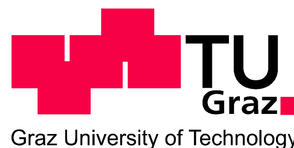
Gernot Kapper, BSc

Development of a library for parallel computation of physical problems and its application to plasma physics

MASTER THESIS

For obtaining the academic degree
Diplom-Ingenieur

Master Programme of
Technical Physics



Graz University of Technology

Supervisor

Ao. Univ.-Prof. Dipl.-Ing. Dr.phil. Martin Heyn

Co-Supervisor

Ass.-Prof. Dipl.-Ing. Dr.techn. Winfried Kernbichler

Institute of Theoretical and Computational Physics

Graz, January 2013

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Zusammenfassung

Die numerische Berechnung physikalischer Problemstellungen ist ein wichtiger Bereich der theoretischen Physik. Trotz der hohen Verfügbarkeit von Rechenleistung moderner Computer können die Lösungsvorgänge komplexer Problemstellungen zeitaufwändige Prozesse sein. Aus diesem Grund müssen jene Lösungsalgorithmen sowohl numerisch korrekt arbeiten, als auch die geforderten Aufgaben in vertretbarer Zeit lösen können.

Diese Masterarbeit untersucht die Möglichkeiten der Parallelisierung physikalischer Probleme mit Hilfe von Mehrkern-Prozessoren sowie Rechenclustern. Dazu wird eine flexible Softwarebibliothek entwickelt, durch deren Einsatz neue und bestehende Lösungsalgorithmen möglichst einfach parallelisiert werden können. Um eine hohe Anpassungsfähigkeit an verschiedene Aufgaben zu gewährleisten, wird diese Bibliothek nach Richtlinien der objekt-orientierten Programmierung in der Programmiersprache Fortran entwickelt. Diese Programmiersprache, welche seit Version 2003 objekt-orientierte Ansätze unterstützt, ist in mathematischen und physikalischen Bereichen weit verbreitet.

Die erste physikalische Anwendung der entwickelten Bibliothek ist die Parallelisierung des bestehenden Programms NEO-2 des Instituts für theoretische Physik - Computational Physics der Technischen Universität Graz, das zur Berechnung von neoklassischem Transport in Plasmen verwendet wird. Dies soll nicht nur als Test zur Stabilisierung der Bibliothek dienen, sondern vor allem die gesteigerte Leistungsfähigkeit durch die Parallelisierung demonstrieren.

Abstract

The numerical computation of physical problems is an important domain of theoretical physics. Despite the availability of high processing power in modern computer systems, the computation of complex tasks can be a time consuming process. In order to obtain the intended results in a reasonable amount of time, the solving algorithms have to be numerically stable and have to deliver results at reasonable expense.

This master thesis investigates the parallelization of physical problems by the use of multi-core processors and computer clusters. Therefore, a flexible software library to parallelize the computation of existing and new physical problems is developed. In order to ensure the adaptability of the library to many kinds of problems, it is implemented in an object-oriented way in the programming language Fortran. This programming language, which supports the object-oriented programming since Version 2003, is commonly used to solve mathematical and physical problems.

The first physical problem to which the parallelization library is applied is the code NEO-2 of the Institute of Theoretical and Computational Physics of the Graz University of Technology. This code computes results which are used to describe neoclassical transport in plasmas. The parallelization of NEO-2 is not only a test case of the implemented library, but also demonstrates that parallelization of software allows to solve problems which could not be computed before due to long calculation times.

Contents

1 Computational physics **1**

1.1 Introduction . . . 1

1.2 Parallel computation . . . 2

1.2.1 Introduction . . . 2

1.2.2 Software requirements . . . 3

1.2.3 Hardware requirements . . . 5

1.3 Computation models . . . 6

1.3.1 Introduction . . . 6

1.3.2 Message Passing Interface . . . 8

1.4 Conclusion . . . 13

2 Analysis of the computational problem **15**

2.1 Introduction . . . 15

2.2 Technical information of NEO-2 . . . 15

2.3 Work flow of NEO-2 . . . 16

2.3.1 Initialization part . . . 16

2.3.2 Core part . . . 18

2.3.3 Finalization part . . . 21

2.4 Conclusion . . . 21

3 Object-oriented analysis **23**

3.1 Introduction . . . 23

3.2 Introduction to objects and classes . . . 23

3.2.1 Class declaration . . . 23

3.2.2 Inheritance . . . 24

3.3 Analysis of the parallelization library . . . 26

3.3.1 Definition of requirements . . . 26

3.3.2 Definition of classes 26
3.3.3 Class diagram of the library 31

4 Object-oriented design 33
4.1 Introduction 33
4.2 Flow charts 33
4.2.1 Scheduling process 33
4.2.2 Client process 35
4.3 System architecture 36
4.3.1 Programming language 36
4.3.2 Tools for the implementation process 36

5 Implementation and testing 39
5.1 Introduction 39
5.2 Numerical integration 39
5.2.1 Problem analysis 39
5.2.2 Adaption of the parallelization library 41
5.2.3 Testing 43
5.2.4 Detailed runtime analysis by profiling 56
5.3 Matrix chain multiplication 60
5.3.1 Problem analysis 60
5.3.2 Adaption of the parallelization library 60
5.3.3 Testing 65

6 Parallelization of NEO-2 73
6.1 Introduction 73
6.2 Analysis of the computational problem 73
6.3 Adaption of the library 76
6.3.1 Definition of the work units 76
6.3.2 Implementation 78
6.4 Performance measurements 79
6.4.1 Definition of the test case 79
6.4.2 Evaluation of the performance 79
6.4.3 Further analysis of the speedup 83
6.4.4 Conclusion of the performance analysis 84

6.5 Verification of the results 84

7 Use of the library 89

7.1 Introduction 89
7.2 Technical information 89
7.2.1 Compilation of the library 89
7.2.2 Source code 90
7.3 Adaption to a specific problem 90
7.3.1 Work units 90
7.3.2 Scheduler 92

8 Conclusion and outlook 95

Acknowledgments 96

A Integration example 99

A.1 Main program 99
A.2 Integration module 101
A.3 Specialized work unit 103
A.4 Adapted scheduler 106

B NEO-2 configuration 109

B.1 Content of neo2.in 109

References 113

1 Computational physics

1.1 Introduction

In this master thesis a library for the parallel computation of physical problems is developed in Fortran 2003. The first purpose of this library is the parallelization of the code NEO-2. This demonstrates the benefits of parallel computation and is also a test to stabilize the library. NEO-2 is a Drift Kinetic Equation solver for neoclassical transport in plasmas based on the method of field line tracing [1].

Since this library should not only be used for this special problem, it is implemented in a generic object-oriented way, so that other physical problems can be solved with little adaptions to the existing code. Therefore, the first part of this work describes the development process of the library which is then applied to plasma physics problems. The aim is to ensure a high degree of flexibility for future applications. After this process it is possible to deliver results which could not be computed so far in a sequential way. The parallelized code offers higher performance by using multi-core processors or computer clusters.

The first chapters of this thesis describe the analysis process of the problem and the implementation of the parallelization library. In order to explain the current solving mechanism of NEO-2, a short description of its working principle is given in Chapter 2. In Chapter 6 the performance of NEO-2 after the parallelization is analyzed. To ensure a straightforward implementation for other problems and codes, a user's manual of the parallelization library is provided in Chapter 7.

1.2 Parallel computation

It is easier to pull a heavy chariot with many oxen,
then growing one giant ox [2,Translated from German].

This quote tells in a descriptive way the reason for developing parallel computers instead of fabricating single-core processors with hundreds of Gigahertz. Single-core machines have physical borders which limit the performance, e.g., the speed of light or issues of heat transport [2].

1.2.1 Introduction

During the last years a new branch called Computational Science has been formed somewhere between theoretical and experimental physics, which can also be noticed in the name of the Institute of Theoretical and Computational Physics of the Graz University of Technology. The trend to have a large number of standard PCs and using them as one parallel computer resulted in the use of those distributed computers for complex tasks, for example SCAN (Super Computers at Night). When using standard PCs as a cluster, a fast connection between them for data transport has to be established. A common communication technique used by parallel algorithms is the Ethernet, while some just use E-Mails for transmitting the data between the nodes. The generic rule is that faster processors need faster connections [2].

A modern connection type between the nodes is called InfiniBand. This is a high bandwidth with low latency interconnection method which makes RDMA (Remote Direct Memory Access) between nodes possible. Beside the hardware, the algorithms also have to be prepared for parallelization. Three sources exist to gain parallelism [2, 3]:

- Physics
  The independence of physical processes, for example two non-interacting events.

- Mathematics
  The independence of mathematical processes, for example the separation of integration domains.

- Programmatic
  The independence of parts of the algorithm itself, for example loops with independent iterations.

There exist some compilers which can automatically parallelize sequential algorithms, but they are limited in their applicability. Therefore, the best performance can only be achieved when the programmer develops the parallel algorithm himself. One library to support the parallelization of a software is the Message Passing Interface, which is described in a later section. To analyze the change of the runtime of a parallelized code, Amdahl's law should be considered [2, 4].

1.2.2 Software requirements

Amdahl's law implies that a program consists of two parts. The sequential part, which is not parallelizable, and the parallel part. The general speedup can be described by Equation (1.1) from the article of Sun and Chen [4].

$$S_{\text{Amdahl}} = \frac{1}{(1-f) + \frac{f}{n}} \tag{1.1}$$

f is the relative parallelizable part of the program and n the number of processes. For a purely sequential program $n = 1$. In Figure 1.1 four different curves for f are plotted against the number of clients. This shows the consequence of even small sequential parts on the maximum speedup:

- $f = 1$ (red line)
  This is the ideal behavior of the runtime of a code with no sequential parts. In this case $S_{\text{Amdahl}} = n$ where the speedup is directly proportional to the number of processes.

- $f = 0.99$ (blue line)
  An interesting effect is seen, if just 1 percent of the code is not parallelizable. This seems to be a very good condition, but the blue line shows a saturation effect at about 256 cores. Therefore, running this code on many thousands of cores will not result in a significantly higher performance.

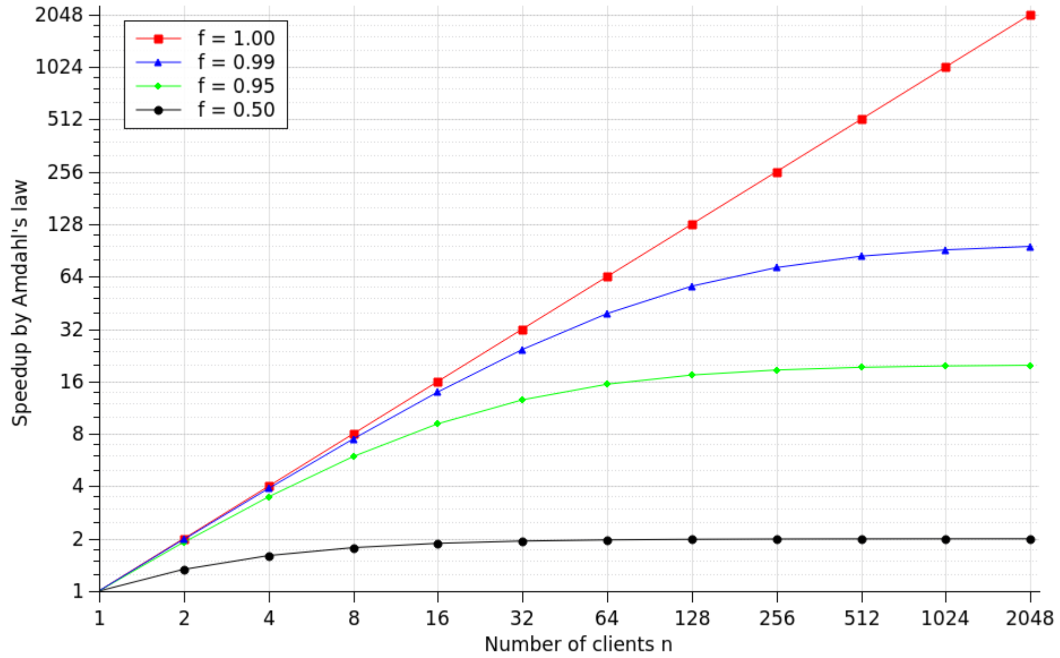


Figure 1.1: Examples for the speedup by Amdahl's law

- $f = 0.95$ (green line)
  If only 5 percent of the code is not parallelizable, then the maximum speedup will be 20.

