{\mathbf{F}_i}{m_i} \tag{11.12}$$

from Equation 11.4 we get $\mathbf{a}_i = {}^{\mathbf{F}_i}/_{m_i}$. More details [30, 68, 96]. With the above approximations the velocity is computed as

$$\mathbf{v}_i(t_k) = \mathbf{v}_i(t_{k-1}) + \Delta t \frac{\mathbf{F}_i}{m_i} \tag{11.13}$$

using Equation (11.12) and the fact that the finite difference approximation of the acceleration is equal to ${}^{\mathbf{F}_i}/_{m_i}$. Finally, the updated position is computed as

$$\mathbf{r}_i(t_{k+1}) = \mathbf{r}_{t_k} + \Delta t \mathbf{v}_i(t_k) \tag{11.14}$$

using Equation (11.11). Note that we used easy methods to update the positions, but more sophisticated methods, e.g. Crank–Nicolson method [24], are available

Exercise 11.1 Look at the equations in this section and try to derive the Equation 11.13 and Equation 11.13 on your own. ■

Exercise 11.2 Implement the N-body problem using the template code[158] on GitHub. ■

12. Peridynamics

Peridynamic, a alternative formulation of continuum mechanics with a focus on discontinuous displacement as they arise in fracture mechanics, was introduced by Silling in 2000 [90, 91]. Models crack and fractures on a mesoscopic scale using Newton's second law (force equals mass times acceleration)

$$F = m \cdot a = m \cdot \ddot{X}. \tag{12.1}$$

12.1 Brief introduction in classical continuum mechanics

We briefly look into the ingredients of classical continuum mechanics which are needed to introduce peridynamics. In Figure 12.1 on the lift-hand side, we see the continuum in the reference configuration $\Omega_0 \subset \mathbb{R}^3$ which is the state where we have no internal forces and we are in the equilibrium. We denote these positions with capitalized $X \in \mathbb{R}^3$ to distinguished with the new position after the deformation $\phi : \Omega_0 \to \mathbb{R}^3$. The deformation implied for example by some external forces moves the continuum from the reference configuration Ω_0 to the current configuration $\Omega(t)$. The new position of X is now $x(t,X)$.

Let us look more closely in the definitions above. The deformation $\phi : [0,T] \times \mathbb{R}^3 \to \mathbb{R}^3$ of a material point X in the reference configuration Ω_0 to the so-called current configuration $\Omega(t)$ is given by

$$\phi(t,X) := id(X) + u(t,X) = x(t,X),$$

where $u : [0,T] \times \mathbb{R}^3 \to \mathbb{R}^3$ refers to the displacement

$$u(t,X) := x(t,X) - X\,.$$

The stretch $s : [0,T] \times \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$ between the material point X and the material point X' after the deformation ϕ in the configuration $\Omega(t)$ is defined by

$$s(t,X,X') := \phi(t,X') - \phi(t,X)\,.$$

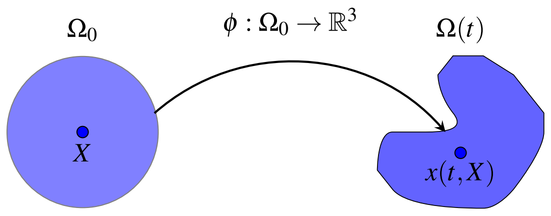


Figure 12.1: The continuum in the reference configuration Ω_0 and after the deformation $\phi : \Omega_0 \to \mathbb{R}^3$ with $\det(\text{grad } \phi) > 0$ in the current configuration $\Omega(t)$ at time t.

We just covered the prerequisites of classical continuum mechanics which are necessary to introduce the peridynamic theory. For more details, we refer to [43, 69].

12.2 Brief introduction in bond-based peridynamics

We can use Newton;s second law (force equals mass times acceleration) and formulate it as

$$\rho(X)a(t,X) := \int\limits_{B_\delta(X)} f\left(t, x(t,X') - x(t,X), X' - X\right) dX' + b(t,X), \tag{12.2}$$

to compute the acceleration $a : [0,T] \times \mathbb{R}^3 \to \mathbb{R}^3$ of a material point at position X at time t. With the pair-wise force function $f : [0,T] \times \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}^3$, the mass density $\rho(X)$, and the external force $b : [0,T] \times \mathbb{R}^3 \to \mathbb{R}^3$. Following assumptions are made

1. The medium is continuous (equal to a continuous mass density field exists)
2. Internal forces are contact forces (equal to that material points only interact if they are separated by zero distance.
3. Conservation laws of mechanics apply
   (a) Conservation of mass
   (b) Conservation of linear momentum

$$f(t, -(x(t,X') - x(t,X)), -(X' - X)) = -f(t, x(t,X') - x(t,X), X' - X)$$

   (c) Conservation of angular momentum

$$(x(t,X') - x(t,X) + X' - X) \times f\left(t, x(t,X') - x(t,X), X' - X\right) = 0$$

12.2.1 Material model

There are several material models available, however, we look into the Prototype Microelastic Brittle (PMB) model, since it was one of the first material models. In this model the assumption is made that the pair-wise force f only depends on the relative normalized bond stretch $s : [0,T] \times \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$

$$s(t, x(t,X') - x(t,X), X' - X) := \tag{12.3}$$

$$\frac{||x(t,X') - x(t,X))|| - ||X' - X||}{||X' - X||}, \tag{12.4}$$

where $X' - X$ is the vector between the material points in the reference configuration Ω_0 and $x(t,X') - x(t,X)$ is the vector between the material point in the current configuration $\Omega(t)$. As a material property, the so-called stiffness constant c is introduced and the pair-wise force function reads as

$$f(t, x(t,X') - x(t,X), X' - X) := c\, s(t, x(t,X') - x(t,X), X' - X) \frac{x(t,X') - x(t,X)}{\|x(t,X') - x(t,X)\|}. \tag{12.5}$$

The pair-wise force function is shown in Figure 12.2a. Which is a linear line with the slope blue. Note that we do not have introduced damage to the material model yet. Therefore, a scalar valued history dependent function $\mu : [0,T] \times \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{N}$ is added to the computation of the pair-wise force

$$\begin{aligned} f(t,x(t,X') - x(t,X), X' - X) :=& \\ & cs(t,x(t,X') - x(t,X), X' - X) \\ & \mu(t,x(t,X') - x(t,X), X' - X) \frac{x(t,X') - x(t,X)}{\|x(t,X') - x(t,X)\|} . \end{aligned} \tag{12.6}$$

with

$$\mu(t,x(t,X') - x(t,X), X' - X) := \begin{cases} 1 & s(t,x(t,X') - x(t,X), X' - X) < s_c \\ 0 & \text{otherwise} \end{cases} \tag{12.7}$$

The pair-wise force function with the damage is incorporated is shown in Figure 12.2b. With the scalar valued history dependent function μ the notion of damage $d(t,X) : [0,T] \times \mathbb{R}^3 \to \mathbb{R}$ can be introduced via

$$d(t,X) := 1 - \frac{\int\limits_{B_\delta(X)} \mu(t,x(t,X') - x(t,X), X' - X) dX'}{\int\limits_{B_\delta(X)} dX'} . \tag{12.8}$$

To express damage in words, it is the ratio of the active (non-broken) bonds and the amount of bonds in the reference configuration within the neighborhood. Note that we have two material properties the stuffiness constant c and the critical stretch s_c. We can related these to continuum mechanics as

$$c = \frac{18K}{\pi\delta} \quad \text{and} \quad s_c = \frac{5}{12}\sqrt{\frac{K_{Ic}}{K^2\delta}} \tag{12.9}$$

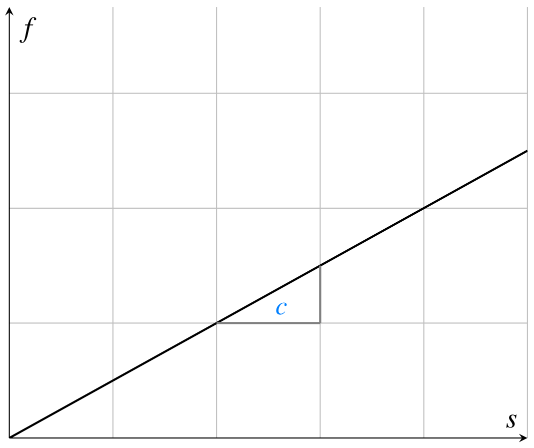
With K is the bulk modulus and K_{Ic} is the critical stress intensity factor.

