Patrick Diehl
Steven R. Brandt
Hartmut Kaiser

Parallel C++

Efficient and Scalable High-Performance Parallel Programming Using HPX

Springer

Parallel C++

Patrick Diehl • Steven R. Brandt • Hartmut Kaiser

Parallel C++

Efficient and Scalable High-Performance Parallel Programming Using HPX

Springer

Patrick Diehl
Center for Computation and Technology
Louisiana State University
Baton Rouge, LA, USA

Steven R. Brandt
Center for Computation and Technology
Louisiana State University
Baton Rouge, LA, USA

Hartmut Kaiser
Center for Computation and Technology
Louisiana State University
Baton Rouge, LA, USA

ISBN 978-3-031-54368-5 ISBN 978-3-031-54369-2 (eBook)
https://doi.org/10.1007/978-3-031-54369-2

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Dr. Diehl dedicates this book to his lovely wife and two exceptional children. He is grateful for the lifelong support of his mother and his sister.

Dr. Brandt dedicates this book to Dr. Lee G. McKnight and Dr. Steve McKnight. They gave me my start in computational science at Murry Hill labs when I was an undergraduate student.

Dr. Kaiser dedicates this book to all of the developers worldwide that have contributed to HPX, and have provided ideas, guidance, and inspiration over the years.

Foreword

Parallelism is the final frontier of computing.

By default, we think sequentially. Parallelism and asynchrony are often seen as challenging and complex. Tools to be used sparingly and cautiously, and only by experts.

But we must shatter these assumptions, for today, we live in a parallel world. Almost every hardware platform is parallel, from the smallest embedded devices to the largest supercomputers. We must explicitly program for that parallelism to achieve the performance we need.

We must change our mindset. Anyone who writes code has to think in parallel. Parallelism must become our default. It must be intuitive and easy to craft correct and performant asynchronous parallel software.

HPX is unique among its peers. It is both a parallel programming philosophy grounded in decades of research and a production software framework developed with the best software engineering practices. It strikes the perfect balance of ease-of-use and performance. Its underlying tenets—such as sending work to data, avoiding or localizing synchronization, and hiding latency—are intuitive and far reaching.

In addition, HPX is intimately tied to Standard C++, the predominant language for parallel programming. It has both influenced the evolution of C++ and been inspired by it. This alignment with the standard makes it seamless and natural to use HPX in C++ software.

This book will teach you how to concretely use HPX. But it will also teach you how to think in parallel. Embracing the HPX mindset will transform how you approach software engineering and computational science.

I began my career working on HPX, and the HPX paradigm continues to influence how I design software every day. I hope it will have the same impact on you.

New York, NY, USA — Bryce Adelstein Lelbach

Preface

Many scientific high performance codes are based on the C++ programming language. For example, in a National Energy Research Scientific Computing Center (NERSC) survey, C++ was the second most used programming language. In European supercomputer centers, courtesy of Partnership for Advanced Computing in Europe (PRACE), showed that C/C++ is the second most used programming language [1]. In a nature survey, "Ten computer codes that transformed science" [2] in 2021 listed FORTRAN to the top. In a second survey, they asked missing choices in the initial top ten, and C++ was ranked third. To conclude, C++ is essential for high-performance computing. A common approach for distributed C++ codes is *MPI+OpenMP* where MPI is used for the communication and OpenMP for the node level parallelism. However, OpenMP is `#pragma`-based and introduces the OpenMP standard on top of the C++ standard. However, with the C++ 17 and C++ 20, the features provided by OpenMP are available in the C++ standard. The 2022 *JET BRAINS* indicates that 41% regularly use the C++ 17 standard and 23% the C++ 20 standard. This book aims to showcase features of the C++ 17 and C++ 20, focusing on high-performance computing. These features can significantly simplify parallel computations using multiple cores. These features' simplified usage of asynchronous programming and dynamic parallelism is an important characteristic. The current C++ standard only allows programming on a single shared memory device. The same programming style in a distributed environment using the C++ standard library for parallelism and concurrency (HPX) will be discussed. The book starts with the single-threaded implementation of the fractal sets, *e.g.* Julia set and Mandelbrot set, using the C++ Standard Library (SL)'s container and algorithms. This code base is used for parallel implementation using low-level threads, asynchronous programming, parallel algorithms, and coroutines. The asynchronous programming will be extended for distributed programming using the C++ standard library for parallelism and concurrency (HPX). Octo-Tiger, an astrophysics code for stellar merger, showcases a portable, efficient, and scalable high-performance application using HPX. The book's audience is advanced undergraduate and graduate students who want to learn the basics of parallel and distributed C++ programming but are not computer science majors. Basic C++

knowledge, like functions, classes, loops, and conditional statements, is assumed as a requirement. Some mathematical methods benefit the understanding, like complex numbers for the fractal set or Taylor expansion. C++ advanced topics, like generic programming, lambda functions, smart pointers, and move semantics, are briefly summarized in the appendix. However, this book is unsuitable as an introduction to C++, and prior essential C++ experience is strongly recommended. We hope to showcase that with the parallel features in the C++ 17 and C++ 20 standards, learning meta-languages, like OpenMP, is unnecessary. Using the C++ standard library for parallelism and concurrency (HPX), the same language features can be extended to distributed codes, obviating the need to learn the Message Passing Interface (MPI). We thank Springer Computer Science for the opportunity to publish this book. We are grateful for the support of Ralf Gerstner during the publication process.

Baton Rouge, LA, USA
January, 2024

Patrick Diehl
Steven R. Brandt
Hartmut Kaiser

Acknowledgments

Dr. Diehl thanks the Department of Mathematics, and the Department of Physics and Astronomy at Louisiana State University for the opportunity to teach the courses the content of this book is partially based on. In addition, he thanks the students for their feedback on the course's content and the course notes. Special thanks go to Christoph Larson for proof-reading the course notes. All authors thank the participants of the short course at 16th U.S. National Congress on Computational Mechanics, 15th World Congress on Computational Mechanics, and 17th U.S. National Congress on Computational Mechanics for providing feedback on the material of this book covered in the half-day training. We would like to thank Alireza Kheirkhahan for the administration of the *Rostam* cluster where we conducted the benchmarks for this book. The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance *Ookami* computing system, which was made possible by a $5M National Science Foundation grant (#1927880). This research used resources of the National Energy Research Scientific Computing Center, the U.S. Department of Energy, Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. This work used computational resources of the Supercomputer Fugaku provided by RIKEN through the HPCI System Research Project (Project ID: hp210311). This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID s1078. Last, we would like to thank Gregor Daiß for the development on HPX's Kokkos features.

Contents

Part I Preliminaries

1 Compiling and Running the Code and Examples in This Book 3
1.1 Using the C++ Explorer 3
1.2 Using CMake and C++ Compiler 4

Part II Introduction to C++ and C++ Standard Library

2 About C++, C++ Standard, and the C++ Standard Library 11
2.1 Brief History of C++, the C++ Standard, and Parallel Programming 11
2.2 Standard Template Library (STL) and C++ Standard Library 14
2.3 C++ Compilers 15

3 C++ Standard Library 17
3.1 Overview of the C++ Standard Library 17
3.2 Containers 19
3.2.1 Vector 20
3.2.2 List 21
3.2.3 List vs. Vector 23
3.2.4 Array 25
3.2.5 Iterators 26
3.3 Algorithms 29

4 Example Mandelbrot Set and Julia Set 33
4.1 Mandelbrot Set 33
4.2 Julia Set 34
4.3 Single Threaded Implementation of the Mandelbrot Set 34

Part III The C++ Standard Library for Concurrency and Parallelism (HPX)

5 Why HPX? 43
5.1 Governing Principles 44
5.1.1 Focus on Latency Hiding Instead of Latency Avoidance 45
5.1.2 Embrace Fine-Grained Parallelism Instead of Heavyweight Threads 45
5.1.3 Rediscover Constraint-Based Synchronization to Replace Global Barriers 46
5.1.4 Adaptive Locality Control Instead of Static Data Distribution 46
5.1.5 Prefer Moving Work to the Data Over Moving Data to the Work 47
5.1.6 Favor Message Driven Computation Over Message Passing 48

6 The C++ Standard Library for Parallelism and Concurrency (HPX) 49
6.1 HPX's Architecture 49
6.2 Applications 53

Part IV Parallel Programming

7 Parallel Programming 59
7.1 An Overview of Parallel Programming 59
7.2 Race Conditions 63
7.2.1 Mutexes and Deadlocks 65
7.2.2 Atomic Operation 68
7.3 Performance Measurements 69
7.3.1 Amdahl's Law 69
7.3.2 Gustafson's Law 71
7.3.3 Speedup and Parallel Efficiency 72
7.3.4 Weak Scaling and Strong Scaling 74
7.4 Memory Access 75
7.5 Parallelism Computer Architectures 75
7.5.1 Pipelined SIMD 76
7.5.2 Single Instruction Multiple Data (SIMD) 77

8 Programming with Low Level Threads 79
8.1 Implementation of the Fractal Sets 80

9 Asynchronous Programming 85
9.1 Advanced Synchronization in HPX 89
9.2 Implementation of the Fractal Sets 93

10 Parallel Algorithms 99
10.1 Parallel Algorithms in HPX 100
10.1.1 Combining Parallel Algorithms and Asynchronous Programming 101
10.1.2 Single Instruction Multiple Data 102
10.2 Additional Parameters for the Execution Policies 104
10.3 Implementation of the Fractal Sets 104

11 Coroutines 109
11.1 Implementation of the Fractal Sets 113

12 Benchmarking the Fractal Set Codes 117

Part V Distributed Programming

13 Distributed Computing and Programming 123
13.1 Overview of Distributed Programming in C++ and Asynchronous Many Task Systems 124
13.2 Data Distribution 127
13.3 Distributed Input and Output 129
13.4 Serialization 129
13.5 Message Passing 131
13.5.1 Implementation of the Fractal Set Using MPI 133
13.5.2 Implementation of the Fractal Set Using MPI+OpenMP 140
13.6 Benchmark of MPI and MPI+OpenMP 144

14 Distributed Programming Using HPX 147
14.1 Active Messaging 148
14.1.1 Plain Action 149
14.1.2 Components and Actions 150
14.1.3 Receiving Topology Information 156
14.2 Serialization 159

15 Examples of Distributed Programming 163
15.1 Distributed Implementation of the Taylor Series of the Natural Logarithm 163
15.2 Distributed Implementation of the Fractal Set 169
15.3 Improved Distributed Implementation of the Fractal Set 173
15.4 Benchmark of the Distributed Implementation of the Fractal Sets 174

16 Some Remarks on MPI+OpenMP and HPX 179

Part VI A Showcase for a Portable High Performance Application Using HPX

17 Accelerator Cards 185

18 Octo-Tiger, a Showcase for a Portable High Performance Application 187
18.1 Synchronous Communication vs. Asynchronous Communication 189
18.2 Acceleration Support 189
18.3 HPX-Kokkos and Vectorization 192

Part VII Conclusion and Outlook

19 Conclusion and Outlook 197
19.1 Distributed Programming 198
19.2 Outlook 200

Appendix 203

A Advanced Topics in C++ 205
A.1 Generic Programming 206
A.2 Lambda Functions 207
A.3 Move Semantics 209
A.4 Placement New 210
A.5 Smart Pointers 211
A.6 Ranges 214

B Supplementary Header Files 217

C Software and Hardware Documentation 223

References 225

Glossary 233

Index 235

Acronyms

AGAS	Active Global Address Space
ALUs	Arithmetic Logic Units
AMR	Adaptive Mesh Refinement
ANSI	American National Standards Institute
APEX	Autonomic Performance Environment for Exascale
APGAS	Asynchronous Partitioned Global Address Space
API	Application Programming Interface
ARB	OpenMP Architecture Review Board
BLAS	Basic Linear Algebra Subroutines
BSP	Bulk Sequential Parallelism
CAF	Coarray FORTRAN
COCOMO	**Co**nstructive **Co**st **Mo**del
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUPTI	CUDA Profiling Tools Interface
CVS	Comma-separated Values
DVS	Delimiter-Separated Values
EPI	European Processor Initiative
EVE	Expressive Vector Engine
FLOPS	Floating Point Operations Per Second
FMM	Fast Multipole Method
FPGAs	Field Programmable Gate Arrays
GASNet	**G**lobal-**A**ddress **S**pace **Net**working
GID	**G**lobal **Id**entifier
GOS	Global Object Space
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HDF	Hierarchical Data Format
HIP	Heterogeneous Interface for Portability
HP	Hewlett-Packard
HPE	Hewlett Packard Enterprise

HPX	C++ Standard Library for parallelism and concurrency
HSFC	Space Filling Curve Partitioning
IBM	International Business Machines
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
LSU	Louisiana State University
MIMD	Multiple Instruction and Multiple Data
MISD	Multiple Instruction and Single Data
MPI	Message Passing Interface
MTBF	Mean Time Between Failures
Mutex	**Mut**ual **Ex**clusion
NetCDF	Network Common Data Form
OpenCL	Open Compute Language
OpenGL	Open Graphics Library
OpenMP	Open Multi-Processing
PAPI	Performance Application Programming Interface
Parcel	**Par**allel **C**ontrol **El**ement
PBM	Portable Bitmap File Format
PETSc	Portable, Extensible Toolkit for Scientific Computation
PGAS	Partitioned Global Address Space
PnetCDF	Parallel-NetCDF
POSIX	Portable Operating System Interface
PPL	Parallel Patterns Library
PU	Processing Units
PUPS	Pixel Updates Per Second
RCB	Recursive Coordinate Bisection
RDMA	Remote Direct Memory Access
RISC	Reduced Instruction Set Computer
RMA	Remote Memory Access
SDK	Software Developer Kit
SIMD	Single Instruction Multiple Data
SISD	Single Instruction and Single Data
SL	Standard Library
STL	Standard Template Library
SVE	Scalable Vector Extension
TBB	Threading Building Blocks
TCP	Transmission Control Protocol
UMA	Uniform Memory Access
VTK	The Visualization Toolkit
XML	Extensible Markup Language

Part I
Preliminaries

In this section, we start with the preliminaries of the book. First, we prepare the reader to compile and run the code snippets and the examples of this book. In addition to reading the text, we strongly advise the reader to download the code on GitHub® and experiment with it.

Why? Consider this analogy. Here, at Louisiana State University (LSU), college football is a major event. We even have a tiger, Mike *VII*, living on campus. Each semester we have taught C++ programming to LSU students, we asked them whether they think one can become a top athlete by just studying the rules and watching games or tailgating. All agree that a player has to go onto the football field and play the game in order to acquire any skill. This is not specific to football and holds for every sport from golf to chess.

Then we tell them that it is the same with coding. To become a better programmer, one needs to "go onto the field" and write code. Just like football or any sport, programming is learned by doing to write better code.

So we encourage the reader to access all the code snippets and examples on GitHub®. We provide a Docker image with all dependencies installed and the reader can run all examples in the book within the Docker image using plain C++ code or the C++ Explorer to run the examples in within Jupyter notebooks with the Cling extension.

Chapter 1
Compiling and Running the Code and Examples in This Book

We provide two ways to run the code examples in the book. First, we provide the option to run the examples within Jupyter notebooks with the Cling extension to run C++ code in the notebook cells. The authors developed a tool, the C++ Explorer, to make C++ programming more accessible to students without the struggle to install a C++ compiler or Integrated Development Environment (IDE). For more details about using the C++ Explorer and GitHub® classroom for teaching C++ and HPX, we refer to [3]. Second, the option to use CMake and your favorite C++ compiler. The Jupyter notebooks and plain source files for all examples in the book are available on GitHub®.[1]

1.1 Using the C++ Explorer

All the code in this book runs on the C++ Explorer, an open source environment maintained on GitHub®.[2] The interested student can build this environment using the `docker-compose.build.yml` file provided inside the repository, or may download the image `stevenrbrandt/cxxex-src` from Docker Hub and start it with `docker-compose.yml`.

Docker is available on Windows, Mac, and most Linux distributions. Note that on some Linux distributions, Docker is replaced by podman, however, the same command line options are used. Docker is a type of container software that leverages an existing Linux kernel to create a complete environment. Docker permits a developer to completely control the environment of the install (i.e. the exact version of the operating system compilers, which packages are installed, etc.). Docker

[1] https://github.com/ModernCPPBook/Examples.

[2] https://github.com/stevenrbrandt/CxxExplorer.git.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_1

greatly simplifies complex builds and installs, such as what is required for the C++ Explorer.

Many different options are needed to run a Docker environment in a user friendly way. For example, by default, Docker runs in a mode where it discards all generated files when it is shut down. This may not be what a student wants or expects from such an environment. The `docker-compose` tool simplifies the launching of Docker by providing a complete and working set of startup options to docker.

To start a Docker instance of the C++ Explorer from Bash, please run the following commands:

```
$ curl -LO https://raw.githubusercontent.com/stevenrbrandt/
    CxxExplorer/master/docker-compose.yml
$ docker-compose up -d
```

The `curl` command simply retrieves the content of the *docker-compose.yml* file. You can, alternatively, download this same file using your browser.

The `docker-compose up -d` command launches Docker silently in the background (that's what the `-d` option is for).

If you wish to see the console output for your Docker process you can run this command:

```
$ docker-compose logs
```

This should only be needed for debugging purposes. After starting the Docker image, a tab with the JupyterHub instance will be opened in the browser. Figure 1.1 on page 5 shows a Python notebook using the C++ 17 cling kernel to compile C++ code. For more details, about the C++ Explorer and Telegram Bot, we refer to [3].

1.2 Using CMake and C++ Compiler

Using the C++ Explorer, the examples in the book can be compiled using CMake and a C++ compiler. For compiling the C++ examples, a recent C++ compiler, e.g. clang, gcc, or MS Visual Studio; is needed. For more details about the compiler versions, we refer to Appendix C. To compile the C++ examples, please execute the following code snippet

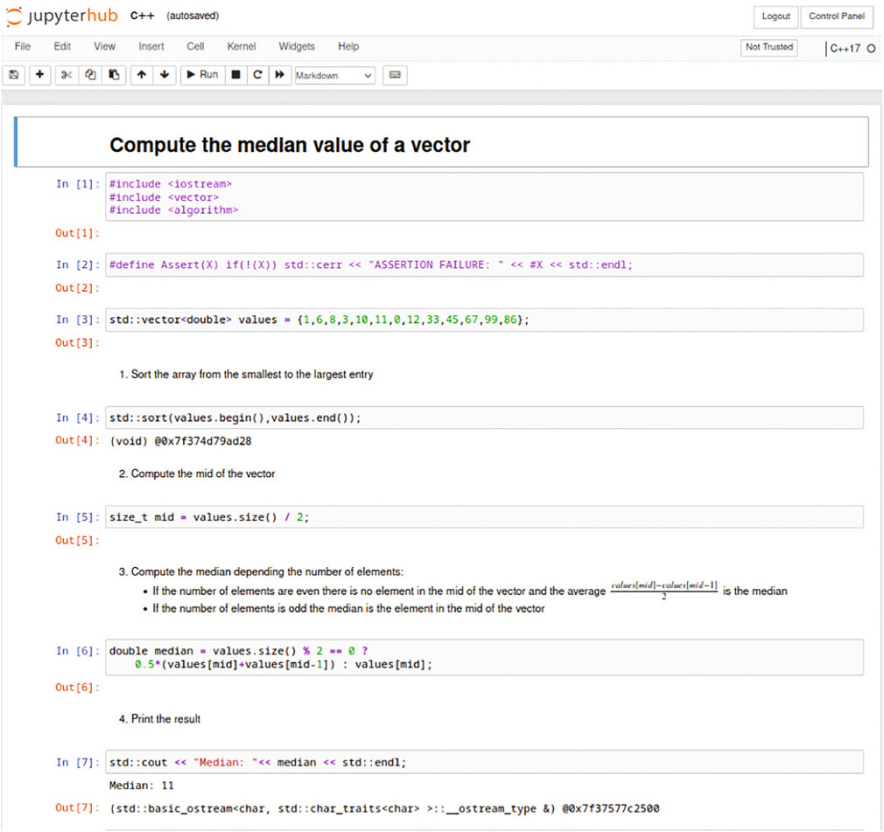


Fig. 1.1 Example of a python notebook running C++ code using Cling on JupyterHub with the C++ Explorer

Listing 1.1 Compiling the examples with cmake

```
1 $ git clone https://github.com/ModernCPPBook/Examples
2 $ cd Examples
3 $ mkdir build
4 $ cd build
5 $ cmake ..
6 $ make
7 $ ctest
```

In Line 1 of Listing 1.1 on page 5, we use `git` to clone the latest version from the GitHub® repository. In Line 2, we change to the directory with the code. In Line 3 and Line 4, we make a new directory `build` and switch to this directory to build the

code. In Line 5, we use `cmake` to configure the default build options and in Line 6 the code is compiled using `make`. For more details about CMake, we refer to [4]. Finally, we call `ctest` in Line 7 to run tests for all examples to make sure that the code was compiled correctly. Note that the OpenMP example will not work on Mac OS since Apple Clang does not come with default OpenMP support. The following CMake options are available to build additional examples:

- `WITH_HPX` – Build HPX examples (Default: `OFF`)
- `WITH_HPX_COROUTINES` – Build HPX example with coroutines (Default: `OFF`)
- `WITH_HPX_SIMD` – Build parallel algorithm examples using SIMD (Default: `OFF`)
- `WITH_MPI` – Build the MPI example for the fractal set (Default: `OFF`)
- `WITH_OPENMP` – Build the MPI + OpenMP example for the fractal set and OpenMP examples (Default: `OFF`)

For the parallel algorithms using the C++ Standard Library, currently only the GNU compiler collection and Microsoft Visual Studio support this experimental feature. For more compilation details, we refer to Chap. 10.

For the HPX examples one needs to compile HPX with all its dependencies using a C++ compiler, which supports the C++ 17 standard. To support coroutines, a C++ compiler which supports the C++ 20 standard is needed. For simplicity, we provide a Docker image, the same we use for the C++ Explorer, with all the compiled libraries. The following code snippet shows how to use the Docker image to compile the codeIn Line 1 of Listing 1.2 on page 6 the recipe to compile the docker image is downloaded and in Line 2 the Docker image `cxxex-src-nbk` is build using `docker-compose`. In Line 3 the Docker image is started and a bash shell is requested. In Line 4, we use `git` to clone the latest version from the GitHub® repository. In Line 5, we change to the directory with the code. In Line 6 and Line 7, we make a new directory `build` and switch to this directory to build the code. In Line 8, we use `cmake` to configure various build options and in Line 9 the code is compiled using `make`. For more details about CMake, we refer to [4]. Finally, we call `ctest` in Line 10 to run tests for all examples to make sure that the code was compiled correctly.

Listing 1.2 Building HPX

```
1  $ curl -LO https://raw.githubusercontent.com/stevenrbrandt/
       CxxExplorer/master/docker-compose.build.yml
2  $ docker-compose -f docker-compose.build.yml build --pull
3  $ docker exec -it cxxex-src-nbk bash
4  $ git clone https://github.com/ModernCPPBook/Examples
5  $ cd Examples
6  $ mkdir build
7  $ cd build
8  $ cmake -DWITH_HPX=ON -DWITH_SIMD=ON -DWITH-COROUTINES=ON ..
9  $ make
10 $ ctest
```

To run the image on HPC clusters, singularity [5] an open source tool for operating system level virtualization, is available. Singularity is an option to avoid running the container without root rights. Note that podman has this feature too, but is not as common on HPC clusters. The following code snippet shows how to pull the image using singularity and open a shell within the image

```
singularity pull cxx.sif docker://stevenrbrandt/cxxex-src
singularity shell cxx.sif
Apptainer>
```

Part II
Introduction to C++ and C++ Standard Library

In this part of the book, we provide the foundations of the C++ programming language and the C++ Standard library.

Note that this book does *not* provide a basic introduction to the C++ programming language. We assume the reader has a working knowledge of how to construct functions, loops, and if/else constructs.

All coding examples in this book represent working code that has been tested on our C++ Explorer docker image. In many cases, we use the appendix to provide the explicit boilerplate needed to run the examples.

The text of this book follows this general outline. We start with the history of the C++ programming language and the C++ Standard library. After that, we look from a birds-eye view at the C++ Standard library to provide a general overview of its functionality. Next, we dive into the algorithms library, container library, and iterator library. These are the most commonly used parts of the C++ Standard library in this book. Later in Chap. 10, we look into the parallel algorithms for shared memory parallelism.

In the course of this book we will focus on the coding of the Mandelbrot set and Julia sets as guiding examples. We will start with a serial implementation using the C++ Standard Library in Sect. 4.3. After that, we will extend this example with asynchronous programming in Chap. 9, parallel algorithms in Chap. 10 and coroutines in Chap. 11; for shared memory parallelism. Lastly. we will show a distributed memory implementation using the C++ standard library for concurrency and parallelism (HPX). For a better understanding these codes, the reader should familiarize themselves with the mathematics behind the Mandelbrot and Julia set.

Chapter 2
About C++, C++ Standard, and the C++ Standard Library

2.1 Brief History of C++, the C++ Standard, and Parallel Programming

C++ is based on the programming language C created by Dennis Ritchie from 1969 to 1973 while employed at Bell Laboratories. In 1973, Ritchie's compiler was used to compile the Unix kernel for the PDP-11 computer. It was the first time an operating system had been written in a language other than assembler.

Unfortunately, until 1989 there was no official C standard, and various compiler implementations resulted in unspecified behavior. However, there was an unofficial specification, the K& R C, based on the book [6] written by Kernighan and Ritchie in 1978. Over the years, many language features were added to the C programming language. Not all compilers supported all these features. So there arose a consensus that an official C standard was needed.

A committee was formed by the American National Standards Institute (ANSI) in 1983. They passed the standard *ANSI X3.159-1989 Programming Language C* in 1989. This was the first official standard, called C 89, for the C programming language. However, this standard was released by ANSI and was not an international standard. Therefore, the International Organization for Standardization (ISO) adapted the C 89 standard and released it as ISO/IEC 9899:1990 in 1990. Thus, the first international standard, called C 90, was released. Note that C 89 and C 90 are the same standard with just a different name. Later, the C 99 (ISO/IEC 9899:1999) standard was released. In 2011, 2018, and 2023 the C 11, C 18, and C 23 standards were released.

From 1990 onward, the C language was standardized, and therefore all compilers which wanted to be standard conforming had to follow the C standard. This greatly simplified the writing of code for different machines and architectures. It could be argued that realizing the value of standards in programming was the greatest technical advance of the time.

Concurrently, In 1978 Bjarne Stroustrup, while working at AT&T, began to develop "C with classes." Stroustrup wanted to write efficient system programs in

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_2

the style of Simula67. Therefore, he added better type checking, data abstraction, and object-oriented programming to C.[1] For more details, we refer to [7]. A first version was used in 1983 internally and later renamed to C++. Later, in 1985 the first commercial implementation was released. However, there was no official standard yet, but Stroustrup released a book [8], much as Kernighan and Ritchie did.

As we discovered with standardization, software works better to the extent that the programmer can separate concerns. Standards helped users divide worries about compiler implementation and machine architecture from the logic of their code. In a similar way, many of the C++ language features and libraries represent further efforts to divide concerns, allowing programmers to focus on fewer issues at once.

Classes allow the basic logic for types such as strings, iostreams, or complex numbers to be kept separate from both the code that uses them and the code that implements them. While it is possible to program in an object-oriented style using C (largely because of its support for structs), C++ provides additional support for the object-oriented paradigm through operator overloading, constructors and destructors, private data, and consistency checks.

Before templates, programmers wishing to use types like vectors or maps had to re-implement similar logic multiple times for each type. This reimplementation creates a maintenance challenge as each replicated piece of logic has to be updated whenever a code modification is required. Templates and generic programming changed this. Again, it is possible to write generic programs in C. In the early days of C++, classes such as maps were made to work through heavy use of the C-preprocessor. However, through templates, C++ provides better support and error checking for this style of programming.

Standards, as well as classes and generics, make it possible for programmers to look at the individual parts of a program without needing to understand whole sections of code at once. By allowing programmers to focus on the pieces, the task of composing large and complex codes becomes tractable.

Parallel programming, however, remains a challenge. Why so?

As part of the programming community's recognition of the value and power of standards, the MPI standard was born in 1994. While we will not spend much time on this language-independent library for parallel programming, we note that it remains the most widely recognized and implemented standard for distributed parallel programming in any language. For a brief period of time, standard C++ bindings were attempted for MPI, but these turned out to be problematic and were removed in the MPI 3.0 standard.

Unfortunately, while MPI makes distributed programming reliable on a variety of platforms and for a variety of languages, it does not make it particularly easy to write parallel code.

In 1997, the OpenMP standard began to evolve. It settled on a FORTRAN and C/C++ standard way of achieving parallel performance on a single node by the year 2005. One nice feature of OpenMP was its use of declarative parallelism within

[1] https://www.stroustrup.com/bs_faq.html#invention.

comments or pragmas. This mechanism made it easier to separate the logic of the code from the parallelism.

Using both OpenMP and MPI, it is fairly straightforward to implement a bulk-synchronous style of programming in which a group of threads does work and exchanges information in lock-step. Such codes can scale well and achieve high performance on nearly any supercomputer. They also frequently employ barriers in which all threads and processes synchronize at some point.

Many programmers were not satisfied with what they could achieve with these mechanisms. C++ 11 brought us threads, mutexes, atomics, futures, and asynchronous method calls, providing us with a standard for parallel programming within a single node. It brought us smart pointers to help clean up data held by more than one thread.

Both C++ 17 and C++ 20 made significant additional features available to programmers, including parallel algorithms.

However, achieving full use of any machine remains a challenge to this day. This book will explore what is possible at present, giving guidance on programming methodology and style as appropriate.

We note that programmers should, as best they are able, try to cleanly isolate the parallel logic of their code from the calculation they are trying to perform. The discipline of modularization, of separation of concerns, remains as the key to creating successful large scale programs.

There is also an effort to standardize teaching C++. The standing document for SG20: Guidelines for Teaching C++ to beginners[2] (ISO P1389) is one attempt for guidelines for teaching introductory C++. In stage 1, the fundamentals, the following fundamentals covered in this book are mentioned: containers, ranges, lambda functions, and iterator usage. In stage 2, the proposal recommend the following C++ features: smart pointers and parallel algorithms. In stage 3, the proposal recommends the following C++ features: generic programming and templates without template meta programming. This book goes further with a focus on parallel and distributed programming. Figure 2.1 on page 14 summarizes the history of the C++ standard and highlights the features used in this book.

Important

The most important aspect of the C++ standard is that the C++ standard is just a technical document specifying the language features and the C++ Standard library. However, the C++ standard is not an implementation of the C++ Standard Library or a compiler.

[2] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1389r1.html.

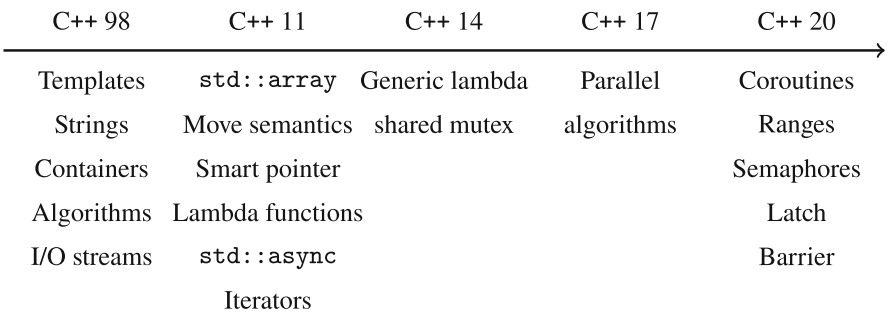


Fig. 2.1 This figure shows the various C++ standards beginning with C++ 98 in 1998 and continuing up to the C++ 20 in 2020. For each standard, a subset of the new features covered in this book are listed

2.2 Standard Template Library (STL) and C++ Standard Library

Dave Musser had a first draft for a generic library using generic programming in 1971. Dave Musser and Alexander Stepanov coined the term generic programming in [9]. Alexander Stepanov worked on Musser's idea of a generic library and a first library using the programming language Ada was developed in 1987. Stepanov and Lee named their library the Standard Template Library (STL) while employed at Hewlett-Packard (HP). Later STL was released as open source. The STL was the first generic library for data structures and algorithms in C++. The STL was designed around the four ideas: generic programming, abstractness, value semantics, and the Von Neumann computation model [10]. Nowadays, the STL is no longer maintained. However, Stepanov and Lee reached out to the C++ standardization committee in 1993 and presented their library for inclusion in the C++ standard. After some back and forth the proposal was approved in July 1994 at the ANSI/ISO committee meeting. These three researchers laid the foundations for the specification of the C++ Standard Library defined in the C++ standard. In addition, Stepanov and McJones wrote the book *Elements of programming* for practical programming based on a solid mathematical foundation [11].

Note that the Standard Template Library and the C++ Standard Library are two different things. The C++ Standard Library is the one specified by the C++ standard. Many implementations of the C++ Standard Library specified by the C++ standard are available. Table 2.1 on page 15 lists the most common active ones. There is one commercial implementation of the C++ Standard Library, Cray® C++ Standard Library, by Cray®. All other implementations are available as open source. Microsoft® releases the Microsoft® C++ Standard Library used for the MSVC tool set and the Visual Studio IDE. This is the most commonly used implementation for Windows operating systems. The second C++ Standard library is developed by NVIDIA® as the NVIDIA® C++ Standard Library. This library is special since the algorithms and data structures can be used on CPUs and NVIDIA® GPUs. All other

Table 2.1 List of the various implementations of the C++ Standard library. Except for one library, all are available under an open source license

Name	Acronym	License
Cray® C++ Standard library	–	Commercial
Microsoft® C++ standard library	MSVC STL	Apache license v2.0
NVIDIA® C++ standard library	libcu++	Apache license v2.0
GNU C++ standard library	libstdc++	GPLv3
LLVM C++ standard library	libc++	Apache license v2.0
C++ standard library for parallelism and concurrency	HPX	Boost software license 1.0

libraries work solely on CPUs. The most common C++ standard library on Linux operating systems is the GNU C++ Standard Library. Another C++ Standard library is the LLVM C++ Standard Library. The last implementation of the C++ Standard library is HPX, the C++ Standard Library for Parallelism and Concurrency. In this book, we will use the GNU C++ Standard Library and Microsoft C++ Standard Library for the examples. For the distributed examples, we will use the C++ Standard Library for Parallelism and Concurrency (HPX). For more details and which library features are available in each of the C++ Standard libraries, we refer to.[3]

> **Important**
>
> The Standard Template Library (STL) and the C++ Standard Library (SL) are two different things.

2.3 C++ Compilers

The following C++ compilers are available. Note that are more compilers available, however, we restricted ourselves to those most well-known and most used on supercomputers. First, we consider compilers provided by companies. On Cray® supercomputers, the HPE® Cray® Compiler is used, especially when using the Message Passing Interface (MPI) for distributed computing. The `icc` compiler by Intel® provides special optimization for Intel® CPUs. Likewise, the IBM® Xl compiler provides special optimization for the IBM® Power™ architecture, the AOCC compiler by AMD® provides special optimization for AMD® CPUs, the compiler `fcc` by Fujitsu® provides special optimization for the A64FX™ architecture, and `armclang++` by Arm® for the Arm® architecture. Microsoft®

[3] https://en.cppreference.com/w/cpp/compiler_support.

Table 2.2 List of C++ compilers showing the vendors that supply them, whether they support Windows and/or a *nix operating system, and the C++ standard they support. Note that we put 20 in the last column even if the compiler only implements a subset of the library features of C++ 20

Compiler	Vendor	Windows	*nix	C++ standard
AOCC	AMD®		✓	20
`clang++`	LLVM project	✓	✓	20
CC	HPE® Cray®		✓	17
`g++`	GNU project	✓	✓	20
`icc`	Intel®	✓	✓	20
NVIDIA® HPC SDK	NVIDIA®		✓	17
Visual C++	Microsoft®	✓		20
IBM® XL	IBM®		✓	14
fcc	Fujitsu®		✓	14
`armclang++`	Arm®		✓	17
Apple® Clang	Apple®		✓	20

provides Visual C++ for Windows operating systems. Finally, NVIDIA® provides the `pgc++` compiler, which is part of the NVIDIA® HPC SDK.

The following community driven compilers are available. First, `g++` by the GNU Project. This compiler is widely used on Linux as the default C++ compiler. Another compiler is `clang++` by the LLVM Project. Table 2.2 on page 16 summarizes all C++ compilers and the latest supported C++ standard.

The examples in this book are tested with the Visual C++, `clang++`, and `g++` compilers. For more details about the recommended compilers and versions, we refer to Appendix C. However, all compilers supporting C++ 17 or C++ 20 should work due to the ISO standardization of C++.

Chapter 3
C++ Standard Library

3.1 Overview of the C++ Standard Library

Figure 3.1 on page 18 sketches some of the components, namely algorithms, iterators, atomic, ranges, coroutines, input/output, thread support, and containers, of the C++ Standard Library (SL). Note that most of the components are provided by the C++ 17 standard and two of them are provided by the C++ 20 standard. For simplicity, we omit the C++ 17 standard information. Furthermore, we only showed the components, we will use in the book. First, the container component[1] provides the following data structures: unordered associative containers, associative containers, and sequence containers. In the current version of the SL maps and sets are provided as associative containers. As unordered associative containers unordered set and unordered maps are provided. The most important containers for this book are the sequenced containers like vector, list, and array. We will have a closer look at these containers in Sect. 3.2. For more details about data structures we refer to [12, Chapter 2]. The next two components are used to work on these containers. The iterator[2] component provides six iterators for working on a sequence of values, e.g. a list or any of the containers. Depending on the work on the sequence of values the following operations are defined: read or write access, random access, increment or decrement. We will have a closer look at iterators in Sect. 3.2.5. With the C++ standard the iterator component will be extended by iterators based on concepts[3] which will be different from the C++ 17 iterators. In addition, the ranges[4] component will become an extension and generalization of the existing C++ iterators. We introduce the usages of ranges in Sect. A.6, however, we do not yet use the ranges since not all major C++ compilers support them. The

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

[2] https://en.cppreference.com/w/cpp/header/iterator.

[3] https://en.cppreference.com/w/cpp/language/constraints.

[4] https://en.cppreference.com/w/cpp/ranges.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_3

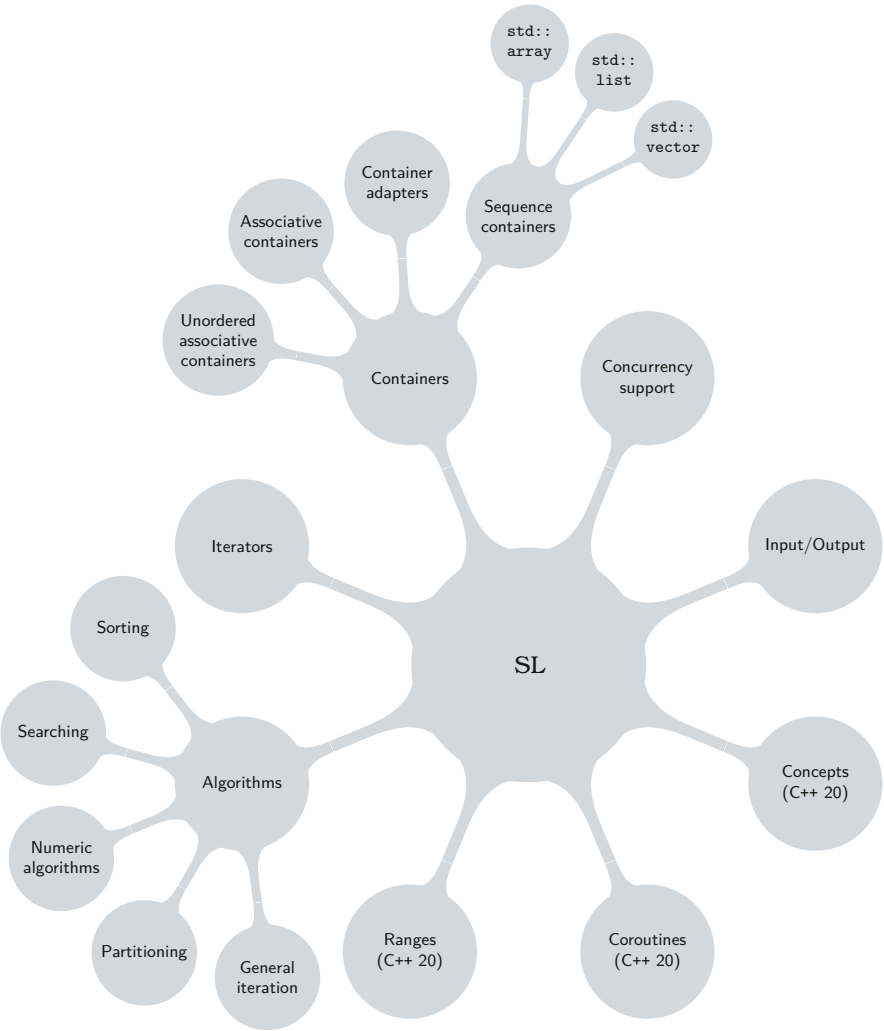


Fig. 3.1 Overview of the C++ standard library (SL): algorithms, containers, iterators, and functions. Note that this book will mainly focus on the algorithms, iterators, and containers

second component acting on containers are the algorithms.[5] We cherry-picked the algorithms we will use in this book:

- Sorting: Ordering the sequence of values with respect to their order, e.g. $<$, $\leq$, or $\geq$.
- Searching: Searching for specific values within the sequence of values.

[5] https://en.cppreference.com/w/cpp/algorithm.

- Numeric algorithms: Compute for example the sum of all values within the sequence of values.
- Minimum/maximum operations: Find the minimum or maximum value within the sequence of values.
- General iteration: Iteration of the sequence of values instead of using for example an `for` loop.

Note there are many more algorithms available and all of these algorithms work on all of the containers. We will look into some of the algorithms in Sect. 3.3. For more details about algorithms, we refer to [12]. Coroutines are introduced in the C++ 20 Standard and are stackless functions and their execution can be suspended and restarted later on, we will introduce coroutines in Chap. 11. To avoid race conditions and dead locks, the component concurrency support[6] is introduced. For more details, we refer to Chap. 7. For more details about the SL, we refer to [13]. We believe the most important takeaway from the introduction is the following: Have a look at the SL and if you can not find the container or algorithm you are looking for, you should ask if you really need to container or algorithm.

Important

The most important takeaway of this section is the following: Have a look at the SL and if you can not find the container or algorithm you are looking for, you should ask yourself if you really need that container or algorithm.

3.2 Containers

In this section, we will introduce on example to showcase the need of containers to store data. Let us assume, we want to compute the average a of some amount of elements n

$$a = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad x \in \mathbb{R}. \tag{3.1}$$

In Listing 3.1 on page 20 the implementation of the computation of the average is shown using the header `#include <iostream>`[7] of the C++ Standard Library (SL). This header provides us with the functionality to write and read from the standard streams. In Line 7 the values are read from the standard input stream and stored in the variable `x` as long as the user does not insert an empty string. The

[6] https://en.cppreference.com/w/cpp/thread.

[7] https://en.cppreference.com/w/cpp/io.

Listing 3.1 Example for the computation of the average of n numbers

```
1  #include <iostream>
2
3  double sum = 0;
4  size_t count = 0;
5  std::vector<double> vals = {1, 3, 7, 2.2, 1.8};
6
7  for (auto x : vals) {
8    sum += x;
9    ++count;
10 }
11
12 std::cout << "Average: " << sum / count << std::endl;
```

reading of the values happens by using `std::cin`[8] which is the standard input stream. In the next line all the values are accumulated in the variable `sum` and we count the total input using the variable `count`. After collecting all numbers provided by the user, we finally can compute the average. we use the standard output stream `std::cout`[9] to print the average, see Line 12.

Note, that we do not have to store the user's input while computing the average. However, to compute the median, we do need to store the user's input. The median is the middle value of a sorted list. To implement this calculation we need a vector to store the list and a sort algorithm. Before we look into the algorithms in Sect. 3.3, we will look at three commonly used containers.

3.2.1 *Vector*

From the mathematical perspective the `std::vector` provided by the header `#include <vector>`[10] is comparable to the mathematical definition of a vector

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

Note that in C++ a vector starts with the index zero and the last index is always $n - 1$, however, the length of the vector is n. Containers in C++ are homogeneous data structures and can contain only elements of the same type. For the usage of a vector, let us have a look into the computation of the average, see Eq. 3.1 on page 19. Listing 3.2 on page 21 shows the implementation using the vector container. In Line 6 a vector with the values 1.1, 2.3, 5.4, and 3.2 is initialized. Note

[8] https://en.cppreference.com/w/cpp/io/cin.

[9] https://en.cppreference.com/w/cpp/io/cout.

[10] https://en.cppreference.com/w/cpp/container/vector.

Listing 3.2 Example for the computation of the average of n numbers using the `std::vector` container

```
1 #include <numeric>
2 #include <iostream>
3 #include <vector>
4
5 size_t count = 0;
6 std::vector<double> values = {1.1, 2.3, 5.4, 3.2};
7
8 double sum = std::accumulate(values.begin(), values.end(), 0.0f);
9 std::cout << "Average: " << sum / values.size() << std::endl;
```

all containers are generic and the data type needs to be specified at initialization. In Line 8 we compute the sum over all elements using the `accumulate`[11] algorithm provided by the SL. The begin and end iterators are given by `values.begin()` and `values.end()`. We will discuss iterators in Sect. 3.2.5. For more details about the algorithms, we refer to Sect. 3.3. Note, that if we did not use the SL, we would need to write a `for` loop here to iterate over the vector and compute the sum. Here, we would produce more lines of codes and it is not as obvious as for the `std::accumulate` algorithm what is implemented there. In Line 9 the average is computed by using the function `size` which returns the amount of elements within the vector. For example the complexity for inserting or removing an element in a vector is $O(n)$ and for the list the complexity is $O(1)$ [12, 14].

Important

You should consider using vectors for the following:

- For small collections (a few thousand elements), it may not matter much.
- For collections which require frequent random access reads.

3.2.2 *List*

Another dynamically sized container is `std::list`[12] provided by the header `#include <list>`. From the API perspective, the vector and list are identical in most cases and one can just replace `std::vector` with `std::list` in the code. Therefore, we will not redo the computation of the average example for

[11] https://en.cppreference.com/w/cpp/algorithm/accumulate.

[12] https://en.cppreference.com/w/cpp/container/list.

Listing 3.3 Example for the computation of the median of n numbers using the `std::list` container

```
1  #include <iostream>
2  #include <list>
3  #include <algorithm>
4  typedef std::list<double>::size_type list_size;
5  ---------------------------------------------------
6  std::list<double> values = {2, 7.7, 3, 9.2, 1.4};
7  double x = 0;
8
9  values.sort();
10 list_size mid_index = values.size() / 2;
11 auto mid = values.begin();
12 std::advance(mid, mid_index);
13
14 double median = 0;
15
16 if (values.size() % 2 == 0) {
17   auto mid_one = values.begin();
18   std::advance(mid_one, mid_index + 1);
19   median = 0.5 * (*mid + *mid_one);
20 } else
21   median = *mid;
22
23 std::cout << "Median: " << median << std::endl;
```

the list. However, we will do a new example for the computation of the median, since we see some differences here. The median for a sorted list of numbers $v = \{v_i, \ldots, v_n \,|\forall v_i < v_{i+1}\}$ is given as

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

Listing 3.3 on page 22 shows the computation of the median using a list.

Note that the `std::sort` requires the random access iterator, and so does not work. Instead, we use the member function `std::list::sort`. In Line 10 we calculate the index of the middle element of the list by using `values.size()` which is identical for a `std::vector`. For a standard vector we could use the access operator `values[i]` to access the element at index i, however, this is the key property of the random iterator that we are missing. In Line 11, we get the iterator pointing to the first element of the list by using `values.begin()`. In Line 12 we use `std::advance`[13] to advance the iterator to the element at the middle of the list. In

[13] https://en.cppreference.com/w/cpp/iterator/advance.

Line 21, the value of the element at the middle is accessed by using the dereference operator `*mid`.

For more details about iterators, we refer to Sect. 3.2.5.

Important

You should consider using lists for the following:

- If you do not need to access the data structure randomly
- If you need to insert or delete elements at arbitrary positions while preserving the order.

3.2.3 *List vs. Vector*

Here, we will have a closer look at the difference of the usage of vectors and lists. The decision to use one or the other can make a big difference in performance. The following are points to consider while choosing a list or vector:

1. For the same amount of elements, vectors need less memory since only the elements are stored. The list, on the other hand, must store two pointers per data element.
2. Inserting elements at arbitrary positions takes less time for lists.
3. Random access of data is faster for the vector, since elements are stored sequentially in memory. Memory can only be traversed in sequential order for lists.

For more details, we refer the reader to [12, 14]. To investigate the difference in computational time for these two data structures, we look at Fig. 3.2 on page 24. These results were obtained using gcc 12 on a single node and the average out of ten runs for each number of elements is plotted. For all the figures, the number of elements in the list or vector varied from 10, 10^2, 10^3, up to 10^9. In Fig. 3.2 on page 24a, we compare the algorithm `std::fill` to assign all the elements within both containers with one. Here, we see a small overhead for the `std::list` compared to the `std::vector`. In Fig. 3.2 on page 24b, we compare inserting a new element at the end of the container by using `push_back`. Here, we see even for smaller sizes a large overhead for the `std::vector`. For the next two benchmarks, we look into arbitrary access by using a random number between zero and the list size. We use the same random number as the index for both containers. In Fig. 3.2 on page 24c, we insert an element at some randomly chosen index by using `insert`. Here, the `std::vector` has worse performance compared to the `std::list`. In Fig. 3.2 on page 24d, an randomly chosen element is accessed for both containers. Here, the `std::vector` shows better performance, since this

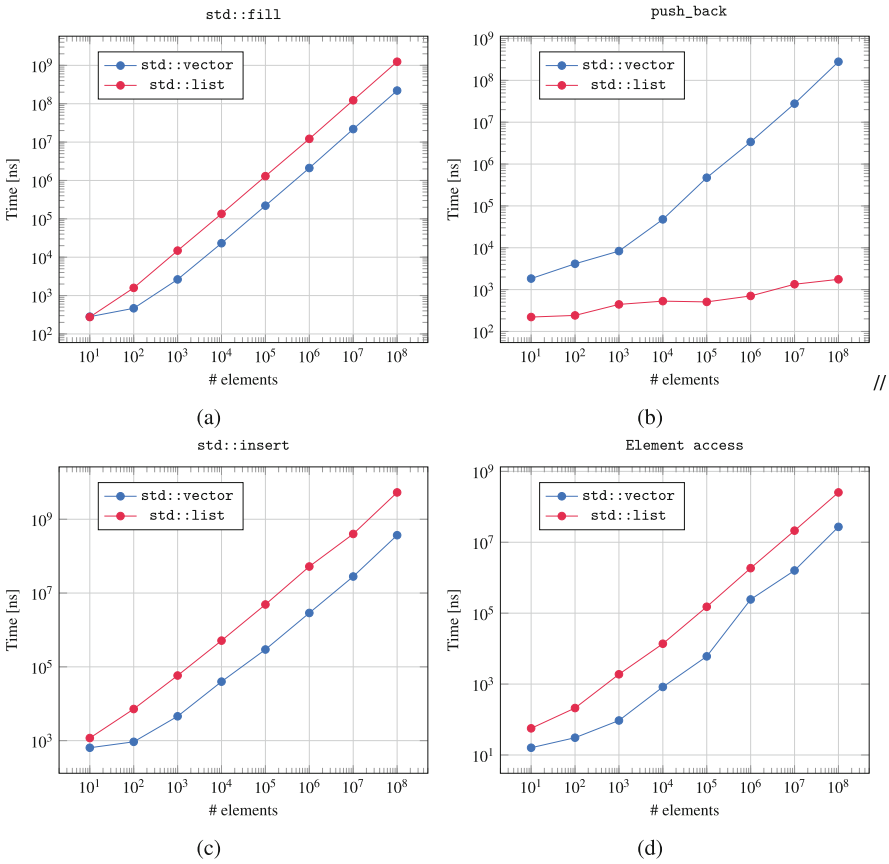


Fig. 3.2 Benchmark of `std::list` and `std::vector` containers: (**a**) filling the containers using `std::fill`. (**b**) shows the time for adding an element using `push_back`. (**c**) shows the time for inserting an element at a random position. (**d**) shows the time for accessing an arbitrary element

container is optimized for this task. These few examples were only some educational showcase to emphasize that the choice of container can make a difference. The source code for the benchmark is available here.[14] So be careful in your choice of container for your code, since there can be performance differences.

[14] https://github.com/ModernCPPBook/Examples/blob/main/benchmark/benchmark_containers.cpp.