- $f = 0.50$ (black line)
  When f approaches 0.5 the maximal speedup will be two and the use of a large amount of processors just results in a waste of energy.

As a result, there is a balance between the effort of parallelization and the benefit of runtime speedup. This balance has to be optimized. The maximum speedup can be calculated by taking the limit $n \to \infty$ in Equation (1.1) to obtain Equation (1.2) [4]. The resulting equation implies that the sequential part of the program $(1 - f)$ defines the value of saturation of the maximum speedup. In Chapter 6 it is shown that the definition of f for an existing code could be nontrivial.

$$\lim_{n \to \infty} \frac{1}{(1-f) + \frac{f}{n}} = \frac{1}{1-f} \tag{1.2}$$

1.2.3 Hardware requirements

In the book of Gropp et al. [2] it is stated that software and hardware have to be prepared for parallel computing. As mentioned before, the trend is towards the connection of standard PCs to build a cluster and use them for daily purposes, e.g., running office applications, while parallel computations run in the background. This is the way the cluster of the Institute of Theoretical and Computational Physics is managed. Several workstations are available for desktop applications and running parallel jobs in the background. Connected by Ethernet, the data transmission between the nodes is not as fast as InfiniBand, but adequate for most purposes.

In an early stage of the development of this master thesis only the local multi-core processors were used, without transporting data over the network. The available test system hardware was an Intel®-i7 processor with four cores, each one with two threads, realized with Hyper-Threading - Technology. The article of Leng et al. [5] describes the influence of Hyper-Threading in computer clusters. It is stated that the improvement of the performance is related to the cluster configuration and strongly related to the kind of the code.

The Linux command `lscpu` delivers Program Output 1 on the test system. The output means that the operating system gets the information that eight CPUs are available, while four of them are real cores and the other four emulated ones. In the next stage of development the used cores of the parallelization library were spread over nodes connected by Ethernet. Additionally, for performance tests of NEO-2 the Vienna Scientific Cluster - 2 has also been used.