12.3 Discretization

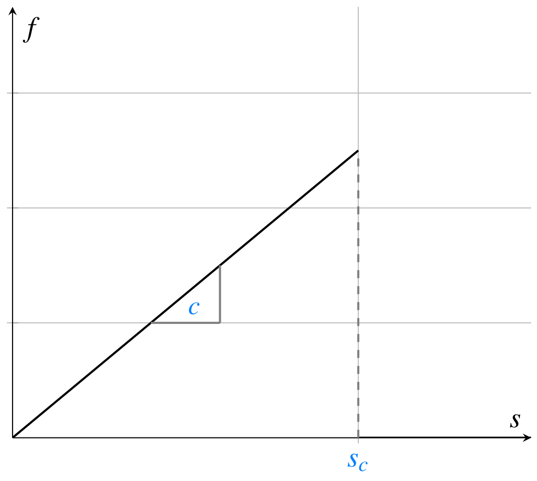
To discretize the peridynamic equation of motion (12.2), the so-called EMU nodal discretization (EMU ND) [77] is used. All material points X are placed at the nodes $\mathbf{X} := \{X_i \in \mathbb{R}^3 | i = 1, \dots, n\}$ of a regular grid in the reference configuration Ω_0, see Figure 12.3. We assume that the discrete nodal spacing Δx between X_i and X_j is defined as $\Delta x = \|X_j - X_i\|$ and is constant in all directions. For all material points at the nodes $\mathbf{X} := \{X_i \in \mathbb{R}^3 | i = 1, \dots, n\}$ a surrounding volume $\mathbf{V} := \{ \mathbf{V}_i \in \mathbb{R} | i = 1, \dots, n\}$ is assumed. These volumes are non overlapping $\mathbf{V}_i \cap \mathbf{V}_j = \emptyset$ and recover the volume of the volume of the reference configuration $\sum_{i=1}^n \mathbf{V}_i = \mathbf{V}_{\Omega_0}$. Using this assumptions the integral sign in the peridynamic equation of motion is replaced by a sum and reads as

$$\rho(X_i) a(t,X_i) = \sum_{X_j \in B_\delta(X_i)} f(t, x(t,X_j) - x(t,X_i), X_j - X_i) \, d\mathbf{V}_j + b(t,X_i). \tag{12.10}$$

The discrete interaction zone $B_\delta(X_i)$ of X_i is given by $B_\delta(X_i) := \{X_j | \, ||X_j - X_i|| \leq \delta\}$ which means that all the materials point within the circle in Figure 12.3 exchange pair-wise forces



(a) Sketch of the pair-wise linear valued force function f with the stiffness constant c as slope.



(b) Sketch of the pair-wise linear valued force function f with the stiffness constant c as slope and the critical bond stretch s_c.

Figure 12.2: Linear elastic pair-wise force (a) and the pair-wise force function with the notation of damage (b)

.

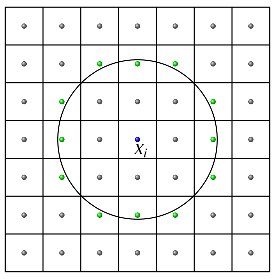


Figure 12.3: Discrete mesh node X_i on the equidistant grid and its interaction zone $B_\delta(X_i) := \{X_j | ||X_j - X_i|| \leq \delta\}$.

with the discrete material point X_i.

From the computational aspects, we have to store the discrete interaction zone $B_\delta(X_i)$ for all discrete material points. To do so, we use two nested `std::vector` data structures. For each discrete node we have `std::vector<size_t>` to store the index of the neighboring discrete nodes. Since we have to store this information for all discrete nodes, we have a nested vector `std::vector<std::vector<size_t>>`. Now, we can use a direct sum, see Listing 11.1, to compute the acceleration a for all our nodes. Note that we need the displacement $u(t,X)$ to compute the pair-wise force $f(t, x(t,X_j) - x(t,X_i), X_j - X_i)$. We use a central difference scheme

$$u(t+1,X) = 2u(t,X) - u(t-1,X) + \Delta t^2 \left(\sum_{X_j \in B_\delta(X_i)} f(t,X_i,X_j) + b(t,X) \right) \tag{12.11}$$

to compute the actual displacement $x(t,X) := x(t-1,X) + u(t,X)$. Note that for the first time step, we assume $x(t-1,X_i) = X_i$ and $u(t-1,X_i) = u(t-1,X_i) = 0$ as the initial values.

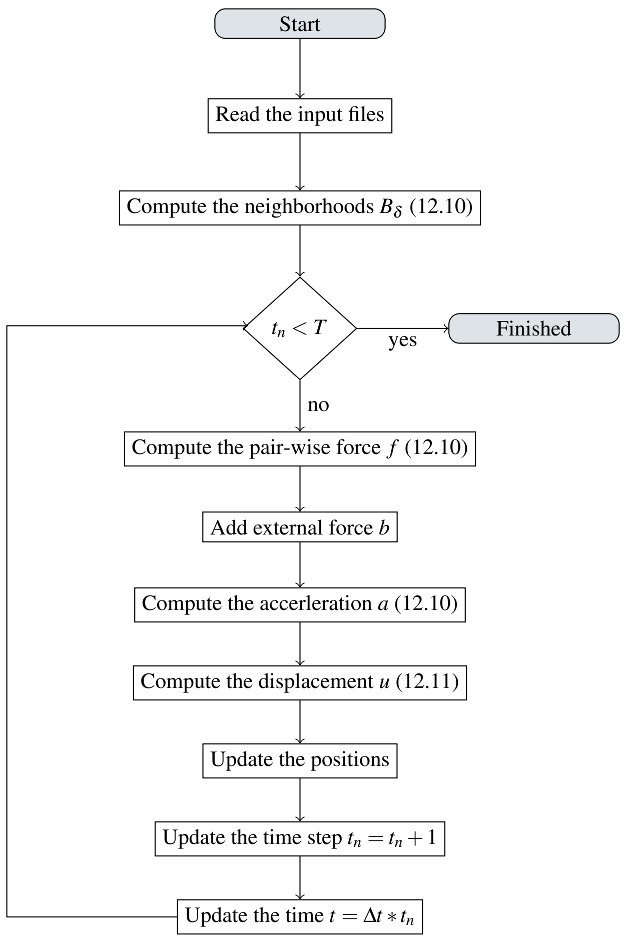
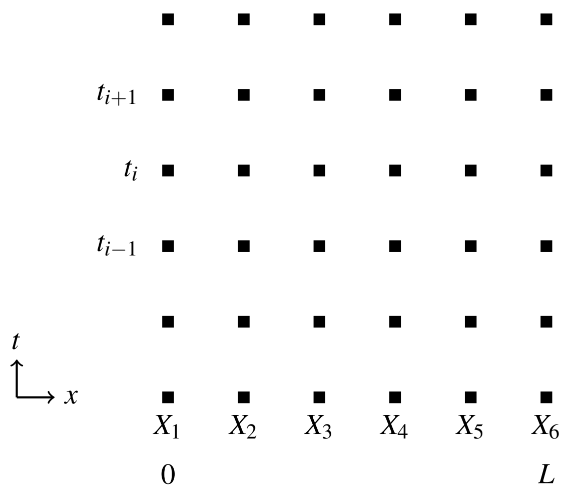

Start
Read the input files
Compute the neighborhoods B_δ (12.10)
$t_n < T$
yes
Finished
no
Compute the pair-wise force f (12.10)
Add external force b
Compute the accerleration a (12.10)
Compute the displacement u (12.11)
Update the positions
Update the time step $t_n = t_n + 1$
Update the time $t = \Delta t * t_n$

Figure 12.4: Flow chart for the Peridynamic simulation.

12.4 Algorithm

Figure 12.4 shows the flow chart for the peridynamic simulation. The first step is to read the input file to obtain the material and discretization properties. Next, the discrete neighborhood for each of the nodes in computed and the neighbors are stored in a nested vector `std::vector<std::vector<size_t>>`. After these steps, the computation is started using a loop. Note that in the first computation the pair-wise forces F are zero, since no external force b was applied. In the next step, the external force b is applied and the acceleration a is computed. Note by adding the external force to the nodes, the acceleration of the nodes is not zero anymore. Now, the displacement u is computed and the positions are updated. Last, the time step and time is updated.


t_{i+1}
t_i
t_{i-1}
t
x
X_1
X_2
X_3
X_4
X_5
X_6
0
L

13. One-dimensional heat equation

For the example for distributed computing, we look into the one-dimensional heat equation. The heat equation reads as

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{13.1}$$

where α is the diffusivity of the material. The heat equation computes the flow of heat in a homogeneous and isotropic medium. For more details about the mathematics and physics of the heat equation, we refer to [17]. For the distributed computing example, we look into the easiest case which is the one-dimensional heat equation. We assume a one-dimensional bar of length L and Eqiation 13.1 reads as

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad 0 \leq x \leq L, t > 0. \tag{13.2}$$

To solve this one-dimensional heat equation, boundary conditions are required

- $u(0,t) = u_0$
- $u(L,t) = u_L$
- $u(x,0) = f_0(x)$.

First, a value at the beginning of the bar and at the end of the bar are given which are constant over time. For all other positions within the bar we apply an arbitrary value at time t equal zero. However, to solve the heat equation from the numerical perspective, we have to discretize the equation in space and time. For the discretization in space a so-called discrete mesh

$$x_i = (i-1)h, \quad i = 1, 2, \ldots, N$$

where N is the total number of nodes and h is given by $h = {}^{L}/_{N-1}$, see Figure 13.1. The next step is to the discretization in time. Therefore, the approximation of the the second

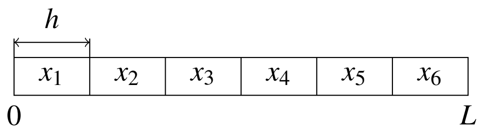


Figure 13.1: Discretization in space using a so–called discrete mesh for the one-dimensional heat equation.