Listing 3.4 Example for the usage of the array as language feature

```
#include <stdlib.h>

// Define the length of the array
const size_t size = 6;

// Generate the array
double array[size];

// Fill the array
for (size_t i = 0; i < size; i++) {
  array[i] = i;
}

// Print the array
for (size_t i = 0; i < size; i++) {
  array[i] = i;
}
std::cout << "last element: " << array[size - 1] << std::endl;
```

3.2.4 *Array*

For storing a fixed amount of elements, which needs to be known at compile time, two options are provided in C++. First, the language feature, like `int* array = double[5];`[15] can be used. Let us have a look at Listing 3.4 on page 25. Here, we use the array that is provided by the language. To fill the array with values a `for` loop is used. The same for printing, we need a `for` loop to iterate over all values. Instead of writing all these `for` loops, the usage of the algorithm in Sect. 3.3 is handy. Therefore, the container `std::array` provided by the header `#include <array>`[16] is needed. The array provided as a language feature and as a container are very similar. The major difference is the usage of the algorithms. In Listing 3.5 on page 26 the same code is written using the container array and the algorithms. The first `for` loop to fill the array with values can be replaced by `std::iota`. This function fills the vector starting with zero for the first element and increases by one for each following element. The second `for` loop can be replaced by `std::for_each` to iterate over each element in the array and call the lambda function for each element. For more details about lambda functions, we refer to Sect. A.2. From the standpoint of readability, the usage of the algorithms is preferable. Since instead of understanding what the loops are doing, the name of the algorithm is self-explanatory. With `array.begin()` and `array.end()` we used

[15] https://en.cppreference.com/w/cpp/language/array.

[16] https://en.cppreference.com/w/cpp/container/array.

Listing 3.5 Example for the usage of the `std::array` as container and the algorithms of the C++ SL

```
#include <algorithm>
#include <array>
#include <iostream>
#include <numeric>

// Define the length of the array
std::array<double, 6> array;

// Fill the array
std::iota(array.begin(), array.end(), 0);

// Print the array
std::for_each(array.begin(), array.end(), [](double value) {
  std::cout << value << " ";
});
std::cout << std::endl;
```

the iterators which we will introduce in the next chapter. For more details about the algorithms, we refer to Sect. 3.3.

3.2.5 *Iterators*

The C++ Standard Library (SL) provides iterators using the header `#include <iterator>`.[17] Five major iterator types, see Fig. 3.3 on page 27, are available: (1) the `output` iterator type iterates forward over the container by using the increment operator ++ and can write an element only once using the dereference operator *; (2) the `input` iterator type is read only and iterates forward over the container by using the increment operator ++ and can access an element multiple times using the dereference operator *; (3) the `forward` iterator type combines the `input` and `output` iterator types; (4) the `bidirectional` iterator type is like the `forward` iterator type, but it adds the `--` decrement operator to iterate over the container in both directions; (5) the `random access` iterator type extends the `bidirectional` iterator type with random access by allowing the programmer to add an integer. Note that the C++ pointer type is a random access iterator.

Depending on the container type, different iterator types are available. See Table 3.1 on page 27.

[17] https://en.cppreference.com/w/cpp/iterator/iterator.

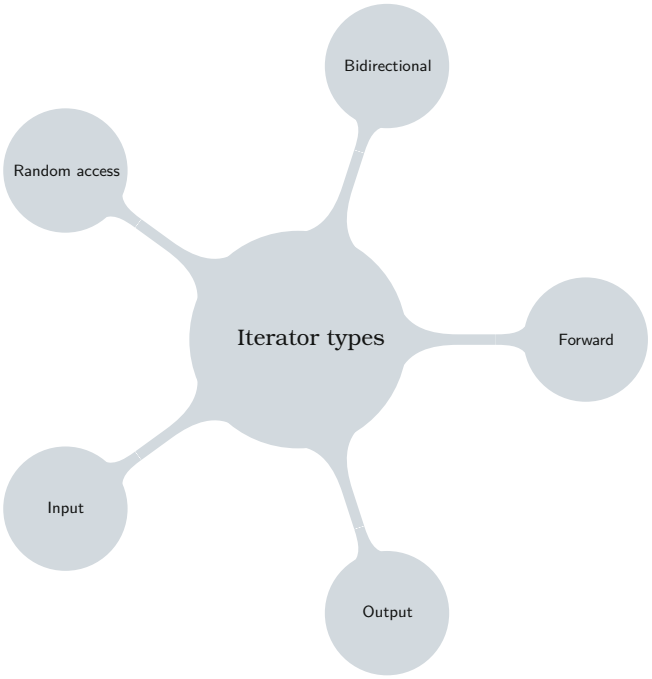


Fig. 3.3 The SL provides iterators with five different iterator types: input, output, forward, bidirectional, and random access. The following classification distinguishes the types: input and output ⊂ forward ⊂ bidirectional ⊂ random access. The random access iterator type has the most features and the input and output types have the fewest features

Table 3.1 SL containers and the available iterator type. Note that there are three containers with no available iterator type: stack, queue, and priority queue

Container	C++ definition	Iterator type
Double linked list	`std::list`	Bidirectional
Vector	`std::vector`	Random access
Set	`std::set`	Bidirectional
Multiset	`std::multiset`	Bidirectional
Map	`std::map`	Bidirectional
Multimap	`std::multimap`	Bidirectional
Dequeue	`std::dequeue`	Random access

Now, let us look into the usage of iterators. The iterator to the first element of the sequence is given by `std:begin()` and the last element of the sequence by cppstd::end(), respectively. Listing 3.6 on page 28 shows how to access all elements of a list using iterators. In Line 8 the iterator to the first element of the list is assigned to the type `std::list<int>::iterator` by using `std::begin (values)`. In Line 10 a `for` loop is used to iterate over the list as long as the iterator

Listing 3.6 Example for the usage iterators to print the elements of a `std::list`

```
1  #include <list>
2  #include <iostream>
3
4  int main() {
5    std::list<int> values = {2, 7, 3, 9, 1};
6
7    // Accessing the iterator to the first element
8    std::list<int>::iterator it = std::begin(values);
9
10   for (; it != std::end(values); it++)
11     // Accessing the element using the dereference operator *
12     std::cout << *it << std::endl;
13
14   std::cout << "------" << std::endl;
15
16   for (const int value : values) {
17     std::cout << value << std::endl;
18   }
19
20   std::cout << "------" << std::endl;
21
22   for (int index = 0; const int value : values) {
23     std::cout << "Index=" << index << " Value= " << value
24               << std::endl;
25   }
26 }
```

is not equal to `std::end(values)` which indicates the end of the list. In Line 12 the dereference operator is used to access the value of the element at the current iterator. Another possibility is to use a range-based `for` loop, see Line 16. With C++ 20 a range-based `for` loop can be written with an initializer. In Line 22 the initializer `int index` is added and initialized with zero. While the loop is iterating over the elements of the list, the variable `index` is incremented by one and indicates the index of the element.

The iterator type makes a big difference to the way we access the elements of a container. For example, for `std::vector` or `std::array` the subscript operator `[]` can be used to access an element by index, whereas for `std::list` it cannot.

Listing 3.7 on page 29 shows the different ways to access the element. In Line 10 for `std::vector` the index operator is used. Accessing the element in the `std::list` container, however, needs additional lines of code. In Line 13 the iterator to the first element of the list is given by `std::begin(list)`. With `std::advance` the iterator is advanced to the fourth element in the list, see Line 14. In Line 15 the dereference operator is used to access the value of the fourth element.

Listing 3.7 Example for different element access of `std::vector` and `std::list`

```
#include <list>
#include <vector>
#include <iostream>

// Generate the vector and list
std::vector<double> vec = {1, 2, 3, 4, 5, 6};
std::list<double> list = {1, 2, 3, 4, 5, 6};

// Access the element of the vector
std::cout << "using []: " << vec[3] << std::endl;

// Access the element of the list
std::list<double>::iterator begin = std::begin(list);
std::advance(begin, 3);
std::cout << "using advance: " << *begin << std::endl;
```

3.3 Algorithms

In this section, we will look into the algorithms provided by the C++ standard library, since these are the basis for the parallel algorithms, see Chap. 10. We will follow Sean Parent's CppCon 2013 talk (it's worth watching) in which he advises against using "raw loops." By using algorithms from the standard library, you create code that is shorter, easier to reason about and maintain, and probably more efficient. In addition, it may help programmers to get in the habit of reusing code rather than writing everything from scratch themselves.

As an example, Sean used the slide function. Slide takes a range of elements, specified by the first two arguments, and moves them to the position given by the third argument. For an illustration of what this algorithm does, see Fig. 3.4 on page 29. One might be tempted to implement this by removing the elements at the start location then inserting them at the end location.

The return value of the function consists of a pair of iterators that point to the new location of the original range. See Listing 3.8 on page 30.

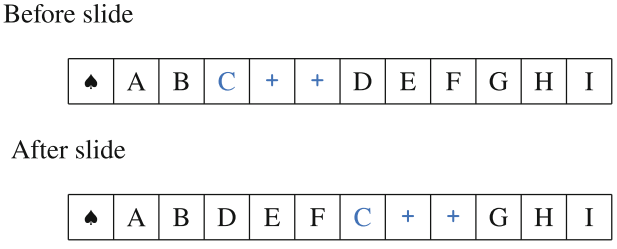


Fig. 3.4 A range of three elements (the blue text C++) is selected and slid three spaces to the right

Listing 3.8 Example of a raw loop

```
#include <vector>

template <typename T, typename V>
std::pair<T, T> slide1(V &v, T b, T e, T p) {
  auto n = e - b;
  typedef typename std::iterator_traits<T>::value_type e_type;
  std::vector<e_type> v2;
  for (auto i = 0; i != n; ++i) {
    v2.push_back(*b);
    v.erase(b);
  }
  T p2 = p;
  for (auto i = 0; i != n; ++i) {
    v.insert(p, v2.back());
    v2.pop_back();
    p2++;
  }
  return {p, p2};
}
```

The problem with this code is not that it is wrong (it isn't). It's probably even reasonably efficient if one is working with `std::list` objects. It is, however, highly inefficient for `std::vector`. Worse, it offers no opportunity for parallelism.

On the other hand, if one realizes that all a slide is doing is rotating the subset of the range of values, the slide function can be re-written without loops using the `std::rotate`[18] algorithm. See Listing 3.9 on page 31.

The new version is about fifty percent shorter, does not need the container to be passed (we needed it for Listing 3.8 on page 30 in order to use the `insert` and `erase` functions), and it does not have to allocate a vector. It turns out that rotating the elements of a container is a reasonably difficult algorithm to implement efficiently, much less parallelize. By rewriting `slide` to use the algorithm library, we reap those benefits as well.

The lesson is that programmers should familiarize themselves with the algorithms available in the C++ standard library and try to use them where possible.

The second example is the computation of the Taylor series of the natural logarithm ln. The Maclaurin series for the natural algorithm [15] reads as

$$ln(1-x) = -\sum_{n=1}^{\infty} \frac{x^n}{n} = -x - \frac{x^2}{2} - \frac{x^3}{3} - \ldots, \quad |x| < 1. \tag{3.4}$$

[18] https://en.cppreference.com/w/cpp/algorithm/rotate.

Listing 3.9 Implementation to slide a selection of elements within a `std::vector` to a new position

```
#include <algorithm>

template <typename T>
std::pair<T, T> slide2(T first, T last, T pos) {
  if (pos < first) {
    return {pos, std::rotate(pos, first, last)};
  } else {
    T hi = pos + (last - first);
    return {std::rotate(first, last, hi), hi};
  }
}
```

Listing 3.10 Implementation of the Taylor series of the natural logarithm using `std::vector` as the container and the algorithm `std::for_each` of the STL

```
#include <algorithm>
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <numeric>
#include <string>
#include <vector>

const std::size_t n = 10;
const double x = .372;
std::vector<double> parts(n);
std::iota(parts.begin(), parts.end(), 1);

std::for_each(parts.begin(), parts.end(), [](double &e) {
  e = std::pow(-1.0, e + 1) * std::pow(x, e) / (e);
});

double result = std::accumulate(parts.begin(), parts.end(), 0.0);
```

In this example, we again avoid the raw `for` loop to make it easier to write parallel code using the parallel algorithms in Chap. 10. Fortunately, the Taylor series for natural logarithm does not contain the factorial operation ! (present for the series for $\sin(x)$ and $\cos(x)$) which is tricky to implement efficiently without a loop. Listing 3.10 on page 31 shows the computation solely using SL algorithms. In Line 11 the vector is initialized with the size of n with is the maximal value of the finite sum in Eq. (3.4). In Line 12 we use the function `std::iota`[19] to fill the vector with the values from one to $n - 1$. In Line 14, we compute the value of each part of the sum and store them in the vector. In Line 18, the sum of all parts is accumulated.

[19] https://en.cppreference.com/w/cpp/algorithm/iota.

Important

The most important takeaway of this section is the following:

- The SL algorithms are efficient implementations and it will be hard to compete against them with your own algorithm.
- The code has less lines by using the SL algorithms and might be easier to understand.
- Most of the SL algorithms can be executed in C++ 17 in parallel easily by adding one additional argument. Thus, the usage of algorithms makes it easy to extend the code to shared memory parallelism.

Chapter 4
Example Mandelbrot Set and Julia Set

4.1 Mandelbrot Set

The Mandelbrot Set [16] is the set of complex numbers, c, for the iterative function given as follows:

$$z_{n+1} := z_n^2 + c, \quad z, c \in \mathbb{C}. \tag{4.1}$$

In the above equation, n refers to the iteration number. The value of z_0 is always 0. Substituting into the equation, the value of z_1 is $z_0^2 + c$, the value of z_2 is $z_1^2 + c$, and so on. The Mandelbrot Set consists of those values c for which this function does not go to infinity. In our visualizations, we will cut off the iteration if, at any point, the value of $|z_n| > 2$ or $n > 255$. We will then assign a color to the image based on the final value of n. Let us do one example. First, we will consider whether a point which is not in the Mandelbrot set $c = i \in \mathbb{C}$:

$$\begin{aligned}
z_1 &:= 0^2 + i = i \text{ // where i represents} \sqrt{-1} \\
z_2 &= z_1^2 + c = i^2 + i = -1 + i \\
z_3 &= (-1 + i)^2 + i = -2i + i = -i \\
z_4 &= (-i)^2 + i = 1 + i \\
z_5 &= (1 + i)^2 + i = 2i + i = 3i
\end{aligned}$$

So after five iterations, $|z_5| > 2$ and the value $c = i$ is in the Mandelbrot set. Second, we will consider the point $c = 0.2$ and compute the iterations again as $z_1 = 0.24$, $z_2 = 0.258$, and $z_3 = 0.267$. The value for the next iteration is growing, but slowly grows to ≈ 0.27. Since this number is less than two the value $c = 0.2$ is not in the Mandelbrot set. Algorithm 1 on page 34 sketches the computation to decide if the value $c \in \mathbb{C}$ is included in the Mandelbrot set. The visualization of this set of

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_4

Algorithm 1 Algorithm for the Mandelbrot set to compute if a value $c \in \mathbb{C}$ is included in the set

```
procedure MANDELBROT(c ∈ ℂ)
    max ← 80 ▷ Note that we need to define a maximal number of iterations, since if a number is not in the Mandelbrot set the algorithm would never terminate.
    z ← (0.0, 0.0) ∈ ℂ
    for i = 0; i < max, i++ do
        z = z^2 + c
        if |z| > 2.0 then
            return i
        end if
    end for
    return 0
end procedure
```

numbers was due to Mandelbrot in 1980. In this text, our primary interest in the Mandelbrot image is as an artificial workload. Each line of the image represents a variable amount of "embarrassingly parallel" work which can be divided up either within a machine or across machines. A key point here is that assigning the same number of rows to each of several processes may not provide a balanced amount of work.

4.2 Julia Set

Alternatively, we plot a Julia [17] set which uses the same equation given for the Mandelbrot set, except instead of fixing $z_0 = 0$ and plotting the number of iterations for all complex values c, we instead fix c to some number and plot all complex values of z_0. This produces a very similar workload to the Mandelbrot set, but produces a completely different image. The purpose in providing both sets is simply that students often find it interesting to explore the parameter space created by the computation and discover new images.

4.3 Single Threaded Implementation of the Mandelbrot Set

In this section, we will create our first implementation of the Mandelbrot and Julia set. For the equations and the mathematical details, we refer to Chap. 4. Here, we focus on the implementation details using the C++ standard library. This example will be single threaded and no parallelism will be added. However, this example is the basis for the whole book. We will extend this example for shared memory parallelism using parallel algorithms in Chap. 10 and asynchronous programming in Chap. 9. Lastly, the asynchronous programming example will be extended to

distributed memory using the C++ standard library for concurrency and parallelism (HPX) in Chap. 15. We encourage the reader to study this example here carefully, since it is essential to get a basic understanding of the code to understand the extension for parallel and distributed programming.

Before, we dig into the code, we will look into some of the utilities. First, the generate images of the fractal sets are stored in the Portable Bitmap File Format (PBM).[1] This might not be the most efficient file format to store images, however, it is easy to implement and no external library is needed. Therefore, this file format is often used for educational purposes. We provide the header `#include <pbm.hpp>`, see Listing B.2 on page 218 in Appendix B, with some minimal functionality to add pixel values to the image and save the image to hard disk. We will not go into the details of the file format. Instead, we direct the interested user to the Netpbm file format specification.[2] However, we like to mention one feature we use here. Sometimes it is beneficial to return multiple values from a function. For example when we convert the number of iterations to the corresponding RGB color scheme. In that case, we want to return three integer values to represent the red, green, and blue values. Here, the `std::tuple` provided by the header `#include <tuple>`[3] can be used as a fixed size container for heterogeneous types. The function `std::make_tuple` wraps the values into a tuple.

All the parameters to control the generating of the images are in the header `#include <config.hpp>` in Listing B.1 on page 217 in Appendix B. The following parameters with their default values are available:

- `const size_t max_iterations = 80;` – The maximal amount of iterations per pixel.
- `const size_t size_x = 3840;` – The size of the image in pixels in the x direction.
- `const size_t size_y = 2160;` – The size of the image in pixels in the y direction.
- `const int max_color = 256;` – The maximum number of distinct colors.

Figures 4.1 on page 36 and 4.2 on page 37 were computed using the code with these default values. Note that we need the mathematical constant π in our code. The constant π is provided by `std::numbers`.[4] In addition, Euler's number e, $\sqrt{2}$, $\sqrt{3}$, and other numbers are available. To change the default parameters or other options, the following environment variables can be set, before executing the code

- `MAX_ITER=` – The value of `max_iterations`.
- `SIZE_X` – The value of `size_x`.
- `SIZE_Y` – The value of `size_y`.

[1] http://netpbm.sourceforge.net/doc/pbm.html.

[2] https://en.wikipedia.org/wiki/Netpbm.

[3] https://en.cppreference.com/w/cpp/utility/tuple.

[4] https://en.cppreference.com/w/cpp/symbol_index/numbers.

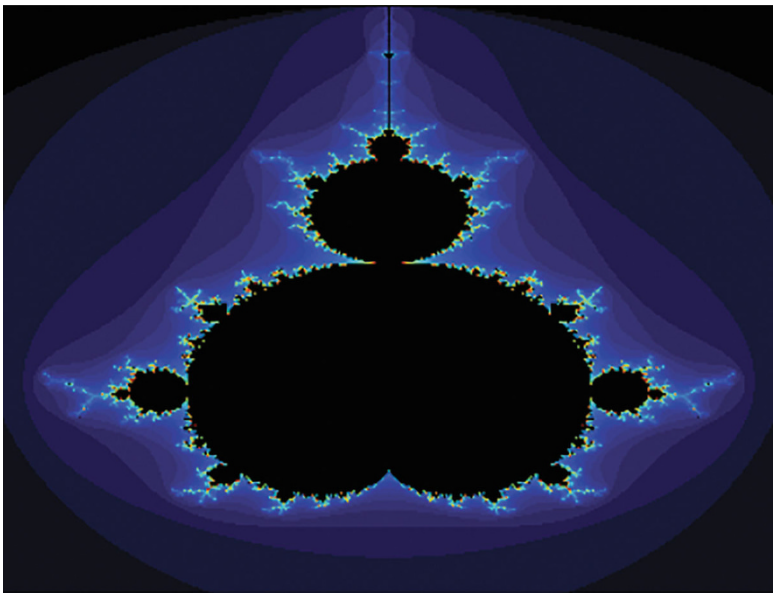


Fig. 4.1 Mandelbrot set computed for 400 × 600 pixels using maximal 80 iterations. The images were generated using the serial code shown in Sect. 4.3. We zoomed into the image to highlight the area with the most interesting features

- `MAX_COLOR` – The value of `max_color`.
- `TYPE` – The type of the set (Mandelbrot or Julia).
- `NUM_THREADS` – The number of threads to use.
- `OUTPUT` – If 1 the PBM file is to be written, 0 if it is not.
- `C_REAL` – The real component of the value of c.
- `C_IMAG` – The imaginary component of the value of c.

Next, we consider the compute kernels for the Julia and Mandelbrot fractal sets. The compute kernels are defined in the header `#include <kernels.hpp>`, see Listing B.3 on page 220 in Appendix B. Listing 4.1 on page 38 shows both of the compute kernels look very similar in their structure and Listing B.3 in the appendix shows the complete code. Note that in Eq. 4.1 the numbers z and x are complex numbers $\mathbb{C}$. A complex number has the form $a + bi$ where a and b are real numbers $\mathbb{R}$ and i is the imaginary number. The numeric library of the C++ standard library provides `std::complex`[5] a data structure to represent complex numbers. The data structure is a generic one and we use it with the `double` data type as `typedef std ::complex<double> complex;` for simplicity. Line 5 of Listing 4.1 on page 38

[5] https://en.cppreference.com/w/cpp/numeric/complex.

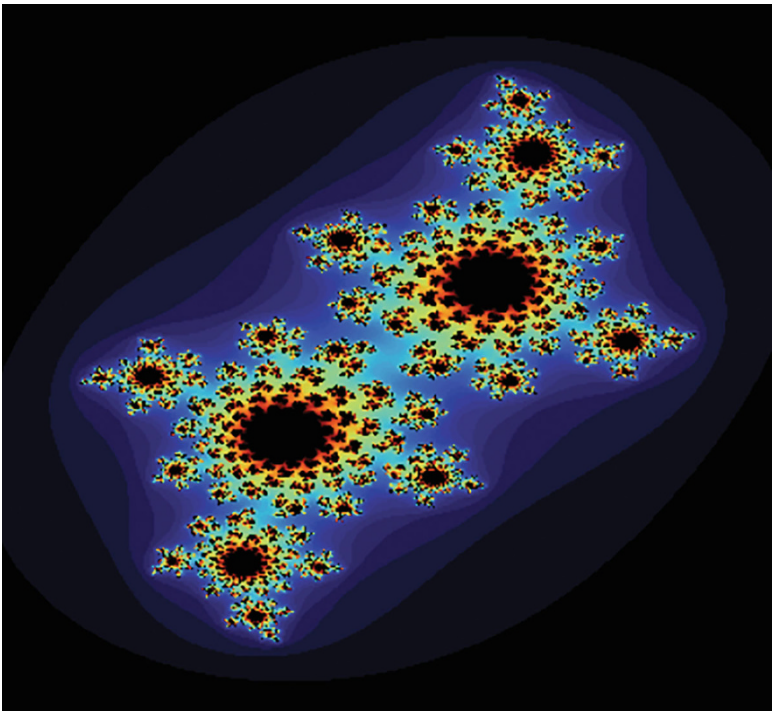


Fig. 4.2 Julia set computed for 400 × 600 pixels using maximal 80 iterations and c=(-0.4.0.6). The images were generated with the serial example code shown in Sect. 4.3. We zoomed into the image to highlight the area with the most interesting features

shows the compute kernel for the Mandelbrot set. Here, as the input parameter the complex number $c \in \mathbb{C}$ is given as the input parameter. The parameter $z \in \mathbb{C}$ is set to zero. In the `for` loop the value of $z = (0, 0)$ is updated by $z^z + c$, see Eq. 4.1 on page 33. After each update the absolute value $abs(z)$ is computed. if the absolute value is larger than two, the pixel is outside of the Mandelbrot set and zero is returned. This means that the color of the image will be black. Otherwise, the number of iterations are returned to compute the color at this pixel. For more details about the algorithm, we refer to Sect. 4.1. Line 17 shows the Julia compute kernel. In that case $c = (-0.4, 0.6) \in \mathbb{C}$ is fixed and we vary $z \in \mathbb{C}$. The same rules for coloring and stopping criteria are used as for the Mandelbrot set. For more details about the algorithm, we refer to Sect. 4.2.

Now, we finally get to the serial implementation using the above ingredients, see Listing 4.2 on page 39. In Line 8 the `pbm` object, which stores the image, is generated. In this code, we have to use two `for` loops to iterate over all pixels of the image in both directions. In Line 16 the complex number with respect to the x pixel is computed. The number of iterations per pixel is computed in Line 20. Note that this number is between one to `max_iterations`. To store the image

Listing 4.1 Compute kernels for the Mandelbrot ans Julia set

```
#include <complex>
#include <config.hpp>

// Kernel to compute the Mandelbrot set
size_t mandelbrot(std::complex<double> c) {
  std::complex<double> z(0, 0);
  for (size_t i = 0; i < max_iterations; i++) {
    z = z * z + c;
    if (abs(z) > 2.0) {
      return i;
    }
  }
  return 0;
}

// Kernel to compute the Julia set
size_t julia(std::complex<double> z) {
  std::complex<double> c(-0.4, 0.6);
  for (size_t i = 0; i < max_iterations; i++) {
    z = z * z + c;
    if (abs(z) > 2.0) {
      return i;
    }
  }
  return 0;
}
```

in the PBM format, we need to convert this one single number to red, green, and blue color values. This is done in Line 22 using one potential conversion. Many more color schemes are available and you can use your imagination to create a more advanced image. Finally, the color value is stored into the `pbm` object at the corresponding pixel. From the programming perspective the computation of fractal sets is straightforward and no complex algorithms are needed. However, for writing example shared memory or distributed memory implementations, the code is very interesting.

First, let us have a look at the theoretical worst-case complexity of the code. One common way to do this is by using the *big O notation* introduced by the German mathematician Paul Bauchmann in 1984 [18]. Edmund Landau advanced the notation and therefore the notation is also known as Landau notation [19]. The notation states that $f \in O(g)$, which means that the function $|f|$ is bounded from above by the function g up to a constant factor (i.e there exists a c such that $|f| \leq c \cdot g$).

For the fractal computation, we have three `for` loops in our code. Two to iterate over all pixels and the one to iterate within the pixel to determine its color. Thus, we can compute the work as `size_x` times `size_y` times `max_iterations` for all of the three `for` loops. Assuming that `size_x` and `size_y` are both proportional to

Listing 4.2 Example for single threaded computation of the fractal sets

```
#include <pbm.hpp>
#include <config.hpp>
#include <numeric>
#include <kernel.hpp>
#include <algorithm>

// Definition of utility
PBM pbm = PBM(size_x, size_y);

std::vector<size_t> index_(size_x);
std::iota(index_.begin(), index_.end(), 0);
std::for_each(index_.begin(), index_.end(), [](size_t i) {

  complex c =
      complex(0, 4) * complex(i, 0) / complex(size_x, 0) -
      complex(0, 2);

  for (size_t j = 0; j < size_y; j++) {
    // Get the number of iterations
    int value = compute_pixel(c + 4.0 * j / size_y - 2.0);
    // Convert the value to RGB color space
    std::tuple<size_t, size_t, size_t> color = get_rgb(value);
    // Set the pixel color
    pbm(i, j) =
        make_color(std::get<0>(color),
                   std::get<1>(color),
                   std::get<2>(color));
  }

  // Save the image
  pbm.save("image_serial_" + type + ".pbm");
});
```

a parameter n (i.e. that the aspect ratio of the image stays the same as we increase the size) but assuming that `max_iterations` is constant, we conclude that the *big O notation* for our code is $O(n^2)$. So computing Mandelbrot or Julia set images can consume a sizable amount of computational work. Therefore, the fractal sets are a good educational example for parallel and distributed computing.

Part III
The C++ Standard Library for Concurrency and Parallelism (HPX)

In this part of the book we examine the C++ standard library for concurrency and parallelism (HPX). In contrast to bulk sequential programs written in the past, HPX relies on asynchronous programming, overlapping communication and computation, over decomposition, and work-stealing to accelerate programs. HPX does this while staying aligned with the latest ISO C++ standard.

In the third part of the book, we will use HPX's asynchronous programming (see Chap. 9), parallel algorithms (see Chap. 10), and coroutines (see Chap. 11) to implement the creation of fractal sets for shared memory parallelism.

Chapter 5
Why HPX?

Before we dig into the question "Why HPX?" we need to investigate "What makes our system *SLOW*?". Hardware vendors provide a theoretical peak performance in floating point operations per second (FLOPS). However, the FLOPS obtained by our applications are rarely even nearly as high as the reported theoretical peak performance. For highly parallel hardware this is particularly true. With more available hardware parallelism, the application needs to scale better to use all the resources of the machine efficiently. In this book, we will look into two forms of scalability: strong scaling (Amdahl's law), which focuses on parallel efficiency; and weak scaling (Gustafson's law), which focuses on keeping efficiency constant as the core count increases. See Sect. 7.3.

To the extent possible, the application must have both excellent efficiency and keep that efficiency as N increases. This requires that nearly all of the code needs to be executed in parallel. In an optimal world, our code would run N times faster or we could run an N times bigger problem in the same amount of time while increasing the number of execution resources by N. However, in practice we see limitations due to the hardware architecture and programming models. We classify the limitations into four categories in Fig. 5.1 on page 44.

- *S*tarvation occurs when there is insufficient concurrent work available to maintain high utilization of all resources.
- *L*atencies are imposed by the time-distance delay intrinsic to accessing remote resources and services.
- *O*verheads work required for the management of parallel actions and resources on the critical execution path, which is not necessary in a sequential variant.
- *W*aiting for contention resolution is the delay due to the lack of availability of oversubscribed shared resources.

A nice visualization for the four factors is the woodcarving "Die vier apokalyptischen Reiter" (Four Horsemen of the Apocalypse) by Albrecht Dürrer from 1511 based on chapter 6, 1–8 of the Book of Revelation in the Bible [20]. From smart

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_5

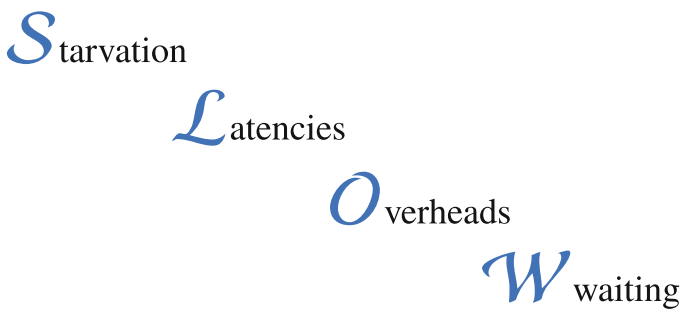


Fig. 5.1 Classification of the four limitations that can make our systems *SLOW*

phones, to desktop computers, to supercomputers, to cloud computing, these four factors will have a huge impact on the scaling and efficiency of our codes.

First, note that today's computers use the Von Neumann architecture which is based on the ideas of John von Neumann published in 1945 [10]. The Von Neumann or Princeton architecture was extended later by the Harvard architecture (which separated instruction and data space). Most programming and execution models widely used today have their foundation on these architectures. In our opinion, these should be the objectives for our codes:

- **Performance**: Applications need to be scalable and efficient.
- **Fault tolerance**: With the expected lower mean time between failures (MTBF) of future systems, we cannot avoid faults, we must handle them.
- **Power**: Minimizing power consumption is essential for various reasons.
- **Generality**: Any system should target a broad set of use cases.
- **Programmability**: With CPUs and GPUs from various vendors, high-level abstractions are essential for portability.

To address the challenge of changing architectures, ParalleX [21] was introduced in 2009. ParalleX provided a set of governing principles for the holistic design of future systems that were chosen to minimize the *SLOW* factors.

5.1 Governing Principles

HPX was designed based on a set of governing principles that were selected in order to be able to best possibly utilize the advantages of ParalleX. We describe those principles in more detail in the following sections. The experience collected while developing and using HPX shows that these principles are important to observe in order to compensate for the *SLOW* factors. Some of those principles are focused on high-performance computing, others are more general.

5.1.1 Focus on Latency Hiding Instead of Latency Avoidance

It is impossible to design a system exposing zero latencies. In an effort to come as close as possible to this goal, therefore many optimizations are targeted towards minimizing latencies instead. Examples for this can be seen everywhere, such as low latency network technologies like InfiniBand, caching memory hierarchies in all modern processors, the constant optimization of existing Message Passing Interface (MPI) implementations to reduce related latencies, or the data transfer latencies intrinsic to the way we use General-purpose computing on graphics processing units (GPGPU) today. It is important to note that existing latencies are often tightly related to some resource having to wait for the operation to be completed. At the same time it would be perfectly fine to do some other, unrelated work in the meantime, allowing the system to hide the latencies by filling the idle time with useful work. Modern systems already employ similar techniques (pipelined instruction execution in the processor cores, asynchronous input/output operations, and many more). What we propose is to make latency hiding an intrinsic concept of the operation of the whole system stack.

5.1.2 Embrace Fine-Grained Parallelism Instead of Heavyweight Threads

If we plan to hide latencies even for very short operations, such as fetching the contents of a memory cell from main memory (if it is not already cached), we need to have very lightweight threads with extremely short context switching times, optimally executable within one cycle. Granted, for mainstream architectures, this is not possible today (even if we already have special machines supporting this mode of operation, such as the Cray XMT). For conventional systems, however, the smaller the overhead of a context switch and the finer the granularity of the threading system, the better will be the overall system utilization and its efficiency. For today's architectures we already see a plethora of libraries providing exactly this type of functionality: non-pre-emptive, task-queue based parallelization solutions, such as Intel Threading Building Blocks (TBB) [22], Microsoft Parallel Patterns Library (PPL), Cilk++ [23], and many others. The ability to suspend a current task if some preconditions for its execution are not met (such as waiting for I/O or the result of a different task), seamlessly switching to any other task which can continue, and to reschedule the initial task after the required result has been calculated, would be the ideal.

5.1.3 Rediscover Constraint-Based Synchronization to Replace Global Barriers

Much of the code written today is riddled with implicit (and explicit) global barriers. By "global barriers," we mean the synchronization of the control flow between several (very often all) threads (when using OpenMP) or processes (MPI). For instance, an implicit global barrier is inserted after each loop parallelized using OpenMP as the system synchronizes the threads used to execute the different iterations in parallel. In MPI each of the communication steps imposes an explicit barrier onto the execution flow as (often all) nodes have to be synchronized. Each of those barriers is like the eye of a needle that the overall execution is forced to squeeze through. Even minimal fluctuations in the execution times of the parallel threads (jobs) causes them to wait. Additionally, it is often only one of the executing threads that performs the actual reduce operation, which further impedes parallelism. A closer analysis of a couple of key algorithms used in science applications reveals that these global barriers are not always necessary. In many cases it is sufficient to synchronize a small subset of the threads. Any operation should proceed whenever the preconditions for its execution are met, and only those. Usually there is no need to wait for iterations of a loop to finish before you can continue calculating other things; all you need is to complete the iterations that produce the required results for the next operation. Goodbye global barriers, hello constraint-based synchronization! People have been building this type of computing (and even computers) since the 1970s. The theory behind what they did is based on ideas around static and dynamic dataflow. There are certain attempts today to get back to those ideas and to incorporate them with modern architectures. For instance, a lot of work is being done in the area of constructing dataflow-oriented execution trees. Our results show that employing dataflow techniques in combination with the other ideas, as outlined herein, considerably improves scalability for many problems.

5.1.4 Adaptive Locality Control Instead of Static Data Distribution

While this principle seems to be a given for single desktop or laptop computers (the operating system is your friend), it is not ubiquitous on modern supercomputers, which are usually built from a large number of separate nodes (i.e., Beowulf clusters), tightly interconnected by a high-bandwidth, low-latency network. Today's prevalent programming tool for those systems is MPI, which does not help with data distribution, leaving it to the programmer to decompose the data to all of the nodes the application is running on.

There are a couple of specialized languages and programming environments based on PGAS (Partitioned Global Address Space) that provide minimal help for

this problem, such as Chapel [24], X10 [25], UPC++ [26], or Fortress [27]. However, all systems based on PGAS rely on static data distribution. This works fine as long as this static data distribution does not result in heterogeneous workload distributions or other resource utilization imbalances. In a distributed system these imbalances can be mitigated by migrating part of the application data to different localities (nodes). The only framework supporting (limited) migration today is Charm++ [28]. The first attempts towards solving this problem go back decades as well, a good example is the Linda coordination language [29, 30]. Nevertheless, none of the other mentioned systems support data migration today, which forces the users to either rely on static data distribution and live with the related performance hits or to implement everything themselves, which is very tedious and difficult. We believe that the only viable way to flexibly support dynamic and adaptive locality control is to provide a global, uniform address space to the applications, even on distributed systems.

5.1.5 Prefer Moving Work to the Data Over Moving Data to the Work

For achieving the best possible performance it seems obvious to minimize the amount of bytes transferred from one part of the system to another. This is true on all levels of the computation. At the lowest level we try to take advantage of processor memory caches, thus, minimizing memory latencies. Similarly, we try to amortize the data transfer time to and from GPGPUs as much as possible. At high levels we try to minimize data transfer between different nodes of a cluster or between different virtual machines on the cloud. Our experience shows that the amount of bytes necessary to encode a certain operation is very often much smaller than the amount of bytes necessary for encoding the data the operation is performed upon. Nevertheless, we still often transfer the data to a particular place where we execute the operation just to bring the data back to where it came from afterwards. Let us take a look at the way we usually write our applications for clusters using MPI (the Message Passing Interface). As its name implies, MPI is focused on orchestrating data transfers between distinct compute nodes. MPI is the tool of choice for portable, distributed programming on clusters, and the methods in its API are fairly straightforward to understand and to use. While MPI makes it easy to pass data through messages, organizing our thoughts around the transfer of data can lead to inefficiencies. Sometimes it is better to move work to data rather than data to work. The concept of Active Messages embodies this principle. While it is possible to implement dynamic, data driven, and asynchronous applications using Active Messages using MPI, it is not the natural way to use the API. If we look at applications that prefer to execute code close to the locality where the data was placed, i.e., utilizing Active Messages (for instance based on Charm++ [28]), we see better asynchrony, simpler application codes, and improved scaling.

5.1.6 Favor Message Driven Computation Over Message Passing

In Active Messaging, any incoming message is handled asynchronously and triggers the encoded action by passing along arguments and, possibly, continuations. The C++ standard library for parallelism and concurrency (HPX) combines this scheme with work-queue based scheduling as described above that allows the system to almost completely overlap any communication with useful work, thereby additionally minimizing latencies.

Important

We use HPX for several reasons:

- HPX implements the latest proposed and released features of the C++ specification. This means that switching a code using the SL to the HPX is as easy as switching the namespace.
- HPX can be made to work inside the Cling C++ interpreter (which is valuable for education), which we used to run and test many examples in this book while the SL library does not.
- HPX uses context switching on blocked threads, preventing threads from becoming idle during calls to `get()` on futures, or `lock()` on mutexes. This feature can increase throughput and avoid some deadlocks.
- HPX supports forward-looking, futures-based extensions for distributed computing, which are not yet specified by any standard.

Chapter 6
The C++ Standard Library for Parallelism and Concurrency (HPX)

The development of HPX started on May 24th 2008. It was begun by the Ste||lar group at Louisiana State University (LSU). After 16 years of HPX development, after a lot of reflection and experience, this book was written. At this point in time (2023), HPX has become a modular, feature-complete, and performance-oriented system targeted at conventional parallel computing architectures, from Raspberry Pis [31], to desktop computers, to supercomputers. For example, HPX has run on ORNL's Summit [32] and Riken's Supercomputer Fugaku [33], which are in the Top 5 of the Top 500 list at the time of this writing.

HPX is developed on GitHub®[1] under the Boost Software license. Though HPX originated at Louisiana State University, HPX's open source community currently has over 127 distinct contributors from all over the world. Figure 6.1 on page 50a shows the commits per month to HPX's GitHub® repository. Figure 6.1 on page 50b shows the contributors per month. Both plots indicate that HPX is actively developed. For more details about HPX's open source community, we refer to [34].

6.1 HPX's Architecture

As we mentioned in Chap. 2, the C++ standard is a technical document specifying the language features and the C++ Standard library. HPX uses the C++ standard and implements a subset of its specification. From one perspective, HPX is just another implementation of the C++ standard library, like the Cray C++ or GNU C++ Standard library, see Table 2.1 on page 15 in Chap. 3. However, from another perspective, HPX is a unique implementation of a distributed, asynchronous, many-task runtime system.

[1] https://github.com/STEllAR-GROUP/hpx.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_6

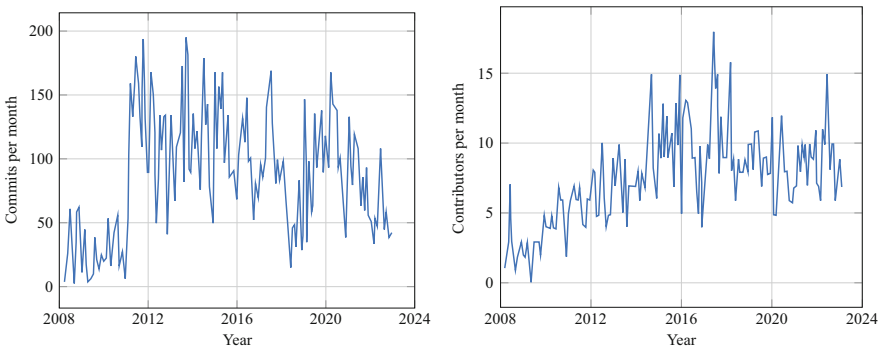


Fig. 6.1 Open source community data: (**a**) shows the commits per month to HPX's GitHub® repository from 128 distinct contributors. (**b**) shows the contributors per month. Both plots indicate that HPX is actively developed. The data was taken from OpenHub at 3/3/2023

Both the distributed capability and the lightweight task scheduling capabilities are special. The latter, because most other implementations of the C++ Standard libraries are based on operating system threads. At its heart, HPX is a very efficient threading implementation using light-weight threads. Figure 6.2 on page 51 shows HPX's architecture with all it components. Note that the last one, the component *C++2z Concurrency/Parallelism API*, is heavily used in this book.

Threading Subsystem
A thread manager creates and controls the light-weight, user-level threads on top of the operating system threads. See HPX [21]. To reduce *S*tarvation, see Fig. 5.1 on page 44, the light-weight threads have extremely short context switching times, especially for short running operations. This reduces synchronization *O*verheads while coordinating work between different threads. HPX thread pools provide a set of scheduling policies to let the user flexibly customize the execution. Automated load balancing is included within HPX by providing work stealing and work-sharing policies. These capabilities are important for high system utilization and scalable code.

Local Control Objects
HPX implements the asynchronous programming paradigm specified in the C++ 11 standard [36] using `hpx::future` and `hpx::async`, see Chap. 9. HPX also implements many of the primitives, e.g. `hpx::latch`, `hpx::barrier`, or `hpx::counting_semaphore`, in the C++ 20 Standard [37] for synchronizing work on different threads. This allows the overlapping of computation and communication. For more details, we refer to Sect. 9.1.

Active Global Address Space
HPX provides a unified API for local and remote execution by supporting distributed objects. Here, each object is annotated with a unique **G**lobal **Id**entifier (GID) which

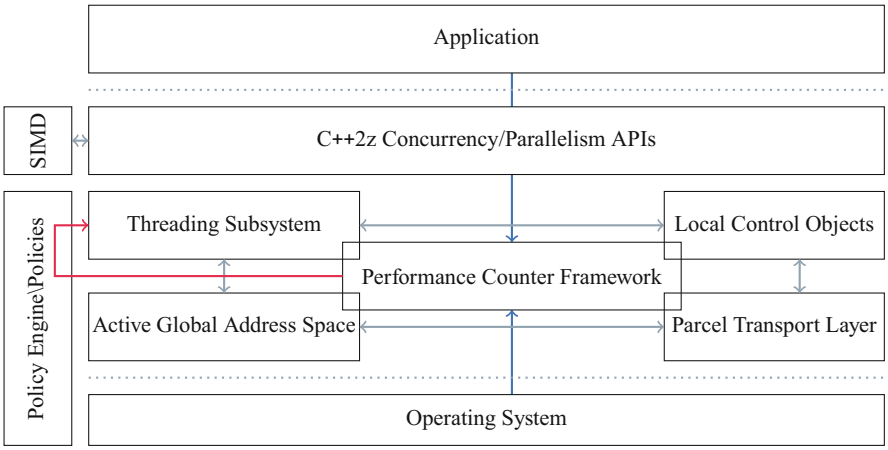


Fig. 6.2 Overview of HPX's architecture: on the bottom is the operating system and on the top the application. In between are HPX's components: the performance counter framework, the globally accessible performance framework, the threading subsystem (a thread manager providing HPX's light-weight user threads), local control objects for synchronization, the active global address space (AGAS) (allows objects and localities to be remotely addressed), the parcel port for providing active messaging, Policy\policies to implement load balancing and work migration, a layer to provide single instruction multiple data (SIMD) within the execution policies, and the C++ 2z concurrency and parallelism APIs. Adapted from [35]

can be resolved by any node in a distributed computation. Therefore, HPX extends the Partitioned Global Address Space (PGAS) [38] model with dynamic runtime-based resource allocation and data placement. Unlike PGAS, HPX's Active Global Address Space (AGAS) [39, 40] allows the application developer to transparently move objects using their GIDs between nodes on a distributed system. This feature is essential for load balancing via object migration.

Parcel Port

A common principle for the communication in distributed applications is message passing using, for example, the Message Passing Interface (MPI), see Sect. 13.1. However, HPX uses the concept of active messaging [41] instead. The parcel port leverages the Active Global Address Space (AGAS) for delivering messages to and launching functions on global objects independent of their current location in the distributed system. For more details, we refer to Sect. 14.1 and reference [42]. All of this functionality is asynchronous and allows the overlapping of communication and computation to be done implicitly. Currently, HPX supports TCP\IP, MPI, LCI [43], and libfabric [44] as communication backends. For a comparison between one-sided communication using MPI and two-sided communication using libfabric, we refer to Sect. 18.1. Examples for distributed programming with HPX are shown in Chap. 15.

Performance Counters
A globally accessible performance framework is integrated within HPX for monitoring in-situ system metrics. The pre-defined counters are registered with the Active Global Address Space (AGAS), which enables the user to query different metrics at runtime. In addition, users can generate application specific performance counters to obtain application specific details, e.g. the total processed sub-grid leaf nodes in Octo-Tiger [45]. HPX provides performance counters for most system components, see Fig. 6.2 on page 51. For example: networking, AGAS operations [39], and thread scheduling. For more details, we refer to [46].

Policy Engine\Policies
For some applications, with for example adaptive mesh refinement (AMR), the runtime environment might change to ensure performance. One example is work migration due unbalanced loads on the computational nodes. However, HPX does not provide work migration as a component yet. Instead, the application developer relies on APEX (the Autonomic Performance Environment for Exascale (APEX) [47]). APEX is included within HPX and can be compiled using the CMake option `-DHPX_WITH_APEX=ON`. With APEX enabled, HPX allows programmers to measure HPX tasks, monitor system utilization, and provides the user with the option to define custom policies that are triggered by events. The usage of the Performance Application Programming Interface (PAPI) [48] can be enabled within APEX by using the CMake option `-DAPEX_WITH_PAPI=TRUE`. Furthermore, APEX supports the tracking of CUDA events using the CUDA Profiling Tools Interface (CUPTI). This option is enabled with the CMake option `-DAPEX_WITH_CUDA=TRUE`. This option allows for the combined performance profiling of CPU and GPU events [49]. Two examples where this is useful are machine-learning-based methods to predict optimal policy engines [50, 51] and parcel coalescing [52].

Single Instruction Multiple Data
Single Instruction and Multiple Data (SIMD) is one of the classifications of parallelism proposed by Flynn [53] in 1972. The classifications are known as *Flynn's taxonomy* in the world of computer science. However, vector processing was missing in Flynn's taxonomy and was covered by Duncan's taxonomy in 1990 [54]. From the early 1990s most CPUs implemented SIMD as a hardware feature and vendor's provided APIs. To access the API, HPX adds the execution policy `hpx::execution::simd` to execute the parallel algorithms of the C++ standard library in Sect. 3.3 using SIMD. In addition, HPX supports the execution policy `hpx::execution::par_simd` to combine thread parallel execution with SIMD, see Sect. 10.1.2. At the time of this writing, HPX supports the experimental feature `std::experimental::simd` [55] of C++ and the library Expressive Vector Engine (EVE) [56].

C++2z Concurrency/Parallelism APIs
HPX implements `hpx::async` and `hpx::future` for asynchronous programming according to the C++ 11 standard [57], see Chap. 9.

HPX also implements all parallel algorithms of the C++ 17 standard [58]. With C++ 17, most of the algorithms of the C++ standard library (see Sect. 3.3) are extended with an execution policy to specify whether the algorithm is executed sequentially (`hpx::execution:seq`) or in parallel (`hpx::execution:par`). Note that the usage of parallel algorithms is discussed in Chap. 10.

HPX extends the standard by allowing programmers to combine the parallel algorithms with asynchronous programming by returning a `hpx::future` from the call of the parallel algorithm. This future can be used to track data dependency, see Sect. 10.1.1. From the C++ 20 standard [37] onward, HPX allows the programmer to extend the asynchronous programming with `coroutines`, see Chap. 11. Furthermore, HPX implements `hpx::jthread`,[2] `hpx::latch`,[3] `hpx::barrier`[4] of the C++ 20 standard. However, these are not covered in this book. HPX also provides an experimental implementation of the sender and receiver pattern based on the current draft of the proposed standard. However, it is unclear when the draft will be finalized and be included in the C++ standard. For some examples of the above features, we refer to [59].

To learn about these and other advanced capabilities in HPX which are not in the current C++ standard, see Sect. 9.1.

6.2 Applications

Let us briefly summarize applications, libraries, frameworks, or tools utilizing HPX at the time of this writing. From the application perspective, HPX was used in astrophysics, computational fracture mechanics, and condensed matter physics. From the library perspective, linear algebra is provided by the Blaze Math Library and SHAD provides high-performance algorithms and data structures. From the framework perspective, HPX is used in FleCSI and Kokkos as one potential backend. The C++ Explorer is a tool used primarily for educational purposes.

In fact, for running and exploring the code examples in this book, the C++ Explorer will be used, see Chap. 1. For all other applications, we provide brief summaries and encourage the interested reader to follow the provided references.

In this book, we will focus on the astrophysics application Octo-Tiger in Chap. 18. Octo-Tiger makes use of the Kokkos framework and GPU support.

[2] https:hpx-docs.stellar-group.orglatesthtmllibscorethreadingapijthread.html.

[3] https:hpx-docs.stellar-group.orglatesthtmllibsfullcollectivesapi/latch.html.

[4] https:hpx-docs.stellar-group.orglatesthtmllibscoresynchronizationapibarrier.html.

- Scientific computing:
  - **Octo-Tiger**[5] [60] models self-gravitating astrophysical fluids on an octree-based adaptive mesh refinement (AMR) mesh. It is ideally suited to modeling the dynamics of binary star systems undergoing mass transfer. It uses a finite volume method to model the hydrodynamics and the fast multipole method (FMM) to compute the gravitational field. It has been used to simulate systems with bipolytropic components [61] and to investigate double white dwarfs as potential progenitors for the R Coronae Borealis stars. Recently, Octo-Tiger was used to study the possibility that the spin of Betelgeuse can be explained by a binary merger [62].
  - **NLMech/PeriHPX**:[6] Peridynamic (PD) [63] is an alternative non-local formulation of classical continuum mechanics with a focus on discontinuities as they arise in crack and fracture mechanics. The successful comparison against experiments of PD has been reviewed in [64]. Due to its non-locality, PD is computationally intensive compared to molecular dynamics and smoothed particle hydrodynamics [65]. NLMech/PeriHPX [66, 67] is an implementation of PD using an asynchronous many-task system.
  - **DCA++**:[7] (Dynamical Cluster Approximation) provides a highly optimized C++ implementation to solve quantum many-body problems in condensed matter physics [68]. The DCA++ software is built on top of three different programming models (MPI, CUDA, and C++/HPX threading), and has numerical library dependencies (BLAS, LAPACK and MAGMA) to expose the parallel computations [69, 70]. In [69], authors reported that DCA++ with HPX threading support achieved a 20% speedup over the DCA++ with C++ standard threading support due to faster context switching and better threading management capability provided by HPX runtime.
- Libraries and frameworks:
  - **Blaze**:[8] Blaze Math Library [71] is a high performance C++ library for linear algebra based on Expression Templates(ETs) [72]. Expression Templates is an abstraction technique that uses overloaded operators in C++ to prevent creation of unnecessary temporaries, while evaluating arithmetic expressions, in order to improve the performance.
  - **SHAD**:[9] SHAD is a high-performance algorithm and data-structure library which provides scalability, flexibility, and portability [73]. It provides general purpose algorithms and data structures which enables users to extend SHAD with applications, such as graph libraries and linear algebra. The interfaces

[5] https://github.com/STEllAR-GROUP/octotiger.

[6] https://github.com/PeriHPX/PeriHPX.

[7] https://github.com/CompFUSE/DCA.

[8] https://bitbucket.org/blaze-lib/blaze/src/master/.