Program Output 1 Hardware information of test system

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
CPU(s):                8
Thread(s) per core:    2
Core(s) per socket:    4
CPU socket(s):         1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 26
Stepping:              5
CPU MHz:               3066.824
```

1.3 Computation models

1.3.1 Introduction

There are two major ways to deal with the memory of the running processes in a parallel environment [2].

- **Shared-Memory-Application**
  This simple model allows every process of the parallel environment to access the same memory space. Though it definitely has advantages that processes can read results from other ones, unfortunately especially writing operations have to be coordinated very well. In such cases lock mechanisms to ensure that two or more processes are not trying to write to the same variable in the memory at the same time are needed. Some high level programming languages can hide these locks from the user. In Figure 1.2 the concept of the model is shown. It has to be considered that the processes have to run on the same physical machine to gain access to the same memory. Because of the difficulty to build computers with more than 20 to 30 processors this method has its definite limits [2].

  One famous implementation of this model is OpenMP - not to be confused with OpenMPI. OpenMP is an interface for shared-memory parallelism for C, C++ and Fortran codes. A variation of the shared-memory model is given, if processes have a local memory and can also access the shared memory [2, 6].

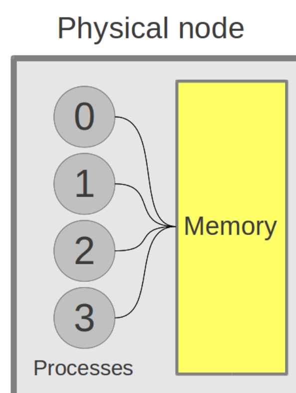


Figure 1.2: Shared-memory model

- **Local-Memory-Application (Message Passing)**
  The case where every process has its own memory space in a parallel environment is realized in a Message Passing system. This concept is shown in Figure 1.3. The only way to share data between processes is sending and receiving messages. That is where the name Message Passing comes from. An important attribute of this model is that the transmission of data from the local memory of one process to the local memory of another process involves operations on both sides. That means that the sender has to call the sending operation and the receiver has to call an appropriate receiving operation. This model has the advantage that the programmer does not have to care about independent variables for every process. Because of the local memory approach there is no need for mechanisms that prevent clients from writing to the same memory space. Otherwise, this type of memory behavior has disadvantages when different clients need the same data, which can happen for initial conditions of physical problems. The handling of this case is discussed in Section 7.3. The most frequent reason for software errors in shared-memory applications is the unintended overwriting of memory space. This problem is prevented in the Message Passing model because of the own memory of each process [2].

  A specification for Message Passing is the Message Passing Interface, in short MPI. The use of MPI makes it possible to bind processes running on different physical machines through a network connection. This is possible, because it does not matter if a send-receive-operation passes data between two cores of a multi-core machine or two cores of different nodes in the network. This is a significant advantage and can turn many connected standard PCs into one supercomputer. The programmer does not have to care about if communicating processes are located on the same physical machine or not, because the required MPI commands are the same [2, 7].

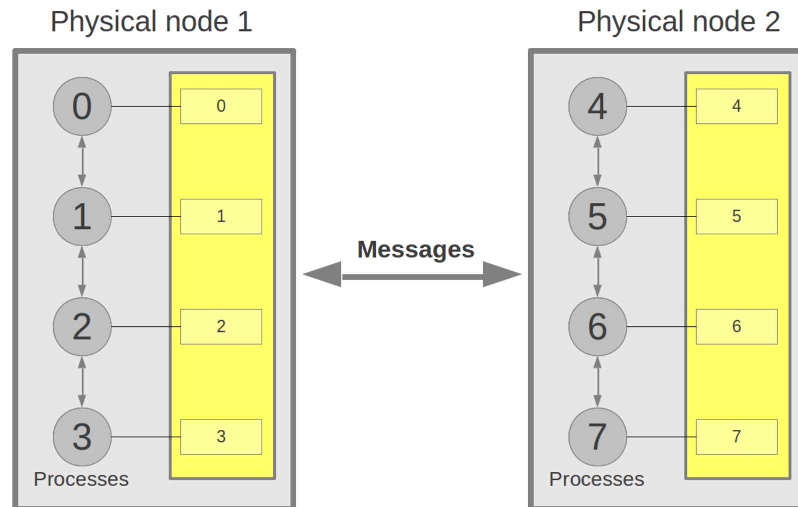


Figure 1.3: Message Passing model

Several other models exist, e.g., remote-memory operations. These operations make it possible to access the memory of a remote machine without calling a method at the remote side. This method requires special hardware and is therefore not described in detail in this thesis [2].

1.3.2 Message Passing Interface

The following sections contain basic information of the official MPI-2.2 report from the Message Passing Interface Forum [7].

Introduction

It is important that MPI is a specification, not an implementation. The implementation used for this master thesis is OpenMPI, which will be explained later. The specification standard includes, e.g., point-to-point communication, data types, collective operations, process groups, language bindings for Fortran, C, C++, and parallel file I/O [8].

Communication between processes always has to take place if a part of the address space of one process has to be copied to the address space of another process. It has already been explained that such operations involve both processes.

Semantic terms

In order to introduce a few MPI methods, especially those for point-to-point communication, a few semantic terms have to be explained. The MPI specification defines blocking and non-blocking operations [7,Chapter 2]:

- Blocking operations
  In order to send a message, the user has to pass the data which should be transmitted to the sending method of MPI. A method is defined as blocking, if it waits until the user's data can be reused, e.g., overwritten or deallocated, after the method returns to the main program. It is relevant to know that this does not imply that the data have already been transmitted to the receiver, they remain in a buffer until the receiver calls an appropriate receive function.

- Non-blocking operations
  A non-blocking method may finish before the user is allowed to modify the data to be sent. This means that the method pushes the data into a buffer while the main program continues. As a consequence, other MPI methods are required to check if a current ongoing process has already been finished. By non-blocking routines it is possible to write code which offers good performance, because the program continues its execution while sending a message.

In addition to blocking and non-blocking methods there are local and non-local ones. Shortly explained, the finishing of a local procedure only depends on the local executing processes and is not linked to an action of another process [7,Chapter 2].

Point-to-point communication

In order to understand the mechanism of MPI, it is necessary to explain the simplest form of Message Passing, the point-to-point communication. Every data exchange between processes has to be fulfilled by message-operations. The simplest kind of these operations are the basic send- and receive-methods [7,Chapter 3].

When calling the standard sending operation `MPI_SEND` the following input arguments have to be passed:

- Send buffer
  The initial address in the memory of the data which should be sent. It can be a single variable, an array, a pointer to an address in the memory or a specially prepared buffer by a packing algorithm. The mentioned packing algorithm is used in this thesis to transmit complex data types.

- Count
  The number of elements in the send buffer, e.g., 1 for a scalar.

- Data type
  The data type of the elements in the send buffer, e.g., `MPI_INTEGER`.

- Destination
  The target of the message (called rank in the MPI specification) also has to be passed. The standard send command is based on a point-to-point mechanism, i.e., only one receiving rank is allowed. The rank is an unique integer number starting with zero for the master process.

- Tag
  To categorize the type of the message a user-defined integer, called tag, has to be passed. The tag can be used to filter incoming messages or to allocate space before receiving the data.

- Communicator
  A communicator can be used to separate the parallel processes into groups. The predefined communicator `MPI_COMM_WORLD` defines the set of all processes.

The client process will start the receiving as soon as the associated MPI receive method `MPI_RECV` is called with the same arguments as on the side of the sender. Some constants, e.g., `MPI_ANY_TAG` or `MPI_ANY_SOURCE`, are defined for disabling the tag- or the source-filtering. In such cases, the additional argument of the receive method, called status, can be used to probe the tag or the source of the message. It is important to understand that sending a message to a client

will not cause an interruption on the receiving side. The message will wait in a buffer and the programmer decides when it will be received by calling the appropriate commands [7,Chapter 3].

Communication concepts

In addition to the transmitted data, every MPI message contains a description of the sender, the receiver and the tag. In MPI it is possible to check if a message is waiting to be delivered before actually receiving it. This operation is called probing. After probing for a message with specific source, tag and data type arguments, a following receive command, called with the same arguments, receives exactly the message, which has been probed before. This can be used to get the length of the message in order to allocate space before receiving it or to perform other operations while waiting for a message. The process running the master instance has rank zero. The rest of the nodes is enumerated from 1 to $n-1$, where n is the number of nodes in the parallel environment [2, 7].

Data types

MPI supports basic data types of C and Fortran, like integers, doubles, characters, and some others. However, complex data types, e.g., structures, sometimes also have to be transmitted. For this case a functionality, called derived data type, exists to construct user-defined types [7,Chapter 4].

When analyzing NEO-2 it has been discovered, that there are structures which contain allocatable arrays and matrices. This means that the number of elements can change during the runtime. In this master thesis no satisfying method has been found to use MPI derived data types with this requirements of dynamically growing or shrinking arrays. Therefore, the MPI pack and unpack routines have been studied to manage these requirements. The packing mechanism provides functionality which is not provided by MPI otherwise, e.g., the sending of complex data structures or the development of libraries which lie on top of MPI. Packing the data lets the user define the layout of the send buffer of messages. The basic principle is that the sending operation gets split into two single operations. One for the preparation of the buffer and another one for the transmission of the packed buffer. If data is packed to a buffer,

instead of sending them directly, performance issues may arise. During this work it has been observed that the pack algorithms offer good performance and that they do not influence the program's runtime significantly [7,Chapter 4].

One-sided communication

For point-to-point communication both, the sender and the receiver, have to call appropriate methods. A noteworthy technology is the Remote Memory Access, which needs actions on only one side of the transmission. This is a special form of communication mechanism, which extends the usual functionality of point-to-point communication methods. Remote Memory Access enables the sender or the receiver to define all communication parameters of the transmission process. That means the sender can define the memory address, where the receiver should store the communicated data. All involved processes have to create a so-called window in their memory. This window contains the data that can be remotely changed. Three different communication calls are supported [7,Chapter 11]:

- `MPI_PUT`
  Put transfers data from the sender to the remote process. In such cases, no action is needed at the receiving side.

- `MPI_GET`
  This function is similar to the previous one, but works in the other direction. The method passes data from a remote process to the process which called the function.

- `MPI_ACCUMULATE`
  This combines the origin data of the calling process with the data of the remote side, instead of overwriting them. This can be used for creating sums or other mathematical operations involving many processes.

Before the data of the buffer can be accessed, a synchronization call has to be performed. This synchronization call is a barrier-method, i.e., all clients have to wait until the data have been updated and are ready to be accessed.

Additional features

This short summary of the first chapters of the official MPI report [7] and the book of Gropp et al. [2] only gives an overview of the major features of MPI. More features as explained here, like process grouping, collective operations, profiling, file I/O, process topologies, dynamic creation of processes, are provided by MPI. However, since they are not needed to understand the working principle of the library developed in this thesis, they are not explained in detail. A short overview of some additional features is given in the following paragraphs.

Collective operations can be broadcast-, scatter- or gather-processes. They involve more than two ranks and support calculations like minimum and maximum queries. Virtual topologies bind ranks on a grid or some other sort of structures. This can be understood as grouping mechanisms. As mentioned above, debugging Message Passing applications is easier than debugging shared-memory applications because of the local memory of each rank in Message Passing systems. With profiling libraries, which are used in this thesis, send and receive processes can be recorded. The recorded data can be converted to figures which view the communication ways [2].

One feature, not included in the first revision of MPI but in the second, is the dynamic creation and management of processes. Although there is a defined number of initially created processes at the start of a MPI application, there is support to create new processes during the runtime. MPI tries not to overrule the responsibilities of the operating system because it is a communication specification and not a concept for process management. With the provided commands it is possible to start new processes and to connect them with the already existing ones for sending and receiving messages [7,Chapter 10].

1.4 Conclusion

The sequential part of a parallel program limits the maximum speedup. This is shown by Amdahl's law in Equation (1.1). There exist several other speedup models based on Amdahl's law, e.g., Gustafson's law which implies that a larger problem can be solved in the same time on a larger parallel environment [4].

Two different ways to deal with the memory in a parallel application are the shared-memory approach and the Message Passing model. In the case of a shared-memory-Application every process has access to the same memory. This approach can cause software errors when two processes try to write to the same memory space at the same time. In a Message Passing model every process has its own memory space. The only way to exchange data is by passing messages between the processes. This makes it easier to keep track of the data because every sending and receiving operation can be recorded by a profiling tool. A specification of the Message Passing model is the Message Passing Interface (MPI) which is used in this master thesis.

2 Analysis of the computational problem

2.1 Introduction

In order to analyze the generic problems of the parallelization of sequential codes, the program NEO-2 is studied in this chapter. Based on the results of this process, the requirements for the parallelization library are determined. The concept is to develop a more general library and to integrate it then into NEO-2. To parallelize NEO-2 directly would be a simpler option but that would not be a sustainable way for future applications. The library should also be used in codes developed to solve other physical problems. Therefore, the parallelization of NEO-2 is one of the last steps performed in this work and it is well-planned.

2.2 Technical information of NEO-2

NEO-2 is a program that has grown over the time to about 50,000 lines of code (external libraries not included). As mentioned in Section 1.1, the program is a solver for the Drift Kinetic Equation based on the method of field line tracing. The code is written in the programming language Fortran. It follows the Fortran 90 standard, but there still exist some parts written in Fortran 77. The parallelization library to be developed has to be flexible enough so that only minor code changes in NEO-2 have to be done because of the complexity of the code.

Testing scenarios showed that NEO-2 is a memory consuming code. Various program runs have been analyzed and a requirement of many Gigabytes of

memory was found. The solving routines of NEO-2 are based on solving of sparse linear systems of equations. This is done by using high performance libraries like SuperLU or SuiteSparse [9, 10]. It has been shown that for this special kind of problems SuiteSparse is the better choice resulting in shorter runtimes [11]. Due to the fact that the libraries are written in the programming language C, there exist wrapper functions to establish the communication to the Fortran code.

2.3 Work flow of NEO-2

The code can be split into three major parts as shown in Figure 2.1. The sequential work flow is indicated by the arrows between the yellow boxes. These parts are described in the following sections.

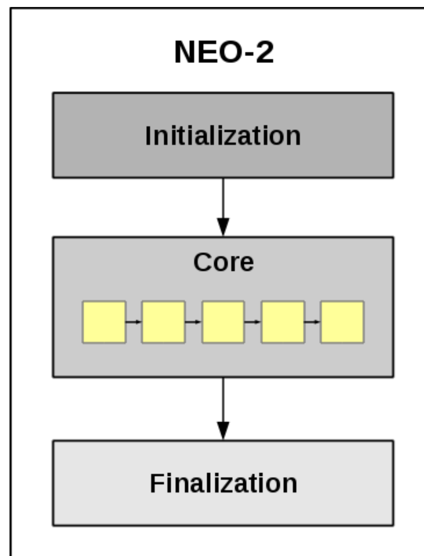


Figure 2.1: Program parts of NEO-2

2.3.1 Initialization part

Operating principle

In this part the program computes a large amount of initial data. At first, the parameters of a magnetic field line are calculated. Then, this field line is