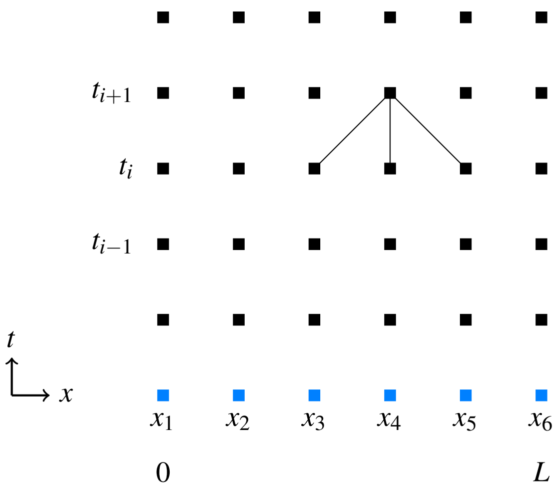


Figure 13.2: Scheme for the discretization in space and time. At time $t = 0$ the boundary conditions are applied at the squares. We use the central difference scheme to compute the new values at time t_{i+1} using the values from the previous time step. The central difference scheme is shown for the discrete mesh node x_4.

derivative $\partial^2 u/\partial x^2$ in Equation 13.2 using a central difference scheme. The first derivation reads as

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1} - u_i}{2h}$$

and the second derivation reads as

$$\frac{\partial u}{\partial x^2} \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{2h}.$$

Meaning we can approximate the second derivation at position x_i using the left neighbor x_{i-1} and the right neighbor x_{i+1}. Note that we do not have the left neighbor at x_0 and the right neighbor at x_L. Here, some special treatment is needed to compute the approximation of the derivation. To avoid this special treatment, we assume that we have a ring instead of a bar in our example. For more details about the finite difference method, we refer to [68, 96]. Now, we combine the discretiztion on time and space, see Figure 13.2, to compute the heat transfer for the next time step. At time $t = 0$ we have applied the boundary conditions on the blue squares. At time t_i we compute the new value for the node x_4 at time t_{i+1} using its neighbor's values at time t_i using the central difference scheme.

13.1 Serial implementation details

First, a function `heat` which implements the central difference scheme in Equation 13.2, see Lines 10–13 in Listing 13.1. Note that for simplicity we renamed α to k. We use the

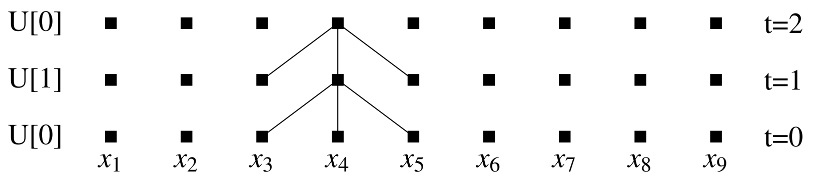


Figure 13.3: Sketch for swapping the partition to reuse the partition vectors to compute the new time step.

keyword `static` in front of the `return` type `double`. We will discuss later why we need to use the `static` keyword. The next step is to look into the data structure (partition) to store the heat values. For the serial implementation the partition is defined as `typedef double partition;` since we store one `double` value per discrete mesh node. For storing all heat values per discrete mesh node a `typedef std::vector<partition> space;` is declared. For the central difference scheme we need the heat values from the previous time step to compute the heat values for the current time step. So we need to have to `space` objects for both time steps, see Line 19 in Listing 13.1. In Line 20–12 the size of the vector is set to the number of discrete mesh nodes nx. Since we have a nested vector `U[t][i]` the first index is the time step t and the second one the index `i`. Thus to set the boundary conditions in Line 24–25 the first argument is zero for $t = 0$ and we iterate over all discrete mesh nodes.

Since we have the initial setup, we can iterate over the time steps using the `for` loop in Line 28. Something tricky happens in Line 30–31 to swap the `space` for the current time step and the previous time step. Figure 13.3 shows how to swap the `space` for each time step. For the initial time $t = 0$ the space `U[0]` holds the current heat values and the space `U[1]` holds the heat values for the next time step $t = 1$. To compute the heat values for the time step $t = 2$ the space `U[0]` is reused to store the next heat values. For swapping the spaces, we use `t % 2` to get the current space and `(t+1) % 2` to get the space for the new heat values. Since we assume a ring, the computation of the first and last elements need a special treatment and all other points are computed the same. The complete source code for the serial example is available here[159]. Choosing following boundary conditions

$$u(x,0) = f(i,0), \text{ with } f(0,i) = i \text{ for } i = 1,2,\ldots,N$$

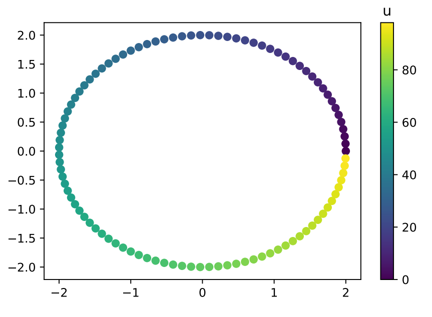
and a heat transfer coefficient $k = 0.5$, time step size $dt = 1.$, grid spacing $h = 1.$, and time steps $nt = 45$ results in the initial conditions, See Figure 13.4a, and the solution, see Figure 13.4b.

13.1.1 Adding grain size control

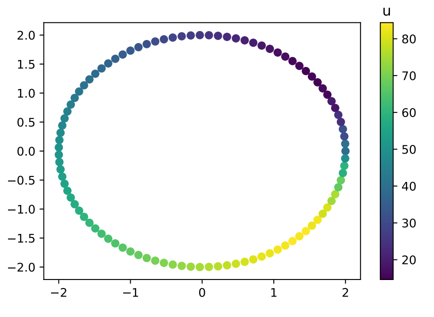
In Figure 13.7 we have seen that we got some speedup with the asynchronous implementation discussed in Section 4.2.1. However, in same cases the granularity (the amount of work) for each core was too small, since we always used one grid point wrapped in a future. Now, we want to extend the code to use partitions of grid nodes, see Figure **??**. In this example we have a grid with nine nodes and we split them into three partitions, which means that each core has to compute the new values for three elements instead of for one element. The first, thing we need to do is to update the `struct participation`, see Listing 13.2. In Line 4 a `std::vector<double>` is added to store the partition. In Line 8 a constructor is added to initialize the vector `data_(size)` with the provided `size_t size`. In Line 12 a second constructor is added to fill the partition with the initial values and the boundary

Listing 13.1: Serial implementation of the one-dimensional heat equation

```cpp
struct stepper
{
    // Our partition type
    typedef double partition;

    // Our data for one time step
    typedef std::vector<partition> space;

    // Our operator
    static double heat(double left, double middle, double right)
    {
        return middle + (k*dt/(dx*dx)) * (left - 2*middle + right);
    }

    // do all the work on 'nx' data points for 'nt' time steps
    space do_work(std::size_t nx, std::size_t nt)
    {
        // U[t][i] is the state of position i at time t.
        std::vector<space> U(2);
        for (space& s : U)
            s.resize(nx);

        // Initial conditions: f(0, i) = i
        for (std::size_t i = 0; i != nx; ++i)
            U[0][i] = double(i);

        // Actual time step loop
        for (std::size_t t = 0; t != nt; ++t)
        {
            space const& current = U[t % 2];
            space& next = U[(t + 1) % 2];

            next[0] = heat(current[nx-1], current[0], current[1]);

            for (std::size_t i = 1; i != nx-1; ++i)
                next[i] = heat(current[i-1], current[i], current[i+1]);

            next[nx-1] = heat(current[nx-2], current[nx-1], current[0]);
        }

        // Return the solution at time-step 'nt'.
        return U[nt % 2];
    }
};
```



(a) Discrete nodes colored with their initial heat value prescribed by the boundary conditions.



(b) Discrete nodes colored with final heat value at the final time $t = 45$.

Figure 13.4: The initial heat values prescribed by the boundary conditions (a) and the final solution after 45 time steps (b).

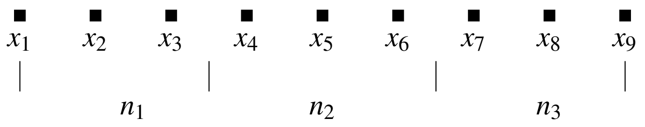


Figure 13.5: Splitting the one-dimensional grid with nine grid nodes $(x_1,\ldots,x_9)$ into three partitions $(n_1,n_2,n3)$ to control the grain size.

values. Since the partition vector is declared as `private`, we use operator overloading in Line 21 and Line 25 to access the elements of the partition. For more details about operation overloading, see Section 1.5.1. In Line 29 a function to obtain the partition size is added.