[9] https://github.com/pnnl/SHAD.

of SHAD algorithms and data structures are similar to the C++ Standard Template Library, offering users the flexibility to adopt SHAD on existing codes and applications.

- **FleCSI**:[10] A compile-time configurable framework designed to support multi-physics application development. FleCSI currently supports multi-dimensional mesh topology, geometry, and adjacency information, as well as n-dimensional hashed-tree data structures, graph partitioning interfaces, and dependency closures. FleCSI introduces a functional programming model with control, execution, and data abstractions that are consistent both with MPI and with state-of-the-art, task-based runtimes such as Legion and HPX.

- Tools:
  - **C++ Explorer**:[11] The C++ Explorer is a tool for teaching C++ at the college level. It grew out of efforts to teach HPX both within our group and at other institutions. Two tutorials were given based on the framework at Supercomputing, and at the USACM16 online school. It has twice been used as the basis of a course at LSU as a Math 4997 course. A paper was published about the C++ Explorer at Gateways [3]. While the C++ Explorer is not about teaching HPX per se (although it has been used for that purpose), it uses HPX to teach advanced C++ parallel programming concepts that are not yet available in the GNU platform.

[10] https://github.com/flecsi/flecsi.

[11] https://github.com/stevenrbrandt/CxxExplorer.

Part IV
Parallel Programming

In Chap. 7 the basic techniques of parallel programming are covered. We study theoretical aspects, such as Amdahl's law, and the new challenges arising with parallelism, such as race conditions and deadlocks and how to avoid them. After that, we focus on the techniques provided by the C++ standard to write parallel code: low level threads (Chap. 8), asynchronous programming (Chap. 9), parallel algorithms (Chap. 10), and coroutines (Chap. 11). Lastly, the performance of all these methods is compared on a single computational node using the following three CPU architectures: Intel®, AMD®, and Arm® A64FX™.

Chapter 7
Parallel Programming

7.1 An Overview of Parallel Programming

This book will focus on the parallel features provided within the C++ language and standard API. However, for completeness we will touch upon other libraries and frameworks that enable one to write parallel C++ code.

Parallel programming was introduced with the C++ 11 standard [36] with the ability to create `std::thread` objects. With threads, no abstraction layer is provided and the programmer has to start and join threads on their own. An example of such low-level thread programming is shown in Chap. 8.

How do `std::thread`s work?

Since before C++ 11, the pthreads library has been a commonly used C library on *NIX operating systems. Pthreads is based on POSIX, a parallel execution model that is independent of any programming language. The application program interface for pthreads is defined by the POSIX.1C thread extension (IEEE Std 1003.1c-1995). Many implementations of the library are available on POSIX compatible *NIX systems. The above-mentioned `std::thread` interface provided by C++ 11 is built on POSIX threads. For more details about POSIX threads, we refer to [74, 75].

But the C++ 11 standards committee was not content to simply provide a low-level, pthread-like interface. At the same time that low-level threads were added to the C++ 11 standard, asynchronous programming was also introduced, freeing the programmer from the need to work on low-level threads. Starting with C++ 11, a function can be launched asynchronously with `std::async` which returns a `std::future` for synchronization. (Note: Asynchronous programming will be described in more detail in Chap. 9.)

With the C++ 17 standard [58], parallel algorithms were introduced. In Sect. 3.3 we introduced the algorithms of the standard library (SL) which operate on containers. A portion of these algorithms were extended for parallel execution by execution policies [76]. The header `#include <execution>` brings in execution

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_7

policies that can be used to cause the algorithm to be executed sequentially or in parallel. Parallel algorithms are described in more detail in Chap. 10.

Parallel algorithms are a higher level of abstraction than `std::async` or `std::thread`, and therefore their usage is preferred when possible, but asynchronous programming provides a flexible way to introduce parallelism into arbitrary code. The C++ standard library for parallelism and concurrency (HPX) provides an extension (not defined in the C++ standard yet) that combines these two styles of parallel programming, allowing the programmer to launch the algorithm asynchronously and get a future back.

With the C++ 20 standard coroutines (provided by the header `#include <coroutine>`) were introduced. Briefly, coroutines provide a way to suspend and resume threads, making it possible to avoid the problems of calling `std::future`'s blocking `get()` call. We describe coroutines in more detail in Chap. 11.

With the C++ 20 standard we now have four different methods for writing parallel code on a single computer. All of these have different flavors, from low level threads, to asynchronous calls, to coroutines, to parallel algorithms. Also, each has different benefits and challenges which we will emphasize in the following chapters where we will examine these methods in more detail. However, one major benefit to using these methods is that all of them are part of the C++ standard and, therefore, using them should result in portable code that integrates well with the `std` library.

One of the most prominent tools for shared memory multiprocessing is Open Multi-Processing (OpenMP) [77]. OpenMP is an extension to the FORTRAN, C, and C++ programming languages by providing compiler directives. The first specification for C and C++ was released in 1998 and the latest version, OpenMP 5.2 [78], was released in 2021. Similar to the C++ standard, compiler vendors implement the OpenMP specification and support compiler flags to activate the compiler directives. Listing 7.1 on page 61 shows how to parallelize a `for` loop using the pragma `#pragma omp parallel for`. Except for Line 8, all other lines are standard C/C++ code. However, while pragmas are defined by the C++ standard, the specific `#pragma`s used by OpenMP are not. In addition to data parallelism, task parallelism has been added to the OpenMP standard. A subset of the task-based programming features in the OpenMP specification are implemented within hpxMP [79], where HPX is used by the compiler as a backend.

In addition to OpenMP, there are other notable tools to enable shared memory multiprocessing. From the language perspective there is CILK. CILK [23] was developed at the Massachusetts Institute of Technology in the 1990s. Later, a spin off company commercialize CILK as Cilk++ and was acquired by Intel in 2009 and released as Intel® Cilk™ Plus. However, Intel discontinued the project and recommended that users switch to OpenMP or Intel's oneAPI Threading Building Blocks. CILK is continued as openCILK. A parallel `for` loop in CILK is shown in Listing 7.1 on page 61 in Line 14. Here, the for loop is replaced by `cilk_for`. Like OpenMP, Cilk is a compiler extension and is activated using a compiler flag. The CILK compiler is based on llvm/clang. To summarize, all of these approaches and new features are not specified in the C++ standard, and using them may make code less portable.

Listing 7.1 Example to compute the power of two for all elements within a array using OpenMP or OpenCILK as language approaches and TBB or PPL as library approaches

```
1  #include <tbb/parallel_for.h>
2
3  int a[100000];
4
5  int I = 1;
6
7  // OpenMP
8  #pragma omp parallel for
9  for (int i = 0; i < 100000; i++) {
10         a[i] = I * i;
11 }
12
13 // CILK
14 cilk_for (int i = 0; i < 100000; i++) {
15         a[i] = I * i;
16 }
17
18 // TBB
19 tbb::parallel_for(tbb::blocked_range<int>(0,100000),
20                         [&](tbb::blocked_range<int> r)
21 {
22     for (int i=r.begin(); i<r.end(); ++i)
23     {
24             a[i] = i*i;
25     }
26 });
27
28 // PPL
29 parallel_for (size_t(0), 100000, [&](size_t i)
30 {
31         a[i] = i * i;
32 });
```

Shifting from language extensions to libraries, there are Intel's oneAPI Threading Building Blocks (TBB) [22] and Microsoft's Parallel Patterns Library (PPL). TBB is a C++ template library enabling task parallelism whereas PPL provides additional features like parallelism and containers.

Listing 7.1 on page 61 in Line 19 shows the usage of `tbb:parallel_for` function to execute the `for` loop in parallel. Listing 7.1 on page 61 in Line 29 shows the execution of the `for` loop in parallel using the `parallel_for` library function. Here, the code depends on one additional library and the API is not standardized. Another widely-used library is [80] which provides high level abstractions for parallelism and data management with a focus on portability across devices. Fore example, kokkos provides a backend for OpenMP, HPX, and C++ threads. Figure 7.1 on page 62 summarizes the most common tools for writing parallel C++ code.

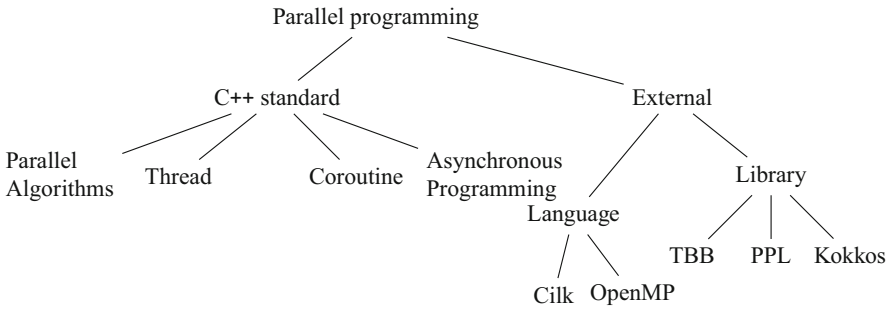


Fig. 7.1 Overview of parallel programming approaches in C++. On the **left** are the approaches included in the C++ standard. These approaches are covered in this book. For completeness, on the **right** we show the external approaches: language features and libraries are shown

Outside of C++, two other programming languages to enable parallelism are Julia and Rust. Julia [81] which has similarities to C++ and Python, started at the Massachusetts Institute of Technology in 2009. Rust [82] is a multi-paradigm, general purpose language that has been adopted by many large companies, including Amazon, Google, and Microsoft. It is the second language to be allowed into the development of the Linux kernel. Rust offers the opportunity to write safer code by making guarantees about memory safety and race conditions. A comparison of Rust, Python, Julia, Chapel, Charm++ and others is available here [83].

Parallel programming is an essential activity for any serious programmer. Many people think of parallel programming as the way to make a program run faster. While you can never have a truly fast and efficient program without parallel programming, cache and vectorization can be just as important. While vectorization will be touched upon in Sect. 7.5, cache considerations are beyond the scope of this text.

What are the reasons to use parallel programming?

- To Run Faster—Already discussed.
- To Coordinate Independent Agents—Programs like multiplayer games and databases typically interface with multiple human users who are trying to modify the same pieces of data at the same time. The agents in these two examples are humans, but they need not be.
- To Save Power—Since the power used on a modern CPU is proportional to the clock speed squared [84], a program can typically get more work done using the same amount of energy by employing more and slower cores, or running more cores at a lower speed.
- To Ensure An Answer—Running on very large machines, the mean time to failure of a given node is sometimes an issue. For a scientific simulation, the loss of data when a node fails is an annoyance. For an air traffic controller, it could cost many lives. In these situations, calculations are sometimes performed redundantly in order to ensure that the answer is delivered.

Regardless of what your reason for employing parallel programming, the capability is there. At one time, a desktop with 2 cores was considered a powerful machine. Now it's hard to imagine anything called a powerful desktop that contains fewer than 6 cores. Compute clusters, at the time of this writing, typically have 48 or more cores per compute node. These cores are on and present, and you'll want to make use of them. In the future, we should expect these numbers to get larger.

7.2 Race Conditions

Unfortunately, parallel programming is also the most difficult type of programming. When many cores attempt to operate on the same data at the same time, unpredictable answers may result. Many people who are new to programming may imagine that programs are deterministic. That is mostly true for sequential programs, but completely untrue for many parallel programs. The answer produced can strongly depend on the order in which events occur, this is called a "race condition."

The most common race condition is exemplified by a pair of threads that attempt to update an integer at the same time. The example code in Listing 7.2 on page 63, when run, essentially produces a random number somewhere between 1,000,000 and 2,000,000.

Why does it do this? It happens because the `count++` operation is not atomic. That is, it is actually composed of three steps. First the computer loads a value into

Listing 7.2 Examples for a race condition

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

double sum = 0;
const int sz = 1000000;
double* data = new double[sz];

void update(int n) {
    for(int i=0;i<sz;i++)
        sum += data[i];
}

int main() {
    for(int i=0;i<sz;i++) data[i] = 1.0;
    std::thread t1(update,1), t2(update,2);
    t1.join(); t2.join();
    std::cout << "final sum=" << sum << std::endl;
    return 0;
}
```

Listing 7.3 Another example for a race condition

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

double sum = 0;
const int sz = 40;
double* data = new double[sz];

void update(int n) {
    for(int i=0;i<sz;i++) {
        sum += data[i];
        std::cout << "working sum=" << sum << " at step=" << i <<
            std::endl;
    }
}

int main() {
    for(int i=0;i<sz;i++) data[i] = 1.0;
    std::thread t1(update,1), t2(update,2);
    t1.join(); t2.join();
    std::cout << "final sum=" << sum << std::endl;
    return 0;
}
```

a register, increments the register, and then stores it back. When two threads (call them A and B) follow this sequence at the same time, it could happen that thread A reads, then thread B reads; then both increment, coming up with the same value; and both store, writing the same value. This would mean that, despite both of them performing an increment, the value of count only increased by one.

Consider another version in Listing 7.3 on page 64 of this same program. This one increments just a few times and prints out messages in between increments. Printing to the screen effectively slows the program down. Because each thread takes such a long pause between increments of `count`, and because `count` is updated far fewer times, there is much less of a chance that the two threads will interfere with each other.

The end result is that it's far less likely that (during the course of one execution of this program) the two threads will interfere with each other when writing to `count`. In my tests, I see `count=40` more than 99% of the time. With failures this infrequent, it's hard to reproduce the problem, and therefore to find and fix a race condition. Indeed, even if you test extensively, you might not know it's there.

For this reason, it is difficult to ever truly be safe from race conditions. What programmers typically do, therefore, is try to avoid low-level parallel programming as much as possible and focus on using parallel algorithms, pieces of code which have been carefully checked, debugged, and maintained.

7.2.1 Mutexes and Deadlocks

Another type of concurrency control available within the C++ framework are mutexes, which are a shortened form of the words "Mutual Exclusion." A mutex is one solution to the problem illustrated by the race condition above.

To use a mutex, include the `#include <mutex>` header file and instantiate one or more variables of type `mutex`, see Listing 7.4 on page 65.

When one or more threads attempt to execute the `m.lock()` call, only one will succeed. All other threads attempting to call this function will wait and do nothing until the thread that called `m.lock()` calls `m.unlock()`. In this way, only one thread at a time executes the blocked region of code. This solves the race condition.

Unfortunately, in this case, it also completely ruins the parallelism. We have returned to a sequential code, but with somewhat worse performance because of the overhead of starting threads and synchronizing them.

It is possible, however, to restructure this code, see Listing 7.5 on page 66, to fix this problem. Since locking and unlocking adds overhead, the idea is to reduce this overhead by doing more work in the locked region.

Notice what we have done. We've exploited the commutative and associative properties of addition to do partial sums on each processor. The associative property

Listing 7.4 Usage of a mutex to avoid the race condition, but execute the code sequential

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

double sum = 0;
const int sz = 1000000;
double* data = new double[sz];

std::mutex m;

void update(int n) {
    for(int i=0;i<sz;i++) {
        m.lock();
        sum += data[i];
        m.unlock();
    }
}

int main() {
    for(int i=0;i<sz;i++) data[i] = 1.0;
    std::thread t1(update,1), t2(update,2);
    t1.join(); t2.join();
    std::cout << "final sum=" << sum << std::endl;
    return 0;
}
```

Listing 7.5 Improved version of the code using a mutex to avoid the race condition

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

double sum = 0;
const int partial_sz = 1000;
const int sz = partial_sz*partial_sz;
double* data = new double[sz];

std::mutex m;

void update(int n) {
    for(int j=0;j<partial_sz;j++) {
        // Do part of the sum on a local variable, partial_sum
        double partial_sum = 0;
        for(int i=0;i<partial_sz;i++) {
            partial_sum += data[i];
        }
        // update the result using the lock
        m.lock();
        sum += partial_sum;
        m.unlock();
    }
}

int main() {
    for(int i=0;i<sz;i++) data[i] = 1.0;
    std::thread t1(update,1), t2(update,2);
    t1.join(); t2.join();
    std::cout << "final sum=" << sum << std::endl;
    return 0;
}
```

tells us that we can add the sequence $1+2+3+4$ as $(1+2)+(3+4)$, adding $1+2$, then $3+4$, then summing those results together. The commutative property tells us that the order doesn't matter. Thus, the sum can be computed as $(3+4)+(1+2)$.

We can sum subsections of an array, and add the partial results together in whatever order the threads finish.

Operations that are associative and commutative, like addition, belong to a class of operations called "reduction operations." They are very common in scientific simulations, and all parallel frameworks have special facilities to take advantage of them. We will look at this class of problem in more detail later. At the moment, what we are concerned with is the mutex.

While mutexes solve this problem nicely and are convenient to use, they are vulnerable to deadlock.

Consider this example in Listing 7.6 on page 67.

Listing 7.6 Example for a deadlock where two threads block each other and the program will never terminate

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

int main() {
    std::mutex locka, lockb;
    std::thread ta([&](){
        locka.lock();
        std::cout << "Got lock A" << std::endl;
        lockb.lock();
        std::cout << "Got lock A then B" << std::endl;
        lockb.unlock();
        locka.unlock();
    });
    std::thread tb([&](){
        lockb.lock();
        std::cout << "Got lock B" << std::endl;
        locka.lock();
        std::cout << "Got lock B then A" << std::endl;
        locka.unlock();
        lockb.unlock();
    });
    ta.join();
    tb.join();
    return 0;
}
```

The problem here occurs when thread `ta` locks `locka`, and thread `tb` locks `lockb`. Each then attempts to get the other lock, but neither relinquishes the lock it has. The program cannot continue.

One way to avoid this problem is to always obtain locks in the same order. That way, you'll never have the situation described above. The C++ standard library has a class called `scoped_lock`[1] which addresses this need, and uses a destructor to automatically free the lock for you.

Note that in the above examples in Listing 7.7 on page 68, we are not actually getting the locks in a different order in time (`std::scoped_lock` will reorder them for us), we are just listing them in a different order within the code.

Does this solve the problem with deadlocks?

No. Though we solved this simple example, the difficulty here is quite insidious. The fundamental problem is that whenever you want to lock a new lock, you must already know what locks your thread currently holds. That means that before you

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

Listing 7.7 Attempt to avoid the deadlock using a scope_lock

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

int main() {
    std::mutex locka, lockb;
    std::thread ta([&](){
        std::scoped_lock slock{locka, lockb};
        std::cout << "Got lock A then B" << std::endl;
    });
    std::thread tb([&](){
        std::scoped_lock slock{lockb, locka};
        std::cout << "Got lock B then A" << std::endl;
    });
    ta.join();
    tb.join();
    return 0;
}
```

lock, you can't simply look at the code you're writing, but must understand the context it is running in. This means that programming with locks is not *composable*.

7.2.2 Atomic Operation

While mutexes can be used to solve synchronization problems like the one above, they are sometimes overkill. A lower level, and possibly more efficient, solution is available through atomics.

The C++ atomics library[2] gives access to a special machine instruction that looks at a value in memory, and if that memory location has the expected value, it updates it to a new value, see Listing 7.8 on page 69.

You might think that this looks less efficient. We have, after all, introduced a `while(true) {...}` into our code which the mutex did not need.

However, this is not so. Mutexes are actually written using atomics, and the `lock()` method will typically contain a `while(true) {...}` style loop.

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

Listing 7.8 Computation of the partial sum using `std::atomic`

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

std::atomic<double> sum = 0;
const int partial_sz = 1000;
const int sz = partial_sz*partial_sz;
double* data = new double[sz];

void update(int n) {
    for(int j=0;j<partial_sz;j++) {
        // Do part of the sum on a local variable, partial_sum
        double partial_sum = 0;
        for(int i=0;i<partial_sz;i++) {
            partial_sum += data[i];
        }
        while(true) {
            double expected = sum.load();
            double desired = expected + partial_sum;
            // try update, if success break
            if(sum.compare_exchange_strong(expected, desired))
                break;
        }
    }
}

int main() {
    for(int i=0;i<sz;i++) data[i] = 1.0;
    std::thread t1(update,1), t2(update,2);
    t1.join(); t2.join();
    std::cout << "final sum=" << sum << std::endl;
    return 0;
}
```

7.3 Performance Measurements

7.3.1 Amdahl's Law

Amdahl's Law [85] places a strong limit on the amount of benefit a program can achieve from a parallel machine. What it measures is the *speedup* of a program. The speedup is the ratio between the time a code takes to run sequentially vs. the time it takes to run in parallel. So if my program runs in 2 seconds using 4 threads, and 4 seconds on 1 thread, then I have a speedup of $s = 2$.

Now, suppose that half my code can benefit from parallelism, and half cannot. Suppose that the half that is parallel can benefit from any amount of parallelism, so that if I run on a machine that can support 100 threads, it will run in 0.01 the time—

almost no time at all. While that sounds great, it means that my overall speedup is about 2, and that's the best I can ever hope too do because half my code cannot speed up.

The following equation is Amdahl's law. It calculates the speedup S_N, based on the ratio of the time it takes to run a program on one thread, T_1, to the time it takes to run on N threads, T_N

$$S_N = \frac{T_1}{T_N}. \tag{7.1}$$

Let us start by imagining a computation called C. This computation could be a physics simulation, the creation of an image, or an effort to balance the books of a large corporation. It represents a substantial set of machine instructions that must be executed to produce an answer to a question. The computation is non-interactive. In other words, it takes input data, performs additions and multiplications, processes loops and conditionals, and produces output data.

To help us understand the speedup of this computation C, let us make some definitions:

- T_N: The time it takes to evaluate C on N threads.
- T_1: The time it takes to evaluate C on 1 thread.
- W: The total computational work in C. Think of this as the number of instructions the machine must process.
- P: The amount of computational work in C that can run in parallel. If, for example, we want to evaluate the mathematical expression $1 + 2 + 3 + 4$, we can evaluate $1 + 2 \rightarrow 3$ at the same time as we evaluate $3 + 4 \rightarrow 7$, and then we can evaluate $3 + 7 \rightarrow 10$ after those prior computations are finished.

If we now assume that in each unit of time, say a nanosecond, our processor can compute one instruction, then we can say $T_1 = W$. That is, the total time in nanoseconds required to evaluate C is equal to the number of instructions, W.

Given this assumption, we can now compute T_N as follows:

$$T_N = (W - P) + \frac{P}{N}. \tag{7.2}$$

In other words, we say that the instructions represented by P can be evenly split between N processors that can each perform one instruction per nanosecond. The value $W - P$ represents the instructions that cannot be performed in parallel. Because they must be evaluated by a single processor, it takes $W - P$ nanoseconds to compute them.

We can now rewrite our speedup as follows:

$$S_N = \frac{T_1}{T_N} = \left(\frac{W}{W - P + {}^{P}/_{N}} \right). \tag{7.3}$$

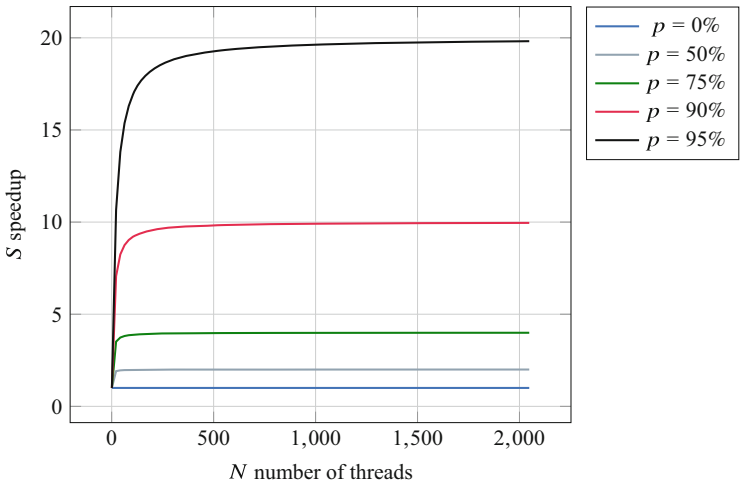


Fig. 7.2 Plot of Amdahl's law for different parallel fractions of the code

Let us now define the parallel fraction, $p = P/W$. Substituting and doing some algebra, one finds the more common and useful form of Amdahl's Law.

$$S_N = \left(\frac{1}{1 - p + {}^{p}/_{N}} \right) \tag{7.4}$$

Substituting into this equation, one finds that if 90% of my code is parallelized, then the speedup of my code can only approach 10 but never reach it.

Roughly speaking, Amdahl's Law says that your program can only benefit from a number of threads $N \equiv {}^{1}/_{(1-p)}$. This means that if you want to make use of a supercomputer with millions of cores, the sequential portion of your code must be vanishingly small. Figure 7.2 on page 71 illustrates Amdahl's Law for different parallel fractions of the code.

7.3.2 Gustafson's Law

Gustafson's Law provides an alternative picture for parallel speedup. In this view, rather than keeping the problem size fixed as we increase N, we run a bigger problem and therefore do more work. Thus, the time it takes to run on a serial system will increase with N, while for a parallel workload, it will remain constant.

This matches many real-world physical simulation codes used in ocean modeling, black hole collisions, galaxy simulations, etc. Because larger machines usually consist of many smaller computers with a high-speed interconnect, a

larger simulation also has access to more memory and it makes sense to run a proportionally bigger problem.

Let us create two proportionality constants, k and ρ and redefine our computational work in order to use them. The constant ρ represents the fraction of code that is parallelizable when N is 1. The value of k is the number of seconds required to run on a problem of size $N = 1$.

$$W = k \cdot (1 - \rho + \rho \cdot N) \tag{7.5}$$

$$P = k \cdot \rho \cdot N \tag{7.6}$$

Substituting these values into the equation we've used before, we get

$$S_N = \frac{T_1}{T_N} = \frac{W}{W - P + P/N} = 1 - \rho + \rho \cdot N. \tag{7.7}$$

Typically, as noted above, this law is applied to codes which have to run on many compute nodes, so the time on a single compute node is largely theoretical. Viewed in this way, the parallel fraction of a code is $p = P/W = N/1-\rho+\rho N$, a value that gets arbitrarily close to 1 (or 100%) as N increases. These sorts of codes can make use of a machine of nearly any size, so long as they scale up the amount of work that they do as they scale up the cores that they are using. In many ways, this is a more realistic way of looking at how scientists run simulation codes.

However, for very large N on very large machines (say $N = 100{,}000$), the problem size is often too large to run on a single node. And even if there were, somehow, enough memory on a single node, the time it would take to run on that single node with that large an N value is impractically long. Thus, the time T_1 would not be practical to measure.

For this reason, Gustafson's Law plots are rarely produced for any real code.

What many scientists do instead is the time a problem of size N on N nodes to the time it takes to run a problem of size 1 on 1 node. In that case, one sees a speedup that is always less than one, but hopefully not much less than 1. This is a "weak scaling" plot, which we will discuss in more detail in Sect. 7.3.4. Figure 7.3 on page 73 illustrates Gustafson's Law for different parallel fractions of the code.

7.3.3 Speedup and Parallel Efficiency

Speedup is a useful thing to measure, but it the amount of speedup you see depends on the value of N, the number of cores you are running on. How well does your application do on 10 cores vs. 100 cores.

Parallel efficiency takes the ratio of the speedup you obtained to the speedup you could have obtained on N cores. Since the ideal speedup on N cores is always N, the

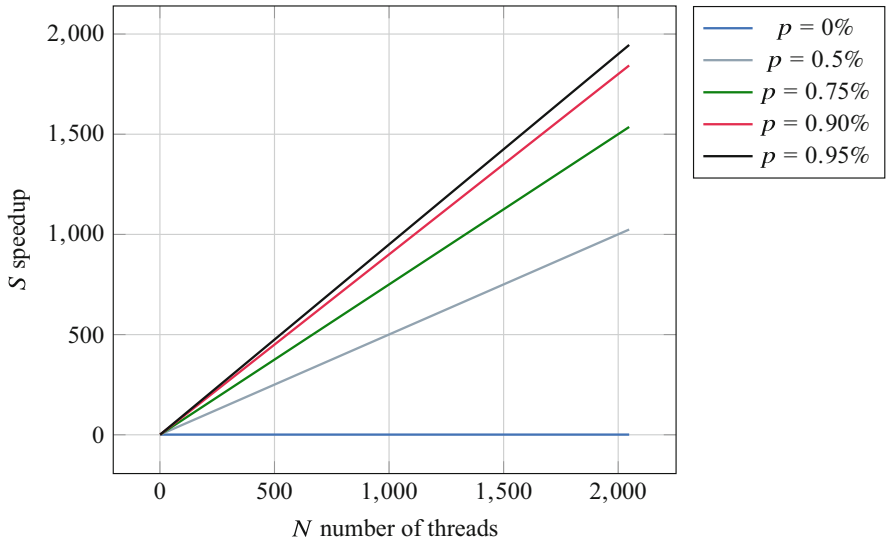


Fig. 7.3 Plot of Gustafson's law for different numbers of threads

parallel efficiency is nothing more than the speedup we actually achieved divided by N.

$$\mathrm{PE}_{\mathrm{Amdahl}} = \frac{1}{N \cdot (1 - p) + p)} \tag{7.8}$$

In the limit of large N, parallel efficiency for an Amdahl-like problem always goes to zero. That is, for a fixed size problem, only a finite number of resources can ever be useful.

On the other hand, for a Gustafson-like prescription in which work increases with N, we get

$$\mathrm{PE}_{\mathrm{Gustafson}} = \frac{1 - \rho + \rho \cdot N}{N}. \tag{7.9}$$

In the limit of large N, Gustafson's parallel efficiency approaches the constant ρ, the faction of code that is parallelizable for a problem of size $N = 1$. For a workload that matches Gustafson's prescription, there is no limit to the amount of resources that could be used.

7.3.4 Weak Scaling and Strong Scaling

The above discussion about scaling laws and parallel efficiency relate to two types of tests one can perform on a large simulation code: strong scaling tests and weak scaling tests.

For the sake of this discussion, imagine that we have a Gustafson-like code, i.e. one for which we can arbitrarily increase the amount of parallel work. Assume, however, that the parameter ρ degrades as N increases. Initially, our $\rho = \rho_0$, but as the core count increases, it eventually falls off to $\rho = \rho_0 - \delta$. See Eq. 7.10 on page 74.

$$\rho(N) = \rho_0 - \delta + \delta exp(-k \cdot N) \tag{7.10}$$

For strong scaling the *problem size* is fixed, and the goal is to make it run with the highest possible parallel efficiency. Initially, the result will be a speedup that resembles an Amdahl's Law plot. A strong scaling test tends to focus on ρ_0 and making it large. It may, however, miss problems caused by δ. See Eq. 7.10 on page 74.

Strong scaling tests, for any real code, will eventually become slower if the core count increases beyond a certain threshold. This happens because parallelism adds additional overhead, and the amount of work is not infinitely divisible. At some point, the overheads (which Gustafson's and Amdahl's Law do not model) become large enough relative to the work being done that they dominate the execution.

This means that the problem with strong scaling arises from its very nature, that the problem size is fixed. This not only limits the number of cores that may eventually be used, but it also means that a given problem cannot run on both a desktop machine and a supercomputer. It will be either too small for one or too big for the other.

Regardless, strong scaling tests are useful because they force the programmer to make ρ_0 as large as possible.

For weak scaling the *problem size per processor* is fixed, and the goal is to show that the execution time does not increase as the problem size grows (i.e. to show that the code scales according to Gustafson's Law).

Thus, weak scaling tests will find inefficiencies that result from scaling problem size in a simulation code (i.e. a large δ). Problems that are still small on 100 cores might be huge at 10,000, and a strong scaling test is unlikely to reveal those kinds of problems.

Weak scaling tests compare the time it takes to run a problem of size N on N cores to the time it takes to run a problem size of 1 on 1 core. While this type of test runs well on machines of all sizes, it tends to hide inefficiency problems (a small ρ_0). Indeed, even a very small ρ_0 will provide a nice flat weak scaling plot.

Therefore, both weak and strong scaling tests are useful. A code should have a high ρ_0 and a small δ.

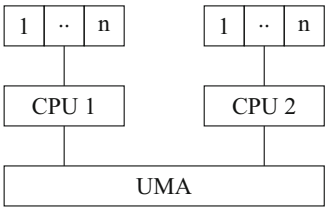


Fig. 7.4 Uniform memory access (UMA)

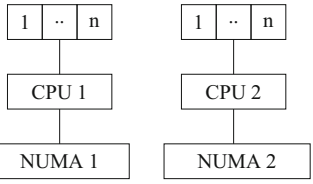


Fig. 7.5 Non-uniform memory access (NUMA)

No single test should provide the sole measure of performance. Beware of scientists who show a weak scaling plot and no other performance data! Their code probably has a very small ρ_0.

7.4 Memory Access

For shared memory parallelism, understanding the memory access scheme is important to understanding the scaling behavior. There are two common memory architectures. First, there is the uniform memory access (UMA), see Fig. 7.4 on page 75, all the CPUs have equal access to memory. Here, the memory access time is uniform. Second, there is non-uniform memory access (NUMA), where groups of CPUs are connected to the memory inside *NUMA domains*, see Fig. 7.5 on page 75. Here, the access of local memory is fast, but the access of non-local memory (i.e. memory from other NUMA domains) is slower. For most desktops and laptops, UMA is the norm. On most HPC resources, however, NUMA is more common. So, if NUMA is not taken into account, it is possible to see a performance drop when the number of cores requested is larger than a single NUMA domain. For more details about memory access, we refer to [86].

7.5 Parallelism Computer Architectures

A common term for the classification of parallel computer architectures is *Flynn's Taxonomy*. It was proposed by Flynn in 1966 [53] and later extended in 1972 [87]. Two of the architectures in Flynn's Taxonomy are *single instruction and single data* (SISD) and *Multiple Instruction and Single Data* (MISD). Today, the CPUs of all supercomputers in the top ten of the Top 500 list are a hybrid of *Multiple Instructions and Multiple Data* (MIMD) and SIMD (the latter part comes from their

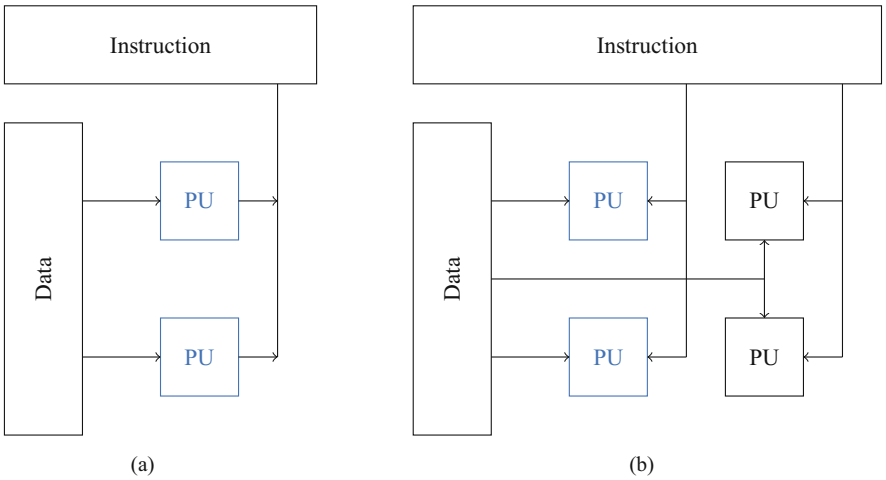


Fig. 7.6 Today's most common approaches to parallelism as described by Flynn's taxonomy: (**a**) Single instructions and multiple data (SIMD) and (**b**) Multiple instructions and multiple data (MIMD)

vector units). However, most of the compute power for the Top500 list comes from GPUs, which essential have SIMD architectures.

Figure 7.6 on page 76 provides an overview of the *Multiple Instructions and Multiple Data* MIMD and *Single Instruction and Multiple Data* (SIMD) architectures. For both architectures on the top is the instructions and on the left the data. The blue boxes show the processing units (PU). For SIMD a single instruction is applied to multiple different data, for more details, see Sect. 7.5.2. For MIMD multiple autonomous processors, different instructions are applied to different data.

Note that pipelined vector processing (*Pipelined SIMD*) is not included in Flynn's taxonomy, since the first processor was introduced in 1977 which is five years after Flynn's second paper was published. However, Duncan's taxonomy [54] from 1990 includes this architecture. On modern computers, a variety of tools (including an OpenMP directive and a proposed C++26 language extension) exist to aid in programming this type of SIMD device.

7.5.1 Pipelined SIMD

One approach to parallel computing is "pipeline parallelism." To see how this works, picture an assembly line in a factory. For the sake of this example, we will imagine assembling a car. The steps are *(1)* assemble the frame; *(2)* install the wheels; *(3)* install the engine; *(4)* install the seats, dashboard, and interior; *(5)* install the body panels. Obviously, this is a vast oversimplification to the process of building any real car, but it provides a good mental picture.

Now imagine that we have five sets of workers to accomplish this task. It is certainly possible for each of the five sets of workers to perform each step in sequence, and the result will be that they can assemble cars five times faster than a single set of workers. One problem with this approach, however, is that each worker must know how to do all the tasks, which means that the car factory's owner must invest more time and resources in training them. In addition, each set of workers will require access to a complete set of parts, meaning that they each must maintain independent supplies of inventory and supplies and require more storage and working space.

To improve the efficiency of the operation, each of the five sets of workers needs to specialize in just one task. This reduces the cost of training the workers, reduces the supply space they need, and the complexity of making sure they each have the required parts and resources. The pipeline should be able to produce cars at the same rate, but with lower overheads. Notice that while each set of workers must operate on each car, they can get the job done at lower cost.

Computer architectures make use of pipe lining as well, normally at a very low hardware level. The ARM processor can use a 3-stage pipeline with steps named `FETCH`, `DECODE`, and `EXECUTE`. At the hardware level, each operation to be performed requires actual circuitry to be fabricated, and so it makes sense to specialize for each task, similar to the training of car assembly workers in the example above. For example this principle is used for AltiVec, the SIMD instructions for IBM®'s power architecture; Neon or Scalable Vector Extensions (SVE), SIMD instructions for ARM® architecture; and AVX for Intel® and AMD® architectures [88].

7.5.2 Single Instruction Multiple Data (SIMD)

CPUs have a lot of sophisticated logic built in to accelerate serial code. They use branch prediction, out-of-order control logic, prefetching of data from cache, etc. While this is generally useful, sometimes it can get in the way of performing a lot of simple tasks quickly. One optimization employed on modern hardware is to have a large set of Arithmetic Logic Units (ALUs), and broadcast a single instruction to the entire group. These ALUs will then all perform that same operation in parallel. The term used for this is Single Instruction and Multiple Data (SIMD).

On a modern CPU, this array of ALUs is called a vector unit. The task of breaking up your code for appropriate execution on a SIMD architecture is handled automatically by the compiler during optimization. While the compiler often does a good job at this, you need to pay attention to compiler output to determine when and where it was successful. Typically, vector performance accounts for a factor of 16 in the peak performance of the CPU. Failure to make use of these units will prevent you from having a fast code. For combining parallelism with vectorization in HPX, we refer to Sect. 10.1.2.

Chapter 8
Programming with Low Level Threads

With the C++ 11 standard low level threads `std::thread`[1] were introduced. This was the first opportunity for writing parallel C++ code using the C++ standard. With the C++ 20 standard `std::jthread`[2] was introduced. This differs from `std::thread` in that `std::jthread` joins automatically on destruction. In some cases, it can be stopped as well. Before we dig into more details, we revisit the implementation of the Taylor series for the natural logarithm in Eq. (3.4). Listing 3.10 on page 31 shows the implementation using `std::for_each` and `std::accumulate` of the SL algorithms.

While using low-level threads, we can not use the SL algorithms from the previous example directly and need to divide the problem into sub problems which each thread can solve independently.

Figure 8.1 on page 80 illustrates the concept of dividing the work into partitions, concurrently computing the partitions, and collecting the results. In this example we want to launch three `std::threads`, e.g. T_1, T_2, and T_3. The `std::vector` in Listing 3.10 on page 31 is sketched as the grid on the top of the figure. Since the computation of each element is independent, we can divide the vector in three mostly equal partitions.

Listing 8.1 on page 81 shows the parallel implementation of the natural logarithm. In Line 13, the total amount of threads is defined which would be three in our illustration. In Line 15, the size of each partition for each thread is calculated. In our illustration we have `n` equal to 18 and `num_threads` equal to 3, so each thread would compute six elements. Next, all three threads are launched to compute their partitions concurrently as sketched in the middle of the figure. In Line 25 we use a `for` loop to launch the threads. In Lines 26–29 the first element and the last element of each partition are computed.

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

[2] https://en.cppreference.com/w/cpp/thread/jthread.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_8

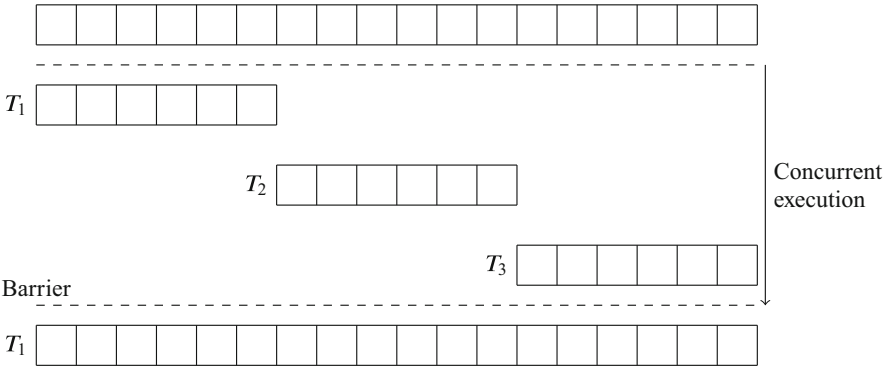


Fig. 8.1 Illustration of the division of work for concurrent computation using `std::thread`. In the top of the image the work items are shown. In this example, we have a `std::vector <double>` with 18 elements. The work items are equally distributed to three threads T_1, T_2, and T_3. The three threads work concurrently at the same time using different cores on their partition size of 6 elements. As each thread finishes, it arrives at a barrier. After all threads are finished, they exit the barrier and one thread T_1 collects the results

In Line 31 the threads are launched. Within each thread, we launch the same `std::for_each` function as in the serial implementation, except that the start and end position of the iterator varies. See Line 33. Note that now `num_threads` cores of our CPU work concurrently on their partitions. After launching each of the threads, we move them into the vector. See Line 38. Before we can accumulate the results, we need to make sure that all threads have finished their work. Therefore, in Line 42 we call `t.join()` on each of the threads. After the joins are complete, the original thread, T_1, collects the results. See Line 45. In this example, we use the SL algorithms to do the sum and leave it to the reader as an exercise to compute the sum in parallel using threads. Using `threads` as we have done in this example is low-level programming. It tends to be complex and error prone. For this reason, we recommend using parallel algorithms when possible or, failing that, asynchronous programming—a higher level abstraction which we will introduce in the next section. For more details about low-level programming, we refer to [89].

8.1 Implementation of the Fractal Sets

Listing 8.2 on page 82 shows the parallel implementation of the fractal set using low-level programming with `std::thread`. First, we generate a `std::vector <std::thread>` to collect all the launched threads. See Line 19. In Line 23, the work in the `X` direction is divided into partitions, just as in the last example. In the `for` loop in Line 25, a `std::thread` for each of the partitions is launched. Within the `for` loop, the first element of the partition (See Line 27) and the last element of the partition (See Line 28) are calculated, respectively. If the partitions cannot

Listing 8.1 Implementation of the Taylor series of the natural logarithm using `std::thread`

```
#include <algorithm>
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <numeric>
#include <string>
#include <thread>
#include <vector>

// Get the number of iterations
size_t n = 6;
// Get the amount of threads
size_t num_threads = 3;
// Compute the partition size for each thread
size_t partition_size = std::round(n / num_threads);
// the value of x
double x = 0.1;

std::vector<double> parts(n);
std::iota(parts.begin(), parts.end(), 1);

using std::thread;

std::vector<thread> threads;
for (size_t i = 0; i < num_threads; i++) {
  size_t begin = i * partition_size;
  size_t end = (i + 1) * partition_size;
  if (i == num_threads - 1)
    end = n;

  thread t([begin, end]() {
    std::for_each(parts.begin() + begin, parts.begin() + end,
                  [](double &e) {
                    e = std::pow(-1.0, e + 1) * std::pow(x, e) /
                        e;
                  });
  });

  threads.push_back(std::move(t));
}

for (thread &t : threads) {
  t.join();
}
// Collect the partial results once all  threads finished
double result = std::accumulate(parts.begin(), parts.end(), 0.);
```

Listing 8.2 Parallel implementation of the fractal set using `std::thread`

```
#include <pbm.hpp>
#include <config.hpp>
#include <kernel.hpp>

#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <numeric>
#include <pbm.hpp>
#include <thread>

-----------------------------------------------------------------------
PBM pbm = PBM(size_x, size_y);

std::vector<size_t> index_(size_x);
std::iota(index_.begin(), index_.end(), 0);

using std::thread;

std::vector<thread> threads;
const size_t nthreads = 4; // Choose the number

// Calculate the size of each partition
const int delx = index_.size() / nthreads;

for (size_t n = 0; n < nthreads; n++) {
  // Split into equal chunks
  const int nlo = n * delx;
  const int nhi = n + 1 == nthreads ? index_.size() : (n + 1) *
      delx;

  thread t([n,nlo,nhi]() {
    std::for_each(index_.begin() + nlo, index_.begin() + nhi, [n,
        nlo,nhi](size_t i) {
      complex c =
          complex(0, 4) * complex(i, 0) / complex(size_x, 0) -
              complex(0, 2);

      for (size_t j = 0; j < size_y; j++) {
        // Get the number of iterations
        int value = compute_pixel(c + 4.0 * j / size_y - 2.0);
        // Convert the value to RGB color space
        std::tuple<size_t, size_t, size_t> color=get_rgb(value);
        // Set the pixel color
        pbm(i, j) = make_color(std::get<0>(color),
                               std::get<1>(color),
                               std::get<2>(color));
      }
    });
  });
  threads.push_back(std::move(t));
}
for (thread &t : threads) {
  t.join();
}
```

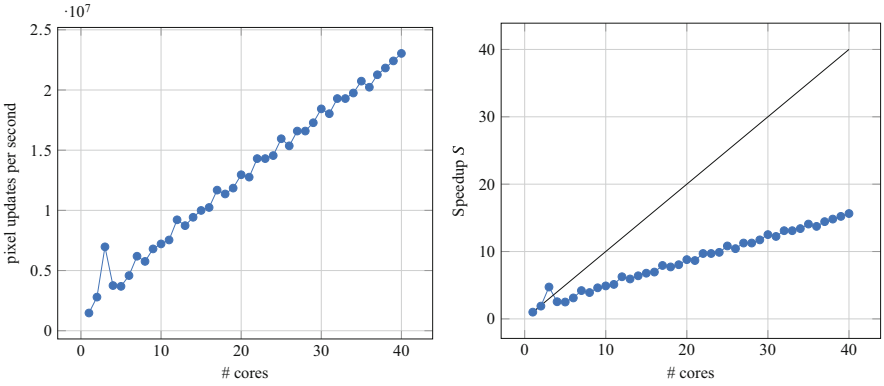


Fig. 8.2 Performance study for the fractal set using HPX's parallel algorithms. (**a**) shows the pixel updates per second for an increasing number of cores. (**b**) shows the speedup S with respect to the execution time on a single core. For each data point, the code was executed ten times and the median was plotted

be equally distributed to all threads, then the last thread gets slightly more work assigned. In Line 47, we use `std::move` to transfer the thread object to the vector and avoid generating a copy, since it is illegal to copy threads. Finally, we join the threads. See Line 50. The practice of splitting the work into equal partitions, launching concurrent threads to work on each partition, and joining the threads to collect the results is a common pattern for low-level thread programming.

Figure 8.2 on page 83 shows the performance of the implementation using `std::thread`. Figure 8.2 on page 83a shows the pixel updates per second (PUPS) with increasing amount of cores. PUPS increases up to 20 cores (with an outlier at three cores). We see a small drop in performance while going from one to two NUMA domains. After that the amount increases again. Figure 8.2 on page 83b shows the speedup S with respect to the execution time on a single core. While we see speedup using low-level threads, more optimal and safer code is possible with parallel algorithms or asynchronous programming.

Important

The most important lesson is that `std::thread` and `hpx::thread` are low-level application programmer interfaces and, if possible, should be avoided. However, the principle of dividing work into pieces, executing the pieces in parallel, and collecting the results remains a common and useful pattern.

Chapter 9
Asynchronous Programming

With the C++ 11 standard asynchronous programming was added as another tool to make use of shared memory parallelism. Asynchronous programming is a higher level abstraction layer, an alternative to the low-level thread implementation in the previous chapter. Before we delve into asynchronous programming, let us take another look at serial execution. Listing 9.1 on page 87 shows the computation of the Ultimate Question of Life, the Universe, and Everything. According to the book "The Hitchhiker's Guide to the Galaxy" from Douglas Adams [90] the computation took 7.5 million years on Deep Thought, the second greatest computer ever. The result was 42. In the serial computation, the function call `compute_answer()` will need to work for 7.5 million years before the result can be printed. In later writing by Douglas Adams, it was realized that the exact text of the question for which 42 is the answer is unknown. A second computation is needed to answer that. In our example, we first perform the computations sequentially, as they were in the books, and next we optimize the computations in the story and compute both in parallel.

For the serial execution, each line of code is executed in order, even if portions of the computation are independent of each other and could be computed concurrently. In this case, `compute_question` and `compute_answer` can be computed in parallel since one does not depend on the other.

For the low-level thread execution, the answer is computed in parallel with the question, and the result is placed in the shared variable `result`. The programmer needs to be careful to remember that the result will not be ready until the thread is joined, but for this simple program, that is not hard.

In asynchronous programming, functions are launched (on a parallel thread) with `std::async`[1] provided by the header `<future>`.[2] The result is a `std::`

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

[2] https://en.cppreference.com/w/cpp/header/future.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_9

`future`,[3] which is a proxy for the value to be computed, a place the value can be stored when the parallel computation finishes. The value of the function launched with `std::async` can be retrieved from the `std::future` by a call to `get()` (which may block).

In the example shown in Listing 9.1 on page 87, we show how asynchronous programming works in detail. Note that for this example, and the remainder of this book, we will normally use `hpx` instead of `std`, since all functions used for parallel computing, such as `hpx::async`, `hpx::future`, `hpx::thread`, etc. are interchangeable with `std::async`, `std::future`, etc.

In the parallel section of the example, a thread executes the code sequentially until it reaches the `hpx::async`[4] function call. The main thread then assigns the computation of the function `compute_answer` to a worker thread, see Line 32. The worker thread will then compute the Ultimate Question of Life, the Universe, and Everything. Since the function `compute_answer` has an integer a its `return` type, the asynchronous function launch returns a `hpx::future<int>`.[5] Within the future the result of the computation is accessed by calling `f.get()`, see Line 35. Here, one of two things could happen. First, if the computation has finished, the future will return the result 42 immediately. Second, if the computation has not finished, the call to `get()` will block and the main thread will wait until the worker thread finishes its computation. However, between the asynchronous function launch and the blocking call to `get()`, the main thread can launch other work while `compute_answer` is running, such as `compute_question`. In principle, each of the functions could contain other asynchronous function calls that perform other computations that might reduce the 7.5 million year wait.

The low-level thread example and the asynchronous example may not look that different, and for this simple example, they aren't. However, for larger scale applications one wants to keep the number of parallel threads to a value less than or equal to what is supported by the hardware for best efficiency. The `async` function can hide this detail by starting a pool of worker threads equal to the count supported by the hardware and farming its tasks out to this pool. Because `async` does not necessarily start a thread, it saves programmers this bookkeeping effort. Also, for larger applications, keeping track of which values are filled in by their respective threads and calling all the appropriate joins could become cumbersome. This is made somewhat worse by the fact that `thread` is not copyable and one must use `std::move()` to transfer it from one location to another. While this is also true of `std::future`, there is a `std::shared_future` which removes this limitation. In short, for larger, more complex applications, `std::async` should be both more efficient and more convenient for the programmer. In this section, we

[3] https://en.cppreference.com/w/cpp/thread/future.

[4] https://hpx-docs.stellar-group.org/latest/html/libs/core/async_base/api/async.html.

[5] https://hpx-docs.stellar-group.org/latest/html/libs/core/futures/api/future.html.

Listing 9.1 Example for sequential program execution and asynchronous program execution for the Ultimate Question of Life, the Universe, and Everything

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

auto compute_answer = []() -> int {
  // Compute the Ultimate Question of Life,
  // the Universe, and Everything.
  return 42;
};
auto compute_question = []() -> std::string {
  // compute the actual 'Ultimate Question'
  // which goes with the answer.
  return "What do you get when multiply 6 by 9?";
};

// Sequential execution
int answer = compute_answer();
// Wait for 7.5 million years...
std::string question = compute_question();
std::cout << "answer: " << answer << " question: " << question <<
     std::endl;

// Low-Level thread execution
hpx::thread answer_thread([&](){
    answer = compute_answer();
});
// While waiting for 7.5 million years...
// concurrent with compute_question
question = compute_question();
answer_thread.join();
std::cout << "answer2: " << answer << " question2: " << question
    << std::endl;

// Asynchronous execution
hpx::future<int> f = hpx::async(compute_answer);
// While waiting for 7.5 million years...
question = compute_question();
auto result = f.get();
std::cout << "answer3: " << result << " question3: " << question
    << std::endl;
```

have only touched on the functionality available for asynchronous programming. More advanced synchronization options are covered in Sect. 9.1.