discretized in parts. These parts are called propagators which finally contain all pertinent physical information for a given part along the field line. In order to manage the data, the program creates linked lists of elements which are connected by a tree structure. A schematic diagram of this data structure is shown in Figure 2.2. In this figure it can be seen that field periods are children of field lines. The field periods are again separated into propagators.

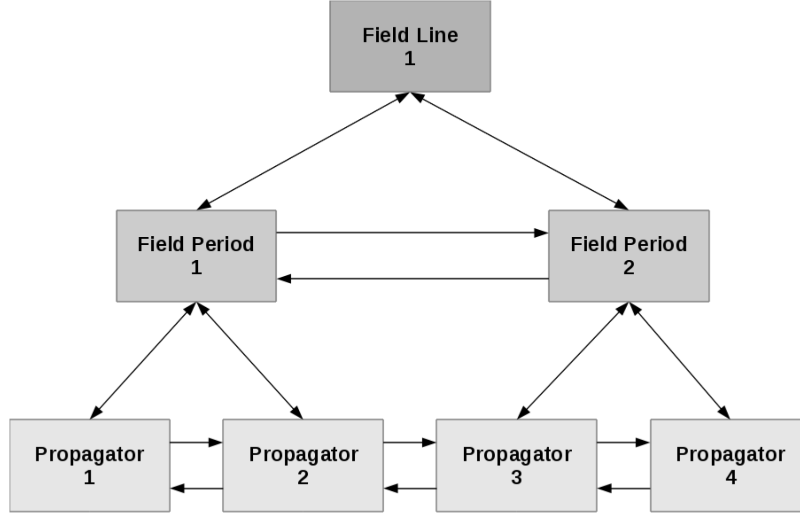


Figure 2.2: Data structure of NEO-2

Sequential parts on clusters

If a parallel code runs on a computer cluster the computing nodes may have to be reserved before the program starts. In this case the reserved number of cores during runtime remains the same whether there is a load or not. As a consequence, the accounted amount of time can be calculated by Equation (2.1). The time t_{cost} is accounted by the cluster provider for doing the computation. t_{wall} is the so-called wall clock time which represents the elapsed time during the calculation. In contrast to t_{cost}, the CPU time t_{CPU} is only accounted if there is a load on the processor.

$$t_{\text{cost}} = n \cdot t_{\text{wall}} \tag{2.1}$$

If the accounting model of the cluster is based on this equation, then it does not

matter if only one core calculates the initial data or if the same calculation is done by every client at the same time. The computation of initial data on each client at the same time has the advantage that packing and sending operations of complex initial data can be avoided.

It has been determined that this initialization part of the code NEO-2 is not parallelizable at reasonable expense.

2.3.2 Core part

Properties of propagators

In the core part the propagators are solved and joined together. Propagators have algebraic group properties which are described in the book of Jänich [12]. It is stated that a group consists of a set G and an operation $(*)$. The following axioms for the set and the operation have to be fulfilled:

- Associativity
  $(ab)c = a(bc)$ for all elements $a, b, c \in G$.
- Existence of a neutral element
  There exists one element $e \in G$ with $ae = ea = a$ for all elements $a \in G$.
- Existence of an inverse element
  For every element $a \in G$ there exists one element $a^{-1} \in G$ with $aa^{-1} = a^{-1}a = e$.

These group properties can be compared to the properties of propagators of NEO-2. There exist a set of propagators P and a joining operation $(*)$ [13]. The operation is defined by Equation (2.2).

$$P_1 * P_2 = P_{1,2} \tag{2.2}$$

The associativity axiom is fulfilled because of Equation (2.3).

$$P_1 * (P_2 * P_3) = (P_1 * P_2) * P_3 \tag{2.3}$$

There is no need for a neutral element in the computation process of NEO-2, but it would be possible to define one [14]. Because of the fulfilled associa-

tivity axiom and a non-commutative operation, the problem is, in some sense, equivalent to a matrix chain multiplication.

Operating principle

In Figure 2.3 the linear work flow of NEO-2 is shown. Time proceeds from top to bottom, where the dotted lines mark the borders of different program states. Gray propagators have been prepared, which means that all information required to solve them is given. The yellow boxes represent solved propagators, which need more memory and are ready to be joined.

The first state of the program represents the state after the initial phase. The number of propagators is known and every propagator contains all information to be solved. Then, the program starts solving the first propagator. This is highlighted by changing the color from gray to yellow at Propagator 1. The algorithm detects no join-able propagators and continues solving the next propagator, which is Propagator 2. At this point two neighboring elements have been solved and can be joined to a new propagator object. For reasons of clarity the new propagator gets the tag 1_2 to indicate its sources. Because of the properties of algebraic groups the resulting propagator is a member of the group and contains all relevant physical information of its origins. As a consequence, the two parent objects are deallocated and the resulting propagator is a new join-able object. Figure 2.3 also explains the built-in memory saving mechanism of NEO-2. At every program state at most two fully computed propagators exist in the memory.

Option for parallelization

As stated in the PhD thesis of Leitold [1] the core part of the program can be parallelized. This can be explained by the associativity of the algebraic group:

$$\underbrace{((P_1 * P_2) * P_3) * P_4}_{\text{Sequential}} = \underbrace{(\underbrace{P_1 * P_2}_{\text{Client 1}}) * (\underbrace{P_3 * P_4}_{\text{Client 2}})}_{\text{Parallel}}$$

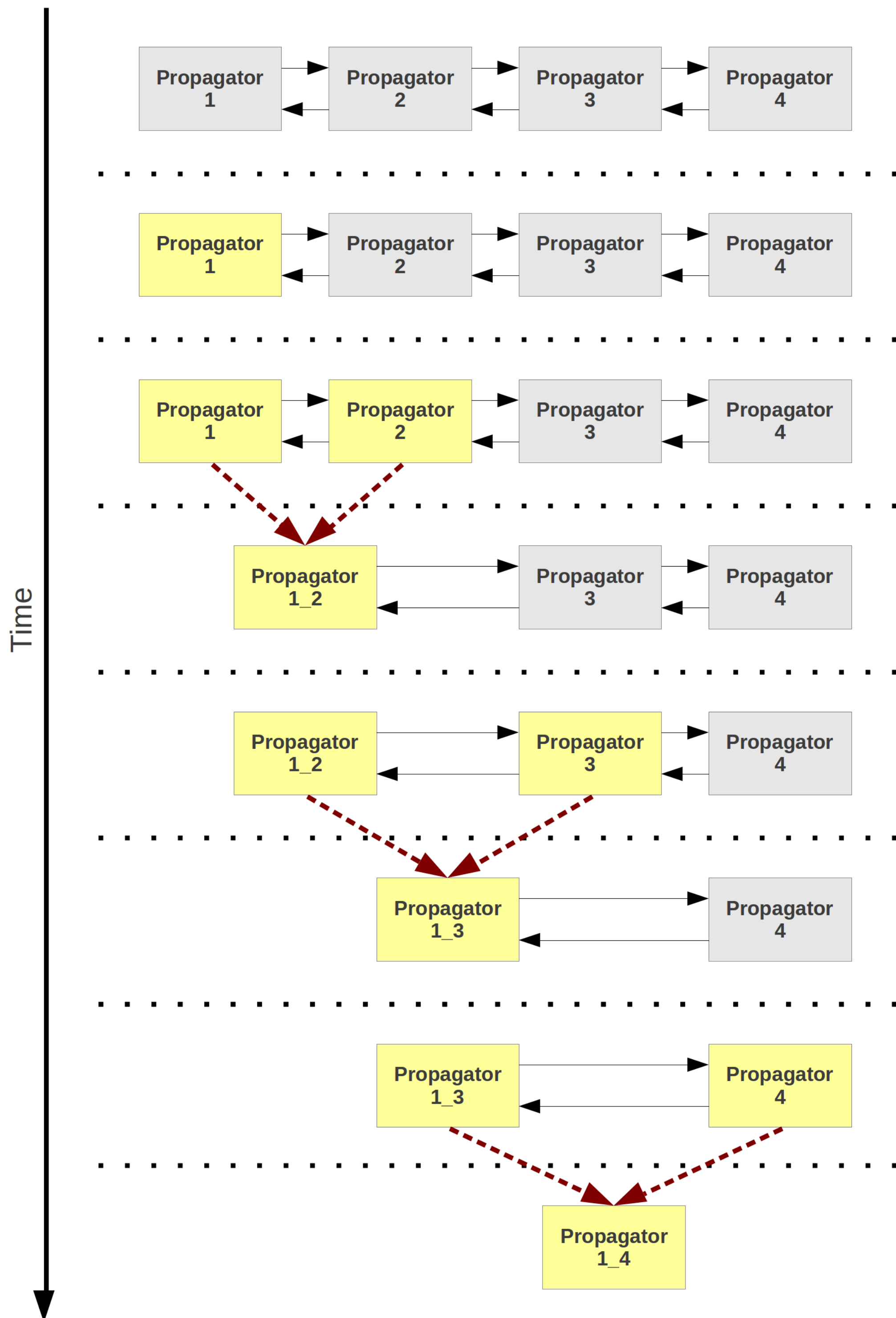


Figure 2.3: Concept of solving and joining propagators

The joining of P_1 and P_2 can be done at the same time as the joining of P_3 and P_4. This parallelization concept is used in this thesis to parallelize NEO-2 and is described in Chapter 6.

2.3.3 Finalization part

The core part of the program continues until only one propagator is left. At this point the code has to perform final sequential computations, e.g., closing the magnetic field line by joining the last propagator with itself and writing the results to files.

2.4 Conclusion

The core part of the program can be parallelized at reasonable expense because of the associativity of the joining operation of propagators. The initial part will be done on all clients at the same time to avoid the packing and sending of the created complex data structure. The finalization part will only run on the master process to summarize the results, e.g., to close the magnetic field line and to write files which contain the physical results.

3 Object-oriented analysis

3.1 Introduction

NEO-2 has been analyzed in order to find an appropriate way to parallelize this specific code. The library to be developed in this thesis is planned to be generic and easily applicable to other codes. Therefore, the development process of the parallelization library is based on the method of object-oriented programming.

The object-oriented programming is separated into different development phases. It consists of an analysis phase, a design phase, and an implementation phase [15].

3.2 Introduction to objects and classes

The object-oriented analysis (OOA) separates the problem into objects which can communicate with each other. These objects can correspond to objects in the real world (e.g. particles) or they can describe programmatic objects (e.g. linked lists). They are created during the program's runtime and are defined by their state and functionality [15].

3.2.1 Class declaration

To create objects during the runtime (also called instances), classes have to be defined. Classes define the structure and the behavior of the objects. A class declaration consists of three parts [15]:

- Name
  The class name is an unique identifier for the class.

- Attributes
  The attributes represent the data of the class. The state of an object is defined by the values of its attributes.

- Methods
  The methods can access the attributes and are used to change them or to interact with other objects. A method has only access to the attributes of the object it is called from.

Example A class Particle may consist of the attributes Position, Velocity and Mass. To change the position, the method move() is created. In Figure 3.1 the UML (Unified Modeling Language) class diagram of this example is shown. The UML is a standard for designing software processes [16]. A class is denoted by a rectangle with three sections. The first one denotes the class name, the second one includes the attributes and the third one shows the methods. The locks on the left side indicate if the attribute or the method can be accessed only by methods of the same class (closed lock) or by all routines (open lock). This can be used to force the user to call an appropriate method to change the value of an attribute, instead of changing the variable directly. During the program's runtime many objects may exist with different values of their attributes.

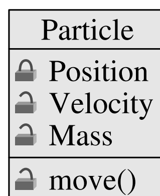


Figure 3.1: Example for a class Particle

3.2.2 Inheritance

An important feature of object-oriented programming is the possibility to inherit classes. The attributes and methods of the parent class are passed to the children, which can have additional attributes and methods. In terms of programming, subroutines accepting an object of the parent class as input argument, will also accept the children of the class. Therefore, it is possible to

write common routines for generic classes which will also work for their children. Those children can be specially designed for a problem [15].

Example To continue the example from above, the class Particle is inherited to obtain the class ChargedParticle. This class inherits all attributes and methods of its parent and has the additional attribute Charge. In order to control the new attribute, the method changePolarity() is added. To show that a child class can also be a parent class, two additional classes (QuasiParticle and VolumeParticle) are added. The class QuasiParticle gets a method to check the validity of the Pauli principle and the class VolumeParticle gets a method to check for a collision with another particle. In an UML class diagram the path of inheritance is shown by an arrow pointing from the child to the parent. An array containing elements of the class Particle, can also contain objects of the classes ChargedParticle, QuasiParticle, and VolumeParticle. The UML class diagram for this example is shown in Figure 3.2.

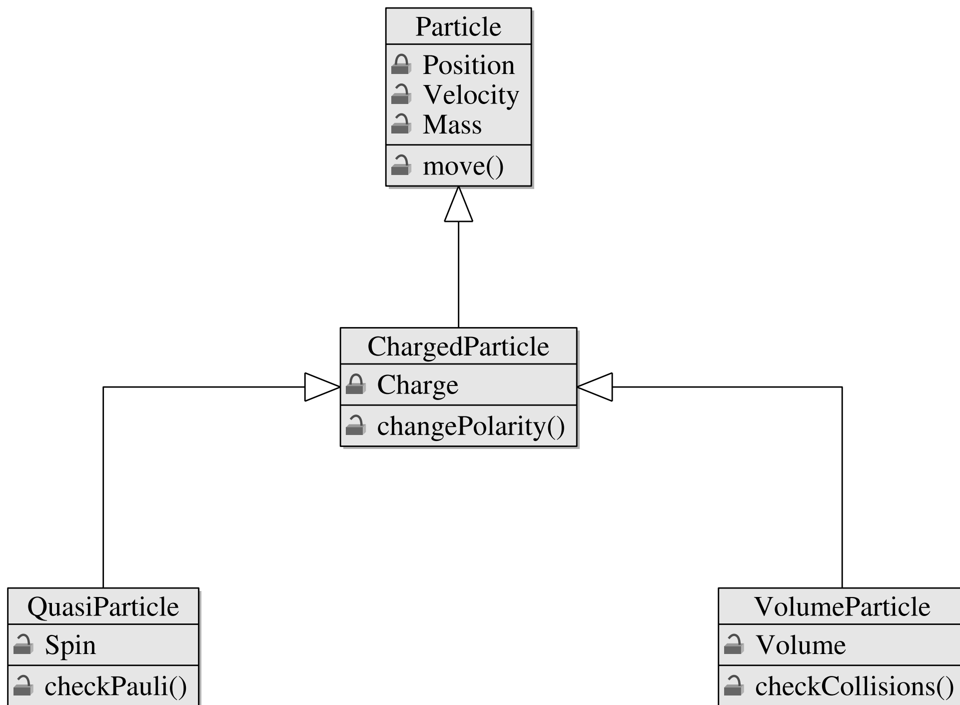


Figure 3.2: Class diagram example with inheritance

3.3 Analysis of the parallelization library

3.3.1 Definition of requirements

The principles of the object-oriented analysis are used to analyze the requirements for the parallelization library to be developed. The key features are defined by the following items:

- It should be possible to define work units, which process different tasks.
- One work unit should be responsible for exactly one task.
- The work units should be automatically distributed among different compute nodes by a scheduler.
- The library should offer the possibility to adapt the behavior of the scheduler through inheritance of the standard scheduler.
- The communication with MPI should be hidden from user.

3.3.2 Definition of classes

The requirements of the library determine the necessary classes, which are defined in the following sections.

Class Workunit

The class Workunit defines a computational task which can be processed separately from other tasks. It should be suited for nearly every kind of problem. Therefore, this class does not imply what the task should be or which data it will contain. The class is defined as an abstract class, so it is not possible to create objects of it during the program's runtime. In order to create a work unit object, the class has to be inherited to get a child class which is specific for the problem. This child defines the task by overwriting the processing subroutine of the generic work unit and contains the data used to solve the task by adding attributes. The scheduler distributes work units among the clients and forces them to run the specific processing routines.

The class Workunit is inherited from the class Packable. The principle of packable classes is that they can pack and unpack themselves to get transmitted. Work units contain data used to run the processing routine and data that represent the results. Because of the problem independent approach, the user has to define the data that are necessary for the work unit by defining the pack routine. This offers the possibility to send the packed work unit to a client. The provided unpack routine of the parent class Packable, which also has to be overwritten by the user, rebuilds the work unit on the client.

In order to provide a dependency model, the class Workunit contains a list of required work units. The scheduler only dispatches work units with fulfilled dependencies, i.e., if all required work units have already been computed.

Example To explain the class Workunit an example of a simple numerical integration is given. A function should be integrated between -1.0 and 1.0. In order to create the integration task, the class Workunit is inherited to get a child class which contains the function to be integrated and variables for the definition of the integration domain and the accuracy of the calculation. The new class is called SimpleIntegrate.

If the computation should be done on two clients, then two objects of the class SimpleIntegrate have to be created. The first one does the integration between -1.0 and 0.0 and the second one between 0.0 and 1.0. The scheduler will distribute these two work units among different clients at the same time. Thereby, the program doubles its performance, when neglecting the time of the send and receive processes and other time consuming processes. After both work units are processed, the results are summed up to get the solution of the full problem.

This example is depicted by the UML class diagram in Figure 3.3. The attributes a and b of the class SimpleIntegrate define the bounds of integration. The number n is the number of iteration points in the integration domain.

Class MPIProvider

An important feature of the library to be developed in this thesis is to hide the native MPI commands from the user by writing appropriate wrapper functions.

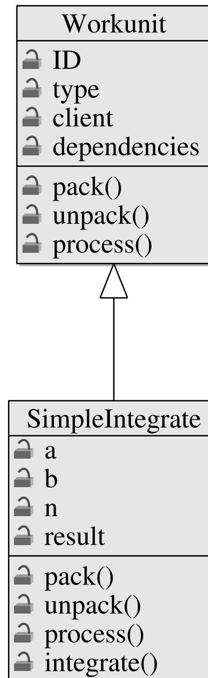


Figure 3.3: Example of inheritance of the class Workunit

The class MPIProvider is a layer between the main program and the MPI implementation. It is defined as a singleton class, i.e., only one global object instance is allowed. Through the use of a global singleton object, every program part can run MPI commands by calling the subroutines of the MPIProvider. If function arguments or the behavior of a specific MPI command changes in future, e.g., due to updates, they only have to be adapted at one place in the program. There is no need to check the whole source code in order to find the places where the modified command has been used.

In addition to the wrapping of commands for MPI initializations, MPI send and MPI receive commands, the class MPIProvider manages the buffers for packing and unpacking of data. In Section 1.3.2 it is explained that the pack and unpack routines of MPI can be used for sending complex data types. Therefore, a buffer has to be prepared so that the data can be pushed into it and passed to a client. There exist generic commands to push variables of common data types into the buffer in order to make the packing of data simple for the user. Then, the data in the buffer are sent by a native MPI send command. The client that receives the message can identify from its tag that the message contains packed

data. To reconstruct the data on the client site, the class provides functions to read data from the received buffer.

Another reason for wrapping the MPI commands by functions of this class is the need to facilitate a monitoring option for every command. This is used in this thesis for performance analysis by profiling, as described in Chapter 5.

Class Scheduler

The class Scheduler manages and distributes work units. This is a generic class with some built-in functionality, but it can also be inherited by the user and be adapted for special problems. The scheduler contains methods to create work units and to add them to a list of waiting jobs. The main scheduling function iterates trough this list and sends the work units to the clients, which reconstruct and process the received work units. Three lists are used to manage the work units:

- `waitingWorkunits`
  Work units that have been added to the list and are waiting for a free client or for fulfilling their dependencies.

- `pendingWorkunits`
  Work units which have already been packed and sent to a client. A client can only process one work unit at one time, so the maximum number of elements in this list is the number of clients.

- `processedWorkunits`
  Work units which have been processed on a client. If a client indicates the completion of a work unit by a message, the scheduler moves this unit from the list of pending work units to the list of processed work units.

Before the scheduler can start the distribution of work units, an initial signal is sent to all processes. This signal triggers a special crafted work unit, called initial work unit, on every client. This can be used to define tasks which create initial conditions, e.g., global definitions needed by all following jobs. If all clients have done this job, the iteration over the waiting work units starts. The scheduler picks the current work unit from the list and depending on the

result of the dependency check, the work unit is either packed and sent or not processed in this loop iteration.

When creating a work unit, it can be defined if it should run on a specific client or not. This can be used for debugging purposes, but in most cases the scheduler should decide on which client the work unit will run. If the desired client is ready, which means that there are no pending work units of this client in the list, the scheduler packs the work unit. This is done by calling the user-defined pack routine. After this process, the work unit is sent to the client. As soon as the client receives the work unit, the unit is rebuilt by the unpack routine and processed by calling the user-defined processing method.