For the swapping scheme between the time steps for the computation of the temperature, see Figure 13.3, some small notification is applied as well. Figure 13.6 shows the principle of the swapping scheme using partitions. The fundamental principle is the same and we have the two space `U` to swap between the current time step and the future time step. However, with introducing the partitions, we have the two spaces per partition. These modifications are shown in Listing 13.3. In Line 4 the `partition` is now of the type `partition_data`, see Listing 13.2. In Line 10, each of the space vector's size is set to the amount of partitions `np`. In Line 14, we access the space vector `U[0]` for the first time step and with `U[0][i]` each partition is accessed. We call the constructor for each partition and assign the initial values and boundary conditions. The full code is available on GitHub[160].

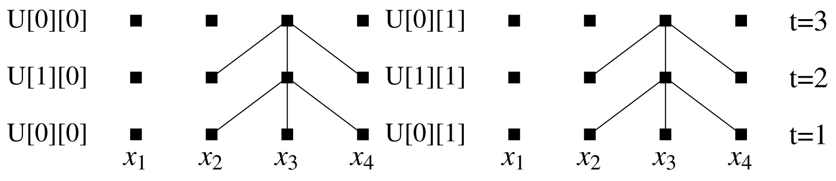


Figure 13.6: Swapping between the partitions using the two spaces `U`. Note that each partition n has his dedicated spaces, however, the fundamental principle stays the same.

Listing 13.2: Serial implementation of the one-dimensional heat equation with grain size control.

```
struct partition_data
{


private:
    std::vector<double> data_;

public:

// Constructor
partition_data(std::size_t size = 0)
      : data_(size)
    {}

partition_data(std::size_t size, double int_value)
      : data_(size)
    {
        double base_value =
              double(int_value * size);
        for (std::size_t i = 0; i != size; ++i)
            data_[i] = base_value + double(i);
    }

//Operator overloading
double& operator[](std::size_t idx) {
     return data_[idx];
  }

double operator[](std::size_t idx) const {
       return data_[idx];
    }

// Util
std::size_t size() const {
       return data_.size();
    }
};
```

Listing 13.3: Serial implementation of the one-dimensional heat equation with grain size control.

```
class stepper
{
    // Our data for one time step
    typedef partition_data partition;
    typedef std::vector<partition> space;

    std::vector<space> U(2);
    for (space& s: U)
        // np is the number of partitions
        s.resize(np);

    // Initial conditions: f(0, i) = i
    for (std::size_t i = 0; i != np; ++i)
        U[0][i] = partition_data(nx, double(i));

    // Return the solution at time-step 'nt'.
    return U[nt % 2];

}
```

13.2 Futurized implementation

HPX provides additional features for asynchronous programming which are not yet in the C++ standard. In this section, we look into these. Therefore, we look into the `struct stepper` of the serial version, see Listing 13.1, and add futures to have asynchronous execution of the solver for the one-dimensional heat equation. Listing 13.4 shows the new `struct stepper` using futures. The first change is that the type `partition`, which was a simple `double` value before, is replaced by `hpx::shared_future`. Note that the `hpx::lcos::future` has the exclusive ownership model and if the future is out of scope, it will be not available anymore. To avoid the out of scope situation, the `hpx::shared_future` has the reference counting ownership model. Here, all references to the object are counted and the object solely goes out of scope if there are zero references. These concepts are equal to `std::unique_ptr`[161] and `std::shared_ptr`[162].

The first feature is the keyword `hpx::make_ready_future`[163], see Line 22 of Listing 13.4. Since the `partition` is now a `hpx::shared_future` the boundary conditions and initial conditions have to be a future too. However, since these are constant values and no computation is needed the future is immediately ready. Since HPX does not know that there is no execution, we can use a `hpx::make_ready_future` ready to propagate this information to the asynchronous execution tree.

Second, since we use futures for the `partition`, but the function to evaluate the central difference `heat(double left, double middle, double right)` takes `double` values as arguments. We either have to change the function to take futures as its arguments and call the `.get()` function inside. To avoid these two things, HPX provides the so-called unwrapping of a function with the keyword `hpx::util::unwrapping`. In Line 24 of Listing 13.4 the function `heat` is unwrapped and the function `Op` takes futures as its arguments. So In

Line 37 of the `hpx::launch::async` we can pass the `current` elements which are of the type `hpx::shared_future<double>` to a function which assumes `double values`.

Third, HPX has the keyword `hpx::dataflow` to use unwrapping for the combination of `hpx::when_all` and `.then`. Imagine you have a vector `std::vector<hpx::lcos::future <int>> futures;` and pass it to `hpx::when_all(futures).then([](auto&& f){});` the vector `futures` will be wrapped in the future `auto&& f`. SO if we want to access the elements of the vector we have to call `f.get()`. An easier approach is to use `hpx::dataflow` as it is done in Line 36 in Listing 13.4. The first argument is `hpx::launch::async` to launch asynchronous and a future is returned. Another possibility is to use `hpx::launch::sync` to launch synchronous. The second argument is the unwrapped function of the `heat` function, see Line 24. The last three remaining arguments are the futures with the values for the central difference scheme evaluation. Before we can return the current solution, we have to call `hpx::when_all` to synchronous all futures of the current solution.

Figure 13.7a shows the execution time of the serial vs the asynchronous implementation for 1 CPU. We clearly see that the execution time even for one CPU is lower. Figure 13.7b shows the execution time for various amount of CPUs for the asynchronous implementation. Here, we can see that for enough grid points the we get some benefit for adding more CPUs which means we have to have enough work to keep the CPUs busy. To obtain better results, we have to extend the code to control its granularity.

13.2.1 Adding grain size control

In Section 13.1.1 the control of the granularity was added to the serial implementation of the one-dimensional heat equation. here, we extend the code with the futurization with the grain size control. First, we extend the `struct partion_data`, see Listing 13.5. In Line 39 we change the class members to a `double []` array and we introduce `size_t size_` to the store the size of the partition. Note that we use a smart pointer `std::unique_ptr` to store the `double []` array. The shared pointer is essential since we need to use the reference counter model to keep track that the array does not go out of scope. For more details about smart pointer we refer to Section 1.11.1. In the constructor, we use the expression `new double [size]` to allocate a double array od the size `size`. Fore more details about the `new` expression, we refer to Section 1.10.1.

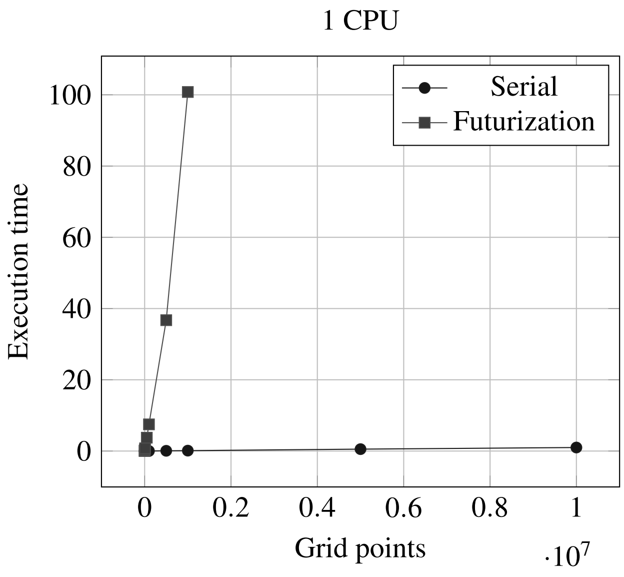
By adding the grain size control to futurized implementation, we see some performance improvement compared to the previous implementation, see Figure 13.7. In Figure 13.8 the number of discrete nodes is fixed to 1000000 and the grain size varies which means the amount of point inside a partition change. For using one and two CPUs, we see the typical curve for the grain size control. Using a grain size of one results in the largest execution time. First, while increasing the grain size the execution times goes down until the so-called sweet spot. At the sweet spot the execution is as its minimum and decreases after. Here, it is important to find this sweet spot which depends on various factors, e.g. the computation, the algorithm, and the architecture of the hardware. So for each amount of discrete nodes one has to find the sweet spot.