For the implementation using asynchronous programming, we will use a concept similar to the one we used for the low-level threads. Figure 8.1 on page 80 follows the principle of dividing the work into equal partitions and executing these partitions in parallel. Listing 9.2 on page 88 shows the slightly modified code using `hpx::async` and `hpx::future` instead of `std::thread` as in Listing 8.1 on page 81. In

Listing 9.2 Parallel implementation of the natural logarithm using `hpx::async` and `hpx::future`

```
1  #include <pbm.hpp>
2  #include <config.hpp>
3  #include <kernel.hpp>
4
5  #include <algorithm>
6  #include <iostream>
7  #include <numeric>
8  #include <hpx/hpx.hpp>
9  ---------------------------------------------------------------
10 const size_t partition_size = 25;
11 const size_t num_partitions = 4;
12 std::vector<double> parts(partition_size*num_partitions);
13 std::iota(parts.begin(), parts.end(), 1);
14 const size_t partitions = 4;
15 std::vector<hpx::future<double>> futures;
16 double x = 0.1;
17 for (size_t i = 0; i < num_partitions; i++) {
18   size_t begin = i * partition_size;
19   size_t end = (i + 1) * partition_size;
20
21   hpx::future<double> f = hpx::async([begin, end]() -> double {
22     std::for_each(parts.begin() + begin, parts.begin() + end, [](
         double& e) {
23       e = std::pow(-1.0, e + 1) * std::pow(x, e) / (e);
24     });
25
26     return std::accumulate(parts.begin() + begin, parts.begin() +
          end, 0.);
27   });
28
29   futures.push_back(std::move(f));
30 }
31
32 double result = 0;
33 for (size_t i = 0; i < futures.size(); i++) result += futures[i].
     get();
34 std::cout << result << std::endl;
```

Line 15 of Listing 9.2 on page 88 we do not store `std::thread` in the vector and store `hpx::future<double>` instead. Note that having a `return` type using `hpx::async` is possible. For a `std::thread` the return value would need to be passed as an argument to the functions, since threads do not allow functions with return values. Here, we compute the partial some of each partition within the asynchronously launched function. Therefore, we added a return type to the lambda function, see Line 26. For the synchronization, we use a `for` loop to iterate over all futures in the vector and collect their partial results, see Line 33. In conclusion, only a few

changes were needed to move from the low-level threads to the higher abstraction level of asynchronous programming.

9.1 Advanced Synchronization in HPX

In this section, we will introduce some of the more advanced asynchronous programming features in HPX, namely `then()`, `when_all()`, and `when_any()`.

In Listing 9.3 on page 90 we see a variation on our Hitchhiker's Guide example with a call to `then()`. This provides an alternate way of accessing the result of `compute_answer`, one which does not block. Instead of waiting for an answer, `then()` adds the function provided by its argument to a vector maintained inside the future. When the future is ready, it calls all the methods in the vector. Note that the lambda we pass to `then` does not take an `int` as we might have expected (see Line 30). Instead, it takes a `future<int>`. However, the call to `result_.get()` on Line 31 can never block because the method is not called until it is ready.

So if it never blocks, why doesn't the lambda function passed to then take an `int` instead of a `future<int>`? The reason is that the computation of the `int` might throw an exception. If that happens, the exception is remembered by the `future<int>` object and rethrown here. So while `result_.get()` will not block on Line 31, it theoretically may throw an exception.

Using `then()` is potentially better than calling `get()` because it avoids blocking. While blocking is far less of a problem with HPX (thanks to HPX's ability to perform context switches on waiting threads), avoiding it is still a performance advantage.

Next, we will implement the computation of the natural logarithm using HPX to show how to use `when_all()`. Listing 9.4 on page 92 shows the prior implementation using HPX. The major difference in the new implementation is in Line 34 where we do the synchronization to collect all partial results. Here, we use `hpx::when_all`[6] and provide a `std::vector<hpx::future>`. In that case the HPX runtime will wait until all of the futures are ready. Once all the futures are ready, the lambda function in the `.then()` function is executed, see Line 35. Here, since all of the futures are ready, we accumulate all of the total results. Note that the `std::vector` added to the `hpx::when_all` function is returned to the `.then()` within a `hpx::future`, therefore, we need to call `.get()` in Line 36. Since we want to print the result in Line 46, we need to call `.get()` in Line 43 since `hpx::when_all` returns a future as well. So we have to make sure that the collection of the partial sums has finished, before we do the printing. The following advanced options for synchronization are available within HPX:

[6] https://hpx-docs.stellar-group.org/latest/html/libs/core/async_combinators/api/when_all.html.

Listing 9.3 Example for sequential program execution and asynchronous program execution for the Ultimate Question of Life, the Universe, and Everything

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

auto compute_answer = []() -> int {
  // Compute the Ultimate Question of Life,
  // the Universe, and Everything.
  return 42;
};
auto compute_question = []() -> std::string {
  // compute the actual 'Ultimate Question'
  // which goes with the answer.
  return "What do you get when multiply 6 by 9?";
};
hpx::future<int> result;
std::string question;

result  = hpx::async(compute_answer);
// While waiting for 7.5 million years...
question = compute_question();
std::cout << "Before get" << std::endl;
std::cout << "answer: " << result.get() << " question: " <<
    question << std::endl;
std::cout << "After get" << std::endl;

std::cout << "======================" << std::endl;

// Asynchronous execution
result  = hpx::async(compute_answer);
// While waiting for 7.5 million years...
question = compute_question();
hpx::future<void> done = result.then([&question](hpx::future<int>
     result_) {
    std::cout << "answer: " << result_.get() << " question: " <<
        question << std::endl;
});
sleep(1); // give the print time to happen
std::cout << "Before get" << std::endl;
done.get();
std::cout << "After get" << std::endl;
```

- `hpx::when_all`
  It returns a `future` that only becomes ready when all of the `futures` passed to `when_all()` become ready. The returned `future` contains a vector of all the ready futures.

- `hpx::when_any`[7]
  It returns a `future` that becomes ready when any of the `future`s passed to `when_any()` become ready. The returned `future` contains a vector of all the futures, as well as an index to the ready future.
- `hpx::when_each`[8]
  It returns a `future<void>` that only becomes ready when all of the `future`s passed to `when_each()` become ready. Additionally, it applies the callback method passed to `when_each()` to each value as it becomes ready.

The function `when_all()` is great if you want to wait for a group of futures that all have the same type, but what if the futures have different types? For this case we have `hpx::dataflow`.[9] Using our Hitchhiker code, we can provide an example that shows how to use it. See Listing 9.5 on page 93. In this example, we launch both the question and answer as asynchronous computations. This gives futures of two different types: `future<int>` and `future<string>`. In the call to `dataflow`, the first argument is a lambda, and the arguments that lambda takes must match the remainder of the arguments to dataflow. The lambda will only be called when all of its arguments (futures) are ready. As before, we use the `sleep(1)` call to make this clear. The print of the question and answer occurs before the `"Before get"` text prints.

For the case in which the programmer can be certain that exceptions will not be thrown during the computation of the futures passed to a function, HPX provides `hpx::unwrapping`.[10] This function "unwraps" the futures by calling `.get()` and passing the results to the function. In Line 20 of Listing 9.6 on page 94, we show the unwrapped version of the lambda call, with no reference to `hpx::future`.

The last feature provided by HPX is `hpx::make_ready_future`.[11] For example to set boundary conditions or initial values in simulations, it can be handy to have a `hpx::future` to set an value without any computational efforts and have this future immediately ready when the `.get()` function is called on it. Listing 9.7 on page 94 shows the usage of a `hpx::future<int>` were a value of one is set. The output in the last line happens right away since the future is ready.

[7] https://hpx-docs.stellar-group.org/latest/html/libs/core/async_combinators/api/when_any.html.

[8] https://hpx-docs.stellar-group.org/latest/html/libs/core/async_combinators/api/when_each.html.

[9] https://hpx-docs.stellar-group.org/latest/html/libs/core/async_base/api/dataflow.html.

[10] https://hpx-docs.stellar-group.org/latest/html/libs/core/pack_traversal/api/unwrap.html.

[11] https://hpx-docs.stellar-group.org/latest/html/libs/core/futures/api/future.html?highlight=make_ready_future#_CPPv4N3hpx17make_ready_futureEv.

Listing 9.4 Parallel implementation of the natural logarithm using `hpx::async`, `hpx::future`, and `hpx::when_all`

```
#include <pbm.hpp>
#include <config.hpp>
#include <kernel.hpp>

#include <hpx/future.hpp>
#include <hpx/hpx.hpp>
#include <hpx/numeric.hpp>
#include <iostream>

---

const size_t num_partitions = 10;
const size_t partition_size = 25;
const size_t n = num_partitions*partition_size;
std::vector<double> parts(n);
std::iota(parts.begin(), parts.end(), 1);

double x=0.1;

std::vector<hpx::future<double>> futures;
for (size_t i = 0; i < num_partitions; i++) {
  size_t begin = i * partition_size;
  size_t end = (i + 1) * partition_size;

  hpx::future<double> f = hpx::async([begin, end]() -> double {
    std::for_each(parts.begin() + begin, parts.begin() + end, [](
        double& e) {
      e = std::pow(-1.0, e + 1) * std::pow(x, e) / (e);
    });

    return hpx::reduce(parts.begin() + begin, parts.begin() + end
        , 0.);
  });

  futures.push_back(std::move(f));
}

double result = hpx::when_all(futures)
    .then([](auto&& f) {
      auto futures = f.get();

      double result = 0;
      for (size_t i = 0; i < futures.size(); i++)
        result += futures[i].get();
      return result;
    })
    .get();

std::cout << "Difference of Taylor and C++ result " << result -
    std::log1p(x)
          << " after " << n << " iterations." << std::endl;
```

Listing 9.5 Example for sequential program execution and asynchronous program execution for the Ultimate Question of Life, the Universe, and Everything

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

auto compute_answer = []() -> int {
  // Compute the Ultimate Question of Life,
  // the Universe, and Everything.
  return 42;
};
auto compute_question = []() -> std::string {
  // compute the actual 'Ultimate Question'
  // which goes with the answer.
  return "What do you get when multiply 6 by 9?";
};
hpx::future<int> answer;
hpx::future<std::string> question;

answer  = hpx::async(compute_answer);
question = hpx::async(compute_question);
hpx::future<void> result = hpx::dataflow([](hpx::future<int> a,
    hpx::future<std::string> q) {
    std::cout << "Q: " << q.get() << " A: " << a.get() << std::
        endl;
}, answer, question);
sleep(1);

std::cout << "Before get" << std::endl;
result.get();
std::cout << "After get" << std::endl;
```

To conclude, HPX provides many useful features for the synchronization of `hpx::futures`. In the next section, we will show different implementations of the fractal sets, both using the features of the C++ standard and the advanced features provided by HPX. It is important to emphasize that the HPX features are not yet in the C++ standard.

9.2 Implementation of the Fractal Sets

We implemented the fractal set code using both the asynchronous programming provided by the C++ standard and using the features provided by HPX. Later, we compare the performance difference of both implementations in Fig. 9.1 on page 95. Listing 9.8 on page 96 shows the asynchronous implementation of the fractal set using either `hpx::` or `std::` depending on whether the symbol `USE_HPX` is defined in the preprocessor. See Line 10.

Listing 9.6 Example for sequential program execution and asynchronous program execution for the Ultimate Question of Life, the Universe, and Everything

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

auto compute_answer = []() -> int {
  // Compute the Ultimate Question of Life,
  // the Universe, and Everything.
  return 42;
};
auto compute_question = []() -> std::string {
  // compute the actual 'Ultimate Question'
  // which goes with the answer.
  return "What do you get when multiply 6 by 9?";
};
hpx::future<int> answer;
hpx::future<std::string> question;

answer  = hpx::async(compute_answer);
question = hpx::async(compute_question);
hpx::future<void> result = hpx::dataflow(hpx::unwrapping(
    [](int a,std::string q) {
    std::cout << "Q: " << q << " A: " << a << std::endl;
}), answer, question);
sleep(1);

std::cout << "Before get" << std::endl;
result.get();
std::cout << "After get" << std::endl;
```

Listing 9.7 A `hpx::future` which is ready immediately by using `hpx::make_ready_future`

```
#include <iostream>
#include <hpx/future.hpp>

auto f = hpx::make_ready_future(1);
/*
* Since the future is ready the output will happen
* without any waiting.
*/
std::cout << f.get() << std::endl;
```

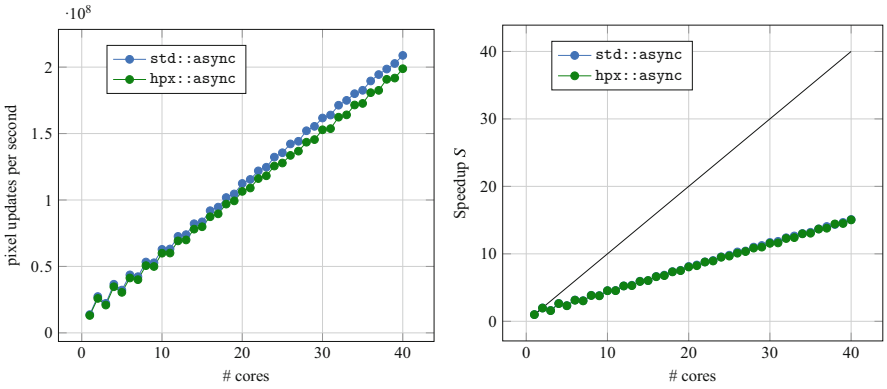


Fig. 9.1 Performance study for the fractal set using asynchronous programming. Figure (**a**) shows the pixel updates per second for an increasing number of cores. Figure (**b**) shows the speedup S with respect to the execution time on a single core. For each data point, the code was executed ten times and the median was plotted. The blue line shows the implementation using `std::async`, and the red line the implementation using `hpx::async`, respectively

Stepping through the code, in Line 41 we declare a `std::vector` to collect all the `hpx::futures<std::vector<int>>`. In Line 47, we launch our computations asynchronously, splitting the work up by partition as before. In Line 48, we store the future in the vector using `std::move`. We do this because futures are not copyable. Next, we wait for all futures to become ready (see Line 50). Note that the use of `wait()` does not retrieve the value from the future, it simply blocks until the future is ready. When all futures are ready, the color of all pixels have been computed and the image can be saved. We avoid measuring the time to save the file for benchmarking purposes.

Figure 9.1 on page 95 shows the performance measurements for the fractal sets implemented using `std::async` and `hpx::async`. Figure 9.1a on page 95 shows the pixel updates per second for the implementation using `std::async` (blue) and using `hpx::async` (green), respectively. For both lines, the number of pixel updates per second increased with the number of cores. However, we see a slightly better performance of `hpx::async` due to the light-weight threads instead of the system threads. Figure 9.1b on page 95 shows the speedup with respect to a single node. In addition to the better performance, the advanced synchronization functionality provided by HPX makes programming more convincing.

Listing 9.8 An asynchronous fractal code that can be compiled and run for either HPX or the standard library

```
#define USE_HPX 1
#include <algorithm>
#include <future>
#include <iostream>
#include <numeric>
#include <pbm.hpp>
#include "config.hpp"
#include "kernel.hpp"

//----------------------------------------------------------------------
#if USE_HPX
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
namespace par = hpx;
#else
namespace par = std;
#endif

//----------------------------------------------------------------------
void launch(size_t begin, size_t end, PBM* pbm) {
  for (size_t i = begin; i < end; i++) {
    complex c = complex(0, 4) * complex(i, 0) /
        complex(size_x, 0) - complex(0, 2);

    for (size_t j = 0; j < size_y; j++) {
      int value = compute_pixel(c + 4.0 * j / size_y - 2.0);
      // Convert the value to RGB color space
      std::tuple<size_t, size_t, size_t> color = get_rgb(value);
      // Set the pixel color
      (*pbm)(i, j) = make_color(std::get<0>(color),
                                std::get<1>(color),
                                std::get<2>(color));
    }
  }
}

int main(int argc, char* argv[]) {
  size_t partitions = get_size_t("NUM_THREADS", 3);
  size_t output = get_size_t("OUTPUT", 1);
  PBM pbm = PBM(size_x, size_y);

  size_t size = std::round(size_x / partitions);
  std::vector<par::future<void>> futures;
  auto start = std::chrono::high_resolution_clock::now();
  for (size_t i = 0; i < partitions; i++) {
    size_t start = i * size;
    size_t end = (i + 1) * size;
    if (i == partitions - 1) end = size_x;
    auto f = par::async(launch, start, end, &pbm);
    futures.push_back(std::move(f));
  }
  for (auto&& f : futures) f.wait();
  auto stop = std::chrono::high_resolution_clock::now();
  auto duration =
```

```
53        std::chrono::duration_cast<std::chrono::microseconds>(stop
              - start);
54    std::cout << duration.count() * 1e-6 << std::endl;
55
56    // Save the image
57    if (output == 1) pbm.save("image_future_parallel_" + type + ".
          pbm");
58
59    return EXIT_SUCCESS;
60  }
```

Important

The most important lesson learned here is that asynchronous programming is an abstraction layer for the low level programming with threads in Chap. 8. Preferable, asynchronous programming should be used to implement the scientific application. However, since asynchronous programming is just some abstraction level the performance will be comparable for most cases.

Chapter 10
Parallel Algorithms

With the C++ 17 standard, parallel algorithms were introduced. We discussed them previously in Sect. 3.3. Starting with the C++ 17 standard, a parallel version of 69 algorithms is provided in the headers `#include <algorithm>`, `#include <numeric>`, and `#include <memory>`. Parallel algorithms were implemented in the GNU compiler collection (gcc) version 9 and kept as an experimental feature in gcc 10. For gcc $\geq$ 11 (the version we tested with the codes in this book), the latest version of parallel algorithms are fully supported and the flag `-cpp=c++17` (the default) is sufficient. The GNU C++ standard library (`libstdc++`) uses Intel's oneAPI Threading Building Blocks (TBB) library for parallelism. Note that TBB needs to be installed independently and is not included in the gcc installation.

For our first example of using parallel algorithms, we revisit the Implementation of the Taylor series for the natural logarithm in Eq. 3.4 on page 30. Listing 3.10 on page 31 shows the implementation using `std::for_each` and `std::accumulate` of the SL algorithms. Listing 10.1 on page 100 shows the parallel implementation. In Lines 14 and 18, the execution policy `std::execution::par` was added to the algorithms as the first argument. The execution policies are provided by the header `#include <execution>`.[1] Now both algorithms are executed in parallel using TBB.

Since the C++ 17 standard, the following execution policies are provided

- `std::execution::seq`
  The algorithm will be executed sequentially using a single thread as in the C++ SL algorithms.
- `std::execution::par`
  The algorithm will be executed in parallel using multiple threads using TBB.
- `std::execution::par_unseq`
  The algorithm is executed in parallel using multiple threads and vectorization is used.

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

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_10

Listing 10.1 Implementation of the Taylor series of the natural logarithm using parallel algorithms

```
#include <vector>
#include <cmath>
#include <numeric>
#include <algorithm>
#include <hpx/hpx.hpp>

using hpx::for_each, hpx::execution::par, hpx::reduce;

const int n = 400;
const double x = 0.1;
std::vector<double> parts(n);
std::iota(parts.begin(), parts.end(), 1);

for_each(par, parts.begin(), parts.end(), [x](double& e) {
    e = std::pow(-1.0, e + 1) * std::pow(x, e) / (e);
});

double result = reduce(par, parts.begin(), parts.end(), 0.);
```

With the C++ 20 standard, the execution policy `std::execution::unseq` was added. Using this policy, the algorithm is executed sequentially, but using vectorization. Additional execution policies, e.g. `std::parallel::cuda` and `std::parallel::opencl` might be added someday. In conclusion, with the parallel algorithms, most of the existing algorithms of the C++ SL can be easily executed in parallel.

10.1 Parallel Algorithms in HPX

The parallel algorithms are implemented by the C++ Standard Library for parallelism and concurrency (HPX) using light-weight threads. To use HPX instead of the C++ SL, one needs to replace the namespace `std` by the namespace `hpx`. Since HPX fully conforms with the C++ standard, there is no need for additional changes. The corresponding header are `#include <hpx/algorithm.hpp>`[2] and `#include <hpx/numeric.hpp>`,[3] respectively.

Let us, revisit the example for computing the power of two for all elements in a array in Listing 7.1 on page 61. Here, we show several different tools that provide a parallel version of the `for` loop. These are similar in function to the `std::for_each` or `hpx::for_each` loops. However, for `for_each`, one

[2] https://hpx-docs.stellar-group.org/latest/html/api/public_api.html?highlight=numeric.

[3] https://hpx-docs.stellar-group.org/latest/html/api/public_api.html?highlight=numeric#hpx-numeric-hpp.

Listing 10.2 Implementation of the Taylor series of the natural logarithm using `hpx::experimental::for_loop`

```
1  #include <array>
2  #include <hpx/parallel/algorithm.hpp>
3
4  std::array<int, 100000> a;
5
6  hpx::experimental::for_loop(
7          hpx::execution::par,
8          0, a.size(), [&](size_t i) {
9          a[i] = i * i ;
10 });
```

cannot iterate over the elements of the container using the index. This is a problem for stencil codes, for example, where one needs to combine values from neighboring indices. HPX provides a parallel version of the `for` loop which is not in the current C++ standard. Listing 10.2 on page 101 shows the usage of the HPX `hpx::experimental::parallel_for`[4] provided by the header `#include <hpx/algorithms.hpp>` to compute the power of two for each element using the index. In Line 8, the range of the loop where 0 is the start index and $a.size()$ the last index, which is similar to the standard `for` loop.

10.1.1 Combining Parallel Algorithms and Asynchronous Programming

There is another feature in HPX which is not in the C++ standard. In HPX the parallel algorithms can be combined with asynchronous programming, discussed in Chap. 9. In that case, the parallel algorithm is started asynchronously and returns an `hpx::future`. Let us revisit the previous example, but let us make the launch of the parallel for loop asynchronous. Listing 10.3 on page 102 shows the modified code. In Line 11 the execution policy `hpx::execution::par` is used as well, but an argument `hpx::execution::task` was added. In that case the `hpx::experimental::for_loop` returns a `hpx::future<void>` and the next line of code is executed. While the asynchronous for loop is running, any other computation could happen (e.g. `compute_question()`). Once the result of the parallel for loop is needed, for example to compute the sum of the result, the returned future is used. In Line 19 the future `f` is used with `.then()` to execute the computation of the sum in serial. We could put a call to `wait()` or `get()` on the

[4] https://hpx-docs.stellar-group.org/latest/html/libs/core/algorithms/api/for_loop.html.

Listing 10.3 Implementation of the Taylor series of the natural logarithm using HPX's parallel algorithms

```
#include <array>
#include <cmath>
#include <hpx/hpx.hpp>
#include <hpx/numeric.hpp>
#include <hpx/parallel/algorithm.hpp>

std::array<int, 100000> a;

hpx::future<void> f =
  hpx::experimental::for_loop(
    hpx::execution::par(hpx::execution::task),
    0,
    a.size(),
    [&](size_t i) {
      a[i] = i * i ;
    });

// Use the future for synchronization
f.then([a](auto&& f) {
        std::cout << hpx::reduce(std::begin(a), std::end(a), 0)
            << std::endl;
});
```

final line of the program, but it is not necessary. HPX detects when there is no more work to do and shuts down automatically.

10.1.2 Single Instruction Multiple Data

Another way to parallelize the code is to use Single Instruction and Multiple Data (SIMD). SIMD is one of the types for parallel processing specified in Flynn's Taxonomy. See [53]. Here, multiple data are collected into a special vector data type, and the same instruction is performed on all the data in the vector at once. Let us consider a `for` loop in C++ to compute $c[i] = a[i] + b[i]$ for $i = 1, \ldots, n$. To use vector operations, the for loop will not be executed line by line for each i in the loop range. Instead multiple data from the loop is packed into a vector

$$c_0 = a_0 + b_0$$
$$c_1 = a_1 + b_1$$
$$c_2 = a_2 + b_2$$

and all of them are evaluated at the same time. Most C++ compilers provide automatic vectorization [91, 92]. The compiler transforms the code, e.g. `for` loops, into SIMD instructions. However, the programmer does not have much control over

Listing 10.4 Using SIMD within HPX's parallel algorithms via execution policies

```
1  #include <array>
2  #include <hpx/hpx.hpp>
3  #include <hpx/hpx_main.hpp>
4  #include <hpx/numeric.hpp>
5  #include <hpx/parallel/algorithm.hpp>
6  #include <hpx/parallel/datapar.hpp>
7
8  int main() {
9    std::array<int, 100000> a;
10
11   std::cout << hpx::reduce(hpx::execution::simd, std::begin(a),
         std::end(a), 0) << std::endl;
12
13   std::cout << hpx::reduce(hpx::execution::par_simd, std::begin(a
         ), std::end(a), 0) << std::endl;
14 }
```

when or to what extent that transformation occurs. One experimental feature of C++ is the vector register `std::experimental::simd`[5] feature. This is a template class that provides support for all built-in numerical data types, e.g. `int`, `float`, and `double`. For more details, we refer to Extensions for Parallelism Version 2 (ISO/IEC TS 19570:2018).[6]

HPX implements the proposed extension to the C++ standard to integrate SIMD with parallel algorithms. Listing 10.4 on page 103 shows two additional execution policies implemented in HPX to provide SIMD to the parallel algorithms. Line 11 shows how the execution policy `hpx::execution::simd` is passed to the algorithm. Line 13 shows the execution policy `hpx::execution::par_simd`, which combines parallelism with vectorization. In addition to `std::experimental::simd`, HPX provides the two following SIMD backends, Vc [93] and Eve [94], respectively. The complete code of the example is available here.[7] Note that you need to compile HPX with a gcc that has a version $\geq$11 or clang with a version $\geq$ 11. In addition, you will need the following CMake options for HPX: `-DHPX_WITH_CXX_STANDARD=20 -DHPX_WITH_DATAPAR_BACKEND=STD_EXPERIMENTAL_SIMD`. For more implementation details, we refer to [55].

[5] https://en.cppreference.com/w/cpp/experimental/simd/simd.

[6] https://en.cppreference.com/w/cpp/experimental/parallelism_2.

[7] https://github.com/ModernCPPBook/Examples/blob/main/hpx/parallel_reduce_simd.cpp.

10.2 Additional Parameters for the Execution Policies

The execution policies for parallelism and SIMD are already in the C++ standard. However, HPX provides additional parameters for the execution policies. While these extensions are not defined by the C++ standard yet, they provide optimization opportunities depending on the application.

For example, the programmer can specify the chunk size, the number of elements a single thread operates on. This parameter might provide a slightly less machine-dependent way to parallelize a computation. By setting a large enough chunk size, each thread is guaranteed to have enough work to offset the overhead it introduces. The following options are provided

- `hpx::execution::static_chunk_size`[8]
  Each thread is assigned a number of elements equal to (at most) the `chunk_size`.
- `hpx::execution::auto_chunk_size`[9]
  The number of elements assigned to each thread is determined after 1% of the total elements are executed.
- `hpx::execution::dynamic_chunk_size`[10]
  The elements are dynamically scheduled among the threads. As each thread finishes, it requests a new chunk.

For more details about the effect of chunk sizes on HPX's parallel algorithms performance, we refer to [95]. Another option to determine the chunk size is to use machine learning techniques [50, 51]. Listing 10.5 on page 105 shows the usage of the chunk sizes with the execution policy `hpx::execution::par` by adding the chunk size by using the function `.with()`. Line 8 shows how to define a static chunk size with a length of ten. In Line 9 the chunk size is passed to the execution policy. Line 13 shows the usage of the dynamic chunk size.

10.3 Implementation of the Fractal Sets

Let us revisit the serial implementation of the fractal set code in Listing 4.2 on page 39. We could try to replace the `for` in Line 18 with `std::for_each()` in Line 12, but `for_each` does not provide us with the loop index. Therefore, we use `for_loop`. Listing 10.6 on page 105 shows the revised code using HPX's parallel algorithms. In Line 11 we see the revised loop.

[8] https://hpx-docs.stellar-group.org/latest/html/libs/core/execution/api/static_chunk_size.html.

[9] https://hpx-docs.stellar-group.org/latest/html/libs/core/execution/api/auto_chunk_size.html.

[10] https://hpx-docs.stellar-group.org/latest/html/libs/core/execution/api/dynamic_chunk_size.html.

Listing 10.5 Using chunk sizes within HPX's parallel algorithms

```
#include <array>
#include <hpx/hpx.hpp>
#include <hpx/parallel/algorithm.hpp>

std::array<int, 100000> a;

// Static chunk size
hpx::execution::static_chunk_size scs(10);
hpx::experimental::for_loop(hpx::execution::par.with(scs), 0, a.
    size(),
                [&](size_t i) { a[i] = i * i; });

// Dynamic chunk size
hpx::execution::dynamic_chunk_size dcs(10);
hpx::experimental::for_loop(hpx::execution::par.with(dcs), 0, a.
    size(),
                [&](size_t i) { a[i] = i * i; });
```

Listing 10.6 Example for the computation of the fractal sets using HPX's parallel algorithms

```
#include <hpx/parallel/algorithm.hpp>

#include <pbm.hpp>
#include <config.hpp>
#include <kernel.hpp>

// Definition of utility
PBM pbm = PBM(size_x, size_y);

hpx::experimental::for_loop(hpx::execution::par,0,size_x, [&pbm](
    size_t i) {


    complex c =
        complex(0, 4) * complex(i, 0) / complex(size_x, 0) -
            complex(0, 2);

    for (size_t j = 0; j < size_y; j++) {
      // Get the number of iterations
      int value = compute_pixel(c + 4.0 * j / size_y - 2.0);
      // Convert the value to RGB color space
      std::tuple<size_t, size_t, size_t> color = get_rgb(value);
      // Set the pixel color
      pbm(i, j) = make_color(std::get<0>(color),
                             std::get<1>(color),
                             std::get<2>(color));
    }
  });

  // Save the image
  pbm.save("image_parallel_" + type + ".pbm");
```

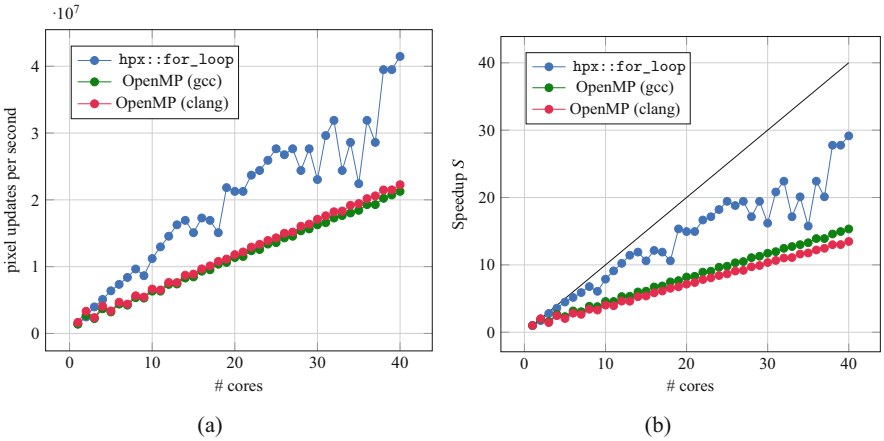


Fig. 10.1 Performance study for the fractal set using HPX's parallel algorithms. Figure (**a**) shows the pixel updates per second for an increasing number of cores. Figure (**b**) shows the speedup S with respect to the execution time on a single core. For each data point, the code was executed ten times and the median was plotted

Figure 10.1 on page 106 shows the performance of the parallel fractal set code using HPX's parallel algorithms. We increased the image size to 38,400 pixels in x-direction and 21,600 y-direction to ensure sufficient work to scale out to all 40 cores of our node. Figure 10.1 on page 106 shows the number of pixel updates per second from 1 to all 40 cores of the machine. Figure 10.1 on page 106 shows the speedup S with respect to the execution time on a single core. The black line indicates the optimal speedup. In conclusion, with only minimal changes in the code, the parallel algorithms made it easy to parallelize this particular code, comparable in ease to writing annotations with OpenMP.

Listing 10.7 on page 107 shows the OpenMP implementation of the code. See Listing 4.2 on page 39. All we did was add the annotation `#pragma omp parallel for` above the `for`. The code is available here.[11] For our benchmark tests, we used the OpenMP implementation of gcc and clang. For more implementation details about using OpenMP, we refer to [96]. For each datapoint in the graph, the code was executed ten times and the median was plotted. For more details about the software versions and hardware, we refer Appendix C.

[11] https://github.com/ModernCPPBook/Examples/blob/main/benchmark/openmp_parallel.cpp.

Listing 10.7 Example for the computation of the fractal sets using OpenMP

```
#include <chrono>

#include <config.hpp>
#include <kernel.hpp>
#include <pbm.hpp>

// Definition of utility
PBM pbm = PBM(size_x, size_y);

#pragma omp parallel for
  for (size_t i = 0; i < size_x; i++) {
    complex c =
        complex(0, 4) * complex(i, 0) / complex(size_x, 0) -
            complex(0, 2);

    for (size_t j = 0; j < size_y; j++) {
      // Get the number of iterations
      int value = compute_pixel(c + 4.0 * j / size_y - 2.0);
      // Convert the smoothened value to RGB color space
      std::tuple<size_t, size_t, size_t> color = get_rgb(value);
      // Set the pixel color
      pbm(i, j) = make_color(std::get<0>(color),
                                    std::get<1>(color),
                                    std::get<2>(color));
    }
  }

// Save the image
pbm.save("image_parallel_" + type + ".pbm");
```

Figure 10.2 on page 108 shows the performance of the Mandelbrot set for different chunk sizes as described in Sect. 10.2. The blue shows the default parallel option, a `hpx::execution::static_chunk_size` of one. The green line shows the pixel updates per second for `hpx::execution::static_chunk_size` of size five. The red line shows the pixel updates per second for `hpx::execution::dynamic_chunk_size` of size five. We see that the dynamic chunk size is marginally faster than the static chunk size. However, both of them show less fluctuation for higher core counts than static with chunk size one. We observed that the chunk size influences the application's performance. However, predicting the optimal chunk size for an application is a challenge since the optimal chunk size depends, for example, on the number of cores or the architecture of the CPU. For an attempt at predicting the optimal chunk size, we refer to [97]. A common option is to run the parallel algorithm using various chunk sizes to identify the sweet spot. Figure 10.2 on page 108 shows the performance using 20 cores and 40 cores for various chunk sizes.

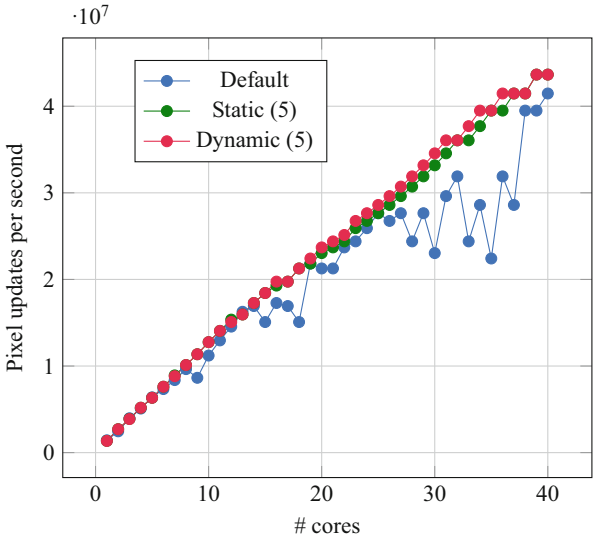


Fig. 10.2 Performance study for the fractal set using HPX's parallel algorithms. The figure shows the pixel updates per second for an increasing number of cores. Here, we use the parallel execution policy with three different chunk sizes to evaluate the influence of the chunk size on the performance

Important

The most important lesson here is that many of the SL algorithms are available as parallel algorithms with the C++ 17 standard. HPX allows for non-standard options such as modifying chunk sizes, `hpx::execution::static_chunk_size`, `hpx::execution:: auto_chunk_size`, `hpx::execution::dynamic_chunk_size`; and asynchronous calls of the algorithms using `hpx::execution::task`. In addition, single instruction multiple data (SIMD) options are available: `hpx::execution::simd` and `hpx::execution::par_simd`.

Chapter 11
Coroutines

Starting with C++ 20, basic support for coroutines[1] was introduced. The basic functionality exists in multiple languages and has been available in Python [98] and Javascript for years. A coroutine is a C++ function which can be suspended and resumed later on. A coroutine is like a C++ function with a special type of `return` value. Coroutines can use one or more of these three new keywords:

- `co_return`
- `co_yield`
- `co_await`

We won't discuss the special return value in detail. However, an `hpx::future` is suitable for use with `co_await` and `co_return`. The statement `co_return` is similar to the `return` statement. If the return type of a function is `hpx::future<int>`, for example, then one can call `co_return 3` or `return hpx::make_ready_future(3)`. The statement `co_yield` returns the expression to the caller and suspends the current coroutine. The statement `co_await` suspends the coroutine and returns the control to the caller. In Listing 11.1 on page 110, we see an example of an extremely inefficient Fibonacci function that uses coroutines. The call to `co_await` functions similarly to the call to future's `.get()` method, except that `co_await` is expected to suspend the thread while `.get()` is expected to block. In HPX, however, even the call to `.get()` will suspend, so the difference is largely syntactic.

We note that for Listing 11.1 on page 110, only one of the three implementations can run with the SL (`fib1`), and that one is slower than any of the three using HPX in gcc 11. In addition, the version from the SL will fail with an error when used with some modestly small argument to Fibonacci. While this is a foolishly slow implementation of Fibonacci, it is good for stressing the threading library's overheads and limitations when using extreme numbers of threads.

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

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_11

Listing 11.1 Parallel Fibonacci done three ways

```
#define USE_HPX 1
#ifdef USE_HPX
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include <coroutine>
namespace par = hpx;
#else
#include <future>
namespace par = std;
#endif
#include <functional>
#include <iostream>

---

int fib1(int n) {
  if(n < 2) return n;
  par::future<int> f1 = par::async(par::launch::async,fib1,n-1);
  auto f2 = fib1(n-2);
  return f1.get() + f2;
}

#ifdef USE_HPX
par::future<int> fib2(int n) {
  if(n < 2) return par::make_ready_future(n);
  par::future<int> f1 = par::async(par::launch::async,fib2,n-1);
  auto f2 = fib2(n-2);
  return par::dataflow([](auto f1_,auto f2_){
    return f1_.get() + f2_.get();
  }, f1, f2);
}

par::future<int> fib3(int n) {
  if(n < 2) co_return n;
  auto f1 = par::async(fib3,n-1);
  auto f2 = fib3(n-2);
  co_return (co_await f1) + (co_await f2);
}
#endif

void test_fib(int n,std::function<int(int)> f) {
  auto start_time = std::chrono::high_resolution_clock::now ();
  int r = f(n);
  auto stop_time = std::chrono::high_resolution_clock::now ();
  auto duration = std::chrono::duration_cast<std::chrono::
      nanoseconds>(stop_time - start_time);
  std::cout << "fib(" << n << ")=" << r << " time=" << (duration.
      count()*1e-9) << " secs" << std::endl;
}

```

```
47 int main() {
48   for(int i=0;i<20;i++) {
49     test_fib(i, fib1);
50     #ifdef USE_HPX
51     test_fib(i, [](int n)->int { return fib2(n).get(); });
52     test_fib(i, [](int n)->int { return fib3(n).get(); });
53     #endif
54     std::cout << std::endl;
55   }
56   return 0;
57 }
```

Note that because of the requirement for the return type for coroutines, the `main` function as well as constructors/destructors of classes or structs can not be coroutines. In addition, one can neither use `auto`[2] for automatic type deduction nor variadic arguments,[3] `double values...`, within coroutines.

Next, we provide a coroutine example based on the previously described asynchronous implementation of the Taylor series of the natural logarithm. See Listing 9.4 on page 92. Listing 11.2 on page 112 shows the new function, `run`, which is needed since we can not use `co_return` and `co_await` within the `main` function. We moved all the functions to split the computations and gather the partial results into this function. The first change is in Line 36, where instead of the function `.then()`, we use the `co_await` statement to suspend the thread while we wait for all futures to be finished. The `co_await` statement returns the vector of futures after all the futures become ready. In Line 38 we use the `co_await` statement to suspend the thread while gathering the results. Finally, we use the `co_return` statement in Line 40 to return the result and the coroutine is suspended. Note that the function `run` has to return a `hpx::future<void>` if we use `co_await` within the function's body, and that's what `co_return` with no argument does. Be aware that functions that return futures can only be used as coroutines while using HPX. Note, that to use coroutines with HPX, you need to compile HPX with gcc $\geq$11 or clang $\geq$ 11 and add the following CMake options `-DHPX_WITH_CXX_STANDARD=20`. In conclusion, the implementation using coroutines is not much different from the one using HPX's asynchronous programming in Chap. 9. This is due to the fact that, syntactically, coroutines provide a similar abstraction level for asynchronous programming using futures.

[2] https://en.cppreference.com/w/cpp/language/auto.

[3] https://en.cppreference.com/w/cpp/language/variadic_arguments.

Listing 11.2 Example for the computation of the Taylor series for the natural logarithm using HPX's futures and coroutines

```
#include <coroutine>
#include <vector>
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include <hpx/numeric.hpp>
#include <hpx/parallel/algorithm.hpp>
#include <iostream>

hpx::future<double> run(size_t n, size_t amount, double x) {
  std::vector<double> parts(n);
  std::iota(parts.begin(), parts.end(), 1);

  size_t partitions = std::round(n / amount);

  std::vector<hpx::future<double>> futures;
  for (size_t i = 0; i < amount; i++) {
    size_t begin = i * partitions;
    size_t end = (i + 1) * partitions;
    if (i == amount - 1) end = n;

    hpx::future<double> f = hpx::async([begin, end, x, &parts]()
        -> double {
      std::for_each(parts.begin() + begin, parts.begin() + end, [
          x](double& e) {
        e = std::pow(-1.0, e + 1) * std::pow(x, e) / (e);
      });

      return hpx::reduce(parts.begin() + begin, parts.begin() +
          end, 0.);
    });

    futures.push_back(std::move(f));
  }

  double result = 0;

  auto futures2 = co_await hpx::when_all(futures);

  for (size_t i = 0; i < futures2.size(); i++) result += co_await
       std::move(futures2[i]);

  co_return result;
}

int main(){
  double x = 0.3;
  int n = 100;
  hpx::future<double> res = run(n,4,x);

```

```
48    std::cout<< "Difference of Taylor and C++ result "
49      << res.get() - std::log1p(x) << " after " << n
50      << " iterations." << std::endl;
51
52    return EXIT_SUCCESS;
53  }
```

11.1 Implementation of the Fractal Sets

Listing 11.3 on page 114 shows the implementation of the fractal set using a combination of asynchronous programming and coroutines. The implementation is based on Listing 9.8 on page 96 in Chap. 9. For coroutines we need a function that returns an `hpx::future`, see Listing 11.3 on page 114. In this function, we use the same `for` loop to launch the asynchronous tasks as in Listing 9.8 on page 96 in Chap. 9. In Line 28 of Listing 11.3 on page 114 we used `hpx::when_all(futures).then[&](auto&& f){...}).get();`, which can now be written more elegantly by using `co_await`. To exit the coroutine, we use `co_return;` in Line 30. Figure 11.1 on page 113 shows the performance measurements for the implementation using coroutines. The performance of the implementation with coroutines is very close to the implementation using `hpx::async`. This is due to the fact that, in HPX, both calls to `.get()` and `co_await` suspend the calling thread.

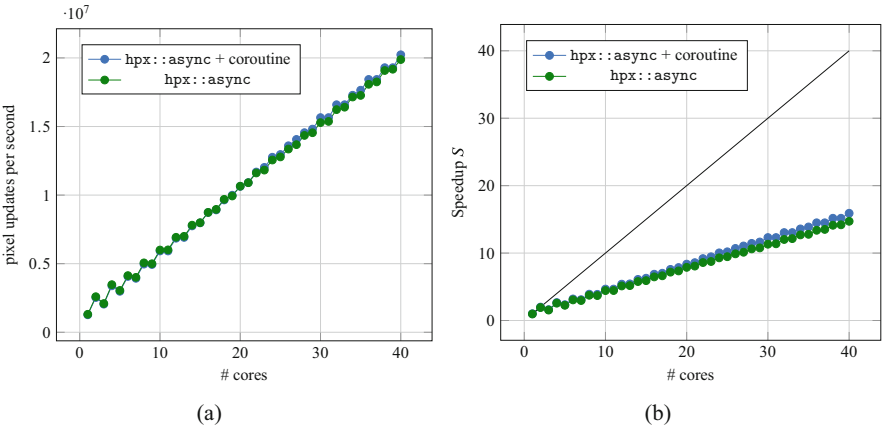


Fig. 11.1 Performance study for the fractal set using HPX's parallel algorithms. Figure (**a**) shows the pixel updates per second for an increasing number of cores. Figure (**b**) shows the speedup S with respect to the execution time on a single core. For each data point, the code was executed ten times and the median was plotted

Listing 11.3 Example for the computation of the Taylor series for the natural logarithm using HPX's futures and coroutines

```
#include <vector>
#include <pbm.hpp>
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include <kernel.hpp>

---

void launch(size_t start, size_t end, PBM *pbm) {

  for (size_t i = start; i < end; i ++)
    pbm->row(i) = compute_row(i);
}

hpx::future<void> run(size_t partitions, PBM *pbm) {
  std::vector<hpx::future<void>> futures;

  size_t size = size_y / partitions;

  for (size_t i = 0; i < partitions; i++) {
    size_t start = i * size;
    size_t end = (i + 1) * size;
    if (i == partitions - 1) end = size_y;
    auto f = hpx::async(launch, start, end, pbm);

    futures.push_back(std::move(f));
  }

  auto futures2 = co_await hpx::when_all(std::move(futures));

  co_return;
}

int main(){

  PBM pbm(size_y, size_x);

  auto start_time = std::chrono::high_resolution_clock::now();

  run(4,&pbm).get();

  auto stop_time = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::
      nanoseconds>(
    stop_time - start_time);
  std::cout << "Total time(s): " << duration.count()*1e-9 << std
      ::endl;