The distribution of the work units runs until all clients are loaded with tasks or no more work units are left. In order to ensure a fair work distribution between the clients, a load-balancing routine is provided. This routine can be overwritten by the user because every kind of problem may need a different load-balancing method. After running this routine, the scheduler waits for responses of the clients. After handling all client responses, the iteration process over the waiting work units starts again until all work units were moved from the list of waiting work units to the list of processed work units. Then, the scheduler executes an user-defined summarizing routine and quits the program.

Dispatcher

This class is a wrapper for the MPI send commands. Two children of this class are defined:

- Command dispatcher
  The command dispatcher is responsible for sending the command to process the initial work unit and for sending a stop signal to the clients as soon as all work units are completed. It is also used for sending messages to indicate the completion of a work unit to the scheduler.

- Work unit dispatcher
  The work unit dispatcher sends and receives work units and calls the methods to rebuild and process them on the client.

Both dispatchers are inherited from the class Dispatcher. They differ in the message tag, which is added to the envelope of the MPI message. With this approach two communication channels between clients exist.

Receiver

Each receiver corresponds to one dispatcher, so there is one receiver for work units and one for commands. One receiver listens for a message with a specific tag and reacts on the type of the message. If a command receiver reads the command to run the initial work unit, the appropriate method of the scheduler is called. If the work unit receiver gets a work unit, it is rebuilt to its original form and gets processed. This class is used for sending tasks from the scheduler to the clients, for exchanging work units between the clients, and for sending back processed work units to the scheduler.

3.3.3 Class diagram of the library

An UML class diagram has been created to show the relations between the defined classes from the section above. The resulting diagram is shown in Figure 3.4. For reasons of clarity, only important attributes and methods are listed in the classes. It should be noted that a class diagram is independent of the programming language, so it should be created before the implementation process. However, changes of the designed classes and relations are common during the implementation because of special features of the specific language or because some functionality might not be implemented in the compiler of this language.

In contrast to the example class diagram in Figure 3.2, new indication elements have been added, called associations. These elements indicate connections between classes, marked by lines between them. For example the class MPIProvider can have zero or more schedulers, but exactly one scheduler belongs to one MPIProvider. The numbers which denote these quantities are called multiplicities [16].

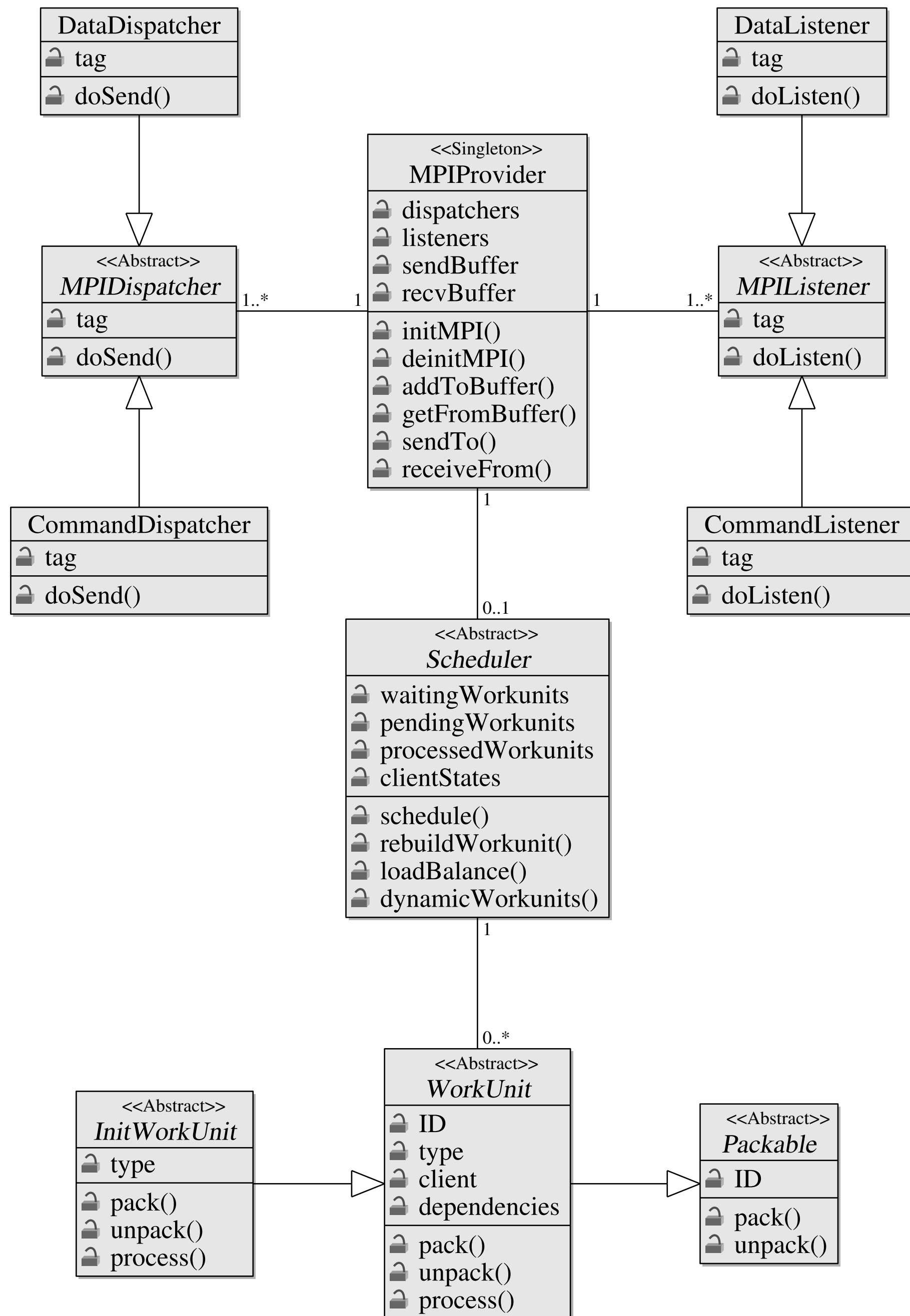


Figure 3.4: Class diagram of the parallelization library

4 Object-oriented design

4.1 Introduction

While the analysis phase described in Chapter 3 is independent of the programming language and the system architecture, in the object-oriented design phase of this chapter they have to be considered. The analysis phase defines classes and how they are related to each other, while the design phase determines how the processes work, e.g., explained by flow charts. That includes the definition of the programming language and the adaption to its special features [15].

4.2 Flow charts

This section contains two flow chart diagrams in order to explain the operation principles of the scheduler and the clients.

4.2.1 Scheduling process

The flow chart of the scheduling process is shown in Figure 4.1. The main loop iterates over the elements of the list of waiting work units. If the end of this list is reached, the scheduler probes for responses of the clients and reacts accordingly. If a work unit is packed and sent (which requires a free client and fulfilled dependencies) the scheduler calls a non-blocking send routine. As explained in Section 1.3, a non-blocking routine may return before the data are sent and enables the scheduler to continue its work while the sending procedure is running in the background.

If the end of the waiting-list is reached, the scheduler calls two user-defined methods to adapt the scheduling process. One routine is provided to add new work units and the other one is designed to balance the load between the clients.

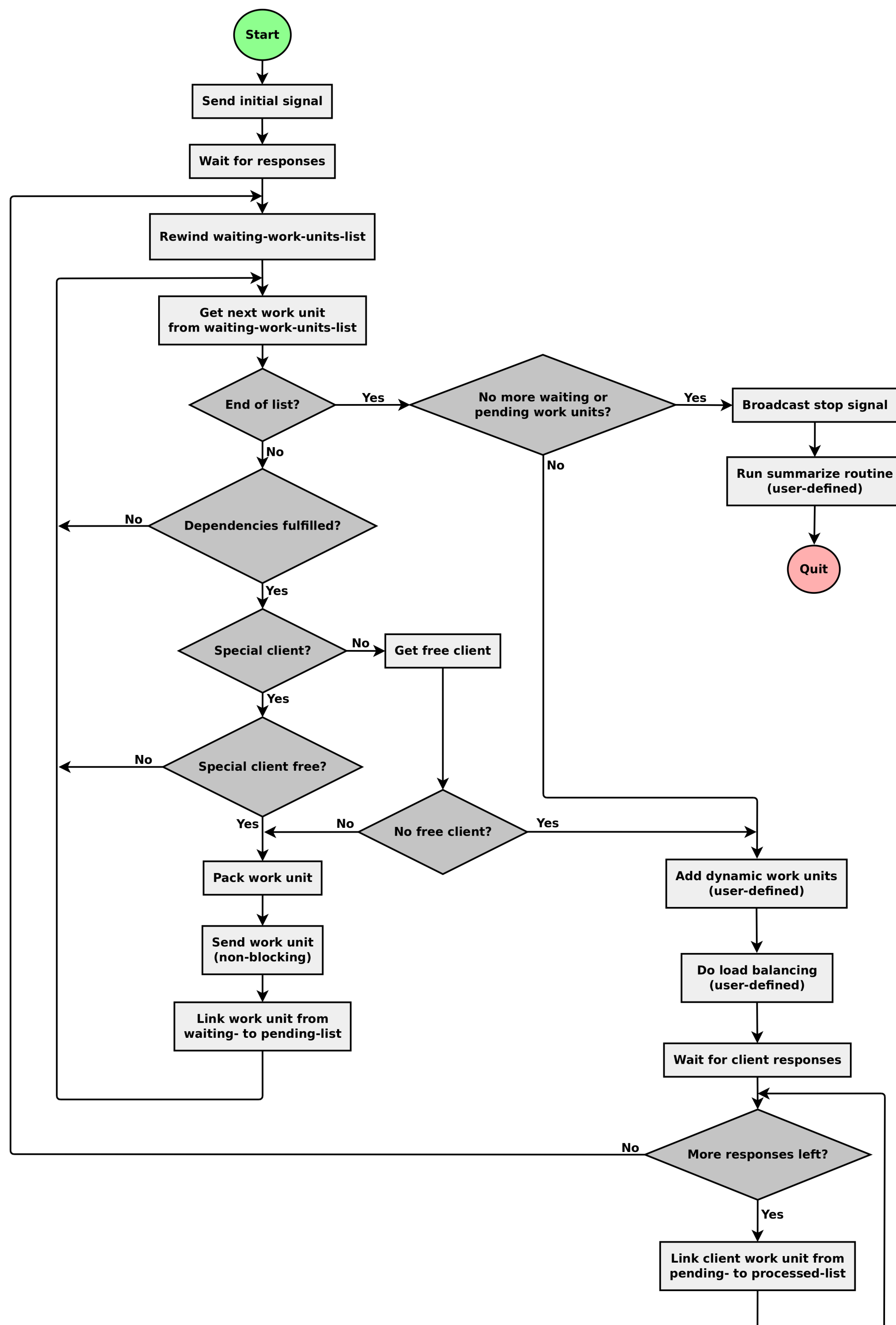


Figure 4.1: Flow chart of the scheduler

Then, the scheduler probes for responses from the clients and after processing them, the iteration process over the waiting work units starts again.

4.2.2 Client process

The client process consists of a loop which runs until the scheduler sends the signal to quit. In this loop the client probes for messages and runs an appropriate receiving function if a message is delivered. The decision which receiving method to run is based on the message tag. The flow chart of this process is shown in Figure 4.2.

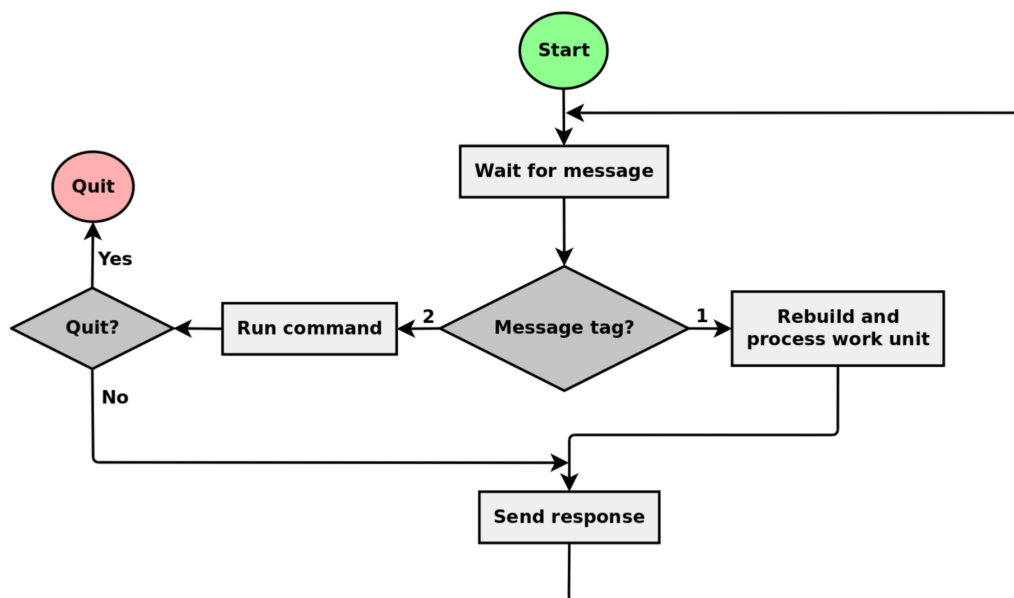


Figure 4.2: Flow chart of a client process

4.3 System architecture

This section describes the programming language, the compiler, and some other tools that were used during the development process.

4.3.1 Programming language

The decision to write the parallelization library in the programming language Fortran 2003 was made due to the following facts:

- Fortran is a common programming language for solving scientific, especially numerical problems [17].
- The code NEO-2 is written in Fortran.
- The Fortran draft 2003 includes mechanisms for object-oriented programming [17].
- MPI defines native language bindings for Fortran [7].

4.3.2 Tools for the implementation process

For managing and building the source code several tools were used. During the development it has been discovered that despite of the many years between 2003 and 2012, there is still little support for Fortran 2003 in many tools.

Compiler

As compiler and linker mainly GFortran is used in this thesis. On the official website of the compiler [18] it is noted that GFortran supports Fortran 2003 standards since version 4.6. Therefore, GFortran in version 4.6.2 and later was used to build the library. This compiler is part of the Gnu Compiler Collection (GCC). It is released under the Terms of the Gnu Public License (GPL) [19]. The license can be found online at [20].

It turned out that a good backward compatibility of the compiler makes it possible to compile NEO-2 with the newest compiler version, while it was compiled with version 4.4.5 of GFortran in the past. For testing purposes of

the parallelized code NEO-2 on the Vienna Scientific Cluster - 2 (VSC-2), the Fortran compiler Intel® Composer XE 2013 were used [21].

MPI implementation

As already mentioned in Section 1.3, the MPI implementation used in this master thesis is OpenMPI. The official website states that this is a high-performance Message Passing library with full MPI-2 standards conformance. OpenMPI implements features as thread safety, dynamic process spawning, 32- and 64-bit versions, and more [8]. It is released under the New BSD-License, which can be found online at [22].

Actually, some other high-performance implementations of MPI exist, e.g., MPICH2 [23]. The decision to use OpenMPI was mainly made due to the fact that it was preinstalled on the development machines. It should be no problem to replace the underlying MPI implementation, because the different implementations have to fulfill the MPI standards. This replaceability is used in this thesis when running NEO-2 on the VSC-2. On this system the library MPICH2 can also be used [21].

Performance analysis

The software package MPE (MPI Parallel Environment) is part of the MPI implementation MPICH2. It supports performance analysis of parallel programs by profiling mechanisms [24]. Although it is included in MPICH2, it is also possible to use it in OpenMPI applications because of the same MPI specification. Using this tool makes it possible to record all send- and receive-commands and to get a graphical analysis of the communication processes between the clients.

Build system

When compiling Fortran sources, the compiler creates module-files. These files are linked together to get an executable file. It can become a problem to use standard make-files because of the order of the creation of the module-files.

Example If module A uses module B, the user has to ensure that the sources of B get compiled before A. If the order is changed after the compilation and

the module-file of module B still exists, it can happen that the linking process works, because the linker does not check the actuality of the module-files. This can cause a situation in which the sources of B were changed but not recompiled, because the compiler and the linker are using the older module-files. A more detailed explanation of this problem including a solution by standard make-files can be found in the appendix of the book of Metcalf et al. [17].

Therefore, the tool CMake for building, testing, and packing applications with an automatic dependency check of source files is used in order to compile the sources in the right order [25]. The release license of CMake can be obtained online at [26]. Several tutorials are given on the official website to learn the syntax and the handling of this build system.

Source revision control

The tool subversion is used to manage the source code of the library. Subversion is an open-source version control system [27]. It is released under the terms of the Apache License 2.0 [28]. During the development of the library, the subversion server of the Graz University of Technology was used.

5 Implementation and testing

5.1 Introduction

This chapter describes the implementation of the parallelization library. In order to test and stabilize the code, two test cases are created. The first one continues the simple integration example of Section 3.3. The second one is a more complex matrix chain multiplication, which is, in some sense, similar to the joining process of propagators in NEO-2.

5.2 Numerical integration

A simple numerical integration can be done by approximating the function with trapezoids [29]. While this case is suited for performance and stability checks of the scheduling algorithm, it does not consider any dependencies between work units, because the integration results of the subintervals can be combined in any order and are, therefore, completely independent.

5.2.1 Problem analysis

The integration of a function is a good example for the parallelization of mathematical and physical problems by splitting the whole problem into minor steps. The test function, which should be integrated, is shown in Equation (5.1). The formula depicts a normal distribution with a standard deviation $\sigma = 1$ and a mean value $x_0 = 0$.

$$y(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \tag{5.1}$$

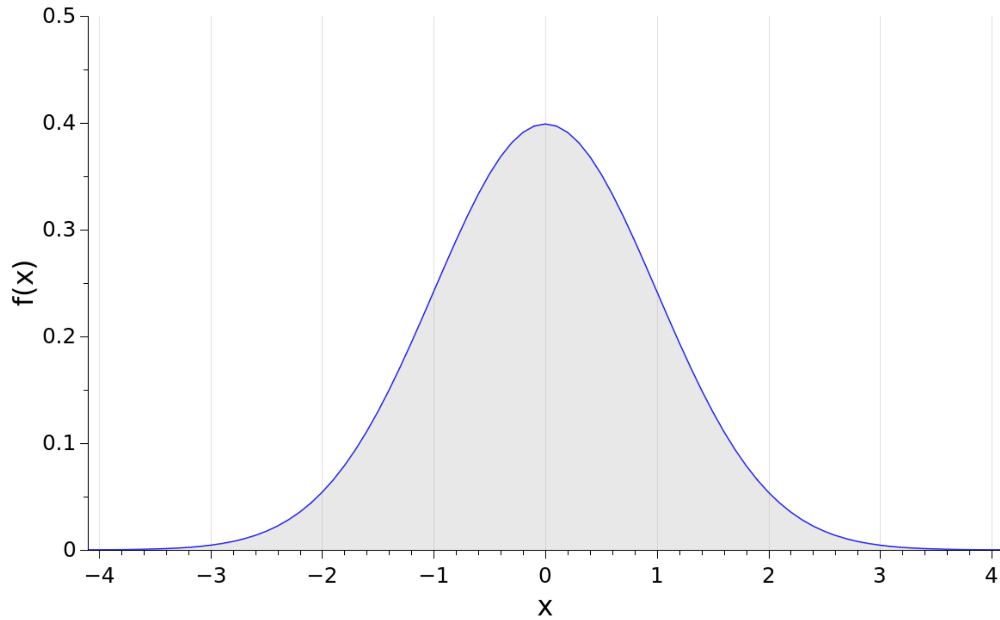


Figure 5.1: Integration test function

Solving method

A simple solving method for the numerical integration of a function is the trapezoid method of Equation (5.2) with the interval width $h = b - a$ [29].

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(a) + f(b)) \tag{5.2}$$

The given formula approximates the interval between a and b with a trapezoid. In order to obtain a more accurate result, the whole integration domain can be split into subintervals. The integral of one subinterval can be approximated by Equation (5.2). The single integration results are summed up to obtain the full solution. An example of an equidistant separation into eight subintervals is given in Figure 5.1.

Transition from sequential to parallel solution

A sequential code would compute the subintervals one by one and then sum them up. In order to speed up the code, these subintervals can be computed on different compute nodes at the same time. This requires a multi-core processor or a computer cluster environment as described in Chapter 1.

5.2.2 Adaption of the parallelization library

This section describes the parallelization of the example by using the developed library as well as the personal experience during the implementation process of the designed classes in Fortran 2003. In order to adapt the library to this kind of problem, the class `WuIntegrate` is inherited from the class `Workunit`. In the class diagram of Figure 5.2 it can be seen that this particular work unit contains attributes to define the bounds of integration of one subinterval. To enable the scheduler to rebuild a received work unit, the scheduler is also inherited to obtain the class `SimpleIntegrationScheduler`. This class overwrites the method of the generic scheduler to allocate an object of the class `WuIntegrate`. The method `summarize`, which is supported for final tasks as soon as all work units are completed, is also overwritten to sum up the integration results of the subintervals.

Class diagram revision In the book of Metcalf et al. [17] it is suggested that every class is encapsulated by its own module. In Fortran two modules can not include each other. This would create a circular dependence during compilation, i.e., the compiler would not know which module to compile first and the compilation process would fail. This becomes a problem, if two classes are related to each other, e.g., in the form of parent and child. If an attribute of the parent class points to a child object, it is not possible for the child class to have a pointer back to the parent object. A work-around for this situation is to place both classes into the same module, but this would be a breach of the principle of modular programming and is therefore not used in this thesis. Another solution would be to use submodules. However, before the Fortran draft 2003 was completed, there was no fully developed solution for submodules available. Therefore, it was decided to postpone this feature and a technical report was created later to describe this enhanced module features, instead of delaying the release of the whole standard [17, 30].

In the case of the parallelization library an issue of the type described above exists between the modules for the scheduler and the work units. In the UML class diagram of the library in Figure 3.4 it can be seen that a work unit belongs exactly to one scheduler, but one scheduler can have many work units.

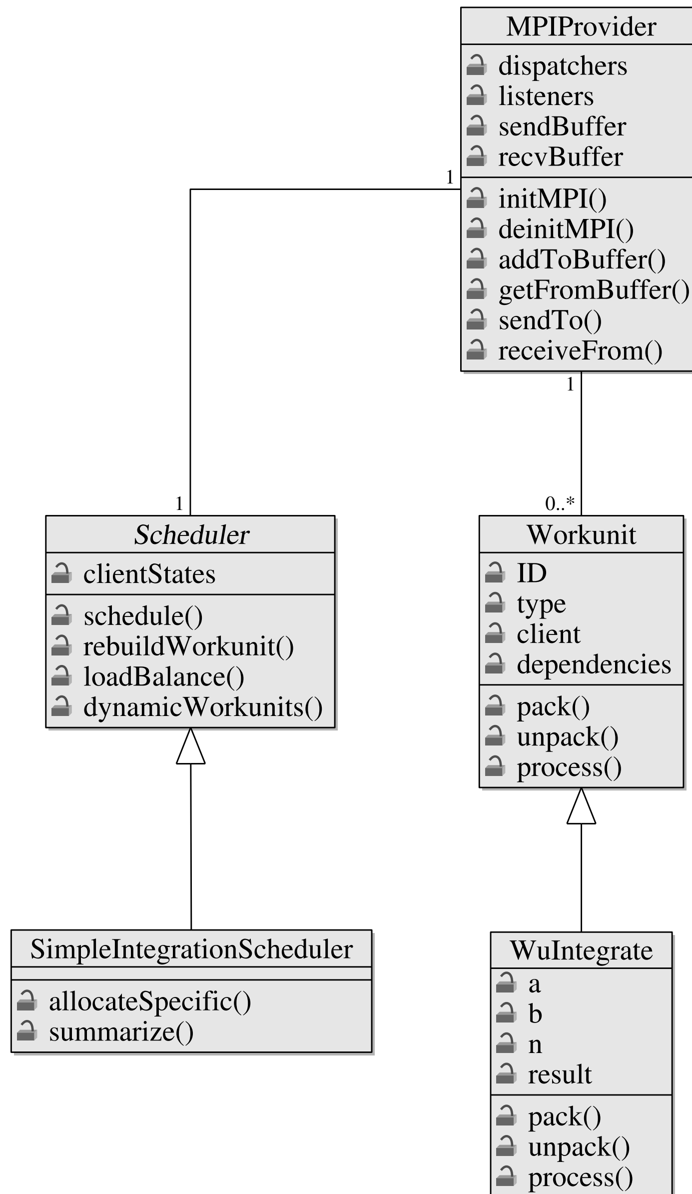


Figure 5.2: Class diagram of numerical integration test case

Therefore, the work units behave in some sense like children of the scheduler. At the time of the development of the parallelization library of this thesis the compiler GFortran did not support submodules [18]. Hence, a revision of the class relations, which were determined in the object-oriented analysis phase, was unavoidable.

The principle of the parallelization library is that a work unit has access to already processed work units in order to use their results for further computations. This requires the work units to gain access to the three lists of work units of the scheduler. This in turn causes a circular dependence because work units and scheduler have to be able to access each other.

To avoid this situation the three lists of work units are moved from the scheduler to the class `MPIProvider`. This enables both, the scheduler and the work units, to access these lists without creating a circular dependency. There is no explicit necessity for this in the integration example, however, it will be necessary for performing a matrix chain multiplication in another example.

5.2.3 Testing

As a test case, the program should integrate the function of Equation (5.1) from $a = -4.0$ to $b = 4.0$. To ensure a longer runtime of the program the number of sampling points in the interval between a and b is set to $n = 2 \cdot 10^9$.

Sequential code

The test of correctness of a sequential algorithm can simply be performed by checking the results. In contrast to a sequential code, in a parallel program the results as well as the runtime have to be considered [2].

Timing-function The MPI implementation provides an internal method to measure the time difference between two calls of a timing-function `MPI_WTIME`. This functionality makes it possible to record the time required for running a local method on one client. In order to do a more accurate runtime measurement for parallel algorithms, additional measurement tools are required. They are introduced by the use of MPE, which is described in Section 5.2.4. The runtime

interpretation has to be done in a careful way because other processes running on the same machine, e.g., from other users, may falsify the program's runtime.

The trapezoid formula used in this test case only consumes little processing power and the tests were performed when the workstations were almost unused. More accurate results can be obtained by running the code on a cluster with reserved processing power and time. If the sequential integration code is run, Program Output 2 is obtained.

Program Output 2 Simple integration test with sequential program