13.3 Distributed implementation

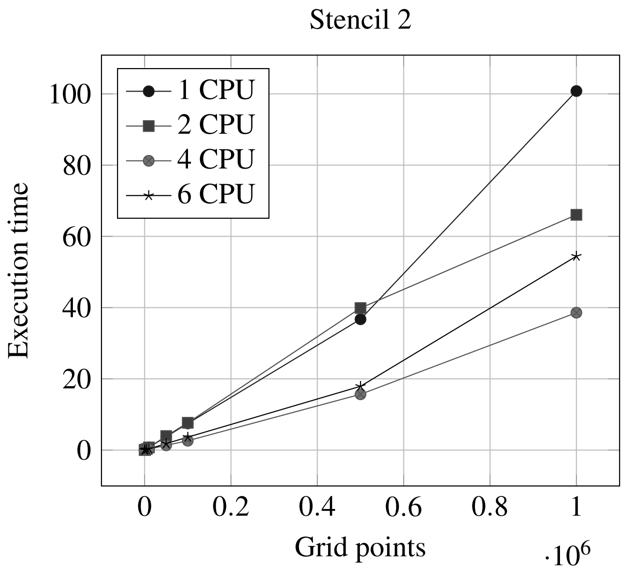
13.3.1 Improving the exchange of partitions

Listing 13.4: Futurized version of the one-dimensional heat equation.

```cpp
struct stepper
{
    // Our partition type
    typedef hpx::shared_future<double> partition;

    // Our data for one time step
    typedef std::vector<partition> space;

    // do all the work on 'nx' data points for 'nt' time steps
    hpx::future<space> do_work(std::size_t nx, std::size_t nt)
    {
        using hpx::dataflow;
        using hpx::util::unwrapping;

        // U[t][i] is the state of position i at time t.
        std::vector<space> U(2);
        for (space& s : U)
            s.resize(nx);

        // Initial conditions: f(0, i) = i
        for (std::size_t i = 0; i != nx; ++i)
            U[0][i] = hpx::make_ready_future(double(i));

        auto Op = unwrapping(&stepper::heat);

        // Actual time step loop
        for (std::size_t t = 0; t != nt; ++t)
        {
            space const& current = U[t % 2];
            space& next = U[(t + 1) % 2];

            // WHEN U[t][i-1], U[t][i], and U[t][i+1] have been computed, THEN we
            // can compute U[t+1][i]
            for (std::size_t i = 0; i != nx; ++i)
            {
                next[i] = dataflow(
                        hpx::launch::async, Op,
                        current[idx(i, -1, nx)], current[i], current[idx(i, +1, nx)]
                    );
            }
        }

        // Return the solution at time-step 'nt'.
        return hpx::when_all(U[nt % 2]);
    }
};
```



(a) Serial vs asynchronous execution



(b) Execution time for various number of CPUs for the asynchronous implementation

Figure 13.7: Comparison of the serial vs asynchronous execution (a) and speed-up for various amount of CPUs (b).

Listing 13.5: Adding the grain size control the futurized one-dimensional heat equation.

```
struct partition_data
{
public:
    explicit partition_data(std::size_t size)
      : data_(new double[size])
      , size_(size)
    {
    }

    partition_data(std::size_t size, double initial_value)
      : data_(new double[size])
      , size_(size)
    {
        double base_value = double(initial_value * size);
        for (std::size_t i = 0; i != size; ++i)
            data_[i] = base_value + double(i);
    }

    partition_data(partition_data&& other) noexcept
      : data_(std::move(other.data_))
      , size_(other.size_)
    {
    }

    double& operator[](std::size_t idx)
    {
        return data_[idx];
    }
    double operator[](std::size_t idx) const
    {
        return data_[idx];
    }

    std::size_t size() const
    {
        return size_;
    }

private:
    std::unique_ptr<double[]> data_;
    std::size_t size_;
};
```

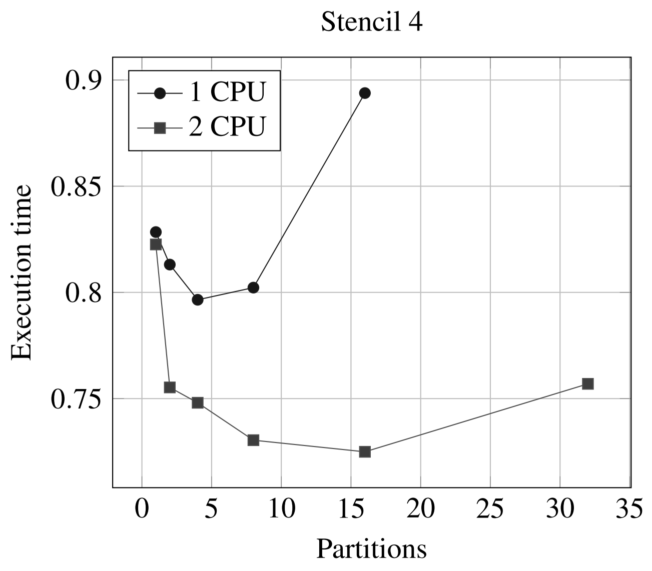


Figure 13.8: Variation of grain size for fixed 1000000 discrete nodes. For one CPU and the two CPUs, we see the typical curve where the execution time goes down, ther sweet spot is reached, and the execution increases after.

Notes

[158]`https://github.com/diehlpkteaching/N-Body`
[159]`https://github.com/STEllAR-GROUP/hpx/blob/master/examples/1d_stencil/1d_stencil_1.cpp`
[160]`https://github.com/STEllAR-GROUP/hpx/blob/master/examples/1d_stencil/1d_stencil_3.cpp`
[161]`https://en.cppreference.com/w/cpp/memory/unique_ptr`
[162]`https://en.cppreference.com/w/cpp/memory/shared_ptr`
[163]`https://hpx-docs.stellar-group.org/latest/html/api.html?highlight=make_ready_future`

VI Linked list

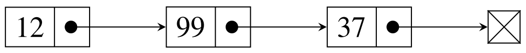


Figure 13.9: A sketch of the linked-list containing three elements. The first pointer points to the second element of the list, the pointer of the second element points to the third element of the list, and the last pointer points to nowhere. This indicates that the end of the list is reached.

This course does not go deep into pointers, since I believe that one should avoid to use pointers a much as possible and use the C++ standard library as much as possible. For more details about the C++ standard library, we refer to Section 3. In this section, we looked into the containers, see Section 3.2, to store values in a `std::vector` or a `std::list`. In this section, we look into the implementation of the `std::list` where pointers are heavily used. We do this for two reasons: 1) To showcase why one should avoid to use pointers and use the containers instead and 2) I believe it is important that you understand how the `std::list` is implemented. In most implementations the `std::list` is implemented as a doubly-linked list. Due to the doubly-linked list, a forward and backward iterator is available. However, for this exercise, we focus on a singly-linked list which relates to the `std::forward_list`[164]. For more details about different types of lists, we refer to [3].

Figure 13.9 sketches the data structure and the usage of pointers. Each element of the list contains a value in this example a integer value and a pointer of the element's type. For example the first element contains the value 12 and point to the second element of the list. The second element contains the value 99 and points to the third element. For any additional element in the list, the same principle would hold, except of the last element of the list which point to nowhere. This is needed to determine the end of the list. Following operations are most commons for lists

- Creating an empty list (`std::list<int> list;`)
- Check if a list is empty (`list.empty();`)
- Prepending an element to a list (`list.push_front(42);`
- Appending an element to a list (`list.push_back(42)`;)
- Getting the first element of the list (`list.front();`
- Accessing the element at a given index
- Deleting the last element (`list.pop_back()`;)
- Deleting the first element (`list.pop_front();`)

As a reference the corresponding methods for the `std::list`[165] are shown. Fore more details, we refer to [61, Chapter 2].

14. Implementation

In this section, we look into the implementation of a singly-linked list using raw C++. Note that this in an exercise to showcase why you should avoid pointers if possible, because handling them will get messy.

Data structures

One list element in Figure 13.9 contains one the data of the type `T` and one pointer to the next element or a `nullptr` at the last element. Listing 14.1 shows the implementation of the list element using a `struct data`. For more details about the struct, we refer to Section 1.6.2. In Line 10 the content of the list is stored in the variable `element`. In Line 12 the pointer `data<T>* next` is used to link to the next list element. Note that the pointer is initialized to `nullptr` since we assume that the element is inserted as the end of the list and points to nowhere. We add one constructor which assigns the value of the element. To make the list generic, we use generic programming and adding the template `typename T`. For more details for generic programming, we refer to Section 1.8. For the initialization of the list, we would use `std::list<double>` using the C++ STL and `data<double> * myList`.

Now, since we have the data structure for the element of the, we need the wrapper class to hide the pointers from the user, as the C++ STL does. Listing 14.1 shows the `struct myList` with a pointer to the struct `struct data` which points to nowhere `nullptr` if the list is empty or to the first element of the list. The first constructor in Line 24 will generate an empty list. The second constructor will generate a list of size one and pointing to the first element. For the example in Figure 13.9 this pointer would point to the element containing the value 12. Now we can generate an empty list `myList<int> mylist;` or a list of size one `myList<int> mylist = myList<int>(42);`. The corresponding expression using the C++ STL are `std::list<int> list;` and `std::list<int> list = 42;`. The easiest function to implement is the `empty()` function which is shown in Line 32. Here, we check if the first pointer is equal to the `nullptr` and if so, the list is empty.