  // Save the image
  pbm.save("image.pbm");
}
```

Important

The most important lesson learned here is that the C++ 20 standard introduced coroutines, i.e. the ability to suspend and resume threads, to the language. For parallelism using `co_await` and `co_return`, the implementation using coroutines is not very different than the one using `.get` in Chap. 9. In principle, `co_await` suspends a thread and `.get()` blocks. This means that threads using coroutines should be better able to keep all cores of a machine busy. In HPX, however, both methods result in suspended threads, so the distinction is less important.

Chapter 12
Benchmarking the Fractal Set Codes

After introducing all four methods for parallelism in the C++ standard, namely low-level threads (Chap. 8), asynchronous programming (Chap. 9), parallel algorithms (Chap. 10), and coroutines (Chap. 11), see Fig. 7.1 on page 62, we compare their performance. First, we use one node with 2 × Intel® Xeon® Gold 6148 CPU @ 2.40 GHz with 20 cores per socket. Figure 12.1 on page 117 shows the pixel updates per second for all four methods. The blue line shows the implementation using `std::thread` and the green shows the implementation using `std::future`. Note that these measurements are very close and the lines overlay each other.

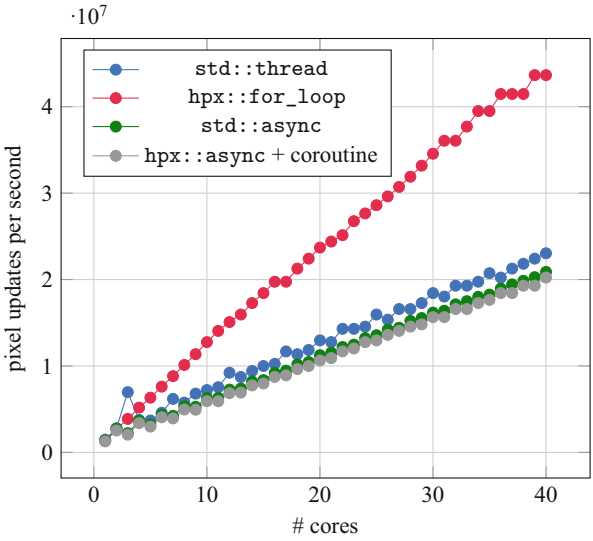


Fig. 12.1 Comparison of the pixel updates per second for the four different parallel implementations: `std::thread`, `std::future`, `coroutine`, and `hpx::for_loop` as presented in this chapter

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_12

For smaller numbers of cores (up to ten cores) the performance is comparable to the other higher-level abstraction methods. After that, the implementation using low-level threads or asynchronous programming is slower compared to the other methods.

The red line shows the implementation using HPX's parallel algorithms `hpx::for_loop`. Here, the least changes in the code were necessary to make the code run in parallel. This method easily has the best performance of the four. The gray line shows the implementation using coroutines with HPX's future. This method had the worst performance.

To investigate the portability of the parallel implementations, we executed the same code on several architectures: Intel® Xeon® Gold 6148 CPU @ 2.40 GHz, AMD® EPYC™ 7H12 CPU @ 3.2 GHz, and ARM® A64FX™ CPU @ 2.2 GHz. Figure 12.2 on page 119 shows the pixel updates per second for the four different implementations. For the Intel and AMD CPUs, we see that the performance is comparable for all of the implementations. However, we see that the AMD® EPYC™ CPU is slightly slower than on the Intel® Xeon® CPU. For the ARM® A64FX™ CPU, we see that the performance is much slower than on the other two architectures. This aligns with our common experience that lower performance is observed on ARM®. The exact causes are still under investigation. The compiler and dependencies used for these benchmarks are listed in Appendix C.

To summarize, we investigated four different implementations for parallelism provided directly by the C++ standard. No language extension like the commonly used OpenMP was necessary, and only pure C++ code was written. While HPX is an external library and not part of the SL, it is conformant with the C++ 17 and C++ 20 standard. The major benefits from using HPX is that the light-weight threads allow the code to run faster compared to the C++ standard library implementation and the library extensions it provides. We emphasize that all four codes were implemented for parallel execution for educational purposes. We did not optimize the codes, which is a separate topic not covered in this book.

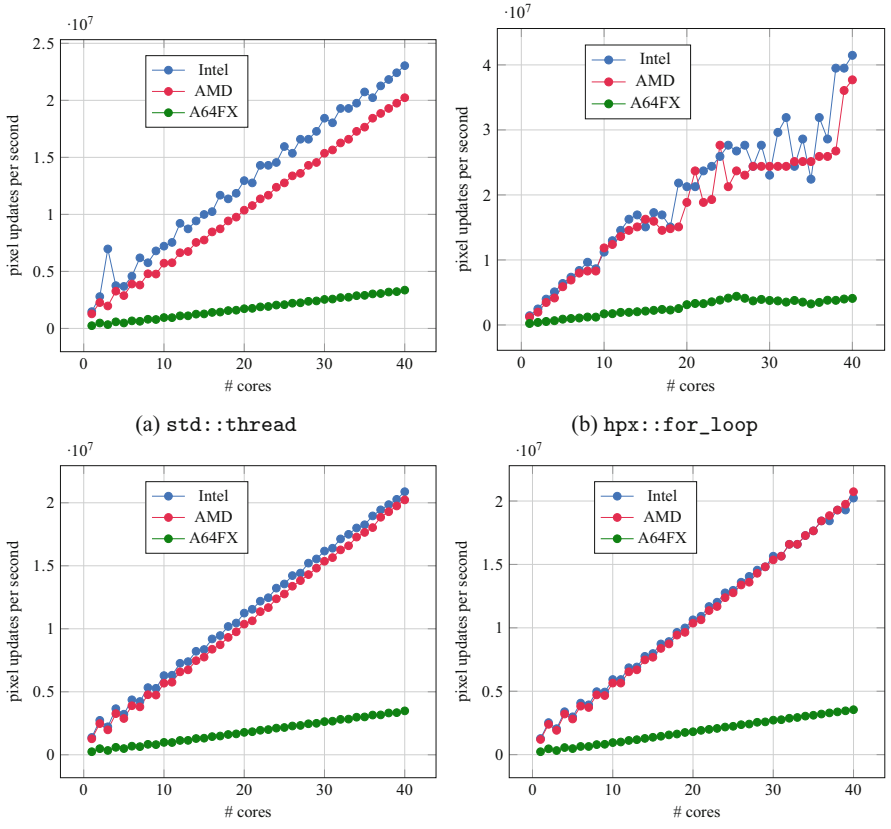


Fig. 12.2 Portability study of the parallel implementations on different architectures: Intel® Xeon®, AMD® EPYC™, and ARM® A64FX™. Comparison of the pixel updates per second for the four different parallel implementations: `std::thread`, `std::future`, `coroutine`, and `hpx::for_loop` as presented in this chapter

Part V
Distributed Programming

In this section,we examine aspects of distributed programming. First, we provide an overview of distributed programming in C++ and asynchronous many-task systems in Sect. 13.1. We briefly examine data distribution and load balancing in Sect. 13.2, and distributed input and output in Sect. 13.3. Second, we cover the Message Passing Interface (MPI) and implement the fractal set using MPI and *MPI+OpenMP* in Sect. 13.5. Third, we consider the distributed programming features of HPX in Chap. 14 and implement the fractal set code in Chap. 15.

Chapter 13
Distributed Computing and Programming

Previously, we considered shared memory parallelism using a single computational node. In this chapter, we study distributed programming. We use the definition for distributed computing as given in [99]: "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." For more details about distributed systems, we refer to [99]. Figure 13.1 on page 124 sketches the components of a distributed system, two computational nodes which are wired to a switch or router. One networking technology used in HPC is Ethernet which is standardized as IEEE 802.3. The data transfer rate for Ethernet (IEEE 802.3) is 10 Mb/s and for fast Ethernet (IEEE 802.3) is 100 Mb/s. To increase the data transfer rate and reduce the latency, other options are available for use in supercomputers or data centers. First, Mellanox® InfiniBand™ allows for a data transfer rate of 200 Gbit/s and a latency of $\approx$ 0.6 ns on ORNL's Summit. Second, HPE® Slingshot® 10 allows for a data transfer rate of 400 Gbit/s on NERSC's Perlmutter. Second, Fujitsu® Torus fusion (Tofu D) on Riken's Fugaku allows for a date transfer rate of 108.9 Gbit/s and a latency of $\leq$ 0.54 ns. For more details about networking technologies used in HPC, we refer to [100]. The following conceptual programming models are used for communication between nodes in HPC: Message Passing (e.g. as described in the MPI standard [101]), and Global Address Space, or GAS models. The GAS models discussed in this book are the following: Partitioned Global Address Space, or PGAS (as implemented in Gasnet [102]); and Active Global Address Space or AGAS [39] (which is what HPX uses. It is similar to PGAS but allows for moving objects).

We note here that the from the programmer's point of view, distributed systems are indistinguishable from processes running within a single node. Since processes share no memory with each other by default (although there are various hooks in the operating system to permit the sharing of memory), they generally must communicate through the same kinds of remote operations, e.g. message passing. Using multiple processes within the same node as part of a computation can be useful for testing, for dealing with NUMA domains, and other performance issues.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_13

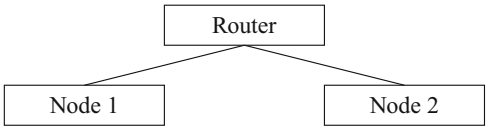


Fig. 13.1 Overview of the components of a distributed system. We have two computational nodes which are connected to a router or switch and they send messages to each other over the network. Common networking technologies used in HPC are the following: Mellanox® InfiniBand™, HPE® Slingshot® or Fujitsu® Torus fusion (Tofu D) Common programming models in HPC are the following: the Message Passing Interface (MPI), Partitioned Global Address Space (PGAS), or Active Global Address Space (AGAS)

The following additional tasks need to be considered when writing distributed programs: the exchange of messages, the distribution of data, and serialization of data. In Sect. 13.1, we provide a overview of distributed programming in C++ and introduce various asynchronous many task systems (AMTs). In Chap. 14 the C++ standard library for parallelism and concurrency (HPX) is used to implement the distributed computation of the fractal set. In Sect. 13.2, we discuss the principle of domain decomposition and introduce libraries that help with this task. In Sect. 13.5.1, we study the Message Passing Interface (MPI).

13.1 Overview of Distributed Programming in C++ and Asynchronous Many Task Systems

The Message Passing Interface (MPI) is the most widely used tool for distributed computing in HPC. It is a well-recognized standard API for passing messages between processes on the same computer or over the network. The MPI standard effort started in 1991 and in 1992 the "Workshop on Standards for Message Passing in a Distributed Memory Environment" [103] discussed what was needed. The first draft proposal of the MPI standard was written by Jack Dongarra, Tony Hey, and David W. Walker after the workshop. After some iterations, the draft standard was presented at the Super Computing conference in 1993. Finally, the first MPI standard was released in June 1994 [104]. The MPI standard continues to be developed and maintained by the MPI Forum, the standards organization for MPI. As of this writing, the latest MPI Standard 4.0 [101] was released in June 2021 and the MPI Standard 4.1 [105] was released in August 2023. Since the MPI standard is a document specifying the API, many implementations are available. OpenMPI and MPICH are the most widely known. Many vendors, e.g. Cray®, Intel®, Cray®, Fujitsu®, and IBM®, provide implementations as well. One challenge with C++ programming is that MPI uses a C-like interface and no standard C++ interface is available (although there is Boost MPI). Another important term with respect to distributed programming is *MPI+X* where MPI us used to send messages over the network and *X* is used for the shared memory parallelism on the node. The most

common value of X is probably $OpenMP$. However, X is not restricted to OpenMP and other tools, e.g. CUDA™ HIP™, or SYCL™ for GPUs, can be used as well.

Since roughly 2004 [106] with the end of increasing clock speeds, computational power has come from multiple cores, parallelism rather than raw speed. Some examples of chips with high core counts include Intel®'s Knight's Landing with up to 74 cores. On NERSC's Perlmutter and ORNL's Frontier the GPU nodes have AMD® CPUs with 64 cores. The Perlmutter CPU-only nodes have two AMD® CPUs with 128 cores in total. The Wafer Scale Engine 2 from TSMC has an astonishing 850,000 AI optimized cores on a single chip. For comparison, ORNL's Frontier, the world's fastest supercomputer at the time of this writing has 606,208 CPU cores counting all its nodes—but not counting its GPUs.

In addition, to the CPU cores, Frontier has four AMD® GPUs per node, resulting in 37,888 GPUs with 8,335,360 cores. Because of their size, power consumption is an important factor to consider in designing HPC machines [107]. The Green 500 list[1] ranks systems by efficiency, floating point operations per Watt. As of this writing, the machine Henri is currently the most efficient supercomputer on that list. Frontier, however, is not far behind.

To use modern HPC machines, we have to orchestrate a massive amount of parallelism, and we have to do it as efficiently as possible. The majority of the HPC community is pursuing an *MPI+X* strategy to tackle these challenges [108]. But as the hardware continues to evolve, getting the full performance of the machine continues to become a more complex and difficult task. Many programmers have difficulty making use of Phis or GPUs. Most of the early GPU-based clusters were highly under-utilized as many programs could not (and still can't) make use of the GPUs.

Regardless, it is a safe bet that parallelism in future systems will be increased by an order of magnitude. We will briefly introduce alternatives in the remainder of this section.

Another distributed programming model is the partitioned global address space (PGAS) [38] model. In PGAS, the programmer sets and gets values from memory on remote nodes, instead of message passing using *get* and *put*. This is a form of Remote Memory Addressing (RMA).

This is intended to translate into Remote Direct Memory Access (RDMA) communication, a protocol supported by modern networks. In RDMA, the network card takes control of the bus and reads and writes memory directly instead of mediating access through the CPU. Examples for PGAS systems are: Fortran (since the 2008 standards [109]), OpenSHMEM [110], UPC [111], CoarrayC++ [112], DASH [113], and GlobalArray [114]. One extension of PGAS is the Asynchronous Partitioned Global Address Space (APGAS) [115] model. This allows synchronous and asynchronous execution of remote tasks as well as data access. Some examples for APGAS systems are Chapel [24], UPC++ [26], Legion [116], and X10 [25]. For more details on asynchronous programming, we refer to Chap. 9.

[1] https://www.top500.org/lists/green500/.

To access the RDMA layer (if it exists), GAS languages need library support. Since MPI 3.0, there has been support for one-sided communication (RMA) in MPI [117]. The **G**lobal-**A**ddress **S**pace **Net**working (GASNet) [102] was designed specifically to provide remote memory access functionality for GAS languages. For example, Chapel, Legion, Coarray FORTRAN (CAF), and UPC\UPC++ can use GASNet. CAF is specified in ISO/IEC 1539-1:2010[2] and there are some implementations using MPI available.

Another distributed programming model within the GAS family is Active Global Address Space (AGAS) [21]. AGAS differs from PGAS in that objects in the global address space are dynamically relocatable. The C++ standard library for concurrency and parallelism (HPX) uses the AGAS programming model for distributed communication. HPX provides the following options for the base-level communication layer (not visible to programmers): (1) synchronous messages using MPI, (2) asynchronous messaging using libfabric [118], and (3) LCI—fast and generic concurrent communication engine [43]. In addition, HPX allows messaging using its own library based on the transmission control protocol (tcp) layer, but this is not recommended for high performance applications. For more details about HPX, we refer to Sect. 6.

Above, we have said that CAF and AGAS can be implemented on top of MPI. This may seem contradictory, since we have stated that MP and GAS are distinct programming models. The distinction is between what is visible to the programmer and the details of the underlying implementation. Because MPI is such a widely used standard and because it is so heavily developed and optimized, it can be thought of as an "assembly language" for distributed programming. While it may not always be the most efficient way to make a GAS programming environment work, MPI is hard to beat in terms of its wide deployment and reliable implementations.

Another model in this space is the global object space (GOS) runtime system, which is used by Charm++ [28]. As in AGAS, GOS has migratable, globally addressed objects.

To summarize, there are many asynchronous many task run time systems available, some solely for shared memory parallelism (which we will not discuss since here we are interested in distributed computing). For a detailed comparison of various AMTs, we refer to [119].

However, we will emphasize the major differences between HPX and other AMTs:

- HPX is influenced by the C++ standard and influences the C++ standard, so its API will seamlessly integrate with other C++ codes and libraries.
- HPX uses AGAS, a global address space like PGAS runtime systems but with migratable objects. This provides greater independence of data and work and benefits resiliency and migration.
- HPX provides a unified approach for multi-threaded, SIMD, heterogeneous, and distributed parallelism.

[2] https://www.iso.org/standard/50459.html.

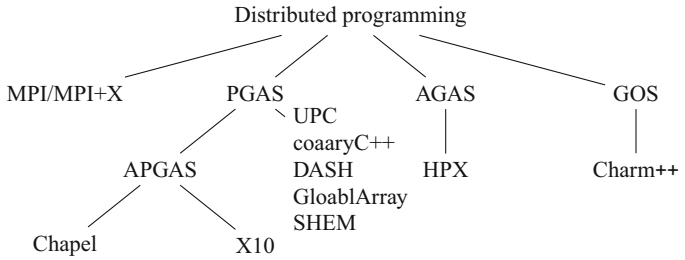


Fig. 13.2 Overview of distributed programming approaches in C++. From left to right: Message passing using the message passing interface (MPI) and Remote Memory Access (RMA) using partitioned global address space (PGAS). For asynchronous programming, the asynchronous partitioned global address space (APGAS) of PGAS is available. Other asynchronous options are active global address space (AGAS) and global object space (GOS) runtime systems

Figure 13.2 on page 127 provides an overview of all distributed programming models and runtime systems. Note there is no functionality in the C++ standard for distributed programming. Unfortunately, there is also no ongoing discussion on this topic in the C++ committee, so no such feature is likely to be added in the near future. One of the few libraries providing distributed capabilities that conform with the C++ standard is HPX.

13.2 Data Distribution

For distributed computing, we need to decompose the domain (the domain is usually a 2-D or 3-D array of floats or doubles) to the computational nodes. Figure 13.3 on page 128 shows the computational domain Ω on the left. To distribute a computation, the domain needs to be divided into sub domains and each of the sub domains is assigned to a single node. In this figure, the domain is decomposed onto four computational nodes. For simplicity, we split the domain at the center into four sub domains Ω_1 to Ω_4.

In this example, the work is not equally distributed to the four nodes, since the sub domains Ω_1 and Ω_3 are larger. In this case the computational nodes with the smaller sub domains are likely to finish their work earlier and have to wait. Note that this issue arises due to an irregular domain. To distribute the work more equally, libraries—like Metis [120] and its MPI-based parallel version ParMetis [121]; or Trilinos's Zoltan [122, 123]—provide static and dynamic load balancing.

Two common algorithmic options for load balancing are geometric portioning, e.g. Recursive Coordinate Bisection (RCB) [124], and Space-Filling Curve Partitioning (SFCP) [125–127]. For more details about SFCP, we refer to [128]. The left side of Fig. 13.4 on page 128 shows the domain Ω and the Hilbert curve, providing one example of a space-filling curve. On the right is the domain discretized with a mesh generated from Gmsh [129] and decomposed using Metis. In that case,

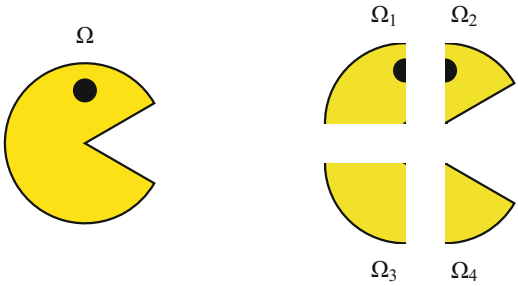


Fig. 13.3 On the **left** the computational domain Ω which is too large, e.g. due to memory consumption, to be simulated on a single computer. On the **right**: To simulate on four computational nodes, the domain Ω is decomposed into four sub domains Ω_1 to Ω_4. Each of these domains is assigned to one of the computational nodes. However, the work for each of the computational nodes is not equal, since sub domains Ω_1 and Ω_3 are larger and so these nodes have more work to do. This is called a domain decomposition imbalance

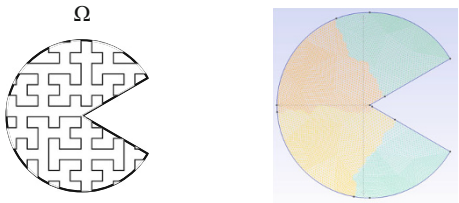


Fig. 13.4 On the **left** the computational domain Ω is filled with a Hilbert curve and provides one example of a space-filling curve. On the **right**: The domain is discretized with a mesh from Gmsh and decomposed with Metis. The colors indicate the subdomain assigned to each computational node. Here, the area or work load is balanced

the work is fairly well-balanced between the four computational nodes. In some simulations, e.g. scientific simulations, the mesh gets more highly refined near features of interest. This is called adaptive mesh refinement (AMR). Adaptive mesh refinement was proposed by Berger et al. [130, 131] for astrophysics. However, nowadays it is widely adopted in other fields. We note that ParMetis provides functionality for Adaptive Mesh Refinement (AMR) and large scale simulations.

Because the domain changes over time, dynamic load balancing is sometimes needed to redistribute the domain as the simulation continues. Load balancing for non-local diffusion using HPX's performance counters has been studied in [132]. For the distributed examples using MPI and HPX in this book, we implement the domain decomposition ourselves, since the image of the fractal set is a regular domain and can be easily equally distributed.

13.3 Distributed Input and Output

In the previous section, we covered how to decompose the domain into sub domains and assign them to the computational nodes. However, we want to avoid that the any one node (let's call it a *I/O node* because it reads and/or writes data on behalf of the others) reads the domain from a single file, which can be very large, and sends the data over the network to all other nodes. This would introduce a large *O*verhead before the simulation can start. The same holds for saving the simulation results. To avoid the communication of the domain, there are file formats for storing large distributed data sets. Here, the I/O node writes the head file, and all other nodes write their sub domain to dependent files. At the beginning of the simulation, the I/O node reads the head file and only sends the meta information for the other nodes, and other nodes read their corresponding files. One commonly used file format is Hierarchical Data Format (HDF) [133] developed by the non-profit organization HDF Group. Another format is Network Common Data Form (NetCDF) [134] and The Visualization Toolkit (VTK) [135]. Yet another is ADIOS2 [136]. All of these file formats provide libraries with a C or C++ API. Most of the libraries have MPI extensions for parallel or distributed computing, e.g. Parallel-NetCDF (PnetCDF) [137]. For adaptive meshes, we use SILO[3] with the HDF5 file format in Octo-Tiger. MPI provides *MPI IO* [138] for parallel and distributed file processing. However, *MPI IO* is a rather low-level API. We do not cover these file format and libraries in the book, but mention them since they are needed to write scalable and efficient distributed applications.

13.4 Serialization

Another important topic with respect to distributed computing is serialization of data structures. Within a single computational node, data structures are allocated in local memory. It is usually straightforward to send simple fixed-size arrays of basic types (e.g. double, float, int, etc.), across the network as raw bytes. Even a struct that contains no pointers might be sent that way, although there are possible dangers. However, variable-sized containers (such as `std::vector`), classes, and structs, usually cannot be safely sent without special effort. For a `std::vector` of doubles, for example, that might first mean sending the length, then the raw array. For a class containing an `int` and a `const char *`, that might mean sending first the `int`, then the length of the `const char *`, then the bytes in the `const char *`. In short, before sending the data, the programmer needs to describe the layout of the data structure to the distributed programming library they are using. This process of turning a struct (of possibly non-contiguous memory) into

[3] https:silo.readthedocs.ioenlatest.

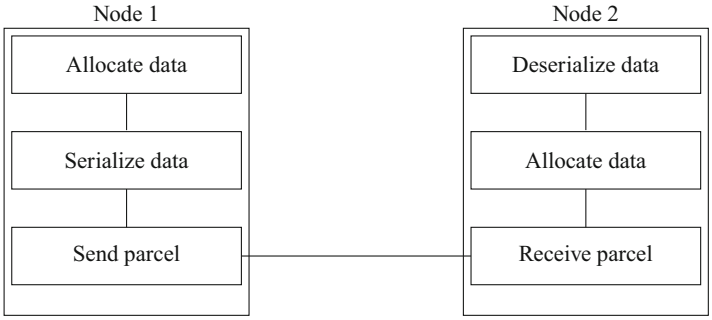


Fig. 13.5 Sketch of the communication between the nodes (localities). The data structure is serialized into a one-dimensional stream which can be wrapped in a parcel and sent over the network

an array of bytes to transmit across the network is called serialization. Figure 13.5 on page 130 illustrates the communication between two computational nodes using serialization.

HPX already knows how to serialize most common data structures such as `std::vector` and `std::map`. As long as you are not trying to send custom data structures across the network, you don't need to think about it.

Since it is C-based, MPI does not work with objects in the C++ sense. Therefore, MPI does not have the same concept of serialization. Instead, programmers can send sequences of messages. The `MPI_Type_create_struct` allows for the sending of user-defined types. HPX works similarly to Boost's serialize package, requiring your custom class to have a `serialize` method that conforms to certain rules.

For more details about serialization, we refer to the C++ FAQ [139].

Common serialization libraries are Boost.Serialization,[4] S11,[5] or Cereal.[6] In addition, *gSOAP* [140, 141] is a compiler-based approach to auto generate Extensible Markup Language (XML) serialization for some annotated C++ data types. However, this approach is mostly used for web services. Fun fact, in some programming languages, e.g. Ruby or OCaml [142], serialization is called *marshalling* and deserialization is called *unmarshalling*.

Important

The most important takeaways of this section are the following:

- Understanding that different approaches for distributed programming exist, namely MPI\MPI+X, partitioned global address space (PGAS), active global address space (AGAS), and global object space (GOS). However, in this book,

[4] https://www.boost.org/doc/libs/1_83_0/libs/serialization/doc/index.html.

[5] http://s11n.net/.

[6] https://uscilab.github.io/cereal/.

we go more deeply into AGAS, which is used by HPX, than we do into MPI, which we cover briefly.
- One important aspect of distributed programming is data distribution, e.g. domain decomposition and load balancing; and distributed input and output. We have only scratched the surface of these topics, but these are necessary for efficient distributed applications.
- To send data over the network, independent of the library used for distributed programming, the data has to be serialized.

13.5 Message Passing

In this section, we will consider an implementation of the fractal set using the Message Passing Interface (MPI).

In MPI, we have two common communication patterns: *collective basics* and *point-to-point basics*. For the collective basics one example is the MPI function `MPI_Bcast`.[7] Here, one node passes the message to all other nodes in the group. Examples of the point-to-point basics are the MPI functions `MPI_Send`[8] and `MPI_Recv`[9]. Here, a node passes a message using `MPI_Send` to one specific node using the rank of the node in the logical group of MPI processes `MPI_Comm`. The default communicator is `MPI_COMM_WORLD`. To receive the message, the recipient process must call `MPI_Recv`. Note that these are the standard-mode blocking functions and `MPI_Ibcast`,[10] `MPI_Isend`,[11] and `MPI_Irecv`[12] are the standard-mode non-blocking counterparts. For more details about the MPI functions, we refer to the MPI standard [101]. We will use the standard-mode non-blocking MPI functions to implement the distributed fractal set. We will use the point-to-point communication in Listing 13.2 on page 136 to send the computed pixels from each subordinate note to the I/O node and store them in the PBM format on the hard disk.

Let us consider the concept of distributing the work onto several computational nodes. Figure 4.1 on page 36 shows the image of the Julia set. In Fig. 13.6 on page 132 we divided it for computation on three nodes. We use the same principle as for the shared memory parallelism shown in Fig. 8.1 on page 80, however, each partition is computed on a different computational node. One simple approach

[7] https://www.open-mpi.org/doc/v4.1/man3/MPI_Bcast.3.php.
[8] https://www.open-mpi.org/doc/v4.1/man3/MPI_Send.3.php.
[9] https://www.open-mpi.org/doc/v4.1/man3/MPI_Recv.3.php.
[10] https://www.open-mpi.org/doc/v3.0/man3/MPI_Ibcast.3.php.
[11] https://www.open-mpi.org/doc/v3.0/man3/MPI_Isend.3.php.
[12] https://www.open-mpi.org/doc/v3.0/man3/MPI_Irecv.3.php.

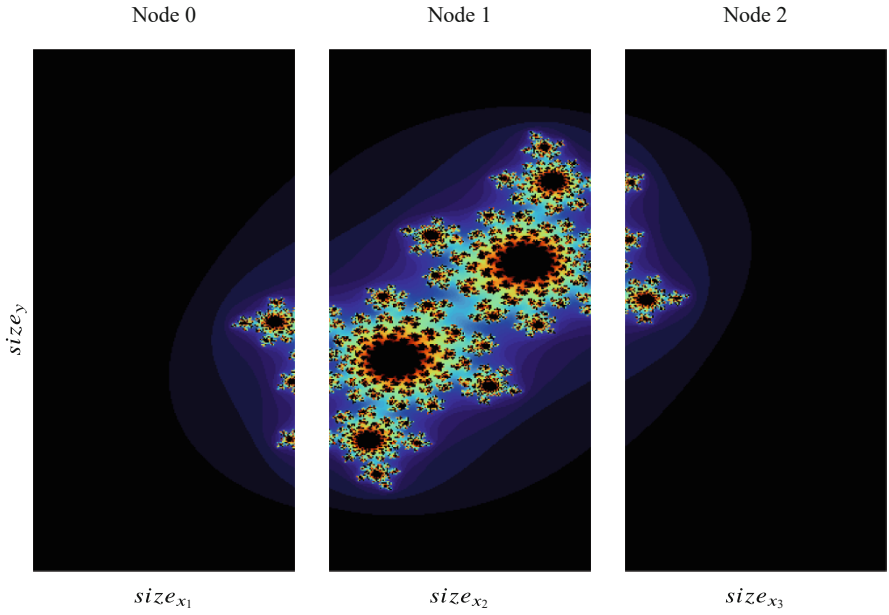


Fig. 13.6 Composition of the pixels of the Julia set on three nodes N_0, N_1, and N_2. Each of the nodes computes the full height of the image $size_y$, however, the width of the image is split into three nearly equal parts $size_{x_1}$, $size_{x_2}$, and $size_{x_3}$

for the domain decomposition is to split the number of pixels in the x-direction $size_x$ and keep the number of pixels in the y-direction $size_y$ undivided. Note that more sophisticated approaches are possible, see Sect. 13.2. In this example, each of the nodes computes one third of the image. Once the computation is finished, the two delegate nodes send a message containing their data (pixels with color values) to their I/O node using `MPI_Isend`, and the I/O node will receive the messages using `MPI_Irecv`. Note that we need to use a flat data structure here, like a `std::vector<int>`, and cannot use the two dimensional data structure, e.g. `std::vector<std::vector<int>>`, as it exists in the PBM class. So, instead, we send the raw data of each line. After the I/O node collects all the pixels of the sub domains, the I/O node can put them into the PBM format and store the image. Note that sending all the pixels to the I/O node becomes expensive for large image sizes or large domains in general. In that case, the sequential effort of collecting and writing the data would dominate the overall computation. Remember Amdahl's law in Sect. 7.3. One solution to this problem would be to use a distributed file format, see Sect. 13.3. Here, each node would write its assigned data to a file that describes the portion of the work it contains and how to join it with the files from the other nodes. The duties of the I/O node would then be reduced to providing text output on the console.

Here you might object that we have only deferred the problem of collecting and writing the data to a single file. We still have to do that computation, and it's still sequential. That is true. However, typically, on supercomputers, programs are run in batch jobs on shared resources where the user must specify the number of nodes to use for the duration of the computation. If we request 100 nodes to run a job and then spend ten percent of our time collecting and writing a file, we are forcing 99 nodes to be idle while we do it. Assembling these data files after the job finishes prevents this enforced idleness.

The PBM image format does not support writing distributed files. Second, the usage of any distributed I/O library would be beyond the scope of this book. For educational reasons, we write our own simple file format to store the pixels of each node and provide a program `combine_image`, see Listing 13.4 on page 139, to combine these files to one PBM image. In the following, we will implement the fractal set using MPI (Listing 13.1 on page 133) and *MPI + OpenMP* (Listing 13.6 on page 142).

13.5.1 Implementation of the Fractal Set Using MPI

Listing 13.1 Implementation of the fractal set using the Message Passing Interface

```
#include <mpi.h>
#include <chrono>

#include <config.hpp>
#include <pbm.hpp>
#include <kernel.hpp>
#include <util.hpp>
#include <io.hpp>

using std::chrono::high_resolution_clock;
int main() {
 // Initialize the MPI environment
 MPI_Init(NULL, NULL);
 // Get the number of processes or nodes
 int mpi_size;
 MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
 // Get the rank of the process
 int mpi_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
 // Start the timer
 high_resolution_clock::time_point start_time;
 PBM pbm;
 if (mpi_rank == 0){
   start_time  = high_resolution_clock::now();
 }
 pbm = PBM(size_y, size_x);

 std::size_t iter_start, iter_end;
 split_work(mpi_size, size_x, mpi_rank, iter_start, iter_end);
```

```
30
31  for (size_t iter = iter_start; iter < iter_end; iter++) {
32    pbm.row(iter) = compute_row(iter);
33    assert(pbm.row(iter).size() == size_y);
34    if(mpi_rank > 0) {
35       MPI_Request req;
36      MPI_Isend((void*)pbm.row(iter).data(),size_y,MPI_INT,0,iter,
          MPI_COMM_WORLD,&req);
37    }
38  }
39
40  // Save the image
41  if (mpi_rank == 0) {
42     save_image(pbm, mpi_size, size_x, size_y);
43  }
44  // Stop the timer
45  if (mpi_rank == 0){
46    auto stop_time = std::chrono::high_resolution_clock::now();
47    auto duration = std::chrono::duration_cast<std::chrono::
        nanoseconds>(
48        stop_time - start_time);
49    std::cout << "Time: " << duration.count()*1e-9 << std::endl;
50  }
51  // Finalize the MPI environment.
52  MPI_Finalize();
53 }
```

Now let us study an implementation of the fractal set using the message passing interface (MPI). Later, we will compare this implementation with the HPX implementation in Chap. 14. In Chap. 16, we make some remarks on the two different approaches.

Listing 13.1 on page 133 shows the implementation using MPI. In Line 13, the MPI environment is initialized using `MPI_Init`.[13] In Line 16 the total number of processes or nodes `int mpi_size` is requested from the MPI environment using `MPI_Comm_size`.[14] This gives us the number of partitions in which to divide the image. In Line 19, the rank `int mpi_rank` of the node is requested from the MPI environment using `MPI_Comm_rank`.[15] The rank is a number in the range 0–`mpi_size-1`, a unique value that identifies the process that calls this function. The argument `MPI_COMM_WORLD` is the group of all MPI processes. It is possible

[13] https://www.open-mpi.org/doc/v3.1/man3/MPI_Init.3.php.

[14] https://www.open-mpi.org/doc/v3.1/man3/MPI_Comm_size.3.php.

[15] https://www.open-mpi.org/doc/v4.0/man3/MPI_Comm_rank.3.php.

to make communicators that perform operations on a subset of the total nodes available, but we will not discuss that functionality.

A common convention is that rank zero is the I/O node, and that is what we do here. In Line 24 the start time is recorded. Note that this only happens on the I/O node (`mpi_rank == 0`), since we need to measure the total computation time only once. In Line 29, the range for each of the nodes is computed. Note that calculation of the range follows the same principle as for the example for asynchronous programming in Listing 9.8 on page 96 in Chap. 9. For simplicity, the function `split_work` is not shown here, but can be found in `util.hpp` in Listing B.4 on page 222 in Appendix B.

In Line 31 of Listing 13.1 on page 133, each node iterates over its assigned work. In Line 32, the compute kernel in the file `kernel.hpp` in Listing B.3 on page 220 in Appendix B is called. If the compute kernel is executed on the I/O node, the `row` can be stored into the `pbm` data structure directly, since this data structure is in the memory of the I/O node. For all other nodes, we use the function `MPI_Isend` to send the computed row of pixels to the I/O node, see Line 36 of Listing 13.1 on page 133. The first argument is the pointer to the row data of the vector `std::vector<int> row`, the second argument is the size of the vector `row`, the third argument the data type, the fourth argument the rank of the destination, the fifth argument the tag (a value which must match between sender and receiver), and the last argument for error handling. Recall that we use the standard-mode non-blocking function calls. In Line 41 the image is saved to the hard disk by the I/O node. We will discuss saving the image in more detail later. In Line 49 the timer is stopped and the duration is printed to the console using the standard output stream. To end the computation without errors, we call `MPI_Finalize`[16] in Line 52. For more implementation MPI details, we refer for example to [143].

Let us consider how to collect the distributed pixel data and save them to the image. The function `save_image` is shown in Listing 13.2 on page 136. In Line 7, we iterate over all workers, excluding the I/O node. The I/O node does not need to pass messages to itself. In Line 9, the messages for all rows for each node are collected. In Line 10 the `std::vector<int> v` is used to store the received row of pixel data is allocated and resized in Line 11. In Line 13, we use `MPI_Recv` to collect the messages passed in Line 36 of Listing 13.1 on page 133 to the I/O node. Note that all messages are sent using a standard-mode blocking function. The first argument is the pointer to the data from the `std::vector<int> v` where the received data is stored on the I/O node, the second argument is the size of the data, the third argument is the data type, the fourth argument the rank of the sender, the fifth argument the tag (an integer which much match between sender and receiver),

[16] https://www.open-mpi.org/doc/v4.0/man3/MPI_Finalize.3.php.

Listing 13.2 Utility functions to save the PBM image

```
#ifndef IO_HPP
#define IO_HPP

void save_image(PBM &pbm, size_t mpi_size, size_t size_x,
                size_t size_y) {
  std :: size_t iter_start , iter_end ;
  for(int worker=1;worker<mpi_size;worker++) {
    split_work(mpi_size, size_x, worker, iter_start, iter_end);
    for(size_t iter=iter_start;iter < iter_end; ++iter) {
      std::vector<int> v;
      v.resize(size_y);
      MPI_Status status;
      MPI_Recv((void*)v.data(),size_y,MPI_INT,worker,iter,
          MPI_COMM_WORLD,&status);
      assert(pbm.row(iter).size() == v.size());
      pbm.row(iter) = v;
    }
  }
  // Save the image
  pbm.save("image_mpi.pbm");
}
#endif
```

the sixth argument the MPI group (i.e. `MPI_COMM_WORLD`, and the last argument is for error handling (which we do not use and, therefore, pass a `nullptr`). Note that we cannot pass the `std::vector` data structure directly to `MPI_Recv` and need to pass the raw pointer. In Line 15 of Listing 13.2 on page 136 the received pixel values are stored to the `pbm` data structure. In Line 19 the image is stored to the hard disk.

Note that sending the data from the worker nodes in Line 36 of Listing 13.1 on page 133 to the I/O node and collecting the data in Line 13 of Listing 13.2 on page 136 might scale well for smaller image sizes. However, for larger images, the *O*verhead of sending the messages dominates the overall scaling. A common approach in HPC is to use a distributed file format, see Sect. 13.3. The PBM file format we use to store the image does not support this feature. In general, none of the common file formats do. We could use libraries, like the Visualization Tool KIT (VTK), to store the image in parallel and avoid sending the rows to the I/O node. However, we prefer to keep things simple for educational purposes. Listing 13.3 on page 137 shows the improved code based on Listing 13.1 on page 133. The header `#include <fstream>`[17] is included in Line 3 of Listing 13.3 on page 137 for using `std::ofstream`[18] to write a file with the partial data for each node, see Line 23. Thus, each node generates a file with its rank in its name, e.g. rank one will generate

[17] https://en.cppreference.com/w/cpp/header/fstream.

[18] https://en.cppreference.com/w/cpp/io/basic_ofstream.

Listing 13.3 Implementation of the fractal set using the Message Passing Interface with distributed IO

```
#include <mpi.h>
#include <chrono>
#include <fstream>

#include <config.hpp>
#include <kernel.hpp>
#include <util.hpp>

using std::chrono::high_resolution_clock;

int main() {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);
  // Get the number of processes or nodes
  int mpi_size;
  MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
  // Get the rank of the process
  int mpi_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
  // Start the timer
  high_resolution_clock::time_point start_time;
  std::ofstream outfile("data_"+
    std::to_string(mpi_rank)+".part");
  if (mpi_rank == 0)
    start_time  = high_resolution_clock::now();

  std::size_t iter_start, iter_end;
  split_work(mpi_size, size_x, mpi_rank, iter_start, iter_end);

  for (size_t iter = iter_start; iter < iter_end; iter++) {
    std::vector<int> row = compute_row(iter);

    assert(row.size() == size_y);
    outfile << iter << " ";
    for(auto color : row)
      outfile << color << " ";
      outfile << "\n";
  }

  // Stop the timer
  if (mpi_rank == 0){
    auto stop_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::
    nanoseconds>(stop_time - start_time);
    std::cout << "Time: " << duration.count()*1e-9 << std::endl;
  }
  outfile.close();
  // Finalize the MPI environment.
  MPI_Finalize();
}
```

the file `data_1.part`. The major change is in Line 35 of Listing 13.3 on page 137. Here, we write the index of the row to the file of each node using the operator <<. Note we separate the index and all pixel values by a space. Other formats like Comma-separated values (CSV) are common. However, the PBM format uses spaces also. Recall that in the previous example we called `MPI_Isend` and sent each computed row of pixel values to the I/O node. Here, we avoid sending the data over the network, and thereby reduce the *O*verhead. In Line 47, we close the file. Note that we need to call `close()`[19] to write the content of the file to the hard disk, otherwise the file will be empty.

Of course, we now need a code to combine the partial files written by each rank into one single PBM file. Listing 13.4 on page 139 shows the code that does this. In Line 3 we include the header `#include <pbm.hpp>`, and in Line 25 we instantiate the PBM object. Note we use the default image size from the header `#include<config.hpp>`. Alternatively the user can use `export SIZE_X` and `export SIZE_Y` to set the image size. We use `std::stoi`[20] to convert the string numeral to an `int` value. In Line 29 of Listing 13.4 on page 139, a `while` loop is used to process each of the files. The header `#include <fstream>`[21] is included in Line 1 to use `std::ifstream`[22] to read the file content in Line 31. The file is read in the next line. In Line 38 and Line 39 the index `int index` and the `std::vector<int> row` are defined. Note we want to read these values from the file and store them into these two variables. In Line 42 we read the file line by line using a `while` loop and `std::getline`[23] to store the line in `std::string s`. Note that once the end of the file is reached, the function will return `false` and the `while` loop terminates. In Line 44 the function `split` defined in Line 5 is called. In Line 7 we use `istringstream`[24] with the `string line`. In Line 13 a `while` loop is used to iterate over the items of the `stream` using `getline`. Note that the third argument `' '` indicates that we want to separate the content by spaces. The first entry is the index and all other elements are the pixel values. Note that one could check the length of pixel values or whether all values are numbers to guard against error. Some programming languages, like Python, allows for multiple `return` values. C and C++ allow for only one. To overcome this restriction, a `std::tuple`[25] can be used. The function `std::make_tuple`[26] can be used to instantiate a tuple. In Line 46 and Line 47, `std::get<int>`[27] and `std::get<std::vector<int>>`

[19] https://en.cppreference.com/w/cpp/io/basic_ofstream/close.
[20] https://en.cppreference.com/w/cpp/string/basic_string/stol.
[21] https://en.cppreference.com/w/cpp/header/fstream.
[22] https://en.cppreference.com/w/cpp/io/basic_ifstream.
[23] https://en.cppreference.com/w/cpp/string/basic_string/getline.
[24] https://en.cppreference.com/w/cpp/io/basic_istringstream.
[25] https://en.cppreference.com/w/cpp/utility/tuple.
[26] https://en.cppreference.com/w/cpp/utility/tuple/make_tuple.
[27] https://en.cppreference.com/w/cpp/utility/tuple/get.

are used to access each value of the `std::tuple` (Note that we could have used `std::get<0>` and `std::get<1>` just as easily). In Line 48 the `row` is assigned to the corresponding index in the PBM image. Finally, the image is saved to hard disk in Line 53.

Listing 13.4 Code to combine the distributed image data into a single PBM image

```
1  #include <fstream>
2  #include <iostream>
3  #include <pbm.hpp>
4
5  std::tuple<int, std::vector<int>> split(std::string line) {
6
7    std::istringstream stream(line);
8    std::string index;
9    std::vector<int> row;
10
11   std::string s;
12   bool first = true;
13   while (getline(stream, s, ' ')) {
14     if (first) {
15       index = s;
16       first = false;
17     } else
18       row.push_back(std::stoi(s));
19   }
20   return std::make_tuple(std::stoi(index), row);
21 }
22
23 int main(int argc, char **argv) {
24
25   PBM pbm = PBM(size_y, size_x);
26
27   size_t id = 0;
28
29   while (true) {
30
31     std::ifstream file;
32     file.open("data_" + std::to_string(id) + ".part");
33     if(!file.good()) {
34         std::cout << "Files read: " << id << std::endl;
35         break;
36     }
37
38     int index;
39     std::vector<int> row;
40
41     std::string s;
42     while (std::getline(file, s)) {
43
44       auto res = split(s);
45
46       index = std::get<0>(res);
```

```
47        row = std::get<1>(res);
48        pbm.row(index) = row;
49      }
50      id++;
51    }
52
53    pbm.save("image_combined.pbm");
54
55    return 0;
56  }
```

Figure 13.7 on page 140 shows the timing from 1 node to 16 nodes for writing the PBM image by the I/O node (Listing 13.1 on page 133), and the distributed I/O where each node writes a separate file (Listing 13.3 on page 137). We used `export SIZE_X=39860` and `export SIZE_Y=21600` resulting in 860,976,000 pixels. First, we observe that the distributed file I/O (green line) is only slightly faster with two nodes than the single file I/O code (blue line). However, for node counts of 4 or greater, the *O*verhead reduction for the distributed I/O makes a huge difference. Second, the *O*verhead for sending and receiving the data with MPI dominates the performance. Therefore, the blue line flattens out soon. In conclusion, distributed I/O is beneficial even for small, educational examples.

13.5.2 Implementation of the Fractal Set Using MPI+OpenMP

In the previous implementation of the fractal set, we focused on *intra-process* parallelism. Those processes may be on the same node or on different nodes. When it is used between nodes, that is *intra-node* parallelism. On a 64 core node, one could run 64 MPI processes to achieve parallelism, but it is more common to use

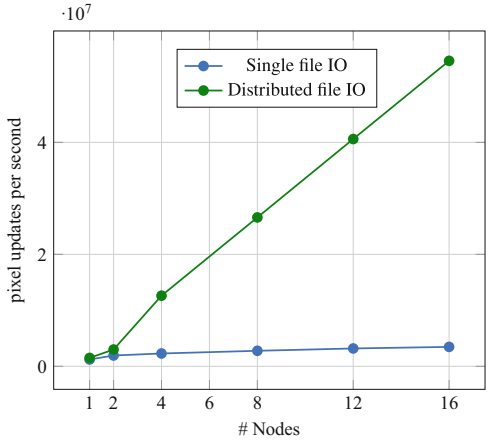


Fig. 13.7 Comparison of the pixel updates per second for single file I/O (green line) and distributed file I/O (blue line). Note that we added the time to combine the partial files to a single PBM file to the distributed I/O for a fair comparison

MPI+OpenMP. Open Multi-Processing (OpenMP) is a `#pragma`-based compiler extension for thread-based parallel programming. For more details about OpenMP, we refer to Chap. 7. If we were running on a cluster with 64 nodes, we could use one MPI process per node and 64 threads of OpenMP. Surprisingly, though, this tends not to give the best results. While the performance details of any code are complex, it is often useful to use at least one MPI process per NUMA domain in order to minimize cross-domain traffic. So if our 64 core node had two 32 core NUMA domains, it might make sense to use 2 MPI processes per node with 32 OpenMP threads per node, or 4 MPI processes with 16 OpenMP threads per node. What combination works best depends on the details of a given code and machine and will require experimentation.

Listing 13.5 on page 141 shows the compute kernel with the `#pragma omp parallel for` in Line 1. However, we avoid the complexity of having multiple OpenMP threads write to the file at the same time. Instead, we use a single thread for I/O. One way to accomplish this is to use `#pragma omp critical`. This pragma will cause the following block to execute as if it were protected by a `std::mutex`. In Line 6, the code is effectively "locked" by the beginning of the critical block, and in Line 13 the code is unlocked. Because OpenMP allows for various loop options including chunk sizes, the code could be optimized further. For more details about chunk sizes, we refer to Sect. 10.2. For more implementation details, we refer to [96].

However, using the locked scope containing the `#pragma omp critical` section introduces a barrier that will affect the scaling of the code. The code may improve as OpenMP threads are added, but the overhead due to starting and stopping the critical section and forcing some of the threads to wait while this happens will slow us down. Let us remove the critical section and write out the file after computing all the pixels. Listing 13.6 on page 142 shows the final implementation of the MPI+OpenMP fractal set. The code is based on the improved

Listing 13.5 Implementation of the fractal set using MPI + OpenMP: Using a critical section for the distributed I/O

```
1  #pragma omp parallel for
2  for (size_t iter = iter_start; iter < iter_end; iter++) {
3    std::vector<int> row = compute_row(iter);
4
5    assert(row.size() == size_y);
6    #pragma omp critical
7    {
8      outfile << iter << " ";
9      for(auto color : row) {
10       outfile << color << " ";
11     }
12     outfile << "\n";
13   }
14 }
```

MPI code in Listing 13.3 on page 137. Until Line 26 of Listing 13.6 on page 142 the code is identical to the MPI-only code. The first change in Line 28 is a `std ::vector<std::vector<int>>` to store the pixel values for each row. Note that `iter_end-iter_start` is the length of the part of the image assigned to the node. The next change is the `#pragma omp parallel for` in Line 30 above the `for` loop in Line 31. Now, the `for` loop runs in parallel if there is more than one OpenMP thread. Once the computation is done, the file is written sequentially in Line 35 to Line 42. Apart from one pragma, the code is the same as the MPI code. The key change from Listing 13.5 on page 141 to Listing 13.6 on page 142 was to decouple the computation and the writing to the file.

Listing 13.6 Improved implementation of fractal set using MPI+OpenMP without using a scope lock

```
1  #include <mpi.h>
2  #include <chrono>
3  #include <fstream>
4  #include <config.hpp>
5  #include <kernel.hpp>
6  #include <util.hpp>
7
8  using std::chrono::high_resolution_clock;
9
10 int main() {
11   // Initialize the MPI environment
12   MPI_Init(NULL, NULL);
13   // Get the number of processes or nodes
14   int mpi_size;
15   MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
16   // Get the rank of the process
17   int mpi_rank;
18   MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
19   // Start the timer
20   high_resolution_clock::time_point start_time;
21   std::ofstream outfile ("data_"+std::to_string(mpi_rank)+".part"
         );
22   if (mpi_rank == 0)
23     start_time  = high_resolution_clock::now();
24
25   std::size_t iter_start, iter_end;
26   split_work(mpi_size, size_x, mpi_rank, iter_start, iter_end);
27
28   std::vector<std::vector<int>> data(iter_end-iter_start);
29
30   #pragma omp parallel for
31   for (size_t iter = iter_start; iter < iter_end; iter++) {
32     data[iter-iter_start] = compute_row(iter);
33   }
34
```

```
35    size_t iter  = iter_start;
36    for( auto row : data){
37       outfile << iter << " ";
38       for(auto color : row)
39         outfile << color << " ";
40       outfile << "\n";
41       iter++;
42    }
43
44    // Stop the timer
45    if (mpi_rank == 0){
46      auto stop_time = std::chrono::high_resolution_clock::now();
47      auto duration = std::chrono::duration_cast<std::chrono::
            nanoseconds>(
48      stop_time - start_time);
49      std::cout << "Time: " << duration.count()*1e-9 << std::endl;
50    }
51    outfile.close();
52    // Finalize the MPI environment.
53    MPI_Finalize();
54  }
```

Finally, to run the fractal set code, we can use `mpirun`[28] or `mpiexec`.[29] Listing 13.7 on page 143 shows the instructions on how to run the fractal set code on two nodes for the intra-node parallelism and using 20 OpenMP cores for the intra-node parallelism. In Line 2 to Line 4 the simulation is configured by specifying the size of the image and the type of the fractal set. For more configuration parameters, we refer to Chap. 4. In Line 6 the number of cores OpenMP can utilize is given by setting the environment variable `OMP_NUM_THREADS`[30] to 20. To allow MPI to use the OpenMP threads, we need to add the option `--bind-to none`. Finally, in Line 8, we use `mpirun` to run the code on two nodes (`-N 2`).

Listing 13.7 Sample bash script to run the distributed MPI-based fractal set code

```
1  #!/bin/bash
2  export SIZE_X=10000000
3  export SIZE_Y=10000000
4  export TYPE=julia
5  # Let OpenMP use 20 cores
6  export OMP_NUM_THREADS=20
7  # Run on two nodes with one core per node
8  mpirun --bind-to none -N 2 ./mpi_fractal
```

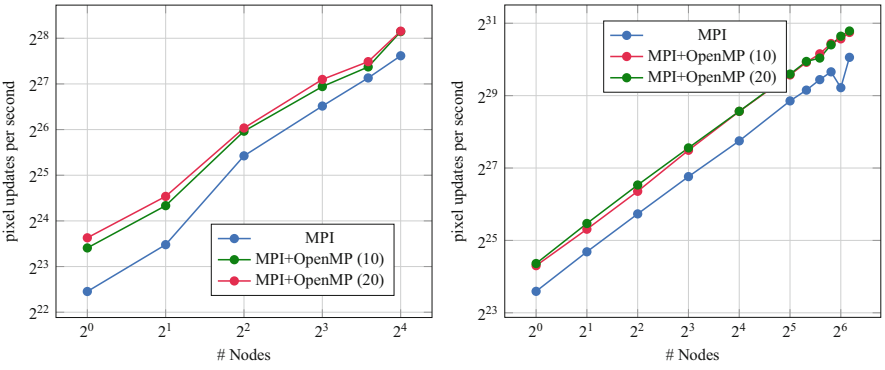
[28] https://www.open-mpi.org/doc/v3.0/man1/mpirun.1.php.

[29] https://www.open-mpi.org/doc/v3.0/man1/mpiexec.1.php.

[30] https://www.openmp.org/spec-html/5.0/openmpse50.html.

13.6 Benchmark of MPI and MPI+OpenMP

After implementing the fractal set in MPI and MPI+OpenMP, we compare the performance of both implementations. However, we have shown in Fig. 13.7 on page 140 that writing to a single file slows down the overall performance. Therefore, we use the code in Listing 13.1 on page 133 with the distributed I/O. First, we run the MPI code with one rank (one core) per node and do not exploit any inter-node parallelism. Second, we use MPI+OpenMP to account for inter-node parallelism using OpenMP. Here, we run with five cores (`export OPENMP_NUM_THREADS=5`) and ten cores (`export OPENMP_NUM_THREADS=10`). For the image size, we use `export SIZE_X=39860` and `export SIZE_Y=216000` resulting in 8,609,760,000 pixels. Figure 13.8 on page 144a shows the scaling on Intel® Gold 6148 Skylake and Fig. 13.8 on page 144b on the Arm® Arm® A64FX™, respectively. On the Intel nodes, the blue line is the MPI code with one MPI rank per node. The `for` loop to compute the pixels runs sequentially on each of the nodes. The code scales from 1 to 16 nodes. The green line is the MPI+OpenMP code using one MPI rank per node but the `for` loop is parallelized with `#pragma omp parallel for` using ten threads. Here, we observe some improvement by adding the intra-node parallelism with ten threads. However, for the red line the improvement by adding ten additional cores is less. One factor here is that the portion of the parallel code is small and the dominant part of the code, the writing of the file, is serial. This again relates to Amdahl's law in Sect. 7.3. Figure 13.8 on page 144b shows the pixel updates per second on the Arm® A64FX™ CPUs. Again the blue line is the MPI code with one MPI rank per node. Here, we see scaling from 1 to 72 nodes. We see one outlier at 64 nodes. The green line is the MPI+OpenMP code using one MPI rank per node and



(a) Intel® Xeon® Gold 6148 Skylake (b) Arm® A64FX™

Fig. 13.8 Performance study for the fractal set using the distributed implementation based on MPI, and MPI+OpenMP. Figure (**a**) shows the pixel updates per second for an increasing number of nodes on the Rostam cluster using Intel® Xeon® Gold 6148 Skylake CPUs. Figure (**b**) shows the pixel updates per second for an increasing number of nodes on the Ookami cluster using Arm® A64FX™ CPUs

ten OpenMP threads. The code shows more pixel updates per second than the MPI code. The red line is the MPI+OpenMP code with one MPI rank and 20 OpenMP threads. Up to 8 cores we see some marginal improvement. After that both codes are very close. Again, this could possibly be due to the small portion of the parallel code with respect to the serial writing of the file. It might be possible to improve both implementations further, but this would be out of the scope of this book.

Chapter 14
Distributed Programming Using HPX

In this chapter, we will examine the distributed programming features provided by HPX. For programming a single node, HPX supports using low-level threads in Chap. 8; asynchronous programming in Chap. 9, the parallel algorithms in Chap. 10; and coroutines in Chap. 11. Here, HPX conforms with the C++ standard.

However, when it comes to distributed computing, there is no specification in the current C++ standard. As of this writing, we are not aware of any discussion within the C++ standardization committee to add distributed programming. Recall that it took until C++ 17 for parallel algorithms to be included and until C++ 20 for coroutines to be included. Maybe with the C++ 26 standard, this issue will begin to be addressed. We will discuss these possibilities in Chap. 19.

HPX tries to always be ahead of the curve and provide the current C++ standard features and even some proposed features. In this chapter, we introduce components and actions, part of HPX's API for remote function calls. Components and actions extend the familiar concept of asynchronous programming in Chap. 9 to distributed programming.

We want to emphasize that HPX relies on other implementations for it's underlying communication (which is invisible to the programmer). HPX offers, instead, a new high-level abstraction based on futures. Right now, the underlying communication layer that HPX uses must be either the Message Passing Interface (MPI); libfabric [44]; LCI [43]; or the Transmission Control Protocol (TCP). For a performance comparison of these communication options, we refer to Chap. 18. We hope that these distributed programming concepts seed discussions within the C++ standardization community.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_14

14.1 Active Messaging

For remote function calls, HPX uses the concept of active messaging [41], first proposed by Von Eiken, et al. in 1992. Active Messages were developed to reduce communication cost and better overlap computation and communication. The authors claim an order of magnitude increase in performance over Bulk Sequential Parallelism (BSP), in which a computation alternates between phases of communication and computation. In the former, the processors are largely unused, and in the latter, the network is largely unused. By overlapping computation and communication, Active Messages keep both pieces of hardware utilized.

Figure 14.1 on page 148 shows the structure of an active message. The first part of the message is the *destination address* of the process where the function is executed. The second part is the method which is to be executed. The third part, *arguments*, are the function arguments. The last (optional) argument, *continuations*, points to work which will be executed after the function has finished.

HPX uses the term Parcel (**Par**allel **C**ontrol **El**ement) to refer to its implementation of Active Messages. Parcels are sent over the network using the Message Passing Interface (MPI) or libfabric [44] or LCI [43]. For more details, we refer to Sect. 18.1. To transport the parcel over the network to the *destination address*, the parcel is serialized into a stream of bytes. Note that we need to serialize all data structures within the parcel. For more details on serialization, we refer to Sect. 13.4. In an additional effort to reduce overheads and optimize the balance between computation and communication, smaller parcels are sometimes joined together and sent as one. This is called *Parcel Coalescing*. The effect of parcel coalescing on performance has been studied in [52]. However, we will not discuss it further in this book.

Because HPX is a C++ library, it is natural to extend the Active Message concept so that the destination of the message can be a user-defined object instead of another process. To make this possible, HPX uses what it calls Active Global Address Space (AGAS) [21], see Fig. 6.2 on page 51 in Chap. 6. With AGAS, HPX provides a mechanism to address special objects which HPX calls *components* in the global runtime. Each object is assigned a unique **G**lobal **Id**entifier (GID). The GID is consistent through the life time of the object, even if the object is moved to a different process. For more details about accessing the GID, we refer to Sect. 14.1.3.

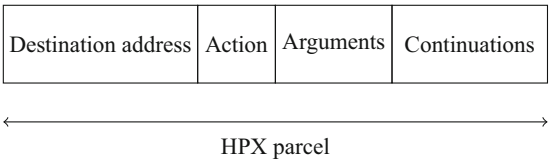


Fig. 14.1 An Active Message contains the following components: *destination address*, the process where the function is executed; *action*, the function to be executed; *arguments* the function arguments; and *continuations*, optional work which is executed if the action finishes

For consistency, localities themselves have a GID so that individual messages can be sent to processes directly.

Typically, when a computation is described by distributed objects communicating by Active Messages, one thinks of the Actor Model [144]. The Actor Model describes a set of remotely addressable objects called actors. Actors can receive asynchronous messages. In response to a message, an actor can do one of three things (1) create more actors, (2) send more messages, (3) and/or change state. In the actor model, actors can only process their messages sequentially. The virtues of the Actor Model are that it is simple, does not deadlock, and is immune from low-level data races.

HPX differs from the Actor Model in that its objects are not restricted to sequential execution of messages (which means that the programmer must take on the complex task of avoiding race conditions and deadlocks in method calls), and that it supports remote calls not attached to any object. This is consistent with C++ being a mixed paradigm language, i.e. one that supports object-oriented programming and procedural programming.

Note that HPX is not alone in leveraging Active Messages and remote objects. Many other frameworks, e.g. UPC\UPC++ [26], Legion [116], Chapel [24], and FORTRAN coarrays; use active messaging via **G**lobal-**A**ddress **S**pace **Net**working (GASNet) [102].

14.1.1 Plain Action

Traditionally, an MPI program works like this: when the program starts, it runs `main` on all ranks (MPI's terminology for a process) and then, depending on the rank, the program decides what it will do. For the case of a simple message exchange (See Listing 14.1 on page 150), one rank decides to send and the other to receive.

HPX, on the other hand, only starts `main` on locality 0 (HPX's terminology for a process). Work is then sent to the other localities as needed. For the case of a simple message exchange (See Listing 14.2 on page 151), it then invokes the message sending function call asynchronously on locality 1 and returns a future whose contents can be received with `get`.

Whether one version of "Hello, world" is easier than the other is not the point of this comparison. The point is to ask which methodology, which way of thinking about coding a distributed program is more natural and convenient. The answer may depend on a variety of factors.

Both of these codes do the same thing. MPI does it more efficiently because it requires the programmer to explicitly identify the buffer of bytes to send and how much to allocate for receiving, low-level details which have to be gotten right for the program to work (note the `size()+1` on the call to `MPI_Send` to make sure the null termination is captured). HPX, on the other hand, handles the message by automatically serializing the return value from the function. That means the msg() task can just as easily be called locally (even asynchronously and/or locally). It also

Listing 14.1 A simple MPI code that sends the message "Hello, world"