```
Integration from a = -4.000 to b = 4.000 with n = 2.0e9 sampling points.
Result =   0.999937

Runtime analysis
----------------
Complete runtime:     66.866 s
Scheduler runtime:     0.000 s
```

The output of the program gives the result of the calculation and the total runtime of the program. The runtime for the scheduler is zero, because the program ran in sequential mode, where no scheduler was needed. The result of the integration is verified by the command `quadl` of the software MATLAB® [31]:

$$\int_{-4}^{4} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx \overset{\text{Test case}}{\approx} 0.999937$$
$$\overset{\text{MATLAB}}{\approx} 0.999937$$

Several program runs delivered nearly the same runtimes, which are shown in Table 5.1. Therefore, it can be assumed that the runtimes are reliable. In order to calculate the speedup of the parallel code, a reference time is defined by Equation (5.4). The error is derived by the standard error, given in Equation (5.5) from the book of Bartsch [29]. The calculated reference time of the sequential code is stated in Equation (5.3).

$$t_{\text{seq}} = (67.06 \pm 0.15)\ \text{s} \tag{5.3}$$

Table 5.1: Runtime of sequential code

n Measurement number
t Program runtime

n	t /s
1	66.87
2	67.64
3	66.96
4	67.02
5	66.83

$$t_{\mathrm{seq}} = \bar{t} = \frac{1}{N}\sum_{i=1}^{N} t_{\mathrm{i}} \tag{5.4}$$

$$\Delta t_{\mathrm{seq}} = \sqrt{\frac{1}{N(N-1)}\sum_{i=1}^{N}(t_{\mathrm{i}} - \bar{t})^2} \tag{5.5}$$

Parallel code

As shown in Figure 5.1, the integral is divided into eight subintervals, i.e., eight work units are required to compute the whole problem. These work units are shown in Table 5.2. Each work unit contains different integration domains, defined by a_{wu} and b_{wu}. In order to obtain the same accuracy from the parallel code as from the sequential code, the number of sampling points $n = 2 \cdot 10^9$ is divided by the number of subintervals. The tasks for creating new work units and specializing the scheduler are described in more detail in Chapter 7.

The scheduler dispatches these work units to the clients until every client is loaded with one job. If a client completes a work unit, it automatically gets the next one from the scheduler. Ideally, all clients get the same number of jobs, but this depends on the processing time of different work units and the processing power of the clients. As soon as all of the work units are processed, the scheduler runs the user-defined summarize-routine to compute the full solution of the problem.

Table 5.2: Work units for parallel integration

wu	Number of work unit
a_{wu}	Left bound of integration
b_{wu}	Right bound of integration
n_{wu}	Sampling points of work unit

wu	a_{wu}	b_{wu}	n_{wu}
1	-4.0	-3.0	$2.5 \cdot 10^8$
2	-3.0	-2.0	$2.5 \cdot 10^8$
3	-2.0	-1.0	$2.5 \cdot 10^8$
4	-1.0	0.0	$2.5 \cdot 10^8$
5	0.0	1.0	$2.5 \cdot 10^8$
6	1.0	2.0	$2.5 \cdot 10^8$
7	2.0	3.0	$2.5 \cdot 10^8$
8	3.0	4.0	$2.5 \cdot 10^8$

One client In order to test the developed scheduler, the program is started in parallel mode on a total number of two processes, i.e., one for the scheduler and one for the client. In this case, the ideal runtime would be the same as for the sequential code. This implies that the processing time of the scheduler, the send- and receive-commands of MPI, the packing processes, and other time consuming processes, are negligible. The result of this test case can be seen in Program Output 3. The runtime is nearly the same as for the sequential code and therefore, not significantly affected by the scheduling process.

Two different performance analysis sections can be seen in the program output of the library. The first one is created by the scheduler and represents the mean time between two processed work units of each client. The second detailed performance analysis is generated by each client and shows more detailed information about the processing of the work units. It can be seen that eight work units have been processed by Client 1. In order to exchange processed work units between the clients, the library provides a special transmission work unit. It is called `DataRequester` and forces the client which received the `DataRequester` to send the requested work unit to a particular client.

In this example there is no need to exchange data between clients, because the integration results of the subintervals do not depend on each other, therefore, the number of data requester objects is zero.

Program Output 3 Simple integration test of parallel program (1 client)