Listing 14.1: Example for a structure for a three dimensional vector.

```
//Struct for the element of a list
template<typename T>
struct data{

data(T in){

element = in;
}

// Element of the type std::list<T>
T element;
// Pointer of type of the class/struct
data<T>* next = nullptr;

};

// Struct for our implementation of the list
template<typename T>
struct myList{

//Pointer to the first element
data<T>* first = nullptr;

myList(){}

myList(T in){

        first = new data<T>(in);

}

bool empty(){

        if (first == nullptr)
                return true;

        return false;
}

};
```

Listing 14.2: Implementation of the push_back function of a linked list.

```
1 void push_front(T element){
2
3 data<T>* tmp = first;
4
5 first = new data<T>(element);
6
7 first->next = tmp;
8
9 }
```

Inserting

Figure 14.1 shows the linked list after the initialization `myList<int> mylist = myList(1);`. Here, we called the contructor in Listing 14.1 and the pointer `data<double>* first` points to the `new data<double>(1)`. Since this element is the last element the pointer `next` is the `nullptr` to indicate that this is the first and last element of the list.

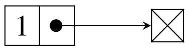


Figure 14.1: The linked list after the initialization `myList<double> mylist = myList<double>(1);`. The pointer `first` point now to the `new data<double>(1)` instead to the `nullptr`.

Inserting at the beginning

Figure 14.2 shows the list after inserting a element at the beginning of the list. In this case the need to manipulate pointer `first` to the first element, see Listing 14.2. In Line 3 we keep a temporary copy `tmp` of the first element. In Line 5 the pointer to the first element points to the new first element. In Line 7 the new first element points to the previous first element which was temporarily stored in the `tmp` pointer.

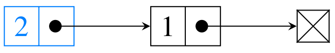


Figure 14.2: The linked list after inserting a new element at the beginning. The pointer `first` point now to `new data(double>(2)` and the pointer `next` of the first element points to the previous first element.

Inserting at the end

Figure 14.3 shows the list after inserting one element at the end of the list. The last element of the list is the element where `next` is the `nullptr`. Listing 14.3 shows the implementation of the `push_back` function. We assign the pointer to the first element to a temporary pointer `tmp` to do not change the pointer to the first element, since we could lose access to the list. To find the last element of the list, the while loop in Line 5 iterates as long as the `next` pointer is not the `nullptr`. If the `next` pointer is the `nullptr` we found the last element. So the `next` pointer points now to the `new data<T>element(2);` and this element becomes the last element. In Figure 14.3 we have the new element colored in blue and the first element points to the second element. The second element points to nowhere.

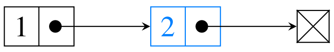


Figure 14.3: The linked list after inserting an element at the end. The pointer `next` of the first element does not point to the `nullptr` and instead points to `new data<double>(2)` and this element point to the `nullptr`, since this is the new last element of the list.

Listing 14.3: Implementation of the `push_back` function of a linked list.

```
void push_back(T element){

data<T>* tmp = first;

while (tmp->next != nullptr)
    tmp = tmp->next;

tmp->next = new data<T>(element);
}
```

Inserting an element

Listing 14.4 shows the implementation of the insertion of an element at a given index. In Line 3 a new element `data<T>* newNode = new data<T>(element);` containing the new element is generated. In Line 8 we have the first special case, since we want to replace the first node. Here, `newNode->next` points to the previous first element which is temporarily stored in `tmp`. Now, the pointer `first` points to the new first node. The next case is that we want to insert at not the first index. Here, we use the `while` loop in Line 17 to find the pointer to the element at the given `index`. Once we found the pointer to the index, we have to check if the pointer is not the `nullptr` which would mean that we want to insert at an index larger as the size of the list, see Line 24. After this check, we can finally insert the new element.

Removing the first element

Listing 14.5 shows the implementation of the `pop_front` function. In Line 3 we check for the case if the first pointer is the `nullptr` and we do not need to delete the first element. In Line 6 the first pointer is stored temporarily in `tmp`, the first pointer points to the second pointer `first-next`, and finally we can delete the `tmp` pointer.

Removing the last element

Figure 14.4 shows the linked list with the last element colored in red which is deleted. Listing 14.6 shows the implementation of the `pop_back` function. In Line 3 the pointer to the first element is stored in the temporary pointer `tmp` and we keep a copy of the previous pointer in the pointer `prev`. In line 6 we search for the element before the last element by using `tmp->next->next` and after the while loop `tmp` points to the second element of the example list. Thus, in Line 10 we can delete the last element `tmp->next` and after that point to the `nullptr` since the second element became the last element.

Listing 14.4: Implementation of the `insert` function of a linked list.

```cpp
void insert(data<T>* first, T element, size_t index){

data<T>* newNode = new data<T>(element);
data<T>* tmp = first;
data<T>* prev = nullptr;

//Case: Replace the head node
if (index == 0 && tmp != nullptr){
    newNode->next = tmp;
    first = newNode;
    return;
}

// Case: search for the node
size_t i = 0;

while(i < index && tmp != nullptr){

    prev = tmp;
    tmp = tmp->next;
    i++;
}

if (tmp == nullptr)
    {
    std::cout << "Index␣" << index << "␣out␣of␣range" << std
       ::endl;
    return;
    }

    prev->next = newNode;
    newNode->next = tmp;
}
```

Listing 14.5: Implementation of the pop_front function of a linked list.

```
void pop_front(){

    if (first == nullptr)
        return;

    // Move the head pointer to the next node
    data<T>* tmp = first;
    first = first->next;

    delete tmp;
}
```

Listing 14.6: Implementation of the pop_back function of a linked list.

```
void pop_back(){

data<T>* tmp = first;
data<T>* prev;

while (tmp->next->next != nullptr){
    tmp = tmp->next;
    }

    delete tmp->next;
    tmp->next = nullptr;
}
```

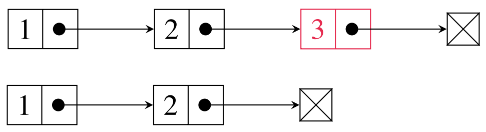


Figure 14.4: In the first row the linked list before the last element was deleted and in the second row the linked list after deleting the last element. The pointer `next` of the second element points now to the `nullptr` since it became the last element. Note that we had to use `delete` to free the memory allocated by the last element.

VII

Appendix

Acknowledgments

First, I like to thank the Louisiana State University's mathematics department for the opportunity to offer this course to a diverse set of students. Second, I would like to thank Dr. Steven R. Brandt for the support with the C++ Explorer to use the Jupyter notebooks to teach C++. Third, I would like thank all the students for providing feedback which helped me to improve the course's materials each semester. Lastly, I would like to thank the following students who contributed to this work: Christoph Larson.

Jupyter notebooks and GitHub classroom

Jupyter notebooks

Some examples for the usage of the Jupyter notebooks are provided here[166] and more details are available here [26].

GitHub classroom

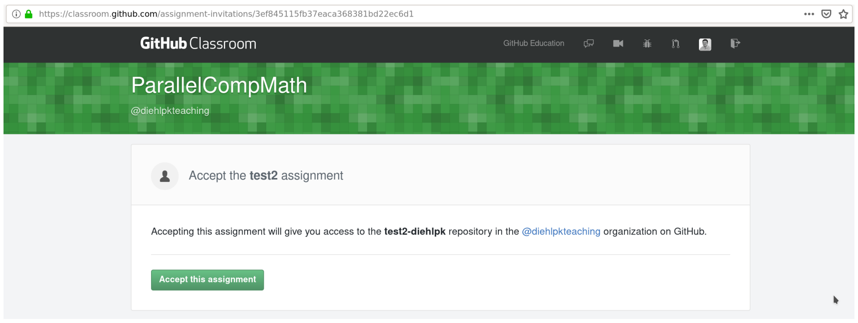
We use GitHub classroom[167] to submit the assignments. In this section, we go through the steps to submit the code to GitHub using git[168]. For a brief overview of the most common git commands, we refer to this cheat sheet[169] and for more details to [67, 92]. The first step to submit the assignments is to get your GitHub[170] account. We recommend to use a user name reflecting your name. If you want to use your local computer to submit the assignments, you have to install git on your computer to follow the following instructions. Note that git is installed on the course's web server, so we recommend to submit from there. Open a terminal on the course's web server and type `git config --global user.name Surname Name` to set your Surname and Name, so one can see who submitted the assignment. Optional you can set your e-mail address using `git config --global user.email you@provider.com`.

If you do not want to enter your password every time to use git, you can generate a ssh key[171] as shown in Listing 14.7. We use the command `ssh-keygen` to generate the public and private key. It is common practice that the ssh-key is related to your e-mail address. We save the private key as `/.ssh/id-rsa-github` and the public key as `/.ssh/id-rsa-github.pub`. To avoid entering the password each time we do a commit to the assignment, we type `ssh-add ~/.ssh/id-rsa-github` to add the key to our key ring. Note that you have to add the content of your public key to GitHub by clicking on Profile -> SSH keys and GPG keys -> New SSH key.