```
#include <mpi.h>
#include <sstream>

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int sender = 1;
  int receiver = 0;
  int tag = 16;
  if(rank == sender) {
    std::ostringstream msg;
    msg << "Hello, world, from " << sender;
    auto m = msg.str();
    std::cout << "Sender is: " << sender << std::endl;
    MPI_Send(m.c_str(), m.size()+1, MPI_CHAR, receiver, tag,
        MPI_COMM_WORLD);
  } else if(rank == receiver) {
    char msgbuf[100];
    MPI_Recv(msgbuf, sizeof(msgbuf), MPI_CHAR, sender, tag,
        MPI_COMM_WORLD, nullptr);
    std::cout << "Receiver is: " << receiver << std::endl;
    std::cout << "Message is: " << msgbuf << std::endl;
  }
  MPI_Finalize();
}
```

means that complex data dependencies that are the result of multiple distributed tasks can be easily and correctly resolved on the fly, as naturally as they would in any asynchronous code running on a single node.

We think there is no question that the HPX method of writing distributed code is easier for some applications.

14.1.2 Components and Actions

Previously, we introduced the concept of active messages and parcels. Within a parcel (which is an Active Message), the first data item is the function to be called on the remote machine. In HPX's terminology, that remotely callable function is called an *action*. An object whose methods are *actions* is called a *component*, and its remotely-callable methods are *component actions*. An *action* that is not part of a *component* is called a *plain action*.

In this section, we study components, component actions, and plain actions. Figure 14.2 on page 151 sketches the principle of components and actions within HPX. The Active Global Address Space (AGAS) [21] contains objects 1 to N. Each

Listing 14.2 A simple MPI code that sends the message "Hello, world"

```
#include <iostream>

#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include <sstream>

std::string msg() {
  std::ostringstream msg;
  msg << "Hello, world, from " << hpx::find_here();
  return msg.str();
}

// Make it possible to call msg() on another location.
HPX_PLAIN_ACTION(msg, msg_action);

int main() {
  auto localities = hpx::find_remote_localities();
  std::cout << "Sender is: " << hpx::find_here() << std::endl;
  std::cout << "Receiver is: " << localities[0] << std::endl;
  // Run the message on locality 1 and get the result in a future
  hpx::future<std::string> f = hpx::async<msg_action>(localities[0]);
  std::cout << "Message is: " << f.get() << std::endl;
}
```

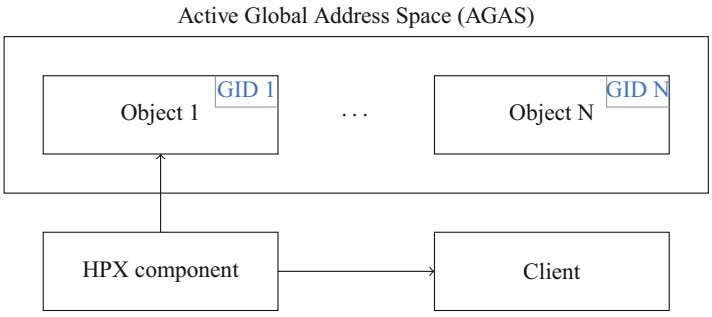


Fig. 14.2 Sketch for HPX component and component actions. An **HPX component** holds a object within the Active Global Address Space (AGAS), its unique Global ID (GID). The component provides an interface to set and access the object. All member functions of the component are registered as **component actions** for remote function calls. The **client** provides the interface to add or access objects locally or remotely using asynchronous programming

object has a unique global identifier (GID) which is constant during its lifetime. Even though components have a global identifier, they must reside in the local memory of one specific process on one computational node. Therefore, in order to facilitate calls from other processes, each object has a *client* interface. The client

has all the same methods as the object it represents but none of its state. When a method is called on a client, the call is encoded into a parcel and sent to the concrete instance of the object, i.e. the *server*.

Because components are potentially migratable, each component that might move must implement the serialization of its data so that it can be turned into a stream of bytes and sent over the network. (For more details about serialization, we refer to Sect. 14.2. For examples, we refer to Listing 14.7 on page 160.)

14.1.2.1 Components and Component Actions

Listing 14.3 on page 153 shows the `struct data_server`, which serves as the HPX component for the object `data`. To make the `struct` a component, we must inherit from `hpx::components::component_base`. In Line 5 a default constructor is defined. Note that a default constructor is required. A common compiler error for programmers creating components is that the object is missing the default constructor.

In Line 30, we use a `vector` to contain the data. For security reasons the vector `data` is declared as `private`. We provide an action to access the data, but the data can not be manipulated by callers of the method.

When this object is generated by HPX, it is assigned a Global ID (GID). In Line 7 a constructor that fills in the vector with `size_t size` elements of value 0 is defined. In Line 12, a second constructor which initializes the vector with `size_t size` elements of value `double const value` is provided.

In Line 18, a method to return a copy of the private `vector<double> data` is implemented. Note that the data returned will be automatically serialized if this function is called remotely. In Line 26, the component action for the `get_data` member function of the object is defined using the macro `HPX_DEFINE_COMPONENT_DIRECT_ACTION`.[1] Note that the action needs to be registered as well as defined, see Line 43. On Line 38, we use the `HPX_REGISTER_COMPONENT`[2] macro to generate the code required to make our `data_server` remotely accessible. This macro also allows us to remotely create the component via `hpx::new_<>()`.[3] Note that the code generation done by HPX is a detail beyond the scope of this book.

[1] https://hpx-docs.stellar-group.org/latest/html/libs/full/async_distributed/api/base_lco.html.

[2] https://hpx-docs.stellar-group.org/latest/html/libs/full/runtime_components/api/component_factory.html.

[3] https://hpx-docs.stellar-group.org/latest/html/libs/full/runtime_components/api/new.html?highlight=hpx%20new_.

Listing 14.3 The `struct data_server` for a HPX component with a single component action

```
struct data_server
    : hpx::components::component_base<data_server> {

  // Construct new instances
  data_server() {}

  data_server(size_t size_) {
    for(int i=0;i<size_;i++)
      data.push_back(0.0);
  }

  data_server(size_t size, double const value) {
    for (size_t i=0;i < size; ++i)
      data.push_back(value);
  }

  // Access data
  std::vector<double> get_data() const {
    return data;
  }

  // Every member function which has to be invoked remotely needs
  // to be  wrapped into a component action. The macro below
  // defines a new type 'get_data_action' which represents the
  // (possibly remote) member function
  HPX_DEFINE_COMPONENT_DIRECT_ACTION(
  data_server, get_data, get_data_action)

  private:
    std::vector<double> data;
};

// Code generation via macros to expose the data_server as a
// component for remote access

// HPX_REGISTER_COMPONENT() exposes the component creation
typedef hpx::components::component<data_server> data_server_type;
HPX_REGISTER_COMPONENT(data_server_type, data_server)

// HPX_REGISTER_ACTION() exposes the component member function
// for remote invocation.
typedef data_server::get_data_action get_data_action;
HPX_REGISTER_ACTION(get_data_action)
```

14.1.2.2 Client

For each component a client is defined. Listing 14.4 on page 155 shows the `struct data_client`, which inherits `hpx::components::client_base` and relates to the component `data_server` in Listing 14.3 on page 153. In Line 4 of Listing 14.4 on page 155 the `typedef` is used for simplification. An empty constructor is defined in Line 6. In Line 13 a constructor to create a new component on the locality *where* and initialize the data with a specific value. Note that we pass the size and the initial value from the constructor to `hpx::new_<>` which does the remote allocation within AGAS. However, we have one additional argument `hpx::id_type where`. With it, we specify on which locality we want to generate the data. For more details about how to receive topology information, we refer to Sect. 14.1.3. We use `hpx::new_<>` to allocate the data within AGAS. Here, we need to pass the `hpx::id_type` to specify if we want to allocate the data on the local memory or remote memory on another node.

Line 17 shows the second constructor which takes the `hpx::id_type` wrapped within a `hpx::future` for possibly remote data. Here, we use `std::move` to move the future containing the id and avoid the copying. For more details about move semantics, we refer to Sect. A.3. The last functionality we need is to access the data. Therefore, we define a helper function in Line 21 to asynchronously call the component action defined in Line 26 of Listing 14.3 on page 153.

14.1.2.3 Using Components and Components Actions

Now, we will use the `data_server` and `data_client` to demonstrate the usage of component actions, see Listing 14.5 on page 156. In Line 11, we look for all remote localities. In Line 15 a vector is initialized with ten `double` values. Since we used the result of `hpx::find_here()`, the data is allocated within local memory. For more details about topology information, we refer to Sect. 14.1.3. In Line 16, we asynchronously access the data on the current locality using the component action `get_data()`. In Line 21, we fill a vector of twenty `double` values with 2 on the first remote locality in the `localities` vector. Note that we need to check if there are remote localities available first. If there are none, the vector won't be created. In Line 22, we asynchronously request the data from the remote locality. In Line 23, we call `.get()` on the future to receive the vector. After that we call `data()` on the vector to access the deserialized data. In Line 27 we access the deserialized data on the local component.

We want to make two remarks on this example. First, this was an educational example for the usage of component actions and was not optimized at all. Second, we use the default serialization available in HPX in this example.

Listing 14.4 The `struct data_client` for a HPX component with a single component action

```
struct data_client
    : hpx::components::client_base<data_client, data_server> {
  typedef hpx::components::client_base<data_client, data_server>
      base_type;

  data_client() {}

  // Create new component on locality 'where' and initialize the
  // held data
  data_client(hpx::id_type where, std::size_t size,
              double initial_value)
      : base_type(hpx::new_<data_server>(where, size,
                                         initial_value)) {}

  // Attach a future representing a (possibly remote) partition.
  data_client(hpx::future<hpx::id_type> &&id)
      : base_type(std::move(id)) {}

  // Invoke the (remote) member function which gives us access to
  // the data. This is a pure helper function hiding the async.
  hpx::future<std::vector<double>> get_data() const {
    return hpx::async(get_data_action(), get_id());
  }
};
```

14.1.2.4 Recap

Let us recap the definitions for distributed computing in HPX:

- **Component**: A `struct` or `class` which contains the object where the actual data is represented. This object is generated in the Active Global Address Space (AGAS) with a unique Global ID (GID). The component is registered using the macro `HPX_REGISTER_COMPONENT`.
- **Component action**: A handle to a member function of a `struct` or `class` which is registered as a component. Each member function is declared as an action within the scope of the `struct` or `class` by using the macro `HPX_DEFINE_COMPONENT_DIRECT_ACTION`. After the definition each action is attached to a component by using the macro `HPX_REGISTER_ACTION`.
- **Client**: A `struct` or `class` with methods to access the component's data synchronously or asynchronously.

Listing 14.5 Example for component and component actions

```
#include <vector>
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>

#include <data_server.hpp>
#include <data_client.hpp>

---

int main(int args, char **argv) {

  std::vector<hpx::id_type> localities = hpx::find_all_localities
      ();

  // Generate ten double values initialized with 1
  // on the local locality, i.e. localities[0].
  data_client local = data_client(localities[0],10,1);
  hpx::future<std::vector<double>> data_local = local.get_data();


  // Generate ten double values initialized with 2
  // on the first remote locality
  if (localities.size() > 0) {
          data_client remote(localities[1],20,2);
          hpx::future<std::vector<double>> data_remote = remote.
              get_data();
          std::vector<double> d_remote = data_remote.get();
      std::cout << "remote: " << d_remote[1] << std::endl;
  }

  std::vector<double> d_local = data_local.get();
  std::cout << "local: " << d_local[1] << std::endl;

  return EXIT_SUCCESS;
}
```

- **Plain action**: A handle for a regular function to be called asynchronously either remotely or locally. A plain action is registered by using `HPX_PLAIN_ACTION` and is not associated with an HPX component.

14.1.3 Receiving Topology Information

Listing 14.6 on page 158 shows various methods for accessing topology information in HPX. HPX provides functions to get the total number of system threads

`hpx::get_os_thread_count`,[4] like `omp_get_max_threads`[5] for OpenMP, see Line 5. Note that the default is that HPX uses all available system threads and `--hpx:threads=10` will restrict HPX to use only ten system threads for its light-weight threads. This is similar to `export OMP_NUM_THREADS=10` to only use ten OpenMP threads.

For distributed programming, HPX provides functions to get information about the total number of localities using `hpx::get_num_localities`[6] which relates in MPI to `MPI_Comm_size(MPI_COMM_WORLD, & mpi_size)` to get the number of ranks, see Line 8. In addition, HPX provides a function `hpx::find_all_localities`[7] to find all available localities, see Line 11. The returned list of localities relates to the default MPI communicator `MPI_COMM_WORLD`. This includes all remote localities and the current locality. To find all remote localities which are not on the same computational node, HPX provides `hpx::find_remote_localities`, see Line 14. With these functions, we can obtain information on the number of system threads or number of computational nodes.

However, HPX uses the active global address space (AGAS) and therefore functions are provided to receive the locality where a function is called or the locality where the object resides in the memory of the locality. The function `hpx::find_here()`[8] provides the global address of the locality the function is executed on, i.e. the current locality see Line 18. With `hpx::find_locality` the locality hosting the component is returned. With `hpx::get_colocation_id` the locality hosting the object with the given address is given. The list below summarizes the functions described above:

- `hpx::find_here()`
  Get the global address of the locality the function is called on.
- `hpx::find_all_localities()`
  Get the global addresses of all available localities.
- `hpx::find_remote_localities()`
  Get the global addresses of all available remote localities.
- `hpx::get_num_localities()`
  Get the number of all available localities.
- `hpx::find_locality()`
  Get the global address of any locality hosting the component.

[4] https://hpx-docs.stellar-group.org/latest/html/libs/core/runtime_local/api/get_os_thread_count.html.

[5] https://www.openmp.org/spec-html/5.0/openmpsu112.html.

[6] https://hpx-docs.stellar-group.org/latest/html/libs/full/runtime_distributed/api/get_num_localities.html.

[7] https://hpx-docs.stellar-group.org/latest/html/libs/full/runtime_distributed/api/find_all_localities.html.

[8] https://hpx-docs.stellar-group.org/latest/html/libs/full/runtime_distributed/api/find_here.html.

Listing 14.6 Functions to receive topology information for distributed HPX applications

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

// Get the number of OS threads
std::cout << hpx::get_os_thread_count() << std::endl;

// Get the number of all localities
std::cout << hpx::get_num_localities().get() << std::endl;

// Get all localities
std::cout << hpx::find_all_localities().size() << std::endl;

// Get all remote localities
std::cout << hpx::find_remote_localities().size() << std::endl;

// Get the global address of the locality the function is
// executed, the current locality.
hpx::id_type here = hpx::find_here();
```

- `hpx::get_locality_id()`
  Get the integer id of the current locality.
- `hpx::naming::get_locality_id_from_id(loc)`
  Get the integer id of `loc`.
- `hpx::get_colocation_id(obj)`
  Get the locality hosting the object `obj`.

Important

The most important lessons learned here are the following:

- HPX uses the Active Global Address Space (AGAS) to address any object using a unique **G**lobal **Id**entifier (GID). This includes localities as well as components. Component ids remain constant even if the object moves.
- HPX only runs `main` on locality. Other localities will remain idle until a remote function is called on them.
- HPX uses active messages for remote function calls. Active messages are called parcels in HPX. The name is an acronym for **Pa**rallel **C**ontrol **El**ement (parcel).
- Remotely executed functions are called actions. There are two types: plain actions (which are functions not attached to any object) and component actions (which are methods of components).

14.2 Serialization

In Sect. 13.4 we briefly discussed the concept of serialization. Most of the time, if we are using standard container types such as `map`, `vector`, `tuple`, etc., and the built-in types such as `double`, `int`, etc., we won't have to think about serialization. It will be automatic. But there are a few cases where it cannot be, e.g. user-defined classes and basic C-style arrays. The latter cannot be serialized by default because there's no easy way to automatically obtain their size.

Recall that if we want to send data from one node to another node over the network, the data needs to be flattened into a stream of bytes. For the cases were automatic serialization is not available, HPX provides several tools. One of these is `hpx::serialization::serialize_buffer<T>`. This is a container with behavior similar to `std::shared_ptr<T[]>`. The main differences are that `serialize_buffer` has a size function and it can be serialized (where `std::shared_ptr<T[]>` cannot).

Some boilerplate functionality is needed to use `serialize_buffer`, e.g. to allocate the memory within the shared memory of the node, the `std::allocator<T>`[9] is used.

For serialization of a user-defined class, the function `serialize(Archive& arg, const unsigned int)` is used.

To illustrate how to implement serialization, we consider one easy example. We create an object to track data sets. It contains a label and a buffer filled with double values.

We want to create a function that generates a container filled with `double` values $\{v_i = i | i = 1, \ldots, n\}$. However, we want to compute the square values $v_i = v_i^2$ of all elements in the container. To do this, we need to serialize the container v to send it over the network using a HPX parcel to the remote locality. On the remote locality where the serialized data is received, the HPX data is deserialized and the computation of the square values is computed.

Listing 14.7 on page 160 shows the `struct data` which stores a label and a number of `double` values within a serializable buffer. For more details about the implementation of HPX's serializable buffer, we refer to [145]. In Line 9 we use the `hpx::serialization::serialize_buffer<double>` and declare for simplicity the `buffer_type`. In Line 47, the `data_` object is defined as `buffer_type`. In Line 40, a function to obtain the size is defined. In Line 36 and Line 38, the access operator `[]` is overloaded to fetch the data from the buffer. The second version is for use with `const data` objects, such as the one used when we overload the `operator<<(std::ostream,const data&)` function in Line 67.

Before we implement the serialization the constructors need to be defined. In Line 12, a default constructor is defined and the buffer is uninitialized. In Line 13,

[9] https://en.cppreference.com/w/cpp/memory/allocator.

the constructor to initialize the buffer is defined and no values are assigned to the buffer. In Line 25 is a constructor to initialize the buffer with a label and a single value. In Line 18, the constructor to initialize the buffer with a list of values is defined. Finally, in Line 31, we initialize the values using a function that takes an `int` (the index of the element), and returns a `double`.

In Line 49 we befriend the `class hpx::serialization::access` (We would not need to do this if our serialization method was public). In Line 51 we finally define the serialization. Note that we pass the buffer `data_` and the `label` are serialized by simply applying them to the archive with `operator&`.

Now, we will use the `struct data` to call a remote function on a different node using the function `square` in Line 58. To call the function remotely, we define a `HPX_PLAIN_ACTION` in Line 76. For details about plain actions, we refer to Sect. 14.1. In Line 79 we generate a `data` object on the local node with ten elements and fill the values using a lambda function. In Line 81 we ask for all remote localities. If we found at least one remote locality in Line 84, we execute the `square` action on the first remote locality. For more details, about topology information, we refer to Sect. 14.1.3. Note that the object `data d` on our local locality needs to be serialized and send over the network to the remote locality. On the remote locality the data is deserialized and processed within the `square` action. That is only possible since we implemented the serialization for the `struct data`. Finally, we have to get a copy of the `data` object back (or we cannot access the newly generated data). See Line 84.

Listing 14.7 Using HPX's serialization for remote function calls

```
1  #include <cstdlib>
2  #include <hpx/hpx.hpp>
3  #include <hpx/hpx_main.hpp>
4  #include <iostream>
5
6  struct data {
7  private:
8    typedef hpx::serialization::serialize_buffer<double>
9        buffer_type;
10
11 public:
12   data() : label("empty") {}
13   data(const std::string& label_,size_t size)
14       : label(label_),
15         data_(std::allocator<double>().allocate(size),
16            size, buffer_type::take) {}
17   data(const std::string& label_,size_t size, double*
18      data__)
19       : label(label_),
20         data_(std::allocator<double>().allocate(size),
21            size, buffer_type::take) {
22     for (size_t i = 0; i < size; i++) data_[i] = data__
23        [i];
24   }
25   data(std::string label_,size_t size, double data)
```

```
26        : label(label_), data_(std::allocator<double>().
27           allocate(size), size, buffer_type::take) {
28      for (size_t i = 0; i < size; i++) data_[i] = data;
29    }
30    data(std::string label_,size_t size, std::function<
31       double(int)> f)
32        : label(label_), data_(std::allocator<double>().
33           allocate(size), size, buffer_type::take) {
34      for (size_t i = 0; i < size; i++) data_[i] = f(i);
35    }
36    double& operator[](size_t id) { return data_[id]; }
37    double operator[](size_t id) const { return data_[id
38       ]; }
39
40    size_t size() const { return data_.size(); }
41
42    std::string get_label() const { return label; }
43    void set_label(const std::string s) { label = s; }
44
45   private:
46    std::string label;
47    buffer_type data_;
48
49    friend class hpx::serialization::access;
50
51    template <typename Archive>
52    void serialize(Archive& ar, const unsigned int) {
53      ar & label;
54      ar & data_;
55    }
56  };
57
58  static data square(data d) {
59    for (size_t i =0; i <  d.size() ; i++) d[i] = d[i] *
60       d[i];
61    d.set_label(d.get_label() + " squared");
62    return d;
63  }
64
65
66  std::ostream& operator<<(std::ostream& o, const data& d
67     ) {
68      o << "data('" << d.get_label() << "',{";
69      for(size_t i=0;i<d.size();i++) {
70          if(i > 0) o << ", ";
71          o << d[i];
72      }
73      return o << "})";
74  }
75
76  HPX_PLAIN_ACTION(square,square_action);
77
78  int main(int args, char** argv) {
79      data d = data("stats",10,[](int n) { return n; });
```

```
80
81      auto remote = hpx::find_remote_localities ();
82      if ( remote.size () > 0) {
83        hpx::future<data> future = hpx::async<
84           square_action>(remote[0],d);
85        data result = future.get();
86        std::cout << result << std::endl;
87      }
88    return EXIT_SUCCESS;
89  }
```

Important

The most important lesson learned here is that serialization is required to send data over the network to remote localities, and that can be done with the `serialize` function.

Chapter 15
Examples of Distributed Programming

In this chapter, we use the distributed implementation of the Taylor series of the natural logarithm as the first example. We will extend an existing shared memory example, using it to provide further examples of serialization and plain data actions. We strongly recommend that the reader has read and understood the two previous chapters, the chapters on active messaging and serialization. See Sects. 14.1 and 14.2. Second, we will expand on the implementation of the fractal sets using the *supervisor-worker* pattern.

The *supervisor-worker* pattern designates one process or thread as the supervisor. The only job of the supervisor is to assign work to the other processes or threads and to gather the results. As each worker finishes the task it was assigned, it returns a result and requests more work. When the worker has no more work, it terminates. The super-visor worker pattern is useful when workloads are highly parallel and uneven.

15.1 Distributed Implementation of the Taylor Series of the Natural Logarithm

In this section, we revisit the shared memory implementation of the Taylor series of the natural logarithm using asynchronous programming from Chap. 9. In Listing 9.8 on page 96 in Sect. 9.2, you can review the code for the asynchronous implementation. Recall from Fig. 8.1 on page 80 in Chap. 8 that we divided the input into chunks and each thread operated on its assigned chunk. However, the complete data was allocated within the memory of the same process. Now each chunk is allocated within the Active Global Address Space (AGAS), see Fig. 15.1 on page 164, and is addressable from other processes. In this example we assume that we have three processes running on three nodes: N_1, N_2, and N_3. During the concurrent execution, each process is working on its assigned chunk C_1, C_2, and C_3. To collect the results, we introduce a global barrier where we wait until all

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_15

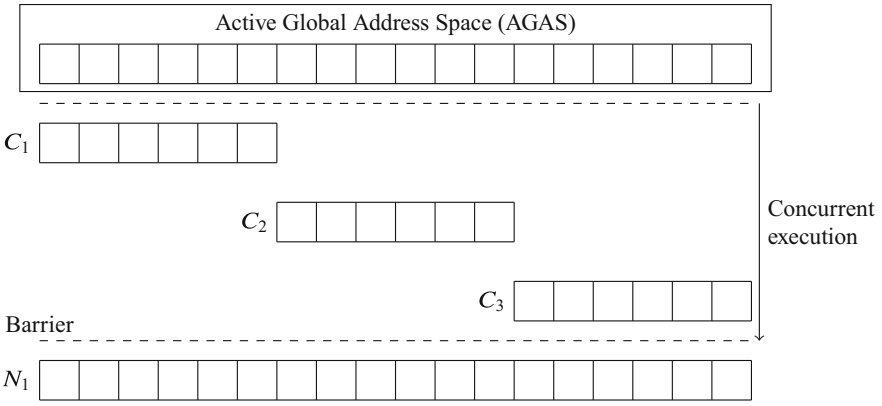


Fig. 15.1 Illustration of the division of work for concurrent computation using HPX's distributed capabilities for asynchronous computing. In the top of the image the work items are shown with Active Global Address Space (AGAS). In this example we have 18 work items in total. The work items are equally distributed to three chunks C_1, C_2, and C_3. Each of the chunks is allocated on the local memory of each process N_1, N_2, and N_3. Within each process, three threads work concurrently on their chunk of 6 elements. After all three nodes are finished, node N_1 collects their results

three processes have finished. Process N_1 then collects the results of the other two processes and combines them with its own. Note that the principle here is very similar, however, we need to use different HPX features for the distributed implementation.

Since HPX provides a unified interface for remote and local function calls, we can use this code as a basis for the asynchronous distributed implementation. In the previous implementation, a lambda function was used and passed to `hpx::async` since we only wanted to call the function to compute the Taylor series element-wise and compute the sum of them locally. For the distributed example, we need to define a plain action to start the work on the other processes. Listing 15.1 on page 165 shows the `static` function `compute` with the arguments `data_client` and `x`. Within the function, we use a `for` loop and compute the sum as we did before within the lambda function. In Line 11 of Listing 15.2 on page 165 we define the function as a plain action. For more details about plain actions, we refer to Sect. 14.1.

Listing 15.2 on page 165 shows the main method of the code. In Line 17, we obtain all localities available to the application. This will determine the number of partitions. Note that we use the same pattern as for the shared memory implementation and divide the computation of the sum into chunks. However, here the chunks are computed on different processes (potentially different nodes) and not necessarily on different cores of the same node. For more details about receiving topology information, we refer to Sect. 14.1.3. The size obtained in Line 18 is analagous to the `MPI_Comm_size` within an MPI application, see Line 16 of Listing 13.1 on page 133. In Line 19 of Listing 15.2 on page 165, the size of each

partition is computed the same as before, however, instead of the number of threads the number of processes is used. In Line 23, the partition assigned to each process is filled with the corresponding values. Note that we use the locality id to define the process in which the data was generated (See Line 29 and Line 31).

Listing 15.1 Definition of the plain action to compute the piece-wise evaluation of the Taylor series and compute the partial sum

```
1  #ifndef TAYLOR_ACTION_HPP
2
3  #define TAYLOR_ACTION_HPP
4
5  static double compute(data_client client, double x) {
6    data d = client.get_data().get();
7    size_t size = d.size();
8    double sum = 0;
9
10   std::cout << d[size - 1] << std::endl;
11
12   for (size_t i = 0; i < size; i++) {
13     double e = d[i];
14     sum += std::pow(-1.0, e + 1) * std::pow(x, e) / (e);
15   }
16   return sum;
17 }
18
19 #endif
```

Listing 15.2 Distributed implementation of the Taylor series for the natural logarithm using HPX

```
1  #include <cstdlib>
2  #include <hpx/hpx.hpp>
3  #include <hpx/hpx_main.hpp>
4  #include <iostream>
5
6  #include "taylor_data.hpp"
7  #include "taylor_component.hpp"
8  #include "taylor_action.hpp"
9  ---------------------------------------------------------------
10
11 HPX_PLAIN_ACTION(compute, compute_action)
12
13 int main(int args, char** argv) {
14   int n = 100;
15   double x = 0.25;
16
17   std::vector<hpx::id_type> localities = hpx::find_all_localities
       ();
18   std::size_t num_partitions = localities.size();
19   size_t partition_size = std::round(n / num_partitions);
20
21   std::vector<data_client> parts(num_partitions);
22
```

```
23    for (size_t i = 0; i < num_partitions; i++) {
24      size_t begin = 1;
25
26      if (i > 0) begin = i * partition_size;
27
28      if (i == num_partitions - 1)
29        parts[i] = data_client(localities[i], n - (partition_size *
              i), begin);
30      else
31        parts[i] = data_client(localities[i], partition_size, begin
              );
32    }
33
34    std::vector<hpx::future<double>> futures;
35
36    for (size_t i = 0; i < num_partitions; i++) {
37      futures.push_back(hpx::async<compute_action>(localities[i],
          parts[i], x));
38    }
39
40    double result = 0;
41
42    hpx::when_all(futures)
43      .then([&](auto&& f) {
44        auto futures = f.get();
45        for (size_t i = 0; i < futures.size(); i++)
46          result += futures[i].get();
47      })
48      .get();
49
50    std::cout << "Difference of Taylor and C++ result "
51              << result - std::log1p(x) << " after " << n << "
                  iterations."
52              << std::endl;
53
54    return EXIT_SUCCESS;
55  }
```

In the shared memory implementation, the `std::vector` was allocated on a single computational node and all the threads operated on chunks of the vector. Here, we need to use components and component actions to remotely allocate the chunks. In Line 34, we use a `std::vector` of `hpx::future<double>` from the launches of the asynchronous function to communicate the partial sums for the Taylor series. In Line 36, we use the exact same `for` loop to launch the plain action instead of the lambda function. Note that we call `hpx::async` differently here than we did for the shared memory implementation.

Note that, for potentially remote calls, `hpx::async<>` requires a template parameter that we need to fill with the plain action or component action. In our case, we provide the plain action `compute_action` which is declared in Line 11 of Listing 15.2 on page 165. The first argument is now the `hpx::id_type` where the function will be executed, and all other arguments are the function parameters of the `static` function in Line 5 of Listing 15.1 on page 165. Note that the `hpx::future` returned by a plain action or component action is the same data structure used for the single node code and only the `hpx::async` call is different. This allows us to combine local and remote function calls in HPX's asynchronous execution graph. From Lines 40–52 of Listing 15.2 on page 165 we use the same code to collect the partial results that we did in the shared memory implementation. In conclusion, we had to make two minor changes to transform the code from a shared memory implementation to a distributed memory implementation.

Up to this point, we have focused on changing the code and ignored the internal workings of the `data_server` component. For completeness, we will now take a look at the component server and component client. Let, us look at the header `#include "taylor_data.hpp"`. Listing 15.3 on page 167 shows the implementation of the `data` object which inherits from `serialize_buffer`, a class HPX already knows how to serialize. The code of the header `#include "taylor_component.hpp"` is shown in Listing 15.4 on page 168. It works basically the same way as the code in Listings 14.3 on page 153 and 14.4 on page 155 in Sect. 14.1. The only real difference is the simpler `data` class.

Listing 15.3 Definition of the `data` object and serialization

```
1  #ifndef TAYLOR_DATA_HPP
2
3  #define TAYLOR_DATA_HPP
4
5  typedef hpx::serialization::serialize_buffer<double> buffer;
6  struct data : public buffer {
7    data(size_t size=0, double value=0.0) :
8      buffer(std::allocator<double>().allocate(size), size, buffer
           ::take)
9    {
10     for(size_t i=0;i < size; i++) (*this)[i] = value+i;
11   }
12 };
13
14 #endif
```

Listing 15.4 Definition of the component client and component server

```
#ifndef TAYLOR_COMPONENT_HPP

#define TAYLOR_COMPONENT_HPP

struct data_server
    : hpx::components::component_base<data_server> {
  // Construct a new instance
  data_server() {}

  data_server(size_t size, double const value) : data_(size,
      value) {}

  // Access data
  data get_data() const { return data_; }

  HPX_DEFINE_COMPONENT_DIRECT_ACTION(data_server, get_data,
      get_data_action)

 private:
  data data_;
};

typedef hpx::components::component<data_server> data_server_type;
HPX_REGISTER_COMPONENT(data_server_type, data_server)

typedef data_server::get_data_action get_data_action;
HPX_REGISTER_ACTION(get_data_action)

// Component client

struct data_client
    : hpx::components::client_base<data_client, data_server> {
  typedef hpx::components::client_base<data_client, data_server>
      base_type;

  data_client() {}

  data_client(hpx::id_type where, size_t size, double init)
      : base_type(hpx::new_<data_server>(where, size, init)) {}

  hpx::future<data> get_data() const {
    data_server::get_data_action act;
    return hpx::async(act, get_id());
  }
};
#endif
```

15.2 Distributed Implementation of the Fractal Set

In this section, we extend the fractal set code for distributed memory using HPX's distributed features of asynchronous programming. Note that these distributed features are not in the C++ standard and will only work with HPX. For the asynchronous programming features in the C++ standard for shared memory parallelism, we refer to Chap. 9. To keep things simple, since the focus here is on HPX's distributed features, we use the most universal approach to parallelism: the supervisor-worker pattern. Here, we follow the idea that the work can be divided into many tasks which can be done mostly independently. As each worker finishes a task, it comes back to the supervisor for a new one, until the work is done. With supervisor-worker, even if the work per task is uneven, or the workers have different clock speeds, the load will automatically balance.

Listing 15.5 on page 170 shows the `main` function for the distributed fractal set. In Line 8, we ask HPX to provide us with the remote localities and store them in an `std::vector`. In Line 11, the current locality will be designated as the supervisor node. Note that which node is supervisor is an arbitrary choice, but with HPX the first thread started is an obvious choice. In Line 14, the number of worker localities is determined.

Until now, all the lines of code were a preparation for the actual distributed computation of the fractal sets. In Line 28, we define a `std::vector` containing `hpx::futures`. Let us glance back at the shared memory implementation in Listing 9.8 on page 96 using asynchronous programming in Sect. 9.2. Here, we used exactly the same `hpx::future` even though it was not distributed.

In Line 31 of Listing 15.5 on page 170 a `for` loop is used to assign the work to the worker pool. The interesting part here is happening in Line 33 where we obtain an `hpx::future` from the call `hpx::async<do_work_action>`. Neglecting the additional parameter in the parentheses, this looks very similar to the call in the previous example on a single node. With the additional argument, we specify the HPX action which we want to launch asynchronously. For the shared memory implementation, we could provide here a function or a lambda function, since everything was in the same process. Here, we specify the name of the action `do_work_action` and the first argument `loc` of the `hpx::async<do_work_action>` defines where the action will be executed. For more details about HPX actions, we refer to Sect. 14.1.2. In Line 34 the futures of the asynchronous function launches are collected in the vector `data` which will be later used for synchronization in Line 37. Note that we use the move semantics `std::move` to avoid copying the future, for more details we refer to Sect. A.3.

In conclusion, the distributed example does not differ too much from the parallel example in Chap. 9. This, again, is due to the fact that HPX provides a unified API for remote and local function calls.

Listing 15.5 Main function of distributed fractal set

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

#include <distributed_fractal_worker.hpp>

---

int main() {
    auto locs = hpx::find_remote_localities();

    // The supervisor
    auto supervisor = hpx::find_here();

    // Compute the number of worker localities
    const int num_workers = locs.size();

    // Make sure we have some workers
    assert(num_workers > 0);

    // The logic below assumes more work than workers
    assert(num_workers < size_x && num_workers < size_y);

    // Allocate the PBM image only for the supervisor
    p = PBM(size_y, size_x);

    start = std::chrono::high_resolution_clock::now();

    // Place to hold futures for current worker tasks
    std::vector<hpx::future<void>> data;

    // Load up workers...
    for(int j=0;j<num_workers;j++) {
        auto loc = locs.at(j);
        hpx::future<void> f = hpx::async<do_work_action>(loc,
            supervisor, next_work_item++);
        data.push_back(std::move(f));
    }
    for(int j=0;j<data.size();j++) {
        data.at(j).wait();
    }

}
```

Listing 15.6 on page 171 shows the action for the workers. Let us have a look at the actiondo_work_action the supervisor calls in Line 33 of Listing 15.5 on page 170 to assign the work to the workers. In Line 60 of Listing 15.6 on page 171 the action is registered as a plain action by calling the macro HPX_PLAIN_ACTION where the first argument is the function name and the second argument is the action name. For more details about actions, we refer to Sect. 14.1.2.

The function do_work defines the work each worker has to perform, see Line 35. The first thing the function does is to check whether work is still available since we only have to compute the vector of pixels as long as we have not computed the full size of the image in the x direction. In Line 39, we ask for the next work item asynchronously by calling hpx::async<get_work_action> and will receive an hpx::future with the index of the next work item. However, while we wait to receive the next work item, the computation of the current assigned work item will be carried out. In this way, we overlap communication and computation to efficiently use our resources. The action next_work_item is defined in Line 12. Note that calls to get_work within the supervisor process might not be running on the same thread. We could, therefore, encounter a race condition in incrementing next_work_item. To guard against this, the variable next_work_item is of type std::atomic <int>, see Line 6. For more details about race conditions, we refer to Sect. 7.2.

Listing 15.6 Functions for the supervisor and worker approach

```
1  #include <pbm.hpp>
2  #include <config.hpp>
3  #include <kernel.hpp>
4
5  std::chrono::high_resolution_clock::time_point start, finish;
6  std::atomic<int> next_work_item(0);
7  std::atomic<int> completed_work_items(0);
8  size_t output = get_size_t("OUTPUT", 1);
9  PBM p;
10
11 // Called by workers to get an item of work from the supervisor
12 int get_work() { return next_work_item++; }
13 HPX_PLAIN_ACTION(get_work, get_work_action);
14
15 // Called by workers to send a result to the supervisor
16 void send_result(const std::vector<int>& result, int item_index)
       {
17   if (item_index < size_x) {
18     p.row(item_index) = result;
19     int n = ++completed_work_items;
20     // If the work is complete
21     // then I can call finish up.
22     if (n == size_x) {
23       auto finish = std::chrono::high_resolution_clock::now();
24       auto duration =
25           std::chrono::duration_cast<std::chrono::microseconds>(
               finish - start);
```

```
26        std::cout << "duration: " << (duration.count() * 1e-6) <<
              std::endl;
27        if ( output == 1)
28            p.save("image_distributed.pbm");
29      }
30    }
31  }
32  HPX_PLAIN_ACTION(send_result, send_result_action);
33
34  // The basic unit of work.
35  void do_work(const hpx::id_type& loc, int item_index) {
36    while (item_index < size_x) {
37      // ask for the next work item asynchronously. That way
38      // we overlap communication and computation.
39      hpx::future<int> fitem = hpx::async<get_work_action>(loc);
40      std::vector<int> result;
41
42      complex c = complex(0, 4) * complex(item_index, 0) / complex(
            size_x, 0) -
43                  complex(0, 2);
44
45      for (int i = 0; i < size_y; i++) {
46        // Get the number of iterations
47        int value = compute_pixel(c + 4.0 * i / size_y - 2.0);
48        // Convert the value to RGB color space
49        std::tuple<size_t, size_t, size_t> color = get_rgb(value);
50        result.push_back(make_color(std::get<0>(color),
51                                    std::get<1>(color),
52                                    std::get<2>(color)));
53      }
54
55      // apply() is like async(), except no future comes back.
56      hpx::apply<send_result_action>(loc, result, item_index);
57      item_index = fitem.get();
58    }
59  }
60  HPX_PLAIN_ACTION(do_work, do_work_action);
```

From Line 40 to Line 53 we do the same computation for the pixels in each column as we do for the shared memory implementation in Listing 9.8 on page 96 in Chap. 9. Note The `for` loop here could be replaced by an `hpx::for_loop`, see Listing 10.2 on page 101 in Chap. 10, to have more parallelism. This would be similar to the concept of *MPI+OpenMP* in Sect. 13.5.1. However, with HPX we have a single API instead of two. Another option would be to just run more MPI ranks (or HPX localities) per node, even one per core. Which combination of threads and processes work best for an application can only be determined by experimentation.

Once we have finished computing all the pixel colors for the current assigned column, we send the `std::vector<int> result` back to the supervisor using the action `send_result_action` in Line 56 of Listing 15.6 on page 171. Note that we use `hpx::apply`[1] here. This is called asynchronously, but we do not get the `hpx::future` back. After the completed work is sent to the supervisor, the worker calls `fitem.get()` to get its next work item. See Line 57. Note that we obtained this future before we did our assigned work. See Line 39. Therefore, it is very likely that the result is immediately available.

The last missing puzzle piece is the action `send_result_action` defined in Line 16. In the `send_result` function the supervisor saves the color values of the pixels at the position `item_index` to the PBM `p` object. Note that only the supervisor allocated this object. In Line 19 we check if the total number of completed work items is equal to `size_x`. If so, we have computed all rows of the image. In that case, we stop the timer and print the timing results. Finally, we save the image to the hard disk if the output flag is enabled. Recall, we use the PBM file format for educational purposes. However, we introduce some *O*verheads by sending the data back from the workers to the supervisor. For a production code, one would use a distributed file format where each node writes its data to a sub file and the supervisor writes a main file which links all these sub files. In this case, the visualization tool would combine the data and produce an image. For more details, we refer to Sect. 13.3.

15.3 Improved Distributed Implementation of the Fractal Set

In the previous implementation of the fractal set, we used collective I/O and sent the pixel data from all nodes to the supervisor node. We use the `HPX_PLAIN_ACTION` with the annotation `send_result_action` In Line 32 of Listing 15.6 on page 171. Function `send_result` in Line 16 receives the data and stores the data to the PBM image `p`. Here, additional *O*verhead is introduced while sending the pixel data to the supervisor node. This might work for smaller image sizes but not for larger image sizes. For parallel, efficient, and scalable high performance applications, we only want to send the necessary data to the supervisor node to minimize the network traffic. For various data formats the C++ libraries provide features for distributed I/O. For more details, we refer to Sect. 13.3. For these examples we use the PBM format. However, as with most image formats, the PBM format doesn't allow for distributed I/O. Other libraries, like VTK or Silo, have features for distributed I/O. To avoid introducing an external C++ library, we will write the pixel data in the Delimiter-Separated Values (DSV) format on each node. After the simulation, we will use the C++ in Listing 13.4 on page 139 in Sect. 13.5.1 to combine the DSV files into a PBM image.

[1] https://hpx-docs.stellar-group.org/latest/html/libs/core/async_base/api/launch_policy.html.

Next we consider the `do_work` function. See Listing 15.6 on page 171. Here, in Line 56, the `send_result_action` is called using `hpx::apply`. The `hpx::apply` is like `hpx::async`, except no future comes back. It is "fire and forget." Instead of sending the pixel data, we want to write the pixel data to a file per node. This approach is similar to the one we used to remove the collective I/O from the MPI-based implementation. See Listing 13.3 on page 137 in Sect. 13.5.1. Listing 15.7 on page 175 shows the modified `main` function of the distributed fractal set. Finally, we add the code to write out data to a local file within `do_work`.

To facilitate this, we add a final argument `id` to the list of arguments to identify the part of the file. See Line 13 of Listing 15.8 on page 176. In Line 14, the worker starts up and opens the file; in Line 33, it appends one vector of data to the DSV file; in Line 39, it closes the file and ensures that all output is flushed to disk.

15.4 Benchmark of the Distributed Implementation of the Fractal Sets

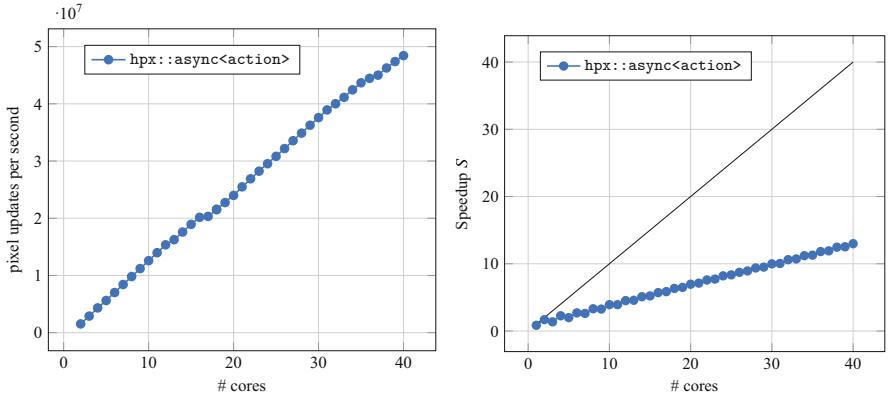
Before we investigate the performance of the distributed implementation, we will study the scaling on a single node and run the distributed code using an increasing number of CPU cores. Figure 15.2 on page 177 shows the scaling on a single node. The image size was set to $39{,}860 \times 21{,}600$ pixels. Here, we observe that using HPX actions on the same node still results in scaling of the code. Now we focus on the scaling on multiple nodes. First, we look into the scaling on a smaller cluster of 16 nodes with two Intel® Xeon® Gold 6148 Skylake CPU. Figure 15.3 on page 177 shows the scaling from a single node up to 16 nodes. The image size was set to $39{,}860 \times 216{,}000$ pixels. The blue line shows the scaling using a single core per node. Thus, we have only inter-node communication while all workers send their work item to the supervisor node. Here, we see scaling, but all other cores of the node are idle and we do not use their compute power. For the next two runs, we used 10 and 20 cores of each node. For the runs with 10 cores per node (green line), we observe scaling up to 12 nodes and for more nodes we have insufficient work and *O*verheads become important. For the runs with 20 cores per node (red line), we observe scaling up to seven nodes. Note that the Rostam cluster is a small test cluster with only 16 nodes. These three examples were chosen to show that enough work is needed to keep the cores busy and to avoid *S*tarvation of the resources. Even for this simple example, we can observe artifacts of the $\mathcal{SLOW}$ terminology in Fig. 5.1 on page 44 in Chap. 5. Figure 15.3 on page 177b shows the scaling on Stony Brook's Ookami Arm® A64FX™ cluster. The green line shows the scaling from 10 cores to 48 cores on each node. Here, we observe scaling up to 20 nodes. Beyond this, we do not have sufficient work compared to *O*verheads. Therefore, we stopped at 26 nodes. The red line shows the scaling where 5 cores per node were used. Here, we scale up to 29 nodes and start to flatten out. However, we could observe that we can scale to a larger number of nodes, since we have more work per node due to less threads

Listing 15.7 Main function of distributed fractal set with non-collective I/O

```
#include <hpx/hpx.hpp>
#include <hpx/hpx_main.hpp>
#include "distributed_fractal_worker_improved.hpp"
#include <fstream>
#include <chrono>

using std::chrono::high_resolution_clock;

---

int main() {
    auto locs = hpx::find_remote_localities();

    // The supervisor
    auto supervisor = hpx::find_here();

    // Compute the number of worker localities
    const int num_workers = locs.size();

    // Make sure we have some workers
    assert(num_workers > 0);

    // The logic below assumes more work than workers
    assert(num_workers < size_x && num_workers < size_y);

    // Measure the time only on the supervisor
    high_resolution_clock::time_point start_time =
        high_resolution_clock::now();

    std::vector<hpx::future<void>> data;
    // Load up workers...
    for(int j=0;j<num_workers;j++) {
        auto loc = locs.at(j);
        hpx::future<void> f = hpx::async<do_work_action>(loc,
            supervisor, next_work_item++, j);
        data.push_back(std::move(f));
    }

    for(int j=0;j<data.size();j++)
        data.at(j).wait();

    std::vector<hpx::future<void>> futures;

    hpx::when_all(futures).get();

    auto stop_time = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::
        nanoseconds>( stop_time - start_time );
    std::cout << "Time: " << duration.count() * 1e-9 << std::endl
        ;
}
```

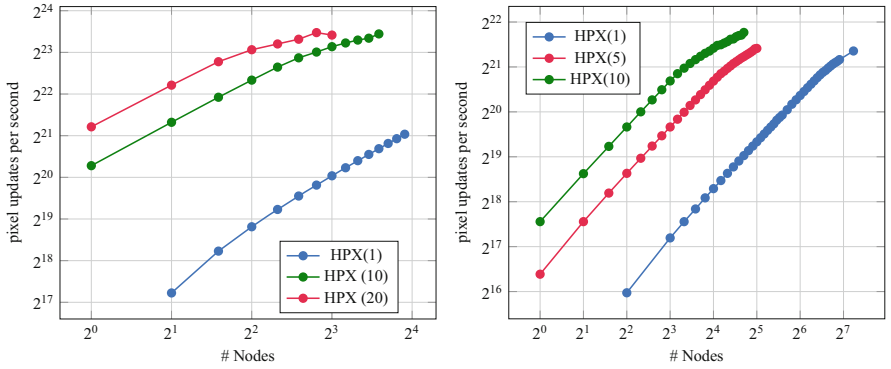
Listing 15.8 Actions for the fractal set with non-collective I/O

```
#include <config.hpp>
#include <kernel.hpp>
std::chrono::high_resolution_clock::time_point start, finish;
std::atomic<int> next_work_item(0);
std::atomic<int> completed_work_items(0);
size_t output = get_size_t("OUTPUT", 1);

---

// Called by workers to get an item of work from the supervisor
int get_work() { return next_work_item++; }
HPX_PLAIN_ACTION(get_work, get_work_action);