```
Program launched by:
mpiexec -np 2 ./SimpleIntegration

Subinterval width:   1.000
Scheduler: All workunits are prepared! Count = 8

Integration from  -4.000 to -3.000 with  2.5e8 sampling points on client 1. Result = 0.001318
Integration from  -3.000 to -2.000 with  2.5e8 sampling points on client 1. Result = 0.021400
Integration from  -2.000 to -1.000 with  2.5e8 sampling points on client 1. Result = 0.135905
Integration from  -1.000 to  0.000 with  2.5e8 sampling points on client 1. Result = 0.341345
Integration from   0.000 to  1.000 with  2.5e8 sampling points on client 1. Result = 0.341345
Integration from   1.000 to  2.000 with  2.5e8 sampling points on client 1. Result = 0.135905
Integration from   2.000 to  3.000 with  2.5e8 sampling points on client 1. Result = 0.021400
Integration from   3.000 to  4.000 with  2.5e8 sampling points on client 1. Result = 0.001318

Scheduler needed     67.090 s for processing all workunits.
Scheduler spent      67.090 s for waiting.
Balanced workunits:       0

The integration result is   0.999937
Scheduler: All jobs are done!

----- PERFORMANCE ANALYSIS (Scheduler) -----
Client    MeanTime [s]
     1          8.292

----- PERFORMANCE ANALYSIS (Client) -----
Client        Workunits   Time[s]   Total[s]  DataRequesters   Time[s]    Packtime[s]
1 on faepop43         8     8.386     67.090               0       NaN            NaN

Runtime analysis
----------------
Complete runtime:          67.124 s
Scheduler runtime:         67.122 s
Time before scheduling:     0.002 s
Time after scheduling:      0.000 s
```

Two clients In order to achieve a higher performance, the code has to run on more than two processes. The output of the parallel program with two working clients is shown in Program Output 4. It was started with a total number of three processes, i.e., one for the scheduler and two for the clients.

In this program output it can be seen that the eight work units, each representing one subinterval of the function, are distributed among the Clients 1 and 2. The scheduler chose automatically which work unit to run on which client. The integration subinterval from -4.0 to -3.0 was sent to Client 1. Almost at

Program Output 4 Simple integration test of parallel program (2 clients)

```
Program launched by:
mpiexec -np 3 ./SimpleIntegration

Subinterval width:   1.000
Scheduler: All workunits are prepared! Count = 8

Integration from  -4.000 to -3.000 with  2.5e8 sampling points on client 1. Result = 0.001318
Integration from  -3.000 to -2.000 with  2.5e8 sampling points on client 2. Result = 0.021400
Integration from  -2.000 to -1.000 with  2.5e8 sampling points on client 2. Result = 0.135905
Integration from  -1.000 to  0.000 with  2.5e8 sampling points on client 1. Result = 0.341345
Integration from   0.000 to  1.000 with  2.5e8 sampling points on client 2. Result = 0.341345
Integration from   1.000 to  2.000 with  2.5e8 sampling points on client 1. Result = 0.135905
Integration from   2.000 to  3.000 with  2.5e8 sampling points on client 2. Result = 0.021400
Integration from   3.000 to  4.000 with  2.5e8 sampling points on client 1. Result = 0.001318

Scheduler needed     33.742 s for processing all workunits.
Scheduler spent      33.741 s for waiting.
Balanced workunits:       0

The integration result is   0.999937
Scheduler: All jobs are done!