For each assignment, you will get an e-mail and should click on the link there, see

Listing 14.7: Setting up a ssh key

```
ssh-keygen -t rsa -C "you@provider.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/diehlpk/.ssh/id-
    rsa): ~/.ssh/id-rsa-github
ssh-add ~/.ssh/id-rsa-github
```

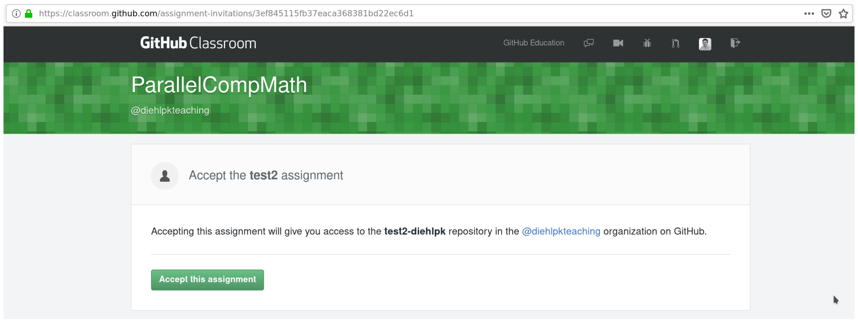

https://classroom.github.com/assignment-invitations/3ef845115fb37eaca368381bd22ec6d1

ParallelCompMath
@diehlpkteaching

Accept the **test2** assignment

Accepting this assignment will give you access to the **test2-diehlpk** repository in the @diehlpkteaching organization on GitHub.

(a) Invitation for the assignment on GitHub.


https://classroom.github.com/assignment-invitations/3ef845115fb37eaca368381bd22ec6d1

ParallelCompMath
@diehlpkteaching

Accept the **test2** assignment

Accepting this assignment will give you access to the **test2-diehlpk** repository in the @diehlpkteaching organization on GitHub.

(b) Confirmation of the acceptance and the link to submit your assignment.

Figure 14.5: Accepting assignments on GitHub classroom.

Figure 14.5a, and accept the assignment, see Figure 14.5b. After accepting the assignment you see a link which will be used to submit your assignment. Listing 14.8 shows how to submit your code to this assignment. Note that you will get a new invitation for each assignment. First, you use `git clone` to clone the repository and after that you change the directory using the command `cd`. For each file, you like to submit you run the command `git add`. Note that you have to do this only once. Using the command `git commit -a` you commit all files and with the command `git push` you send them to the server, so the instructor can see and grade them.

Listing 14.8: Setting up a ssh key

```
git clone https://github.com/diehlpkteaching/test-diehlpk.git
cd test-diehlpk
touch exercise.cpp
git add exercise.cpp
# Work on your exercise
git commit -a
git push
```

Notes

[164]https://en.cppreference.com/w/cpp/container/forward_list
[165]https://en.cppreference.com/w/cpp/container/list
[166]https://github.com/diehlpk/gateways2020
[167]https://classroom.github.com
[168]https://git-scm.com/
[169]https://education.github.com/git-cheat-sheet-education.pdf
[170]https://github.com/
[171]https://www.ssh.com/ssh/key/

Bibliography

Articles

[8] D. A. Bader. “Evolving MPI+X Toward Exascale”. In: *Computer* 49.08 (Aug. 2016), pages 10–10. ISSN: 1558-0814. DOI: `10.1109/MC.2016.232` (cited on page 84).

[9] Josh Barnes and Piet Hut. “A hierarchical O (N log N) force-calculation algorithm”. In: *nature* 324.6096 (1986), page 446 (cited on page 115).

[15] James R Bunch and John E Hopcroft. “Triangular factorization and inversion by fast matrix multiplication”. In: *Mathematics of Computation* 28.125 (1974), pages 231–236 (cited on page 92).

[18] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos”. In: *J. Parallel Distrib. Comput.* 74.12 (Dec. 2014), pages 3202–3216. ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2014.07.003`. URL: `https://doi.org/10.1016/j.jpdc.2014.07.003` (cited on page 84).

[19] André-Louis Cholesky. “Sur la résolution numérique des systèmes déquations linéaires”. In: *Bulletin de la Sabix. Société des amis de la Bibliothèque et de l’Histoire de l’École polytechnique* 39 (2005), pages 81–95 (cited on page 92).

[27] Edsger W Dijkstra. “Over de sequentialiteit van procesbeschrijvingen”. In: *Trans. by Martien van der Burgt and Heather Lawrence. In* (1962) (cited on page 63).

[29] Ralph Duncan. “A survey of parallel computer architectures”. In: *Computer* 23.2 (1990), pages 5–16 (cited on page 73).

[33] Michael J Flynn. “Some computer organizations and their effectiveness”. In: *IEEE transactions on computers* 100.9 (1972), pages 948–960 (cited on page 73).

[35] David Goldberg. “What every computer scientist should know about floating-point arithmetic”. In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pages 5–48 (cited on pages 13, 38).

[45] Thomas Heller. “Extending the C++ Asynchronous Programming Model with the HPX Runtime System for Distributed Memory Computing”. In: (2019) (cited on page 55).

[46] Thomas Heller et al. “Hpx–an open source c++ standard library for parallelism and concurrency”. In: *Proceedings of OpenSuCo* (2017), page 5 (cited on page 55).

[47] Thomas Heller et al. “Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars”. In: *The International Journal of High Performance Computing Applications* 33.4 (2019), pages 699–715 (cited on page 55).

[50] “IEEE Standard for Floating-Point Arithmetic”. In: *IEEE Std 754-2008* (Aug. 2008), pages 1–70. DOI: `10.1109/IEEESTD.2008.4610935` (cited on pages 13, 38).

[51] K. Iglberger et al. “Expression Templates Revisited: A Performance Analysis of Current Methodologies”. In: *SIAM Journal on Scientific Computing* 34.2 (2012), pages C42–C69. DOI: `10.1137/110830125`. eprint: `https://doi.org/10.1137/110830125`. URL: `https://doi.org/10.1137/110830125` (cited on page 89).

[57] Hartmut Kaiser et al. “HPX - The C++ Standard Library for Parallelism and Concurrency”. In: *Journal of Open Source Software* 5.53 (2020), page 2352. DOI: `10.21105/joss.02352`. URL: `https://doi.org/10.21105/joss.02352` (cited on page 55).

[62] A Koenig. “Standard–the C++ language. Report ISO/IEC 14882: 1998”. In: *Information Technology Council (NCTIS)* (1998) (cited on page 12).

[63] Dirk P Kroese et al. “Why the Monte Carlo method is so important today”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 6.6 (2014), pages 386–392 (cited on page 109).

[70] Makoto Matsumoto and Takuji Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pages 3–30 (cited on page 36).

[72] Peter J Mohr, David B Newell, and Barry N Taylor. “CODATA recommended values of the fundamental physical constants: 2014”. In: *Journal of Physical and Chemical Reference Data* 45.4 (2016), page 043102 (cited on page 113).

[77] Michael L Parks et al. “Implementing peridynamics within a molecular dynamics code”. In: *Computer Physics Communications* 179.11 (2008), pages 777–783 (cited on page 119).

[79] Michael J Quinn. “Parallel programming”. In: *TMH CSE* 526 (2003) (cited on page 73).

[80] Chittoor V Ramamoorthy and Hon Fung Li. “Pipeline architecture”. In: *ACM Computing Surveys (CSUR)* 9.1 (1977), pages 61–102 (cited on page 72).

[82] Dennis M Ritchie. “The development of the C language”. In: *ACM Sigplan Notices* 28.3 (1993), pages 201–208 (cited on page 11).

[84] Karl Rupp et al. “ViennaCL—Linear Algebra Library for Multi-and Many-Core Architectures”. In: *SIAM Journal on Scientific Computing* 38.5 (2016), S412–S439 (cited on page 89).

[86] Conrad Sanderson and Ryan Curtin. “Armadillo: a template-based C++ library for linear algebra”. In: *Journal of Open Source Software* (2016) (cited on page 89).

[90] Stewart A Silling. “Reformulation of elasticity theory for discontinuities and long-range forces”. In: *Journal of the Mechanics and Physics of Solids* 48.1 (2000), pages 175–209 (cited on page 117).

[91] Stewart A Silling and Ebrahim Askari. “A meshfree method based on the peridynamic model of solid mechanics”. In: *Computers & Structures* 83.17-18 (2005), pages 1526–1535 (cited on page 117).

[93] Richard Smith. "ISO/IEC 14882: 2017 Information technology—Programming languages—C++". In: *International Organization for Standardization* (2017) (cited on page 12).

[94] Ian Sommerville. "Software engineering 9th Edition". In: *ISBN-10* 137035152 (2011) (cited on page 19).

[100] Peter Thoman et al. "A taxonomy of task-based parallel programming technologies for high-performance computing". In: *The Journal of Supercomputing* 74.4 (2018), pages 1422–1434 (cited on page 55).

[102] Maarten Van Steen and A Tanenbaum. "Distributed systems principles and paradigms". In: *Network* 2 (2002), page 28 (cited on page 83).

Books

[1] Sverre Aarseth, Christopher Tout, and Rosemary Mardling. *The Cambridge N-body lectures*. Volume 760. Springer, 2008 (cited on page 114).