// The basic unit of work.
void do_work(const hpx::id_type& loc, int item_index, int id) {
  std::ofstream outfile("data_"+std::to_string(id)+".part");
  while (item_index < size_x) {
    // ask for the next work item asynchronously. That way
    // we overlap communication and computation.
    hpx::future<int> fitem = hpx::async<get_work_action>(loc);
    std::vector<int> result;

    complex c = complex(0, 4) * complex(item_index, 0) /
                complex(size_x, 0) - complex(0, 2);

    for (int i = 0; i < size_y; i++) {
      // Get the number of iterations
      int value = compute_pixel(c + 4.0 * i / size_y - 2.0);
      // Convert the smoothened value to RGB color space
      std::tuple<size_t, size_t, size_t> color = get_rgb(value);
      result.push_back(make_color(std::get<0>(color),
        std::get<1>(color),std::get<2>(color)));
    }

    outfile << item_index << " ";
    for(auto d : result)
      outfile << d << " ";
    outfile << "\n";
    item_index = fitem.get();
  }
  outfile.close();
}
HPX_PLAIN_ACTION(do_work, do_work_action);
```



(a) (b)

Fig. 15.2 Performance study for the fractal set using the distributed implementation on a single node. Figure (**a**) shows the pixel updates per second for an increasing number of cores on the Rostam cluster using Intel® Xeon® Gold 6148 Skylake CPUs. Figure (**b**) shows the speedup S with respect to the execution time on a single core. For each data point, the code was executed ten times and the median was plotted. The image size was $39,860 \times 21,600$ pixels. Note that we begin with two cores because that is the minimum possible for the supervisor-worker pattern



(a) Intel® Xeon® Gold 6148 Skylake (b) Arm® A64FX™ CPU

Fig. 15.3 Performance study for the fractal set using the distributed implementation on a single node. Figure (**a**) shows the pixel updates per second for an increasing number of nodes on the Rostam cluster using Intel® Xeon® Gold 6148 Skylake CPUs. Figure (**b**) shows the pixel updates per second for an increasing number of nodes on the Ookami cluster using Arm® A64FX™ CPUs

used on the node. The blue lines shows the scaling where a single core per node was used. Here, we started at 2 nodes because one was reserved for the supervisor. In this case, we show scaling data up to 150 nodes. Note that the Ookami cluster is one of the first clusters with Arm® A64FX™ CPUs in the United States and has 178 nodes in total. Here, we observe scaling up to 100 nodes and after that the curve flattens out.

Comparing the runs on the two different architectures, we observe that more pixels per second are processed on Intel® CPUs when using 10 cores per node. This aligns with our general experience that Arm® A64FX™ CPUs are slower. However, they consume less power. For more details we refer to [33] where runs on ORNL's Summit, CSCS's Piz Daint, NERSC's Perlmutter were compared with runs on Riken's Supercomputer Fugaku. Another challenge is to find the sweet spot, which means the optimal number of threads and nodes to obtain the best performance.

In conclusion, we have shown scaling for the distributed implementation of the fractal sets using the parcel port based on the Message Passing Interface (MPI). We showed scaling of the code on Intel® and Arm® A64FX™ GPUs on a small cluster and a medium size cluster. On both CPU architectures, the code scaled well for different numbers of cores per node.

Chapter 16
Some Remarks on MPI+OpenMP and HPX

After implementing the fractal set using the Message Passing Interface (MPI) in Sect. 13.5.1, *MPI+OpenMP* in Sect. 13.5.2, and HPX in Sect. 15.2, we are now at the point where we can compare the three different implementations.

Programming Languages and Interfaces
This book has a focus on C++, and according to a study from 2019, C++ is the dominant language for scientific open source HPC applications [146]. Similar results in other studies [1, 2] support this statement. However, MPI is still a C library and the authors are not aware of any effort to make an official C++ interface available. There is, however, *boost.mpi*[1] which provides a C++-friendly interface to MPI. Note that *boost.mpi* supports the major functionality of the MPI 1.1 standard. So not all the features available in Modern C++ can be used within Boost MPI. In addition, over 80% of the programs studied require the MPI 2 standard and do not use modern MPI features [146]. So, to serve the majority of codes, only a subset of functions would need to be ported to C++. Of course, one does not need a C++ interface to use C-based libraries such as MPI in C++ since C is (mostly) a subset of C++.

With the usage of OpenMP, the game becomes different and we use a `#pragma` -based programming language extension. On the plus side, these extensions are supported by a standard. Also, the pragmas, because they are pragmas, help to visually separate the parallel control logic from the underlying program logic. Yet another bonus to using OpenMP is that it can easily be used to add parallelism to a sequential code. Adapting a sequential code or MPI-only code to benefit from threads would require fairly unobtrusive changes with OpenMP, whereas adapting a codebase to use futures might require more extensive changes. For these reasons, this is probably the only threading model that makes sense for older codes.

[1] https://www.boost.org/doc/libs/1_80_0/doc/html/mpi.html.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_16

Studies have shown that ⅔ of the codes used *MPI+X* where OpenMP is dominant [147]. Soon, *X* might be replaced with CUDA or HIP given the increasing number of GPUs in the latest super computers in the Top 500. It should be noted that OpenMP supports GPUs and GPU programming to some extent. How successful it will ultimately be remains to be seen.

For most scientific applications, Message Passing is dominant. A survey within the US Exascale Computing Project (ECP) showed that 93% expect to use MPI in their application while running on exascale clusters. The remaining 7% plan to use Legion or remote procedure calls [147].

We do not expect this to change anytime soon. Even if HPX or some other API should become the dominant programming model, that new model will still likely use MPI as a reliable backend.

But the question naturally arises, will a new programming paradigm become popular for parallel computing?

Synchronous vs Asynchronous Computing
In general, there are two forms of parallelism that are typically in any parallel application, task-based parallelism (where we start tasks asynchronously and do other work while waiting for the result) and data-parallel (where we parallelize a loop operation). Inherent in the latter is a barrier where all the threads must stop and wait. Sometimes this barrier is essential, and sometimes the programmer includes it more or less by reflex even when it is not needed.

Data parallelism tends to dominate in codes that work on large arrays. However, the addition of `std::async` and `std::future` to the C++ standard make it possible to find parallelism in more ways. It means that, potentially, two data parallel operations can be executed at the same time.

It does us little good to have millions of cores if they're idle most of the time. Having good facilities for asynchronous, task-based parallelism (in addition to data-parallelism) can go a long way toward keeping the cores busy.

We note that many codes lose performance to blocking, to waiting on mutexes or futures. HPX has a unique context-switching ability that means threads waiting on blocking calls can still perform useful work. It also means that any HPX program can run on a single thread, regardless of how many asynchronous tasks it spawns. While running on one thread is not, in itself, especially useful, it means that your code will not be restricted to run on some minimum number of cores, and it means that what cores you do have access to will stay more busy.

One-Sided vs. Two-Sided Communication
One important paradigm for efficient and scalable high performance application, independent form the programming language, is overlapping communication and computation. This avoids *W*aiting and *S*tarvation, two sources of inefficiency. In general, HPX attempts to avoid this issue by trying to keep a surplus of work available to fill in the latencies created by waiting for data to transfer, and by providing the ability to context-switch away from spinning to do that work.

A relatively recent hardware innovation to help with achieving this goal is Remote Direct Memory Addressing (RDMA). While there are many HPC stories about how achieving this overlap with one-sided communication improved performance, and many using asynchronous messages in MPI (which has support for one-sided communication), we would like to share one related to HPX. In Sect. 18.1 we have shown that for the astrophysics application Octo-Tiger that asynchronous communication achieved by switching to HPX increased the code's performance by a factor of around 2.5 for larger node counts with respect to synchronous communication [44]. Note that HPX does not expose the ability to use RDMA directly to the user, but it can use it under the hood. In addition, as of this writing, an experimental parcel port using GASNet is available.

Active Messaging and Message Passing
The Active Message paradigm was invented in the early 1990s [41, 148]. It is a different approach in which messages are objects (or parcels in HPX) and perform processing on their own. Active messages are a light-weight messaging protocol with a focus on reduced $\mathcal{L}$atency.

Palumbo et al. in [148] describe the motivation for their work on Active Messages with the following words: "In traditional computer-based message systems, messages are typically viewed as passive entities with no processing power. The users are thus responsible for explicitly creating, routing and processing each message instance [148]."

Active Messaging is a more dynamic programming model, more data-driven, more asynchronous, less prone to result in a Bulk Sequential Parallel (BSP) code.

While Active Messaging has not become mainstream, it may be a revolution waiting to happen for newer codes. Because HPX is an exascale-capable option for making use of this paradigm, it will be a part of that revolution (should it come to pass), a part of the change that makes computing more dynamic and data-driven.

Distributed Asynchronous Programming
Distributed asynchronous programming, as we envision it, is a high-level programming API that combines asynchronous programming with futures, remote memory addressing, active messages, and serialization.

HPX is an effort to realize that dream within the C++ programming language and standardization process. There are many challenges to realizing this goal: (1) helping users navigate the unique challenges of debugging and performance-tuning a program whose execution is a data-dependency graph that spans multiple nodes, and (2) keeping up with the capabilities of modern hardware and pushing new performance boundaries.

But HPX has an international community of developers who are inspired by what it has accomplished so far and what it dreams to do.

Important

The most important lessons learned here are the following:

- HPX offers a high-level, object-oriented C++-API with a unified approach to distributed and local parallelism based on futures. It also has advanced features for serialization and active messages.
- MPI is a low-level, procedure-oriented C-API with an unparalleled level of standards support, and nearly unbeatable performance on most platforms.
- Because of the raw performance and universal installation of MPI, HPX is sometimes implemented on top of MPI.
- MPI is an API only, allowing multiple organizations to implement what it does in an optimal way for their communities or hardware. HPX, on the other hand, is a complete library, an API plus an implementation.

Part VI
A Showcase for a Portable High Performance Application Using HPX

In this part, we look into Octo-Tiger, an astrophysics program simulating the evolution of star systems based on the fast multipole method on adaptive octrees. Octo-Tiger is a real world application using the HPX features we introduced in this book. However, we neglected to cover how to use accelerator cards in combination with HPX. First. we give a brief introduction to accelerator cards in Chap. 17 and the state-of-the-art within HPX. Second, in Chap. 18, we briefly introduce the physics and solvers used in Octo-Tiger. From a computer science perspective, we consider performance issues. For example, speedups from one-sided vs. two-sided communication.

Chapter 17
Accelerator Cards

Another emerging topic is accelerator cards. These are specialized computation devices distinct from the CPU. In a sense, writing code to utilize these resources is a form of distributed programming, as data needs to be moved from the CPU to the device and the results copied back. There are various efforts to unify memory between these cards and main memory, but they are optional and not available on all accelerator cards.

Accelerator cards come in a variety of flavors. The most well-known are the graphics processing units (GPU). However, there are also Field Programmable Gate Arrays (FPGAs), or specialized devices for artificial intelligence. All of these accelerator cards have one thing in common: all of them are physical devices and have additional cores. Here, we focus on GPUs, since these are the most commonly used accelerators on super computers. Four out of the five fastest supercomputers in the Top500 November 2023 list[1] use GPUs. GPUs have been used for graphics processing for a long time. Their usage for general-purpose computing, however, only began around 2001 when they began to be called General-Purpose GPUs (or GPGPUs) [149]. In the beginning, OpenGL, a language which focused on graphics programming, was used to implement general linear algebra problems on GPUs. The computations ran faster on the GPU than on the CPU [150, 151] showing their potential. The algorithm, however, needed to be reformulated using the features from graphics programming.

The programming of GPUs was later simplified by NVIDIA®'s Compute Unified Device Architecture (CUDA™) introduced in 2007. CUDA™ has a C++-like API and some special language extensions for programming the compute kernels. For more details about CUDA™ programming, we refer to [152]. This started an NVIDIA-led revolution in computing. The last five super computers in the top ten of the Top 500 Nov 2023 list run on NVIDIA GPUs. A few alternatives to CUDA™ exist. There is a vendor-independent programming model called Open

[1] https://www.top500.org/lists/top500/2023/11/.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_17

Compute Language (OpenCL™), and Microsoft® has DirectCompute. In 2016 AMD® released ROCm their C++ Heterogeneous Interface for Portability (HIP). With HIP, programmers can write code for AMD® GPUs and NVIDIA GPUs. For example the fastest supercomputer, Frontier, and the fifth fastest, LUMI, in the Nov 2023 Top 500 list use AMD® GPUs. A new player in the game is Intel®. Kokkos [80] provides a unified API for programming CPUs and GPUs without writing vendor-dependent code for various GPUs. Kokkos supports the following backends: CUDA™, HIP, SYCL™, HPX, OpenMP, and C++ threads. Another option is SYCL™ developed by the Khronos® Group. SYCL™ is a single-source embedded domain-specific language (eDSL) using the C++ 17 standard to provide a higher level abstraction layer for various backends. The following two tools provide SYCL™ support. First there is ComputeCPP™ developed by codeplay®. Here, SYCL™ is translated to OpenCL™ or SPIR-V, the Intel® Graphics Compiler for OpenCL™. Second there is Intel ® DPC++ which uses LLVM/Clang. Here, the SYCL™ code is translated to NVIDIA® CUDA™, AMD® HIP, OpenCL or SPIR-V. In addition, there is support for Intel® Level Zero, the API for the upcoming Intel® CPUs is supported. For a comparative overview between Kokkos and SYCl™, we refer to [153].

Intel® DPC++ is part of Intel®'s oneAPI, a collection of APIs for various tasks across different accelerator cards. Intel® oneAPI includes a math kernel library oneMKL, formerly Intel® MKL, and former Intel® TBB for shared memory parallelism. In addition, libraries for data analytic and deep neural networks exist.

However, this book does not focus on GPGPU programming. Instead, we briefly describe the features provided by HPX at a high level.

HPX offers several opportunities for the integration of GPUs into the asynchronous task graph of HPX. First, the external library *hpxCL*, which integrates CUDA [154] and OpenCL. Here, asynchronous launches of data transfer to and from the GPU, and kernel launches are supported. Second, HPX provides executors [76], to asynchronously integrate Kokkos, CUDA, HIP, and OpenCL kernel launches and data transfer. Third, the tight integration of HPX within Kokkos. This integration is two-fold: *(1)* Kokkos has a HPX backend and *(2)* HPX can launch Kokkos functionality asynchronously. In the first case, Kokkos's parallel execution uses HPX's light-weight threads. In the second case, Kokkos functions are launched asynchronously and can be integrated into an HPX asynchronous task graph.

Finally, we describe Octo-Tiger, an astrophysics code which uses the above features in Chap. 18.

Important

The most important lessons learned here are the following:

- There are multiple GPU vendors: e.g. AMD® Intel® and not just NVIDIA®.
- There are different programming models: CUDA™, and HIP™.
- There are different Abstraction levels: Kokkos, and SYCL™.
- HPX provides ways to access them through *hpxCL*, its executors, and Kokkos.

Chapter 18
Octo-Tiger, a Showcase for a Portable High Performance Application

Up until now, we have discussed the parallelism provided by C++ and HPX using toy examples. Now we will look into Octo-Tiger, a real-world astrophysics code that simulates the evolution of star systems based on the fast multipole method on adaptive octrees.

Octo-Tiger is open source and available on GitHub®.[1] We briefly describe the astrophysics simulated by Octo-Tiger to provide a basic understanding of the numerical methods and data structures it employs. For more details, we refer to Ref. [60].

The adaptive mesh refinement (AMR) in Octo-Tiger resolves the atmosphere between the two stars, as well as the aggregation belt where mass is transferred between the stars. The adaptive octree uses a Cartesian sub-grid within each leaf. Octo-Tiger has 512 cells within a $8 \times 8 \times 8$ cube as the default for each sub-grid. However, the cube size can be configured during compile time. The performance benefits of various sub-grid sizes were studied in [32]. The star system is modeled as a self-gravitating, astrophysical fluid. A coupled solver for hydrodynamics and gravity was used. Finite volumes [155] are used to solve the inviscid Navier-Stokes equations. The Fast Multipole Method (FMM) [156] was used to solve for Newtonian gravity. For implementation details about the hydro kernel, we refer to [157]. Lastly, for a convergence study in a production run we refer to [45].

Octo-Tiger addresses performance portability using Kokkos and HPX. Figure 18.1 shows the relation of performance, portability, and productivity according to [158]. In the vertical direction there is *specialization* and *abstraction*. Previously, specialization meant that the code had one compute kernel for the CPU using OpenMP for parallelism and one CUDA™ kernel for NVIDIA® GPUs. Currently, with the rise of AMD and Intel GPUs in supercomputers, more kernels written in vendor-specific languages, like HIP for AMD® GPUs or SYCL™ for Intel® GPUs, are needed to make the code portable to all supercomputers.

[1] https://github.com/STEllAR-GROUP/octotiger.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_18

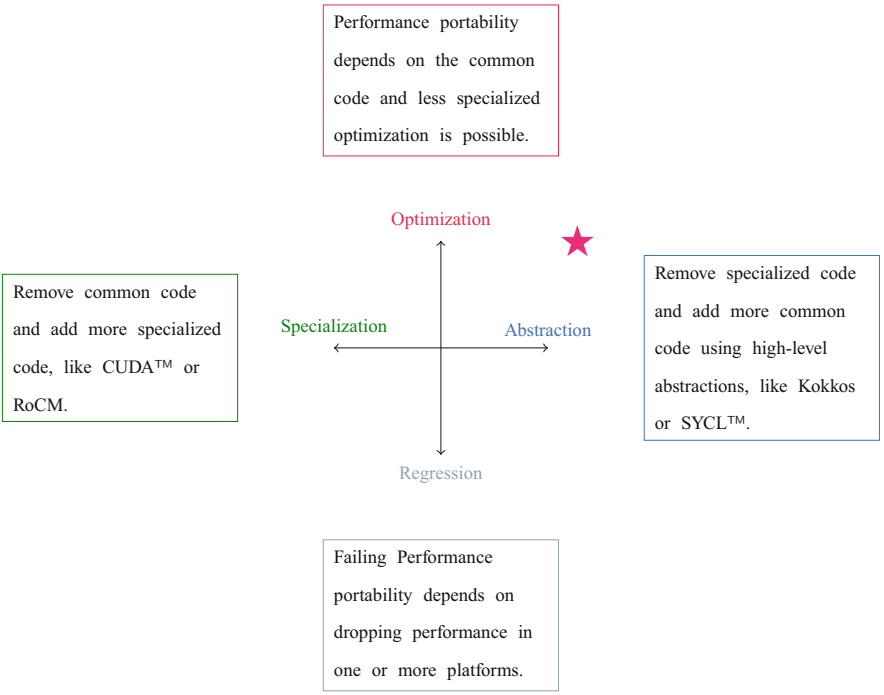


Fig. 18.1 Performance, portability, and productivity. Adapted from [158]

Even with only two versions of the compute kernel it was challenging to keep both versions the same while fixing bugs or adding features. Maintaining even more kernels is simply not scalable. With libraries like Kokkos and SYCL™ high-level abstractions allow the compute kernel to be written once in a generic, vendor-neutral language. Thus, using abstractions results in less-specialized, more-maintainable code.

In the horizontal direction there is *Optimization* and *Regression*. The star (★) in the upper left corner indicates where a code should be in an ideal world. The code should have only one version of each of its compute kernels written in a high-level language, and all the vendor-specific kernels should be highly optimized. We all know there is no free lunch and this scenario is unrealistic. Writing specialized code makes it possible to use special features of the vendor-specific language and write highly tailored code for the best performance on the specific GPU or CPU. Regression means that performance portability drops due to performance issues in one ore more platforms. Maybe new vendor-specific features are not yet provided by the abstraction layer or new hardware requires modifications of the abstraction layer.

Octo-Tiger has run successfully using HPX, Kokkos, and SYCL™, on ORNL's Summit [32], CSCS's Piz Daint [44], NERSC's Cori [159], ORNL's Frontier, and

Riken's Supercomputer Fugaku [33] on various CPU architectures and accelerator cards. In the next sections, we examine a few of the implementation details of Octo-Tiger and how C++ and HPX benefits them.

18.1 Synchronous Communication vs. Asynchronous Communication

In Sect. 13.5, Message Passing using the Message Passing Interface (MPI) was introduced. For distributed computing, HPX provides a parcel port (what HPX calls its backend communications layer) based on MPI using non-blocking, two-sided communication. It also uses he `MPI_T` to obtain a call-back driven interface for event notification. An overview of this functionality is given in [160, 161]. Finally, HPX's parcels are packed into MPI messages using remote memory access (RMA) to keep the overheads to a minimum [145].

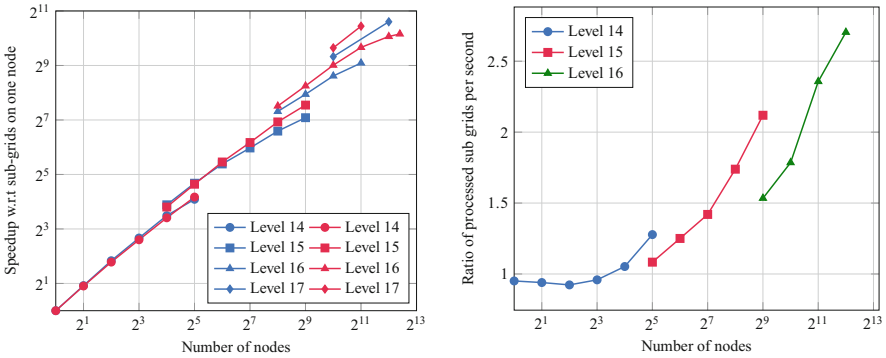
To reduce overheads due to the two-sided communication, HPX has another parcel port using the one-side communication provided by libfabric [118]. Libfabric provides an asynchronous API which blends perfectly with the asynchronicity of HPX's run time system. For more implementation details, we refer to [44]. This parcel port uses the libfabric API directly. However, we also have an LCI[2] parcel port which uses an MPI interface but calls the native libfabric API.

To show the benefit of one-sided communication over two-sided communication, we look at a benchmark on CSCS's Piz Daint [44]. Figure 18.2 shows a comparison of the MPI parcel port and libfabric parcel port on a full system run on Piz Daint using up to 62,000 CPU cores and 2400 NVIDIA® V100 GPUs. Figure 18.2a shows the speedup, S_n, with respect to a single compute node for the sub-grids processed per second for level 14. Figure 18.2b illustrates the ratio of the sub-grids processed per second using the MPI parcelport compared to the libfabric parcelport.

18.2 Acceleration Support

In Chap. 17 accelerator cards were introduced. In the beginning NVIDIA® GPUs were practically the only GPUs and, for example, the supercomputer Summit, which was hosted by Oak Ridge National Laboratory, was the fastest supercomputer in the Top 500 from June 2018 until November 2019. Summit has a combination of IBM® Power™ 9 CPUs and NVIDIA® V100 GPUs. Here, Octo-Tiger supported NVIDIA® GPUs using native CUDA™ calls wrapped into `hpx::futures` for the integration in HPX's asynchronous execution graph [32, 44]. Scaling results up to a

[2] https://github.com/uiuc-hpc/LC.



(a) Relative speedup S_n with respect to a single node for the processed sub-grids on level 14. The blue lines show the simulations using the MPI parcel port and the red lines show the simulations using the libfabric parcel port.

(b) Ratio of the processed sug-grids per second between the two parcel port implementations. Higher numbers indicate that libfabric could process more sub-grid per second.

Fig. 18.2 Comparison of the MPI parcel port and the libfabric parcel port on Piz Daint: (**a**) Relative speedup w.r.t to a single node and (**b**) ratio of the processed sub-grids per second. Adapted from [44]. © ACM 2019; reprinted with permission

full system run on CSCS's Piz Daint using 2400 NVIDIA® P100 GPUs and 62,000 CPU cores are shown here [44].

Summit's successor, Supercomputer Fugaku hosted by Riken in Japan, was the fastest supercomputer from June 2020 until November 2022. Before this, most supercomputers had a combination of CPUs and GPUs. Fugaku, however, introduced a novel Fujitsu® Arm® A64FX™ CPU based on the Arm architecture. Here, there were no GPUs and scientific codes that were heavily optimized for NVIDIA® GPUs faced porting issues. Fugaku was succeeded by Oak Ridge National Laboratory's Frontier in December 2022. Frontier uses a combination of AMD® CPUs and AMD® GPUs. Aurora, hosted by Argonne National Laboratory, will have a combination of Intel® CPUs and Intel® GPUs. With the proliferation of new accelerator cards, performance portability is a new challenge. To avoid writing individual compute kernels for AMD®, Intel, and NVIDIA® accelerator cards, abstraction layers like Kokkos [80] can be used. Kokkos provides abstractions for parallel execution and memory access. Kokkos provides a backend for HPX, and Kokkos compute kernels can be executed using HPX's light-weight threads. However, this does not integrate into HPX's asynchronous execution graph. Therefore, HPX provides an extension to wrap Kokkos function calls into `hpx::futures` [162]. Intel GPUs are programmed using SYCL, a single-source embedded domain-specific language (eDSL). Kokkos provides a backend for SYCL, which was tested using Octo-Tiger [163]. The HPX-Kokkos integration using Octo-Tiger was successfully tested on AMD® GPUs [164] or NVIDIA GPUs [32, 44, 164]; and Arm® A64FX [165] CPUs [33]. Figure 18.3 shows the scaling results on various

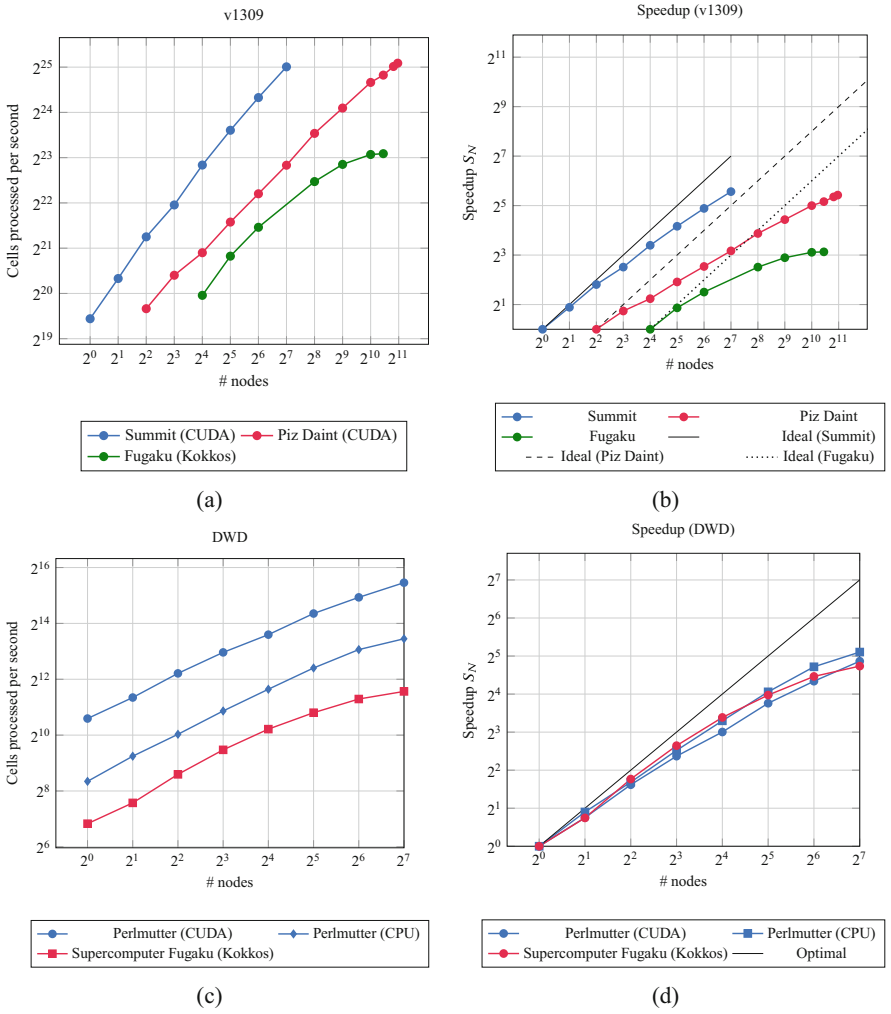


Fig. 18.3 Cells processed per second (**a**) on ORNL's Summit, CSCS's Piz Daint, and Riken's Supercomputer Fugaku; and (**c**) on NERSC's Perlmutter and Riken's Supercomputer Fugaku. The speedup S_N is shown in (**b**) and (**d**), respectively. Adapted from [33]

supercomputers. Figure 18.3a shows the cells processed per second for ORNL's Summit, CSCS's Piz Daint, and Riken's Supercomputer Fugaku; and Fig. 18.3b the speedup, S_N, with respect to the smallest number of nodes the simulation can use. Figure 18.3c shows the cells processed per second on NERSC's Perlmutter and Riken's Supercomputer Fugaku; and Fig. 18.3d shows the speedup, S_N, with respect to a single node. For more details, we refer to [33]. These runs provide a good example of the benefit of the abstraction layers provided by HPX and HPX-

Kokkos. They enable programmers to write maintainable, parallel, efficient, and scalable C++ code for various supercomputers.

18.3 HPX-Kokkos and Vectorization

After discussing the support of various GPU architectures, the support of vectorization using explicit SIMD types is important to address. With Kokkos it is possible to write SIMD code for vectorization. However, on Arm® A64FX™ CPUs, we found that some functionality was missing and could not use SIMD there. Here, we added `std::experimental:simd` provided by the C++ standard as a second option for explicit vectorization. To target Arm® A64FX™, we added support for the Scalable Vector Extension (SVE). Figure 18.4 shows our execution model for launching compute kernels in Octo-Tiger for CPU execution and GPU execution. On the left, we have our HPX application: Octo-Tiger. Between Kokkos and the HPX application, we introduced a middle layer HPX-Kokkos. This middle layer allows us to call Kokkos functions asynchronously and we receive an `hpx::future` which we can use for synchronization with other tasks. HPX-Kokkos internally calls the Kokkos API and, depending on the chosen execution space, does the appropriate thing.

On the CPU, Kokkos' HPX Execution Space is used. Here, the Task 1 up to Task *N* are asynchronously executed on the CPU. Within the execution space, we

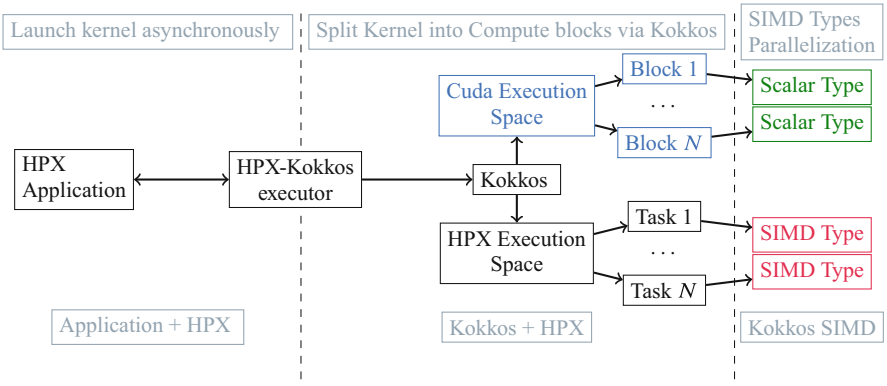


Fig. 18.4 Overview of the integration of HPX and Kokkos. The application developer uses HPX within the application to call computer kernel launches asynchronously using the HPX-Kokkos executor. Depending on the execution space, the kernel is executed on the CPU or GPU. On the top, we show the CUDA™ Execution Space where the kernel is split into blocks which are executed on the NVIDIA® GPU. Note that AMD® and Intel® GPUs are supported too, but for simplicity, we show only NVIDIA® GPUs via CUDA™. On the bottom, the HPX Execution Space is used to split the compute kernel into tasks and use HPX's light-weight threads. The last step is Kokkos SIMD. Adapted from [165]

use Kokkos SIMD to choose the SIMD Type depending on the CPU architecture. This happens automatically, and the application developer does mot need to select the SIMD type, since the SIMD type is set in Kokkos during the build process using CMake. For GPUs, we use Kokkos' CUDA™ Execution Space. Note that for simplicity, we run on CUDA™ GPUs using Kokkos CUDA™, but we could also use Kokkos SYCL™/CUDA™ as well. However, Kokkos and HPX-Kokkos allow for AMD® GPUs using Kokkos HIP or Kokkos SYCL™/HIP™ and for Intel® GPU's using Kokkos SYCL™. Block 1 up to Block N are executed asynchronously on the NVIDIA GPU using the default scalar type, e.g. `double` or `float`. For more details about HPX-Kokkos, we refer to [162]; and for more details about the vectorization, we refer to [165].

Part VII
Conclusion and Outlook

Finally, we conclude on the opportunities the C++ 17 and C++ 20 standard provides for writing shared memory parallel code using pure C++. For the distributed memory parallelism, we look into the C++ standard library for parallelism and concurrency (HPX) which adds distributed features by providing a unified API for local and remote calls. We provide an outlook to the upcoming C++ 23 standard and the features relevant for shared memory parallelism. We do some crystal ball gazing on the future C++ 26 standard and showcase some features which might get accepted and will benefit writing parallel, efficient, and scalable high performance applications.

Chapter 19
Conclusion and Outlook

We hope that this book has provided a useful overview of parallel programming using the C++ 17 and C++ 20 standards, and how using HPX can help you get the most out of these standards.

Several points that we tried to emphasize during the course of this book are as follows:

- Parallel programming is hard: Deadlocks and race conditions can be challenging problems to solve. Even when correct code is achieved, performance may be elusive for a variety of complex reasons. The reason that there are so many different tools for parallel programming is because the task of parallel programming has proven so difficult.
- The importance of widely recognized standards: Without these, it is difficult to write portable code. For distributed programming, the strongest standards are MPI and OpenMP. Because of this, HPX relies on MPI in many cases.
- The importance of the C++ standard: While it is possible to use a C-language library from C++, it misses the benefit of many useful language features. The strongest candidates for distributed programming with respect to C++ language support are Boost.MPI and HPX. We feel that HPX offers an advantage here because it leverages the more natural concept of `future`s for asynchronous programming.
- The importance of writing code at a higher level: Because parallel programming is such a complex and error-prone task, you are more likely to get correct and performant code from the parallel algorithms than if you wrote similar code on your own. Even for serial code involving loops this is often the case.
- Avoidance of blocking: Blocking code can rob you of performance (or even deadlock it). HPX is usually better than the standard library at performing context switches and keeping the threads busy.

As a sort of roadmap to what you have learned, we now provide a brief summary of the various topics we've covered.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2_19

C++ 11	C++ 14	C++ 17	C++ 20
`std::thread`	Generic lambda	Parallel	Coroutines
`std::async`	shared mutex	algorithms	Ranges
Smart pointer			Semaphores
Lambda functions			Latch
			Barrier

Fig. 19.1 Parallel features added to the C++ standard beginning with `std::thread` and `std::async` with the C++ 11 standard, and continuing with the more recent features provided by the later C++ standards. Adapted from [59]

With the C++ 11 standard, `std::thread` and `std::async` were added to allow parallel programming without directly exposing the *pthreads* library. Low-level thread programming was discussed in Chap. 8 and asynchronous programming using `std::future` and `std::async` in Chap. 9. Asynchronous programming provides a more natural way to express parallelism, one that abstracts the application code from the number of threads the machine actually supports.

With the C++ 17 standard parallel algorithms were added. Here, most of the algorithms from the C++ 98 standard, e.g. `std::reduce` or `std::sort`, were parallelized. The way these algorithms divide their work between threads can be controlled through execution policies.

These algorithms act on containers provided by the C++ Standard Library (SL). We discussed the C++ SL in Chap. 3 and the parallel algorithms in Chap. 10.

With the C++ 20 standard coroutines and ranges were added. We discuss coroutines in Chap. 11 and ranges in Appendix A.6. Figure 19.1 provides an overview of the timeline of the current available parallel features in the C++ standard from `std::thread` and `std::async` with the C++ 11 standard to the latest features in the C++ 20 standard.

We introduced the Mandelbrot set and Julia set in Chap. 4 and provided an example serial implementation using the C++ Standard Library. The serial code was later extended for parallel computing using low-level threads, asynchronous programming, and coroutines.

19.1 Distributed Programming

After focusing on shared memory parallelism, the second part of the book focused on distributed memory parallelism. No features for distributed computing are provided in the current C++ standard. To the best of our knowledge, there is no discussion within the C++ standardization committee to target distributed computing. However, many new features for parallel computing will be added. We will discuss these features in the outlook below. The C++ standard library for

parallelism and concurrency (HPX) implements the parallel features of the C++ standard, but extends these for distributed programming. While these features are not in the C++ standard, a C++ compiler supporting the C++ 17 standard or (for some features) the C++ 20 standard is sufficient.

In Chap. 5 we discussed what makes a system *SLOW* and how asynchronous many-task run time systems, like HPX, can address some of these concerns. In Chap. 6 HPX's architecture and some applications, libraries, and tools using HPX were introduced. Here, we focused on the concepts and components of HPX. The implementation details for parallelism are discussed in Chaps. 8–11. Since HPX strictly follows the C++ standard only namespaces and headers are different.

Next, we looked at distributed programming in Chap. 13. We described the concepts of data distribution, load balancing, and distributed input and output. For sending data over the network, we discussed the Message Passing Interface (MPI) as well as the concept of serialization. After this, we provided examples of HPX's distributed features in Chap. 14, namely active messaging to send and receive **Par**allel **C**ontrol **E**lements (parcels) within HPX's Active Global Address Space (AGAS) using components and actions.

In Chap. 15 we implemented distributed versions of the Taylor series and fractal set codes, including serialization. The fractal set was executed on 16 Intel® CPU-based cores and 150 Arm® A64FX™-based cores for performance measurements. In Chap. 16 we provide some remarks on message passing using MPI and MPI+OpenMP and active messaging using HPX.

In the last part of the book, we briefly described the use of accelerator cards and their integration with HPX. See Chap. 17. We used Octo-Tiger, a real-world astrophysics application for stellar mergers, as a showcase for the usage of HPX and GPUs. Note we could only provide an overview of the technical concepts, since implementation details would require an entire book.

In the context of discussing Octo-Tiger, we also provided a high-level description of "performance" portability using HPX-Kokkos on NVIDIA® and AMD® GPUs and Arm® A64FX™ CPUs.

In the appendix, we provided a recap of advanced C++ topics that we used throughout the book, e.g. generic programming, lambda functions, move semantics, smart pointers, etc. We briefly described the new ranges library. The ranges library is part of the C++ 20 standard, but not fully supported by all compilers yet.

We hope reading this book has helped the reader to obtain a better understanding of parallel C++ and scalable high-performance computing. However, as described by Ericson et al. in "The Role of Deliberate Practice in the Acquisition of Expert Performance" [166] practice is needed to become an expert. While this study applied to violin students, the conclusion—that intense practice as well as ability were important—is no less true for programming than for music. Training is essential for improving programming skills. We recommend that you use the examples provided in the book and play with or extend them. All examples are available on GitHub®[1]

[1] https://github.com/ModernCPPBook/Examples.

and instructions for compiling and running the code are available in Chap. 1. Please feel free to open GitHub® issues[2] if you experience any problems with the examples or if you think you have found a bug.

19.2 Outlook

Many new and exciting features have appeared in the C++ 23 and will appear in the upcoming C++ 26 standard. We have only scratched the surface of what's possible using the ranges library introduced in the C++ 20 standard (See Appendix A.6). Once the ranges library is more widely adopted by C++ compiler vendors and extended to cover C++ 23, we expect to make use of them in future revisions of this text.

Two additions that we plan to cover in the next edition are `std::views::cartesian_product`,[3] which makes tuples from multiple containers; and (`std::mdspan`[4]), an abstraction for multi-dimensional data. Mdspan allows programmers to access multi-dimensional data structures from numerical algebra libraries, like Eigen, Portable, Extensible Toolkit for Scientific Computation (PETSc) [167], Blaze [71], or Boost. A C++ Standard dense linear algebra interface based on the Basic Linear Algebra Subroutines (BLAS) based on mdspans is proposed in ISO P1673.[5] Here, several backends for shared memory are implemented. Currently, an HPX backend[6] is under development.

The C++ 26 standard is still in its early stages, but some promising features with relevance for the book might be added to it. One big change might be the addition of schedulers, senders, and receivers (ISO P0443)[7] which provide a new interface for asynchronous programming and might deprecate `std::async`. However, `hpx::async` will remain in HPX. HPX will also continue to follow the future C++ standards and implement new parallel programming features ahead of the standard compilers. We hope that the C++ standardization committee starts the discussion about distributed programming within the C++ standard soon, but we will have to wait and see. Figure 19.2 summarizes the features we have described.

Another interesting aspect to consider is upcoming hardware trends. First, an Intel® GPU might be available soon as a third type of GPU. Here, SYCL might emerge for GPU programming to address Intel GPUs. From the CPU perspective the RISC-V architecture is interesting. RISC-V is an open standard instruction set architecture (ISA) using reduced instruction set computer (RISC) features.

[2] https://github.com/ModernCPPBook/Examples/issues.

[3] https://en.cppreference.com/w/cpp/ranges/cartesian_product_view.

[4] https://en.cppreference.com/w/cpp/container/mdspan.

[5] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p1673r12.html.

[6] https://github.com/STEllAR-GROUP/stdBLAS/tree/hpx_backend.

[7] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r13.html.

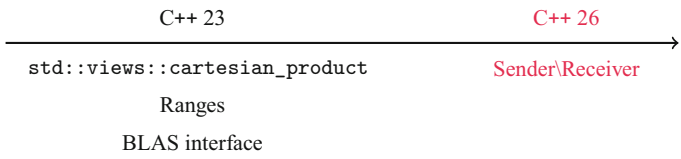


Fig. 19.2 Parallel features in the recent C++ 23 standard, namely `std::views::cartesian_product`, ranges, and a BLAS interface. These features, however, are not available in many compilers as yet. In red are senders and receivers, which could be interesting for parallelism in C++ if the standardization committee approves them. They might appear in the C++ 26 standard. Figure taken from Diehl, P., Brandt, S.R., Kaiser, H. (2023). Shared Memory Parallelism in Modern C++ and HPX. In: Diehl, P., Thoman, P., Kaiser, H., Kale, L. (eds) Asynchronous Many-Task Systems and Applications. WAMTA 2023. Lecture Notes in Computer Science, vol 13861. Springer, Cham.

One party interested in RISC-V is the European Processor Initiative (EPI)[8] which supports European independence in high performance computing. Their vision is a European Exascale machine based on the RISC-V processor. The HPX group has started exploring the usage of HPX on a RISC-V single board computer (SBC) already [168]. Note that while we do not yet have access to HPC-grade hardware for RISC-V, we have been able to use what's available for porting applications to the new architecture. Another current trend is HPC in the Cloud. A prominent example is Microsoft Azure's Eagle supercomputer, a cloud service machine that is number three in the Nov 2023 Top 500 list. Currently, we are investigating the overheads for HPX-Kokkos and Octo-Tiger running within containers such as Docker or Singularity.

This is an exciting time for supercomputing, for parallel programming, and for HPX. We are in a time where enormous progress has been made in tackling large scale programming problems, but also a time in which much remains to be discovered. We hope that this book has helped you understand this landscape as well as to get you started in exploring it. We hope that you will come away with new energy and insight into tackling the performance barriers of your own codes.

Thank you, for reading. Please let us know if you have any thoughts, criticisms, or insights that would make future editions of this book more valuable.

[8] https://www.european-processor-initiative.eu/.

Appendix

laissez le bon temps rouler (let the good times roll)

This book's appendix consists of three parts: advanced topics in C++, supplementary header files, and software and hardware documentation.

Appendix A, "Advanced Topics in C++" briefly summarizes some of the advanced C++ concepts, e.g. generic programming, lambda functions, smart pointers, move semantics, placement new, and ranges. We would like to encourage the reader to familiarize themselves with or refresh their knowledge of these concepts before reading the main part of the book.

Appendix B, "Supplementary Header Files" lists the supplementary header files used throughout the book, e.g. for the fractal set, the pbm files, etc. Note that the PBM generating code is not really important for the subject matter of the book, however, for completeness we provide all header and source files in this appendix to the interested reader.

Appendix C, "Software and Hardware Documentation" provides details about the hardware and software used for the performance measurements in the book. We think reproducible science is a cornerstone of academia. Therefore, we have provided as much information as possible so that a passionate reader can reproduce all plots in the book.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2

Appendix A
Advanced Topics in C++

In this appendix, we briefly summarize some of the advanced C++ concepts, e.g. generic programming, lambda functions, smart pointers, move semantics, placement new, and ranges; which are used in this book for parallel programming using C++ STL and HPX. All the examples are available on the book's GitHub® repo[1] as C++ source files or Jupyter notebooks, respectively. This book will not provide a basic introduction to C++. Instead, it assumes the reader has a basic knowledge of the C++ language and some experience programming in it. From one of the author's teaching experiences, the following books are good sources to learn the basics of C++:

- A. Koenig, *Accelerated C++: practical programming by example* (Pearson Education India, 2000)
- B. Stroustrup, *Programming: principles and practice using C++* (Pearson Education, 2014)
- B. Stroustrup, *A Tour of C++* (Addison-Wesley Professional, 2022)

This book uses the parallel algorithms[2] introduced in the C++ 17 standard (See Chap. 10). However, there are many more features in the C++ 17 standard which do not directly relate to the material in this book. For advanced reading about the C++ 17 Standard, we recommend the following books:

- A. O'dwyer, *Mastering the C++ 17 STL: Make full use of the standard library components in C++ 17* (Packt Publishing Ltd, 2017)
- N.M. Josuttis, *C++ 17: The Complete Guide* (Nicolai Josuttis, 2019)

Since they are in the C++ 20 standard, this book makes use of ranges since these simplify the usage of the parallel algorithms. For advanced reading about the C++ 20 Standard, we recommend the following book:

- N.M. Josuttis, *C++ 20: The Complete Guide* (Nicolai Josuttis, 2022)

[1] https://github.com/ModernCPPBook/Examples.

[2] https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2

A.1 Generic Programming

The term generic programming was coined in 1988 in [9] to describe algorithms which work the same way on different types of data. This means that one does not need to implement the same functionality for various data types, e.g. `double`, `std::string`, `float`, etc. Listing A.1 on page 206 shows one example of generic programming using C++ templates. For example in Line 7 a function `add` is defined to add two double values. In Line 11 the same functionality is implemented for the addition of two `std::string` values. In that case, we had to write redundant code just because we were using different data types.

Listing A.1 Example for the usage function templates

```
1  #include <cstdlib>
2  #include <string>
3  #include <iostream>
4  #include <complex>
5  
6  // Definition of multiple functions
7  double add(double a, double b) {
8    return a + b;
9  }
10 
11 std::string add(std::string a, std::string b) {
12   return a + b;
13 }
14 
15 std::complex<int> add(std::complex<int> a, std::complex<int> b) {
16   return a + b;
17 }
18 
19 // Function template
20 template<typename T>
21 T add_template(T a, T b){
22   return a+b;
23 }
24 
25 // Usage of the functions
26 std::cout << add(1.0, 1.5) << std::endl;
27 std::cout << add("hello ", "world") << std::endl;
28 std::cout << add(std::complex<int>(1,0),
29     std::complex<int>(0,1)) << std::endl;
30 std::cout << "---------" << std::endl;
31 std::cout << add_template<float>(1.0, 1.5) << std::endl;
32 std::cout << add_template(
33     std::complex<double>(1.1,0),
34     std::complex<double>(.1,2.3)) << std::endl;
```

In C++ the concept of generic programming is provided by the usage of the language feature `template<>`[3] which was introduced in the C++ 11 standard.

What we want is an algorithm for addition (which we name `add_template`) which works on any object that defines `operator+()`. In Line 21 the generic implementation of the function `add_template` is shown. Above the function definition, the language feature `template<>` is added (See Line 20). Inside the angle brackets the template parameter, `typename T`, is defined. Note that one or more template parameters can be specified. In this example, the data types `float` and `std::complex<double>` are replaced by the placeholder `T` such that there is one generic implementation of the function `add` independent of the data types. Now, it is possible to call the function with a specific data type, e.g. `add<double>`, or `add<std::string>`, or `add<std::complex<int>>`, respectively. Generic programming is heavily used in the C++ Standard Template library, see Chap. 3. For more details, we refer to [174].

A.2 Lambda Functions

Lambda functions[4] were introduced in the C++ 11 standard. However, with recent standards new features were added. Lambda functions are used to specify an unnamed function object capable of capturing variables in the scope. Listing A.2 on page 208 shows the syntax of the lambda function. The lambda function is introduced using square brackets: `[ ]`. Inside the parentheses, the lambda captures are defined. The following two capture defaults are available:

- `[&]` – Implicit capture and all variables are passed by reference
- `[=]` – Implicit capture and all variables are passed by copy

Note that if you specify only one of these, all variables are passed by reference or copy. If you want to have a mix of pass by reference or copy, you need to provide the information for all of the variables within a comma separated list. The function parameters, as in a regular named function, are given between parenthesis: `()`. In most cases the compiler deduces the return type of the function, however, in some rare cases there are some issues and the return type needs to be specified. This is done by using the arrow operator, `->`, followed by the return type, e.g. `-> int` for returning an integer. As with a regular named function, the body of the function is given between the curly braces.

[3] https://en.cppreference.com/w/cpp/language/templates.

[4] https://en.cppreference.com/w/cpp/language/lambda.

Listing A.2 Syntax of the lambda function

```
[capture_clause](parameters) -> return_type {
  // Definition of the function
}
```

Let us look at one example for the usage of unnamed functions. Listing A.3 on page 208 shows an example to print the values of a vector defined in Line 7. For more details about the `std::vector`,[5] we refer to Sect. 3.2.1. In Line 9 we see the function `print`, which takes one integer as its argument and prints the integer. Now, we use the C++ Standard Template Library `std::for_each`[6] to iterate over the elements of a container, in that case a vector. This will print all the elements line by line by calling the function `print(int value)` and passing each element to that function (See Line 15). However, we had to define a function with the name `print`, which is a utility function and not necessary since we might call this function only once. Another option it to use a lambda function (See Line 20). Here, we use square brackets, `[]`, to introduce the lambda function, since we do not provide any argument we use the default capture, *i.e.* nothing. In the parentheses `(int value)`, we provide the function argument as before. Within the curly braces, we provide the function definition as before. Using lambda functions is quite common within the algorithms provided by the C++ STL (See Sect. 3.3).

Listing A.3 Printing the values of a `std::vector` using `std::for_each` and a lambda function

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

// Vector to be printed
std::vector<int> values = {1, 2, 3, 4, 5};

// Define a named function
void print(int value) {
  std::cout << value << std::endl;
}

// Print each element of the vector using a named function
std::for_each(values.begin(), values.end(), print);

// Print each element of the vector using a unnamed function
std::for_each(values.begin(), values.end(), [](int value) {
  std::cout << value << std::endl;
});
```

[5] https://en.cppreference.com/w/cpp/container/vector.

[6] https://en.cppreference.com/w/cpp/algorithm/for_each.

Listing A.4 Printing the values of a `std::vector` using `std::for_each` and a lambda function. Here, no pre-leading comma symbol is printed

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

int main() {
  // Vector to be printed
  std::vector<int> v = {1,2,3,4,5};

  // Print the first element without a leading commas symbol
  std::for_each(v.begin(),v.end(),
    [init = true] (int x) mutable {
     if (init) { std::cout << x; init = false; }
     else { std::cout << ',' << x; }
  });
}
```

A common example is to print all values of a container, like a `std::vector`, and separate the items with a comma symbol. However, we only want commas between the values and not at the beginning or end. Listing A.4 on page 209 shows a convenient way to do so using a lambda function. We use the function `std::for_each` to iterate over all items of the vector `v` as in the previous example. However, this time we use a slightly different lambda function. First, we add a boolean value `init` and initialize the value to `true` in the capture clause. Second, we add `mutable`[7] to the lambda function. The `mutable` declarator removes the `const` qualifier from the parameter `bool init` which allows us to change the boolean value within the body of the lambda function. For the first element, the value is printed and the boolean value `init` is set to `false` and from now on a comma symbol is printed before the value. The output of the program is now `1,2,3,4,5`.

A.3 Move Semantics

Move semantics were introduced in C++ 11 around ten years ago. However, according to recent surveys among C++ developers, move semantics are considered to be a complex topic. The function `std::move`[8] provided by the header

[7] https://en.cppreference.com/w/cpp/language/lambda.

[8] https://en.cppreference.com/w/cpp/utility/move.

Listing A.5 Example for move semantics

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4
5  std::string name = "Parallel C++";
6  std::vector<std::string> values;
7
8  // Copy the object name to the end of the vector
9  values.push_back(name);
10
11 std::cout << "The content of name after copying is " << name <<
       std::endl;
12
13 // Move the object name to the end of the vector
14 values.push_back(std::move(name));
15
16 std::cout << "The content of name after moving is " << name <<
       std::endl;
```

`#include <utility>` is used to move the ownership of an object using efficient resources to another object. Listing A.5 on page 210 shows an example for the usage of data movement. In Line 5 we have the `std::string` with the content `"Parallel C++"`. In the next line, we define an empty vector containing strings. For more details about the vector container, we refer to Sect. 3.2.1. In Line 9 the string `name` is appended at the end of the vector. Note that we copy the object `name` and append its copy to the end of the vector. Thus, in Line 11 we will see the following: `"The content of name after copying is Parallel C++"`. In some cases, this behavior is wanted, since a copy should be passed and we still need the object `name`. In Line 14 the object `name` is added again to the end of the vector, but here the move semantics `std::move` is used. Note that here the object is moved to the vector and no object is copied. After the code on Line 16 is executed, we will see the following: `"The content of name after copying is"`. Note here the object `name` is an empty string, since we moved the data to the vector and did not copied it. Depending on the application we either want to copy or move the object to the vector.

A.4 Placement New

The `new`[9] expression creates and initializes storage for objects. In Line 2 of Listing A.6 on page 211, the expression `new` is use to generate the array `values` with space in the memory for five integer values. Note that in this case,

[9] https://en.cppreference.com/w/cpp/language/new#Placement_new.

Listing A.6 Example for the expression new and expression delete

```
1  // Usage of the new expression to generate an array of integers
2  int *values = new int[5];
3
4  // Usage of the placement new for struct and classes
5  struct point {
6    double x,y,z;
7
8    point (double x0=0, double y0=0, double z0=0) {
9       x = x0; y = y0; z = z0;
10   }
11 };
12
13 point *vec = new point();
14
15 // Releasing the allocate double array
16 delete[] values;
17
18 // Releasing the allocated struct
19 delete vec;
```

the user does not have to allocate the memory. Here, the `operator new[] (sizeof(integer)*5 + overhead)` is called to allocate the memory. In Line 5 the `struct point` for the coordinates to a point in three dimensional space is defined. In Line 13 a new point object is instantiated by using the expression `new`. Here, the default constructor `point()` is called and the memory is allocated. Understanding this helps to understand the implementation and usage of the `std::vector` in Sect. 3.2.1.