[2] Sverre J Aarseth. *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003 (cited on page 114).

[3] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996 (cited on page 137).

[4] Douglas Adams. *The Ultimate Hitchhiker's Guide to the Galaxy*. Volume 6. Pan Macmillan, 2017 (cited on page 63).

[6] Koenig Andrew. *Accelerated C++: practical programming by example*. Pearson Education India, 2000 (cited on pages 11, 15, 18–21, 23, 35, 43, 48).

[7] Paul Bachmann. *Die analytische zahlentheorie*. Volume 2. Teubner, 1894 (cited on page 115).

[10] Richard Barrett et al. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994 (cited on page 102).

[11] Norman Biggs, Norman Linstead Biggs, and Biggs Norman. *Algebraic graph theory*. Volume 67. Cambridge university press, 1993 (cited on page 92).

[12] William L Briggs, Steve F McCormick, et al. *A multigrid tutorial*. Volume 72. Siam, 2000 (cited on page 98).

[16] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997 (cited on page 80).

[17] John Rozier Cannon. *The one-dimensional heat equation*. 23. Cambridge University Press, 1984 (cited on page 123).

[23] Scott Craig. *LProfessional CMake: A Practical Guide*. Volume 6. Crascit, 2018 (cited on page 25).

[28] Allen Downey. *The little book of semaphores*. Green Tea Press, 2008 (cited on page 63).

[30] Leonhard Euler. *Institutionum calculi integralis*. Volume 1. impensis Academiae imperialis scientiarum, 1824 (cited on page 116).

[32] Richard William Farebrother. *Linear least squares computations*. Marcel Dekker, Inc., 1988 (cited on page 97).

[34] Angelo Christopher Gilli. *Binary arithmetic and Boolean algebra*. McGraw-Hill, 1965 (cited on page 13).

[36] Ananth Grama et al. *Introduction to parallel computing*. Pearson Education, 2003 (cited on page 71).

[37] William Gropp et al. *Using MPI: portable parallel programming with the message-passing interface*. Volume 1. MIT press, 1999 (cited on page 83).

[40] Patricia A Grubel. *Dynamic Adaptation in HPX: A Task-based Parallel Runtime System*. New Mexico State University, 2016 (cited on page 55).

[43] Morton E Gurtin. *An introduction to continuum mechanics*. Academic press, 1982 (cited on page 118).

[44] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010 (cited on page 74).

[48] Magnus Rudolph Hestenes and Eduard Stiefel. *Methods of conjugate gradients for solving linear systems*. Volume 49. 1. NBS Washington, DC, 1952 (cited on page 98).

[49] Andrew Hunt. *The pragmatic programmer*. Pearson Education India, 1900 (cited on page 19).

[53] Nicolai M Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2003 (cited on page 26).

[54] Nicolai M Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012 (cited on page 35).

[58] Brian W Kernighan and Phillip James Plauger. *The elements of programming style*. 1974 (cited on page 12).

[59] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006 (cited on page 11).

[60] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with threads*. Sun Soft Press Mountain View, 1996 (cited on page 80).

[61] Donald Ervin Knuth. *The art of computer programming: Fundamental Algorithms*. Volume 1. Pearson Education, 1968 (cited on pages 44, 115, 137).

[64] Vipin Kumar et al. *Introduction to parallel computing*. Volume 110. Benjamin/Cummings Redwood City, CA, 1994 (cited on page 71).

[66] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. Volume 1. , 2000 (cited on page 115).

[67] Brent Laster. *Professional git*. John Wiley & Sons, 2016 (cited on page 151).

[68] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. Volume 98. Siam, 2007 (cited on pages 116, 124).

[69] I-Shih Liu. *Continuum mechanics*. Springer Science & Business Media, 2013 (cited on page 118).

[71] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013 (cited on page 44).

[73] Cameron Newham and Bill Rosenblatt. *Learning the bash shell: Unix shell programming*. " O'Reilly Media, Inc.", 2005 (cited on page 12).

[74] Isaac Newton. *Philosophiae naturalis principia mathematica*. Volume 1. G. Brookman, 1833 (cited on page 114).

[75] Arthur O'dwyer. *Mastering the C++ 17 STL: Make full use of the standard library components in C++ 17*. Packt Publishing Ltd, 2017 (cited on page 41).

[76] Maxim A Olshanskii and Eugene E Tyrtyshnikov. *Iterative methods for linear systems: theory and applications*. SIAM, 2014 (cited on page 98).

[78] William H Press et al. *Numerical recipes in Fortran 77: volume 1, volume 1 of Fortran numerical recipes: the art of scientific computing*. Cambridge university press, 1992 (cited on page 92).

[81] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*. Volume 42. John Wiley & Sons, 2005 (cited on page 74).

[83] Arnold Robbins. *Bash Pocket Reference: Help for Power Users and Sys Admins*. " O'Reilly Media, Inc.", 2016 (cited on page 12).

[85] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010 (cited on page 84).

[87] John T Scheick. *Linear algebra with applications*. Volume 81. McGraw-Hill New York, 1997 (cited on pages 89, 94).

[89] Ronald W Shonkwiler and Franklin Mendivil. *Explorations in Monte Carlo Methods*. Springer Science & Business Media, 2009 (cited on page 109).

[92] Richard E Silverman. *Git Pocket Guide: A Working Introduction*. "O'Reilly Media, Inc.", 2013 (cited on page 151).

[95] Alexander Stepanov and Meng Lee. *The standard template library*. Volume 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995 (cited on page 41).

[96] John C Strikwerda. *Finite difference schemes and partial differential equations*. Volume 88. Siam, 2004 (cited on pages 116, 124).

[98] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000 (cited on page 12).

[99] Bjarne Stroustrup. *Programming: principles and practice using C++*. Pearson Education, 2014 (cited on page 11).

[101] Roman Trobec et al. *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer, 2018 (cited on page 71).

[104] Steven Watts. *The people's tycoon: Henry Ford and the American century*. Vintage, 2009 (cited on page 72).

[105] Anthony Williams. *C++ concurrency in action : practical multithreading*. Shelter Island, NY: Manning, 2012. ISBN: 9781933988771 (cited on page 79).

In proceedings

[5] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pages 483–485 (cited on page 71).

[14] Jerzy Brzezinski, J-M Helary, and Michel Raynal. "Deadlocks in distributed systems: request models and definitions". In: *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*. IEEE. 1995, pages 186–193 (cited on page 84).

[24] John Crank and Phyllis Nicolson. "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type". In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Volume 43. 1. Cambridge University Press. 1947, pages 50–67 (cited on page 116).

[25] Gregor DaiSS et al. “From piz daint to the stars: simulation of stellar mergers using high-level abstractions”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pages 1–37 (cited on pages 55, 84).

[26] Patrick Diehl and Steven R. Brandt. “Interactive C++ code development using C++Explorer and GitHub Classroom for educational purposes”. In: *Proceedings of Gateways 2020*. Science Gateways Community Institute (SGCI), 2020, page 5. DOI: 10.17605/OSF.IO/QBTJ3 (cited on page 151).

[31] Xin Gui Fang and George Havas. “On the worst-case complexity of integer gaussian elimination”. In: *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*. 1997, pages 28–31 (cited on page 98).

[38] Patricia Grubel et al. “The performance implication of task size for applications on the hpx runtime system”. In: *2015 IEEE International Conference on Cluster Computing*. IEEE. 2015, pages 682–689 (cited on page 57).

[39] Patricia Grubel et al. “Using intrinsic performance counters to assess efficiency in task-based parallel applications”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2016, pages 1692–1701 (cited on page 55).

[41] Paul Grun et al. “A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pages 34–39. DOI: 10.1109/HOTI.2015.19 (cited on page 84).

[52] K. Iglberger et al. “High performance smart expression template math libraries”. In: *2012 International Conference on High Performance Computing Simulation (HPCS)*. July 2012, pages 367–373. DOI: 10.1109/HPCSim.2012.6266939 (cited on page 89).

[55] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. “Parallex an advanced parallel execution model for scaling-impaired applications”. In: *2009 International Conference on Parallel Processing Workshops*. IEEE. 2009, pages 394–401 (cited on page 65).

[56] Hartmut Kaiser et al. “Hpx: A task based programming model in a global address space”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM. 2014, page 6 (cited on page 66).

[65] Gabriel Laberge et al. “Scheduling Optimization of Parallel Linear Algebra Algorithms Using Supervised Learning”. In: *2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*. IEEE. 2019, pages 31–43 (cited on page 57).

[103] Qian Wang et al. “AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs”. In: *SC’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2013, pages 1–12 (cited on page 89).

Index

Listings

List of Figures

List of video lectures

- C++ Lecture 1 - The Standard Template Library
- C++ Lecture 2 - Template Programming
- C++ Lecture 4 - Template Meta Programming
- The C++17 Parallel Algorithms Library and Beyond
- Modern CUDA and C++
- The C++17 Parallel Algorithms Library and Beyond
- The Asynchronous C++ Parallel Programming Model
- Nonlocality in Peridynamics