The memory allocated with the `new` operator can be released using the `delete` operator.[10] In Line 16 the allocated array of double `values` is released by using `delete[]`. Note that the square brackets after delete indicate that we want to release an array instead of a single value. This means calling the relevant destructor for each element. In Line 19 the allocated struct is released by using the `delete`.

A.5 Smart Pointers

Pointers are a controversial topic between C and C++ users. C programmers use pointers heavily and prefer their use to object-oriented programming. Most C++ users avoid them if possible, since they are frequently a source of errors.

[10] https://en.cppreference.com/w/cpp/language/delete.

Pointers are an important feature for the following situations:

- Allocating new objects from the heap,
- Passing functions to other functions,
- Iterating over data structures.

In the C programming language, raw pointers are used for all these tasks. However, many features were added in C++ in order to avoid using raw pointers. New C++ programmers are not the only people who have issues with using raw pointers, it turns out that even experienced programmers do.

For passing functions to other functions, lambda functions can be used instead of function pointers (See Appendix A.2). For iterating over data structures, e.g. vectors, arrays, or lists, iterators can be used (See Sect. 3.2.5). Starting with the C++ 20 standard, ranges can be used as an extension of iterators (See Appendix A.6). In this section, we introduce smart pointers which should be used instead of raw pointers to allocate new objects from the heap. This book avoids the usage of raw pointers for all shared memory parallelism, and only uses smart pointers sparingly.

First, the `std::unique_ptr`[11] provided by the header `#include <memory>`. The unique pointer provides a unique reference to an object. Listing A.7 on page 212 shows the usage of the `std::unique_ptr<double[]>` for an array of double values. Instead of using the pointer `double*` directly, we pass the argument `double[]` as the template parameter to the `std::unique_ptr<double[]>`. To allocate the memory to store the `double` values, we use the placement new `new double[2]` (See Appendix A.4). Note that if we wanted to use a raw pointer, we would use `double* a = new double[2];`. Now, we can use the smart pointer `a` as if it were a regular pointer. For example, we can assign values to the array using

Listing A.7 Usage of the unique pointer to allocate an array of double values

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

[11] https://en.cppreference.com/w/cpp/memory/unique_ptr.

Listing A.8 Usage of the shared pointer to allocate an array of double values

```
1  #include <iostream>
2  #include <memory>
3  #include <array>
4
5  // Allocate a shared pointer to a double array
6  std::shared_ptr<double[]> a (new double[2]);
7
8  // Initialize the value
9  a[0] = 1;
10 a[1] = 2;
11
12 // Create a copy of the reference to array a
13 std::shared_ptr<double[]> b(a);
14
15 // Print the total references to the array
16 std::cout << a.use_count() << std::endl;
17
18 // Release the smart pointer b
19 b.reset();
20
21 // Print the total references to the array
22 std::cout << a.use_count() << std::endl;
23
24 // Create a copy of the reference to array a
25 //std :: unique_ptr < double [] > b ( a ) ;
```

the subscript operator `[]` (See Line 6). However, the C++ compiler will ensure that we make no copies of references to this array.

Let us have a look at Line 10 which is commented out on purpose to avoid a compilation error. Copying the smart pointer `a` to the new smart pointer `b` (if it were allowed) would result in two non-unique references to the memory. In Line 13, we show how to fix that issued by using move semantics to move the ownership array instead of making a copy of it. This is implemented by using `std::move(a)`. Now only the smart pointer `b` points to the memory where the array is allocated. For more details about move semantics see Appendix A.3.

Second, there is the `std::shared_ptr`.[12] With this type of smart pointer, many pointers can point to the same memory location, but the `shared_ptr` will keep track of how many there are and only deallocate the memory when the last reference goes out of scope. This can be handy, for example, if you want to have several threads work with the same memory, but don't know which one will be the last to need access. Listing A.8 on page 213 shows the usage of shared pointers. As in the unique pointer example, a shared pointer to a `double` array is allocated (See Line 5). In Line 13 the shared pointer `a` is copied to a shared pointer `b`. Now both

[12] https://en.cppreference.com/w/cpp/memory/shared_ptr.

shared pointers point to the same location in memory. In Line 16 we use the function `use_count()` to access the count of all references pointing to the same location in memory. In this example, the number 2 will be printed since two shared pointers point to the same memory location. In Line 19, the `reset()` function is used to reset the shared pointer `b` to the `null` pointer. Now, in Line 22, the number 1 will be printed since only one shared pointer, `a`, points to the memory location. The Line 25 is commented out on purpose, since this would result in a compilation error. Here, two pointers a shared pointer and a unique pointer would point to the same memory location.

A.6 Ranges

With C++ 20 the ranges library[13] provided by the header `#include <ranges>` was added to the C++ Standard Template Library. It can be seen as the generalization and extension of the algorithms (See Sect. 3.3) and the iterator library (See Sect. 3.2.5), respectively. Let us look at one basic C++ example to find all even elements in a vector and compute their square roots (See Listing A.9 on page 214). Here, we use the if clause to find the even entries and for all even elements we compute the square root of each element and print them all separated by spaces. With the ranges library, the same code can be written more elegantly. Let us have a look at Listing A.10 on page 215. In Line 9 the `for` loop is extended using `std::views`. First, the `if` clause in the previous example is replaced by `std::views::filter`.[14] The view is a range adapter in our case on the vector `values` and gives an underlying sequence of the vector containing all elements by applying the lambda function to each element. This unnamed function has one function

Listing A.9 Example for finding all even values and compute their square root

```
1  #include <cmath>
2  #include <iostream>
3  #include <vector>
4
5  std::vector<int> values = {0, 1, 2, 3, 4, 5, 6};
6
7  for (int i : values) {
8    if (i % 2 == 0) {
9      std::cout << std::sqrt(i) << " ";
10   }
11 }
```

[13] https://en.cppreference.com/w/cpp/ranges.

[14] https://en.cppreference.com/w/cpp/ranges/filter_view.

Listing A.10 Example for finding all even values and compute their square root using the ranges library

```
#include <cmath>
#include <iostream>
#include <ranges>
#include <vector>

int main() {
  std ::vector<int> values = {0, 1, 2, 3, 4, 5, 6};

  auto tr_values = values
     std::views::filter([](int value)  return value
          std::views::transform([](int value) -> double { return
        std::sqrt(value);});
  for (int i : tr_values)
      std::cout << double(i) << ' ';
  std::cout << std::endl;
}
```

parameter `int i` and returns the result of the test if the function parameter is even. For more details about lambda functions, we refer to Appendix A.2. This view will select all even elements within the vector. Second, the computation of the square root within the body of the `if` statement in the previous example is replaced by `std::view::transform`.[15] Here, the lambda function is applied to each element. Since we used the first pipe `|` to filter out the even elements of the vector and applied the transformation with the second pipe `|`, the square root is only computed for the even elements. The major advantage for the ranges library is algorithm composition.

The example for printing the values of a container can be found in Listing A.4 on page 209 in Appendix A.2.

Important

The most important lesson to be learned here is that you should familiarize yourself with generic programming, lambda functions, move semantics, placement new, and smart pointers before digging into programming with HPX, since these features are heavily used throughout the book. The second most important lesson is that the ranges library is an interesting feature of the C++ 20 standard, but is not heavily used in this book.

[15] https://en.cppreference.com/w/cpp/ranges/transform_view.

Appendix B
Supplementary Header Files

Here, for completion, we present two header files used to implement the fractal set. Listing B.1 on page 217 shows the file *config.h* with the default constants for the fractal sets and some utility functions. This header file is included by all implementations of the fractal sets. Listing B.2 on page 218 shows the utility to write Portable Bitmap File Format (PBM) files. This is used to store the images of the fractal sets. Listing B.4 on page 222 shows utilities for the MPI implementation of the fractal sets.

Listing B.1 Configuration of default constants for the fractal sets and some utilities functions

```
#ifndef CONFIG_HPP
#define CONFIG_HPP

#include <complex>
#include <string.h>
#include <cmath>
#include <iostream>
#include <sstream>

typedef std::complex<double> complex;
```

```
inline size_t get_size_t(const char *varname,size_t
   defval) {
    const char *strval = getenv(varname);
    size_t retval;
    if(strval == nullptr) {
        retval = defval;
    } else {
        std::stringstream ss(strval);
        ss >> retval;
    }
    std::cout << "Using " << varname << "=" << retval
       << std::endl;
    return retval;
}
```


P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2

```

inline std::string get_string(const char *varname,std::
    string defval) {
     const char *strval = getenv(varname);
     std::string retval;
     if(strval == nullptr)
         retval = defval;
     else
         retval = strval;
     std::cout << "Using " << varname << "=" << retval
        << std::endl;
     return retval;
}

inline std::string valid_string(std::string s) {
     if(s == "mandelbrot")
         ;
     else if(s == "julia")
         ;
     else {
         std::cout << "Only 'mandelbrot' or 'julia' are
            valid values for type.";
         exit(1);
     }
     return s;
}


// Definition of constants
const double pi = M_PI;
const size_t max_iterations = get_size_t("MAX_ITER",80)
    ;
const size_t size_x = get_size_t("SIZE_X",3840);
const size_t size_y = get_size_t("SIZE_Y",2160);
const int max_color = get_size_t("MAX_COLOR",256);
std::string type = valid_string(get_string("TYPE","
    mandelbrot"));

#endif
```

Listing B.2 Utility to write Portable Bitmap File Format (PBM) files. This is used to store the images of the fractal sets

```
#ifndef PBM_HPP
#define PBM_HPP

// Header for generating PBM image files
// https://en.wikipedia.org/wiki/Netpbm

#include <cassert>
#include <fstream>
#include <tuple>
#include <vector>
```

```
11
12 #include <config.hpp>
13 ------------------------------------------------------------------------
14 // Function to smoothen the coloring
15 std::tuple<size_t, size_t, size_t> get_rgb(int value) {
16   double t = double(value) / double(max_iterations);
17   int r = (int)(9 * (1 - t) * t * t * t * 255);
18   int g = (int)(15 * (1 - t) * (1 - t) * t * t * 255);
19   int b = (int)(8.5 * (1 - t) * (1 - t) * (1 - t) * t *
20       255);
21
22   return std::make_tuple(r, g, b);
23 }
24 // convert an rgb tuple to an int
25 int make_color(int r, int g, int b) {
26   assert(0 <= r && r < max_color);
27   assert(0 <= g && g < max_color);
28   assert(0 <= b && b < max_color);
29   return (b * max_color + g) * max_color + r;
30 }
31
32 template<typename vector_type>
33 class PBM_ {
34   // width and height
35   int w, h;
36   // a vector of vector if ints to store the pixels
37   typename vector_type::allocator_type a;
38   vector_type values;
39
40   void _init() {
41     // initialize vector to all zero
42     for (int j = 0; j < h; j++) {
43       std::vector<int> row;
44       for (int i = 0; i < w; i++) {
45         row.push_back(0);
46       }
47       values[j] = std::move(row);
48     }
49   }
50
51  public:
52   int width() { return w; }
53
54   int height() { return h; }
55
56   PBM_(){};
57
58   PBM_(int w_, int h_) : w(w_), h(h_), a(), values(h_,a
59      ) { _init(); }
60   PBM_(int w_, int h_, typename vector_type::
61      allocator_type a_) : w(w_), h(h_), a(a_), values(
62      h_,a) { _init(); }
63   ~PBM_() {}
64
```

```
  // get or set a pixel at i, j
  int& operator()(int i, int j) { return values.at(j).
     at(i); }

  // get or set a row at j
  std::vector<int>& row(int j) { return values.at(j); }

  // save to a file
  void save(const std::string& fname) {
    std::ofstream f(fname);
    f << "P3\n";
    f << w << " " << h << "\n";
    f << max_color-1 << "\n";
    for (int j = 0; j < h; j++) {
      auto& row = values.at(j);
      for (int i = 0; i < w; i++) {
        int color = row.at(i);
        for (int c = 0; c < 3; c++) {
          f << (color % max_color) << " ";
          color /= max_color;
        }
        f << "\n";
      }
    }
    f.close();
  }
};

using PBM = PBM_<std::vector<std::vector<int>>>;
#endif
```

Listing B.3 Utility to write Portable Bitmap File Format (PBM) files. This is used to store the images of the fractal sets

```
#ifndef KERNEL_HPP
#define KERNEL_HPP

#include "pbm.hpp"
#include "config.hpp"

// Kernel to compute the Mandelbrot set
inline size_t mandelbrot(complex c) {
  std::complex<double> z(0, 0);
  for (size_t i = 0; i < max_iterations; i++) {
    z = z * z + c;
    if (abs(z) > 2.0) {
      return i;
    }
  }
  return 0;
}

```

```
inline double get_double(const char *varname,double
   defval) {
    const char *strval = getenv(varname);
    double retval;
    if(strval == nullptr) {
        retval = defval;
    } else {
        std::stringstream ss(strval);
        ss >> retval;
    }
    std::cout << "Using " << varname << "=" << retval
       << std::endl;
    return retval;
}

inline std::complex<double> get_const() {
  return {get_double("C_REAL",-.4),get_double("C_IMAG"
  ,.6)};
}

const std::complex<double> julia_const = get_const();

// Kernel to compute the Julia set
inline size_t julia(complex z) {
  for (size_t i = 0; i < max_iterations; i++) {
    z = z * z + julia_const;
    if (abs(z) > 2.0) {
      return i;
    }
  }
  return 0;
}

// Function to compute the Mandelbrot set for a pixel
inline size_t compute_pixel(complex c) {
  // std::complex<double> z(0, 0);
  for (size_t i = 0; i < max_iterations; i++) {
    if (type == "mandelbrot")
      return mandelbrot(c);
    else
      return julia(c);
  }

  return 0;
}

std::vector<int> compute_row(int item_index) {

  std::vector<int> result (size_y);

  complex c = complex(0, 4) * complex(item_index, 0) /
    complex(size_x, 0) - complex(0, 2);

```

```
  for (int i = 0; i < size_y; i++) {
    int value = compute_pixel(c + 4.0 * i / size_y -
       2.0);
    std::tuple<size_t, size_t, size_t> color = get_rgb(
       value);
    result[i] = make_color(std::get<0>(color),
                           std::get<1>(color),
                           std::get<2>(color));
  }

  return std::move(result);
}
#endif
```

Listing B.4 Utility functions for the Message Passing Interface

```
#ifndef UTIL_HPP
#define UTIL_HPP

// Split work evenly as possible. If the work cannot be
// divided exactly equally, then then first few workers
// will have one more item than the remainder.
void split_work(std::size_t workers, std::size_t
   work_items,
                std::size_t worker_num, std::size_t &
                   start,
                std::size_t &end) {
  std::size_t partition_size = work_items / workers;
  std::size_t excess = work_items % workers;
  start = partition_size *  worker_num    + (std::min)(
     worker_num  , excess);
  end   = partition_size * (worker_num+1) + (std::min)(
     worker_num+1, excess);
}
#endif
```

Appendix C
Software and Hardware Documentation

To explore the code, we provide the C++ Explorer, see Chap. 1, to the reader. Here, most of the code can be executed within the Jupyter notebook. The relevant compilers and installations of HPX are provided in the Docker image. Because Cling, in its current version, has issues with `std::async` and `std::future`, we use HPX for all shared memory examples. We think the notebook interface of the C++ Explorer is convenient. If the reader prefers to not use the C++ Explorer, we provide all examples as raw C++ code on GitHub®.[1]

For all of the benchmarks for the fractal sets, we used the compilers, tools, and dependencies in Table C.1 to build HPX 1.8.0. However, a different gcc compiler, CMake version, or Boost version should not change the results significantly. If you would like to build your own HPX or use the HPX package provided by Fedora's package manager *dnf*,[2] we recommend to use the software versions you have available.

The benchmarks were executed on the Rostam cluster at the Center of Computation & Technology at Louisiana State University and on the Ookami Cluster at Stony Brook University. Table C.2 summarizes the hardware of all three CPU architectures. For all data points the code was executed ten times and the average or the median of the computation time was used.

[1] https://github.com/ModernCPPBook/Examples.

[2] https://src.fedoraproject.org/rpms/hpx.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2

Table C.1 List of compilers, tools, and dependencies for building HPX on CCT's Rostam and Stony Brook's Ookami cluster for obtaining performance measurements

gcc 12.1.0	cmake 3.23.2	hwloc 2.7.1	boost 1.79	gperftools 2.10	HPX 1.8.0

Table C.2 Hardware information for CCT's Rostam and Stony Brook's Ookami cluster

CPU	Clock speed [GHz]	Memory [GB]
CCT's Rostam		
Intel ® Xeon® Gold 6148	2.4	96
AMD® EPYC™ 7H12	2.6	527
Stony Brook's Ookami		
Arm® A64FX™	1.8	32

References

1. M. Bull, X. Guo, I. Liabotis, PRACE-1IP Deliverable D **7** (2011)
2. J.M. Perkel, Nature **589**(7842), 344 (2021)
3. P. Diehl, S.R. Brandt, Concurr. Comput.: Pract. Exp., e6893 (2022). https://doi.org/10.1002/cpe.6893. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6893
4. C. Scott, Professional CMake: A Practical Guide (2018)
5. G.M. Kurtzer, V. Sochat, M.W. Bauer, PLoS One **12**(5), e0177459 (2017)
6. D.M. Ritchie, B.W. Kernighan, M.E. Lesk, *The C Programming Language* (Prentice Hall, Englewood Cliffs, 1988)
7. B. Stroustrup, *The Design and Evolution of C++* (Pearson Education India, Noida, 1994)
8. B. Stroustrup, *The C++ Programming Language* (Addison-Wesley, Reading, 1985)
9. D.R. Musser, A.A. Stepanov, *International Symposium on Symbolic and Algebraic Computation* (Springer, Berlin, 1988), pp. 13–25
10. J. Von Neumann, IEEE Ann. Hist. Comput. **15**(4), 27 (1993)
11. A.A. Stepanov, P. McJones, *Elements of Programming* (Addison-Wesley Professional, Reading, 2009)
12. D.E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, vol. 1 (Pearson Education, Upper Saddle River, 1968)
13. A. O'dwyer, *Mastering the C++ 17 STL: Make Full Use of the Standard Library Components in C++ 17* (Packt Publishing Ltd, Birmingham, 2017)
14. M. Zbigniew, *Computational Statistics* (Springer-Verlag, Berlin, 1996), pp. 372–373
15. M. Abramowitz, I.A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, vol. 55 (US Government Printing Office, Washington, 1964)
16. R. Brooks, J.P. Matelski, *Riemann Surfaces and Related Topics: Proceedings of the 1978 Stony Brook Conference, Ann. of Math. Stud*, vol. 97 (1981), pp. 65–71
17. G. Julia, J. Math. Pures Appl. **8**, 47 (1918)
18. P. Bachmann, *Encyklopädie der Mathematischen Wissenschaften mit Einschluss ihrer Anwendungen* (Springer, Berlin, 1904), pp. 636–674
19. E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen*, vol. 96 (American Mathematical Society, Providence, 2000)
20. R. Carroll, S. Prickett, *The Bible: Authorized King James Version* (Oxford Paperbacks, Oxford, 2008)
21. H. Kaiser, M. Brodowicz, T. Sterling, *2009 International Conference on Parallel Processing Workshops* (IEEE, 2009), pp. 394–401
22. C. Pheatt, J. Comput. Sci. Coll. **23**(4), 298 (2008)
23. R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, ACM SigPlan Notices **30**(8), 207 (1995)

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2

24. B.L. Chamberlain, D. Callahan, H.P. Zima, Int. J. High Perform. Comput. Appl. **21**(3), 291 (2007)
25. K. Ebcioglu, V. Saraswat, V. Sarkar, *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, vol. 30 (Citeseer, 2004)
26. Y. Zheng, A. Kamil, M.B. Driscoll, H. Shan, K. Yelick, *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (IEEE, 2014), pp. 1105–1114
27. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.W. Maessen, S. Ryu, G.L. Steele Jr., S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al., Sun Microsystems **139**(140), 116 (2005)
28. L.V. Kale, S. Krishnan, *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '93* (Association for Computing Machinery, New York, 1993), pp. 91–108. https://doi.org/10.1145/165854.165874
29. S. Ahuja, N. Carriero, D. Gelernter, Computer **19**(08), 26 (1986)
30. G. Wells, *Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05)* (2005), pp. 87–98
31. N. Gupta, S.R. Brandt, B. Wagle, N. Wu, A. Kheirkhahan, P. Diehl, F.W. Baumann, H. Kaiser, *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)* (IEEE, 2020), pp. 11–20
32. P. Diehl, G. Daiß, D. Marcello, K. Huck, S. Shiber, H. Kaiser, J. Frank, G.C. Clayton, D. Pflüger, *2021 IEEE International Conference on Cluster Computing (CLUSTER)* (IEEE, 2021), pp. 204–214
33. P. Diehl, G. Dais, K. Huck, D. Marcello, S. Shiber, H. Kaiser, D. Pfluger, *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE Computer Society, Los Alamitos, 2023), pp. 682–691. https://doi.org/10.1109/IPDPSW59300.2023.00116. https://doi.ieeecomputersociety.org/10.1109/IPDPSW59300.2023.00116
34. T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, H. Kaiser, *Proceedings of OpenSuCo*, vol. 5 (2017)
35. H. Kaiser, P. Diehl, A.S. Lemoine, B.A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S.R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, T. Zhang, J. Open Source Softw. **5**(53), 2352 (2020). https://doi.org/10.21105/joss.02352
36. C++ Standards Committee, ISO/IEC 14882:2011, Standard for Programming Language C++ (C++11). Tech. rep., ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee) (2011). https://wg21.link/N3337, last publicly available draft
37. C++ Standards Committee, ISO/IEC 14882:2020, Standard for Programming Language C++ (C++20). Tech. rep., ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee) (2020). https://wg21.link/N4860, last publicly available draft
38. G. Almasi, *PGAS (Partitioned Global Address Space) Languages* (Springer US, Boston, 2011), pp. 1539–1545
39. P. Amini, H. Kaiser, *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)* (IEEE, 2019), pp. 26–33
40. H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (2014), pp. 1–11
41. T. Von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser, ACM SIGARCH Comput. Archit. News **20**(2), 256 (1992)
42. J. Biddiscombe, T. Heller, A. Bikineev, H. Kaiser, *14th International Conference on Applied Computing* (IADIS, International Association for Development of the Information Society, 2017)
43. J. Yan, H. Kaiser, M. Snir, *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23 (Association for Computing Machinery, New York, 2023), pp. 1151–1161. https://doi.org/10.1145/3624062.3624598

44. G. Daiß, P. Amini, J. Biddiscombe, P. Diehl, J. Frank, K. Huck, H. Kaiser, D. Marcello, D. Pfander, D. Pfüger, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2019), pp. 1–37
45. P. Diehl, D. Marcello, P. Amini, H. Kaiser, S. Shiber, G.C. Clayton, J. Frank, G. Daiß, D. Pflüger, D. Eder, et al., Comput. Sci. Eng. **23**(3), 73 (2021)
46. P.A. Grubel, *Dynamic Adaptation in HPX–A Task-Based Parallel Runtime System* (New Mexico State University, 2016)
47. K.A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A.D. Malony, T. Sterling, R. Fowler, Supercomput. Front. Innov. **2**(3), 49 (2015)
48. D. Terpstra, H. Jagode, H. You, J. Dongarra, *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden* (Springer, Berlin, 2010), pp. 157–173
49. P. Diehl, G. Daiss, K. Huck, D. Marcello, S. Shiber, H. Kaiser, J. Frank, G.C. Clayton, D. Pflueger, arXiv preprint arXiv:2210.06437 (2022)
50. Z. Khatami, L. Troska, H. Kaiser, J. Ramanujam, A. Serio, *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware* (2017), pp. 1–8
51. G. Laberge, S. Shirzad, P. Diehl, H. Kaiser, S. Prudhomme, A.S. Lemoine, et al., *2019 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)* (IEEE, 2019), pp. 31–43
52. B. Wagle, S. Kellar, A. Serio, H. Kaiser, *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE, 2018), pp. 1133–1140
53. M.J. Flynn, Proc. IEEE **54**(12), 1901 (1966)
54. R. Duncan, Computer **23**(2), 5 (1990)
55. S. Yadav, N. Gupta, A. Reverdell, H. Kaiser, *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)* (2021), pp. 20–29. https://doi.org/10.1109/ESPM254806.2021.00008
56. J. Falcou, J. Serot, Scalable Comput.: Pract. Exp. **6**(4) (2005)
57. C++ Standards Committee, ISO/IEC 14882:2014, Standard for Programming Language C++ (C++14). Tech. rep., ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee) (2014). https://wg21.link/N4296, last publicly available draft
58. C++ Standards Committee, ISO/IEC DIS 14882, Standard for Programming Language C++ (C++17). Tech. rep., ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee) (2017). https://wg21.link/N4659, last publicly available draft
59. P. Diehl, S.R. Brandt, H. Kaiser, in *Asynchronous Many-Task Systems and Applications*, ed. by P. Diehl, P. Thoman, H. Kaiser, L. Kale (Springer Nature Switzerland, Cham, 2023), pp. 27–38
60. D.C. Marcello, S. Shiber, O. De Marco, J. Frank, G.C. Clayton, P.M. Motl, P. Diehl, H. Kaiser, Mon. Not. R. Astron. Soc. **504**(4), 5345 (2021)
61. K. Kadam, P.M. Motl, D.C. Marcello, J. Frank, G.C. Clayton, Mon. Not. R. Astron. Soc. **481**(3), 3683 (2018). https://doi.org/10.1093/mnras/sty2540
62. E. Chatzopoulos, J. Frank, D.C. Marcello, G.C. Clayton, Astrophys. J. **896**(1), 50 (2020). https://doi.org/10.3847/1538-4357/ab91bb
63. S.A. Silling, E. Askari, Comput. Struct. **83**(17–18), 1526 (2005)
64. P. Diehl, S. Prudhomme, M. Lévesque, J. Peridynamics Nonlocal Model. **1**(1), 14 (2019)
65. P. Diehl, R. Lipton, T. Wick, M. Tyagi, Comput. Mech. **69**(6), 1259 (2022)
66. P. Diehl, P.K. Jha, H. Kaiser, R. Lipton, M. Lévesque, SN Appl. Sci. **2**(12), 1 (2020)
67. P.K. Jha, P. Diehl, J. Open Source Softw. **6**(65), 3020 (2021). https://doi.org/10.21105/joss.03020
68. U.R. Hähner, G. Alvarez, T.A. Maier, R. Solcà, P. Staar, M.S. Summers, T.C. Schulthess, Comput. Phys. Commun. **246**, 106709 (2020)
69. W. Wei, A. Chatterjee, K. Huck, O. Hernandez, H. Kaiser, *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)* (2020), pp. 77–84. https://doi.org/10.1109/ScalA51936.2020.00015

70. G. Balduzzi, A. Chatterjee, Y.W. Li, P.W. Doak, U. Haehner, E.F. D'Azevedo, T.A. Maier, T. Schulthess, *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (IEEE, 2019), pp. 433–444
71. K. Iglberger, G. Hager, J. Treibig, U. Rüde, SIAM J. Sci. Comput. **34**(2), C42 (2012)
72. T. Veldhuizen, C++ Report **7**(5), 26 (1995)
73. V.G. Castellana, M. Minutoli, *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (IEEE, 2018), pp. 442–451
74. D.R. Butenhof, *Programming with POSIX Threads* (Addison-Wesley Professional, Reading, 1997)
75. R. Kay, *Unix Systems Programming: Communication, Concurrency and Threads, 2/E* (Pearson Education India, Noida, 2004)
76. T. Heller, H. Kaiser, P. Diehl, D. Fey, M.A. Schweitzer, *International Conference on High Performance Computing* (Springer, Berlin, 2016), pp. 18–31
77. L. Dagum, R. Menon, IEEE Comput. Sci. Eng. **5**(1), 46 (1998)
78. OpenMP Architecture Review Board. OpenMP application program interface version 5.2 (2021). http://www.openmp.org/mp-documents/spec52.pdf
79. T. Zhang, S. Shirzad, P. Diehl, R. Tohid, W. Wei, H. Kaiser, *Proceedings of the International Workshop on OpenCL* (2019), pp. 1–10
80. C.R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, J. Wilke, IEEE Trans. Parallel Distrib. Syst. **33**(4), 805 (2022). https://doi.org/10.1109/TPDS.2021.3097283
81. J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah, SIAM Rev. **59**(1), 65 (2017). https://doi.org/10.1137/141000671
82. N.D. Matsakis, F.S. Klock II, *ACM SIGAda Ada Letters*, vol. 34 (ACM, New York, 2014), pp. 103–104
83. P. Diehl, M. Morris, S.R. Brandt, N. Gupta, H. Kaiser, in *Euro-Par 2023: Parallel Processing Workshops*, vol. LNCS 14352, ed. by D. Blanco Heras, G. Pallis, H. Herodotou, D. Balouek, P. Diehl, T. Cojean, K. Fürlinger, M.H. Kirbey, M. Nardelli, P. Di Sanzo, D. Zeinalipour. LNCS, vol. 14352 (Springer Nature Switzerland, Cham, 2024), pp. 120–131
84. R. Gonzalez, B.M. Gordon, M.A. Horowitz, IEEE J. Solid-State Circuits **32**(8), 1210 (1997)
85. G.M. Amdahl, in *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference* (ACM, New York, 1967), pp. 483–485
86. H. El-Rewini, M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing* (Wiley, Hoboken, 2005)
87. M.J. Flynn, IEEE Trans. Comput. **100**(9), 948 (1972)
88. D. Kusswurm, *Modern Parallel Programming with C++ and Assembly Language: X86 SIMD Development Using AVX, AVX2, and AVX-512* (Springer, Berlin, 2022)
89. A. Williams, *C++ Concurrency in Action* (Simon and Schuster, New York, 2019)
90. D. Adams, *The Ultimate Hitchhiker's Guide to the Galaxy: Five Novels in One Outrageous Volume* (Del Rey, New York, 2010)
91. D. Naishlos, *Proceedings of the 2004 GCC Developers Summit* (Citeseer, 2004), pp. 105–118
92. H. Finkel, *The LLVM Compiler Infrastructure 2012 European Conference* (2012)
93. M. Kretz, V. Lindenstruth, Softw.: Pract. Exp. **42**(11), 1409 (2012)
94. J. Penuchot, J. Falcou, A. Khabou, *2018 International Conference on High Performance Computing Simulation (HPCS)* (2018), pp. 508–514. https://doi.org/10.1109/HPCS.2018.00086
95. P. Grubel, H. Kaiser, J. Cook, A. Serio, *2015 IEEE International Conference on Cluster Computing* (IEEE, 2015), pp. 682–689
96. R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, J. McDonald, *Parallel Programming in OpenMP* (Morgan Kaufmann, San Francisco, 2001)
97. S. Shirzad, R. Tohid, A. Kheirkhahan, B. Wagle, H. Kaiser, *European Conference on Parallel Processing* (Springer, Berlin, 2022), pp. 456–467

98. Z. Li, E. Kraemer, *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum* (2013), pp. 1304–1311. https://doi.org/10.1109/IPDPSW.2013.193
99. M. Van Steen, A. Tanenbaum, Network **2**, 28 (2002)
100. P.J. Lu, M.C. Lai, J.S. Chang, Electronics **11**(9) (2022). https://doi.org/10.3390/electronics11091369. https://www.mdpi.com/2079-9292/11/9/1369
101. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0* (2021). https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
102. D. Bonachea, P.H. Hargrove, *International Workshop on Languages and Compilers for Parallel Computing* (Springer, Berlin, 2018), pp. 138–158
103. D.W. Walker, Standards for message-passing in a distributed memory environment. Tech. rep., Oak Ridge National Lab., TN (1992)
104. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 1.0* (94). https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps
105. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.1* (2023). https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf
106. H. Sutter, et al., Dr. Dobb's J. **30**(3), 202 (2005)
107. P.E. Ross, IEEE Spectrum **45**(4), 72 (2008)
108. D.A. Bader, Computer **49**(08), 10 (2016)
109. Fortran Standards Committee, ISO/IEC 1539:2010. Tech. rep., JTC1/SC22/WG5 (the Fortran Standards Committee) (2010). N2121, last publicly available draft
110. S.W. Poole, O. Hernandez, J.A. Kuehn, G.M. Shipman, A. Curtis, K. Feind, *OpenSHMEM - Toward a Unified RMA Model* (Springer US, Boston, 2011), pp. 1379–1391. https://doi.org/10.1007/978-0-387-09766-4
111. W.W. Carlson, J.M. Draper, D.E. Culler, K. Yelick, E. Brooks, K. Warren, Introduction to UPC and language specification. Tech. rep., Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999)
112. T.A. Johnson, *7th International Conference on PGAS Programming Models* (2013), p. 54
113. K. Fürlinger, T. Fuchs, R. Kowalewski, *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016)* (Sydney, 2016), pp. 983–990. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0140
114. J. Nieplocha, R. Harrison, J. Supercomput. **11**(2), 119 (1997)
115. V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, O. Tardieu, *The First Workshop on Advances in Message Passing* (2010), pp. 1–8
116. M. Bauer, S. Treichler, E. Slaughter, A. Aiken, *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (IEEE, 2012), pp. 1–11
117. J. Daily, A. Vishnu, B. Palmer, H. Van Dam, D. Kerbyson, *2014 21st International Conference on High Performance Computing (HiPC)* (IEEE, 2014), pp. 1–10
118. H. Pritchard, E. Harvey, S.E. Choi, J. Swaro, Z. Tiffany, *Proceedings of Cray User Group Meeting, CUG*, vol. 2016 (2016)
119. P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, et al., J. Supercomput. **74**(4), 1422 (2018)
120. G. Karypis, V. Kumar, SIAM J. Sci. Comput. **20**(1), 359 (1998). https://doi.org/10.1137/S1064827595287997
121. G. Karypis, V. Kumar, J. Parallel Distrib. Comput. **48**(1), 71 (1998)
122. E.G. Boman, U.V. Catalyurek, C. Chevalier, K.D. Devine, Sci. Program. **20**(2), 129 (2012)
123. K. Devine, E. Boman, R. Heaphy, B. Hendrickson, C. Vaughan, Comput. Sci. Eng. **4**(2), 90 (2002)
124. M.J. Berger, S.H. Bokhari, IEEE Trans. Comput. **36**(05), 570 (1987)
125. M.S. Warren, J.K. Salmon, *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (1993), pp. 12–21
126. A. Patra, J.T. Oden, Comput. Syst. Eng. **6**(2), 97 (1995)

127. J.R. Pilkington, S.B. Baden, CSE Technical Report Number CS94-349 (1994)
128. M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*, vol. 9 (Springer Science & Business Media, Berlin, 2012)
129. C. Geuzaine, J.F. Remacle, Int. J. Numer. Methods Eng. **79**(11), 1309 (2009)
130. M.J. Berger, J. Oliger, J. Comput. Phys. **53**(3), 484 (1984)
131. M.J. Berger, P. Colella, J. Comput. Phys. **82**(1), 64 (1989)
132. P. Gadikar, P. Diehl, P.K. Jha, *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE, 2021), pp. 669–678
133. M. Folk, A. Cheng, K. Yates, *Proceedings of Supercomputing*, vol. 99 (1999), pp. 5–33
134. R. Rew, G. Davis, IEEE Comput. Graphics Appl. **10**(4), 76 (1990)
135. W. Schroeder, K.M. Martin, W.E. Lorensen, *The Visualization Toolkit an Object-Oriented Approach to 3D Graphics* (Prentice-Hall, Inc., Englewood Cliffs, 1998)
136. W.F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, et al., SoftwareX **12**, 100561 (2020)
137. J. Li, W.k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, arXiv preprint cs/0306048 (2003)
138. J.P. Prost, *MPI-IO* (Springer US, Boston, 2011), pp. 1191–1199. https://doi.org/10.1007/978-0-387-09766-4_297
139. C. Marshall, C++ FAQ: "What's this "Serialization" Thing All About?" (2015). https://web.archive.org/web/20150405013606/http://isocpp.org/wiki/faq/serialization. Accessed 11 Sept 2023
140. R.A.V. Engelen, ACM Trans. Internet Technol. **8**(3), 1 (2008)
141. R.A. Van Engelen, K.A. Gallivan, *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)* (IEEE, 2002), pp. 128–128
142. X. Leroy, The zinc experiment: an economical implementation of the ml language. Ph.D. thesis, INRIA (1990)
143. G. Karniadakis, R.M. Kirby II, *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*, vol. 2 (Cambridge University Press, Cambridge, 2003)
144. C. Hewitt, P. Bishop, R. Steiger, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (1973), pp. 235–245
145. J. Biddiscombe, T. Heller, A. Bikineev, H. Kaiser, *14th International Conference on Applied Computing* (IADIS, International Association for Development of the Information Society, 2017)
146. I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, N. Sultana, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19 (Association for Computing Machinery, New York, 2019). https://doi.org/10.1145/3295500.3356176
147. D.E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R.E. Grant, T. Naughton, H.P. Pritchard, M. Schulz, G.R. Vallee, Concurr. Comput.: Pract. Exp. **32**(3), e4851 (2020)
148. R. Palumbo, H. Saiedian, M. Zand, *Proceedings of the 1992 ACM Annual Conference on Communications*, CSC '92 (Association for Computing Machinery, New York, 1992), pp. 367–375. https://doi.org/10.1145/131214.131261
149. D.M. Chitty, *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* (2007), pp. 1566–1573
150. J. Krüger, R. Westermann, *ACM SIGGRAPH 2005 Courses* (ACM, New York, 2005), pp. 234–es
151. J. Bolz, I. Farmer, E. Grinspun, P. Schröder, ACM Trans. Graphics **22**(3), 917 (2003)
152. J. Sanders, E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming* (Addison-Wesley Professional, Reading, 2010)
153. J.R. Hammond, M. Kinsner, J. Brodman, *Proceedings of the International Workshop on OpenCL* (2019), pp. 1–2
154. P. Diehl, M. Seshadri, T. Heller, H. Kaiser, *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)* (IEEE, 2018), pp. 19–28

155. A. Kurganov, E. Tadmor, J. Comput. Phys. **160**(1), 241 (2000). https://doi.org/10.1006/jcph.2000.6459
156. D.C. Marcello, Astron. J. **154**, 92 (2017). https://doi.org/10.3847/1538-3881/aa7b2f
157. D. Pfander, G. Daiß, D. Marcello, H. Kaiser, D. Pflüger, *Proceedings of the International Workshop on OpenCL*, IWOCL '18 (ACM, New York, 2018), pp. 19:1–19:8. https://doi.org/10.1145/3204919.3204938
158. S.J. Pennycook, J.D. Sewall, D.W. Jacobsen, T. Deakin, S. McIntosh-Smith, Comput. Sci. Eng. **23**(5), 28 (2021)
159. T. Heller, B.A. Lelbach, K.A. Huck, J. Biddiscombe, P. Grubel, A.E. Koniges, M. Kretz, D. Marcello, D. Pfander, A. Serio, et al., Int. J. High Perform. Comput. Appl. **33**(4), 699 (2019)
160. M.A. Hermanns, N.T. Hjelm, M. Knobloch, K. Mohror, M. Schulz, Parallel Comput. **85**, 119 (2019). https://doi.org/10.1016/j.parco.2018.12.006. https://www.sciencedirect.com/science/article/pii/S0167819118303314
161. J. Schuchart, P. Samfass, C. Niethammer, J. Gracia, G. Bosilca, Parallel Comput. **106**, 102793 (2021). https://doi.org/10.1016/j.parco.2021.102793. https://www.sciencedirect.com/science/article/pii/S0167819121000466
162. G. Daiß, M. Simberg, A. Reverdell, J. Biddiscombe, T. Pollinger, H. Kaiser, D. Pflüger, *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2021), pp. 377–386. https://doi.org/10.1109/IPDPSW52791.2021.00066
163. G. Daiß, P. Diehl, H. Kaiser, D. Pflüger, *Proceedings of the 2023 International Workshop on OpenCL*, IWOCL '23 (Association for Computing Machinery, New York, 2023), IWOCL '23. https://doi.org/10.1145/3585341.3585354
164. G. Dais, P. Diehl, D. Marcello, A. Kheirkhahan, H. Kaiser, D. Pfluger, *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (IEEE Computer Society, Los Alamitos, 2022), pp. 89–99. https://doi.org/10.1109/P3HPC56579.2022.00014. https://doi.ieeecomputersociety.org/10.1109/P3HPC56579.2022.00014
165. G. Dais, S. Singanaboina, P. Diehl, H. Kaiser, D. Pfluger, *2022 IEEE/ACM 7th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)* (IEEE Computer Society, Los Alamitos, 2022), pp. 10–19. https://doi.org/10.1109/ESPM256814.2022.00007. https://doi.ieeecomputersociety.org/10.1109/ESPM256814.2022.00007
166. K.A. Ericsson, R.T. Krampe, C. Tesch-Römer, Psychol. Rev. **100**(3), 363 (1993)
167. E. Bueler, *PETSc for Partial Differential Equations: Numerical Solutions in C and Python* (SIAM, Philadelphia, 2020)
168. P. Diehl, G. Daiss, S. Brandt, A. Kheirkhahan, H. Kaiser, C. Taylor, J. Leidel, *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23 (Association for Computing Machinery, New York, 2023), pp. 1533–1542. https://doi.org/10.1145/3624062.3624230
169. A. Koenig, *Accelerated C++: Practical Programming by Example* (Pearson Education India, Noida, 2000)
170. B. Stroustrup, *Programming: Principles and Practice Using C++* (Pearson Education, Upper Saddle River, 2014)
171. B. Stroustrup, *A Tour of C++* (Addison-Wesley Professional, Reading, 2022)
172. N.M. Josuttis, *C++ 17: The Complete Guide* (Nicolai Josuttis, 2019)
173. N.M. Josuttis, *C++ 20: The Complete Guide* (Nicolai Josuttis, 2022)
174. D. Vandevoorde, N.M. Josuttis, *C++ Templates: The Complete Guide, Portable Documents* (Addison-Wesley Professional, Reading, 2002)

Glossary

Action A type of handle that allows a function to be called remotely in HPX. There are two types: plain actions and component actions.

AGAS The Active Global Address space is the part of HPX which allows the system to find objects or localities using an object of type `hpx::id_type`. Internally, it uses logic similar to `std::shared_ptr` to determine when the object goes out of scope.

Client A `struct` or `class` with methods to access the component's data synchronously or asynchronously.

Component A globally addressable object in HPX. This object is registered within the Active Global Address Space (AGAS) and given a unique Global ID (GID). The component is registered using the macro `HPX_REGISTER_COMPONENT`.

Component action A type of action which is also a member function of a component. Each component action is defined to be an action for the component by using the macro `HPX_DEFINE_COMPONENT_DIRECT_ACTION`. After the definition, each action is registered as an action by using the macro `HPX_REGISTER_ACTION`.

Global Identifier Each object generated within the HPX runtime system is assigned with a unique **G**lobal **Id**entifier (GID) to be globally accessible in the Active Global Address Space (AGAS). The GID is constant through the life time of the object, even if the object is moved to a different node. The data type of a GID is `hpx::id_type`.

***L*atencies** are imposed by the time-distance delay intrinsic to accessing remote resources and services.

Locality The HPX term for a process (possibly on a remote node). Note that Localities, like components, have a GID. This GID can be used by any process to run an action on that locality.

***O*verheads** work required for the management of parallel actions and resources.

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2

Parallel Control Element In HPX, an action is mediated by a parcel (**Par**allel **C**ontrol **El**ement). Each time an action is called on a remote locality, an HPX parcel containing the destination address, action, arguments, and continuations is generated.

ParalleX describes governing principles to address intrinsic hiding of latency, delivering an abundance of variable fine-grained parallelism using a hierarchical namespace environment.

Parcel See Parallel Control Element.

Plain action A type of action that is used to enable the remote calling of methods that are not part of any component. A plain action is registered by using `HPX_PLAIN_ACTION` and is not associated with an HPX component.

***S*tarvation** occurs when there is insufficient concurrent work available to maintain high utilization of all resources.

***W*aiting** for contention resolution is the delay due to the lack of availability of oversubscribed shared resources.

Index

A
Accelerator cards, 185
Action, 233
Active Global Address Space (AGAS), 50, 123, 126, 150, 233
Active messaging, 51
Adaptive mesh refinement (AMR), 128, 187
Algorithm
  parallel, 99
Amdahl's law, 43, 69, 144
American National Standards Institute (ANSI), 11
Application
  DCA++, 54
  Octo-Tiger, 54
  PeriHPX, 54
Arithmetic Logic Units (ALUs), 77
Asynchronous Partitioned Global Address Space (APGAS), 125
Asynchronous programming, 59, 85
Atomic, 68
Atomic operation, 68
Autonomic Performance Environment for Exascale (APEX), 52

B
Blaze, 54
Boost, 179
  MPI, 179

C
Capture
  implicit, 207
Central Processing Unit (CPU), 185
C++ Explorer, 3, 4, 55
Chapel, 125, 126
Charm++, 126
CILK, 60
Client, 154, 155, 233
Cling, 3
CMake, 3, 4
CoarrayC++, 125
Coarray FORTRAN (CAF), 126
co_await, 109
Comma-separated values (CSV), 138
Communication
  one-sided, 189
Compilers, 15
Complexity, 38
Component, 152, 155, 233
Component action, 152, 233
  define, 152
Compute Unified Device Architecture (CUDA), 185
Concurrency, 52
Container, 19
  dequeue, 27
  list, 21
  map, 27
  queue, 27
    priority, 27
  set, 27
  stack, 27
  vector, 20
co_return, 109
Coroutines, 19, 60, 109
  co_await, 109
  co_return, 109
  co_yield, 109
co_yield, 109

P. Diehl et al., *Parallel C++*, https://doi.org/10.1007/978-3-031-54369-2

C standard, 11
CUDA Profiling Tools Interface (CUPTI), 52

D
DASH, 125
Data driven, 47
Dataflow, 46
Data migration, 47
DCA++, 54
Deadlock, 66
Delete, 211
Delimiter-Separated Values (DSV), 173
Docker, 3
  Hub, 3
Domain decomposition, 127
Duncan's taxonomy, 76

E
Embedded domain-specific language (eDSL), 186
Execution policies, 60, 99

F
Field Programmable Gate Arrays (FPGA), 185
FleCSI, 55
Flynn's taxonomy, 75
Fractal set
  asynchronous programming, 93
  benchmark, 117
  coroutines, 113
  low level threads, 80
  parallel algorithms, 104
Framework
  FleCSI, 55
Function
  lambda, 207

G
GASNet, 126, 149
General-purpose computing on graphics processing units (GPGPU), 45, 47, 185
Generic programming, 206
GlobalArray, 125
Global Identifier (GID), 155, 233
Global object space (GOS), 126
Gmsh, 127
Graphics Processing Unit (GPU), 185
Green 500, 125
Gustafson's Law, 43, 71

H
Heterogeneous Interface for Portability (HIP), 186
Hierarchical Data Format (HDF), 129
HPX, 48, 124, 126
  action, 233
  active global address space, 50
  advanced synchronization, 89
  AGAS, 51
  client, 154, 155, 233
  component, 152, 155, 233
  component action, 155, 233
  dataflow, 91
  execution
    auto_chunk_size, 104
    dynamic_chunk_size, 104
    par_simd, 103
    simd, 103
    static_chunk_size, 104
    task, 101
  for_each, 100
  global identifier, 148, 151
  local control objects, 50
  make_ready_future, 91
  parallel algorithms, 100
  parcel, 148
  parcel port, 51
  performance counters, 52
  plain action, 156, 234
  Policy Engine, 52
  serialization, 159
  serialization buffer, 159
  SIMD, 52
  thread
    manager, 50
    subsystem, 50
  unwrapping, 91
  when_all, 89, 90
  when_any, 91
  when_each, 91
HPX-Kokkos, 190, 192
HpxMP, 60

I
Integrated Development Environment (IDE), 3
International Organization for Standardization (ISO), 11
I/O node, 129
Iterator, 26
  advance, 22
  begin, 27
  end, 27
Iterator type, 26

bidirectional, 26
forward, 26
input, 26
output, 26
random access, 26

J
Julia, 62
Jupyter notebooks, 3

K
Kokkos, 61, 186, 192
SIMD, 192

L
Lambda function, 207
*L*atencies, 43
Latency, 47
Law
Amdahl's, 43, 69
Gustafson's, 43, 71
LCI, 126
Legion, 125, 126
Libfabric, 189
Library
Blaze, 54
SHAD, 54
Light-weight threads, 50
Load balancing, 127
dynamic, 127, 128
static, 127
Local control objects, 50
Locality, 233

M
Macro, 152
define component action, 152
register component, 152
Mean time between failures (MTBF), 44
Memory access, 75
Message Passing Interface (MPI), 45, 46, 123, 124, 131, 189, 199
Bcast, 131
boost, 179
collective basis, 131
Finalize, 135
Ibcast, 131
Init, 134
Irecv, 131
Isend, 131
MPI+X, 124
MPI_Comm_rank, 134
MPI_Comm_size, 134
mpiexec, 143
mpirun, 143
OpenMP, 199
point basics, 131
Recv, 131
Send, 131
Metis, 127
Microsoft DirectCompute, 186
Move, 83
Move semantics, 209
MPI+OpenMP, 141
MPI+X, 124, 180
Multiple Instruction and Multiple Data (MIMD), 75
Multiple Instruction and Single Data (MISD), 75
Mutex, 65, 68

N
Network Common Data Form (NetCDF), 129
Networking
Ethernet, 123
New, 210
Non-uniform memory access (NUMA), 75

O
Octo-Tiger, 54, 187
Open Compute Language (OpenCL), 186
Open Graphics Library (OpenGL), 185
Open Multi-Processing (OpenMP), 46, 60
OMP_NUM_THREADS, 143
OpenSHMEM, 125
Operation
atomic, 68
Operator
decrement, 26
dereference, 26
*O*verheads, 43, 136, 138

P
Paralellism, 52
Parallel Control Element (Parcel), 148, 234
coalescing, 148
Parallel efficiency, 72
Parallelism, 125
intra-node, 140
pipeline, 76
ParalleX, 44, 234

Parcel port, 51
ParMetis, 127
Partitioned Global Address Space (PGAS), 46, 47, 51, 123, 125
Performance counters, 52
PeriHPX, 54
Pipeline parallelism, 76
Placement new, 210
Plain action, 156, 234
Podman, 3
Pointer
  raw, 212
  shared, 213
  smart, 212
  unique, 212
Policy Engine, 52
Portable Bitmap File Format (PBM), 35, 138, 203, 217
Principles, 44
  global barrier, 46
  latency, 45
  lightweight threads, 45
Processing unit, 76
Programming
  generic, 206
Programming language
  C, 11
Pthreads, 59

R
Race condition, 63, 65
Ranges, 214
  transform, 215
  view, 214
Recursive Coordinate Bisection (RCB), 127
Reduction operation, 66
Remote Direct Memory Access (RDMA), 125
Remote Memory Access (RMA), 127, 189
ROCm, 186

S
Scalability
  fault tolerance, 44
  generality, 44
  performance, 44
  power consumption, 44
  programmability, 44
Scaling
  strong, 74
  weak, 74
Scoped lock, 67
Sender & receiver, 200
Serialization, 130, 159
  buffer, 159
Set
  Julia, 34
  MandelBrot, 33
SILO, 129
Single Instruction Multiple Data (SIMD), 52, 76, 77, 102
Single Instruction and Single Data (SISD), 75
Singularity, 7
SLOW, 199
  *L*atencies, 43, 233
  *O*verheads, 43, 136, 138, 233
  *S*tarvation, 43, 234
  *W*aiting, 43, 234
Smart pointer, 211
Space-Filling Curve Partitioning (SFCP), 127
Speedup, 70, 72
Standard
  C, 11
Standard Library (SL), 17, 79
  algorithms, 18, 29
    parallel, 99
  concepts, 17
  concurrency support, 19
  container, 17, 19
  coroutines, 19
  execution policies, 60
  iterator, 17, 26
  iterator type, 26
  jthread, 79
  list, 21
  move, 83
  numeric library, 36
  parallel algorithms, 60
  ranges, 17
  thread, 79
    join, 80
  vector, 20
Standard Template Library (STL), 14
*S*tarvation, 43
Std
  async, 85
  execution
    par, 99
    par_unseq, 99
    seq, 99
    simd, 103
  future, 86
Strong scaling, 74

Supervisor-worker, 163
SYCL, 186

T
Taxonomy
  Duncan's, 76
  Flynn's, 75
Threading Building Blocks (TBB), 99
Threads
  light-weight, 50
Tools
  C++ Explorer, 55
Transmission Control Protocol (TCP), 126, 147
Trilinos, 127
  Zoltan, 127

U
Uniform memory access (UMA), 75
UPC, 125, 126
UPC++, 125, 126

V
Vectorization
  automatic, 102
The Visualization Toolkit (VTK), 129

W
*W*aiting, 43
Weak scaling, 74

X
X10, 125