----- PERFORMANCE ANALYSIS (Scheduler) -----
Client    MeanTime [s]
     1          7.317
     2          7.357

----- PERFORMANCE ANALYSIS (Client) -----
Client        Workunits   Time[s]   Total[s]  DataRequesters   Time[s]    Packtime[s]
1 on faepop43         4     8.365     33.459               0       NaN            NaN
2 on faepop43         4     8.435     33.741               0       NaN            NaN

Runtime analysis
----------------
Complete runtime:         33.825 s
Scheduler runtime:        33.820 s
Time before scheduling:    0.005 s
Time after scheduling:     0.000 s
```

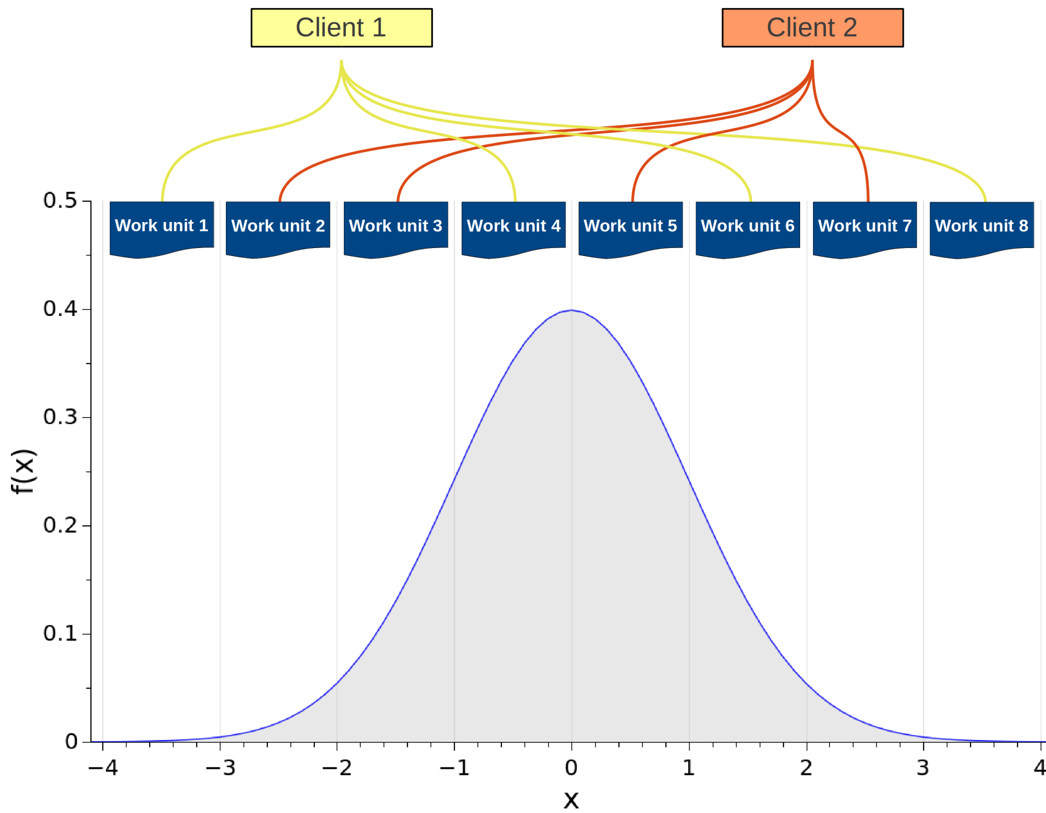


Figure 5.3: Principle of work unit distribution among the clients

the same time, the work unit for the subinterval from −3.0 to −2.0 was sent to Client 2. Apparently, Client 2 processed its work unit faster than Client 1, which can be caused by multiple reasons, so the next subinterval from −2.0 to −1.0 was also sent to Client 2. Figure 5.3 shows the principle of the scheduling process of this example.

The reason for different computation time of work units can be client- or work unit-dependent:

- Client-dependent
  The MPI-based implementation of this work is not limited to multi-core processors. It can also be used with different workstations connected via the network. Therefore, clients do not have to be equal machines. In such cases, a client with less processing power may need more computation time for a particular work unit, than another one. The scheduler of the

parallelization library provides some load-balancing mechanisms to react accordingly to clients with different processing power and distributes less work units to slower clients.

- Work unit-dependent
  The jobs done by the work units are not specified by the library and the scheduler does not care what a particular work unit does compute on a client. Therefore, two work units can represent completely different tasks, which could take a different amount of time to process. Another reason for different computation time arises if the work units should indeed perform the same task but are called with different parameters, e.g., in this integration example all work units integrate other parts of a function.

In Program Output 4 it can also be seen that the program roughly needs half the time as the sequential code. The reference time of the sequential code can be found in Equation (5.3). This proves that the scheduler is fast enough to provide both clients each time with enough input. The complete runtime of the code is $t_{\text{run}} = 33.825$ s, which can be split into three parts.

- Before scheduling
  This time is needed by the initial phase of the code, here 0.005 s.
- Scheduler runtime
  This time is needed until all work units are processed, here 33.820 s.
- After scheduling
  This time is needed for the finalization phase of the code, here 0.000 s.

Performance evolution on one compute node

In this section the evolution of the program's performance when the number of clients is increased is analyzed. In the former program tests the number of work units was eight. While this is a good number to understand the operation principle of the scheduler, it is too small to enable the scheduler to distribute the work units in an effective way to a larger number of clients. If there are less work units than clients, a situation may arise where some clients do not get any work unit to process. Therefore, the number of subintervals is increased to

100 for the following test cases. The runtimes of the program as a function of the number of clients are shown in Table 5.3.

Table 5.3: Run time measurement of integration example on multi-core machine

n	Number of clients
t_n	Program runtime of n-th run
t_{par}	Mean runtime, Equation (5.4)
Δt_{par}	Standard error of mean runtime, Equation (5.5)
S_{ideal}	Ideal speedup of Amdahl's law, Equation (5.6)
S_{calc}	Calculated speedup of the program, Equation (5.7)
ΔS_{calc}	Error propagation of the calculated speedup, Equation (5.9)

n	t_1 /s	t_2 /s	t_3 /s	t_{par} /s	Δt_{par} /s	S_{ideal}	S_{calc}	ΔS_{calc}
1	66.87	67.89	66.87	67.21	0.59	1.00	1.00	0.01
2	34.12	33.54	33.51	33.72	0.34	2.00	1.99	0.03
3	22.81	22.76	22.77	22.78	0.03	3.00	2.94	0.01
4	20.30	20.03	19.73	20.02	0.29	4.00	3.35	0.04
8	13.67	15.08	13.82	14.19	0.77	8.00	4.73	0.26
16	13.60	13.46	13.21	13.42	0.20	16.00	5.00	0.08

All runtime measurements were performed three times. The mean runtime is calculated by Equation (5.4), the standard error by Equation (5.5). The ideal speedup by Amdahl's law (Equation (5.6)) is obtained if the program has no sequential parts ($f = 1$), as explained in Section 1.2.2.

$$S_{\text{ideal}} = n \tag{5.6}$$

The calculated speedup of the program (Equation (5.7)) can simply be evaluated by the ratio of the runtime of the parallel code to the runtime of the sequential code.

$$S_{\text{calc}} = \frac{t_{\text{seq}}}{t_{\text{par}}} \tag{5.7}$$

In order to evaluate the error of the calculated runtime speedup, the error

propagation of independent variables is used (Equation (5.9)) [29].

$$\Delta S_{\text{calc}} = \sqrt{\left(\frac{dS_{\text{calc}}}{dt_{\text{par}}}\Delta t_{\text{par}}\right)^2 + \left(\frac{dS_{\text{calc}}}{dt_{\text{seq}}}\Delta t_{\text{seq}}\right)^2} \tag{5.8}$$

$$= \sqrt{\left(\frac{t_{\text{seq}}}{t_{\text{par}}^2}\Delta t_{\text{par}}\right)^2 + \left(\frac{\Delta t_{\text{seq}}}{t_{\text{par}}}\right)^2} \tag{5.9}$$

Figure 5.4 shows a graphical depiction of the measured runtimes against the number of clients. The ideal and calculated speedups are plotted against the number of clients in Figure 5.5. Due to the linear behavior of Equation (5.6), the ideal runtime is represented by the red straight line. The blue curve, which represents the calculated speedup of the program, shows nearly ideal behavior at a lower number of clients, but shows a saturation effect at a higher number. Since the number of physical cores of the processor of the test machine is four, which can be seen in the hardware information of the test machine in Section 1.2.3, this effect can be interpreted as arriving at the hardware limits. The reason for a breach of the performance already at a number of four clients is that the total number of processes on the machine is five, because there has to be one extra process for the scheduler. Therefore, five processes have to be distributed among four cores, which causes performance issues.

Performance evolution on a computer cluster

The result of the test on one machine shows that the hardware specifications, e.g., the number of physical cores, limit the maximum speedup of the program. In order to avoid such limits and to run the code on more than four clients, the computer cluster of the Institute of Theoretical and Computational Physics of the Graz University of Technology is used. As explained in Section 1.2.3 this cluster consists of workstations connected by Ethernet and enables the users to run parallel tasks. Therefore, an additional argument to the run command `mpiexec` of the MPI-based program has to be passed [32]:

```
mpiexec -np 9 --machinefile machinefile ./SimpleIntegration
```

The content of the user-defined file `machinefile` defines the compute nodes on which the code should run. The option `max_slots` defines the maximum

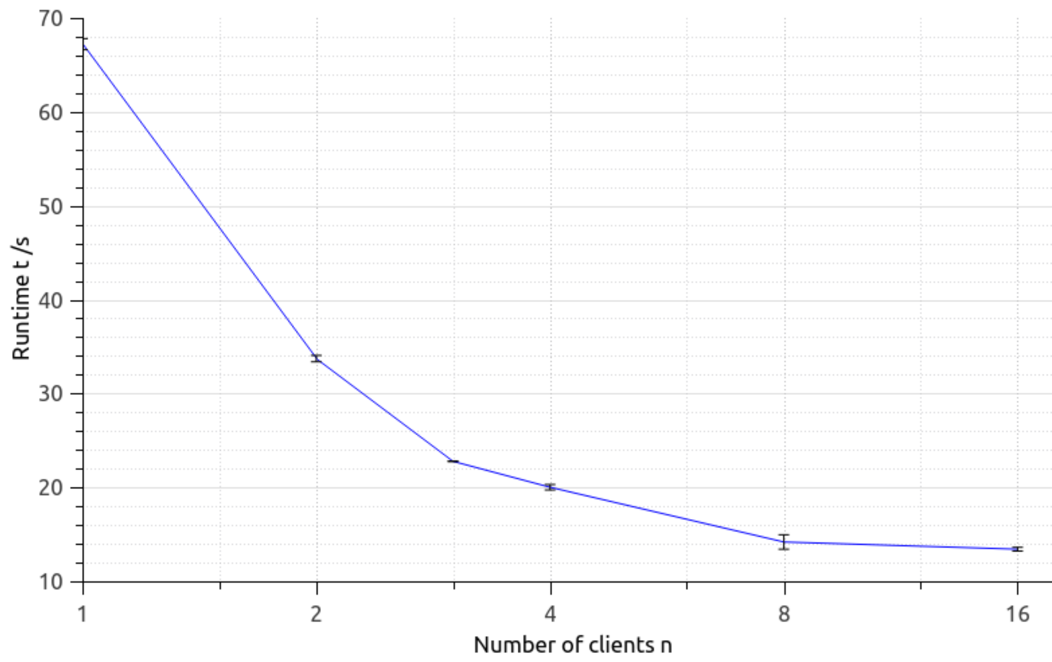


Figure 5.4: Mean runtime plotted against number of clients on one compute node

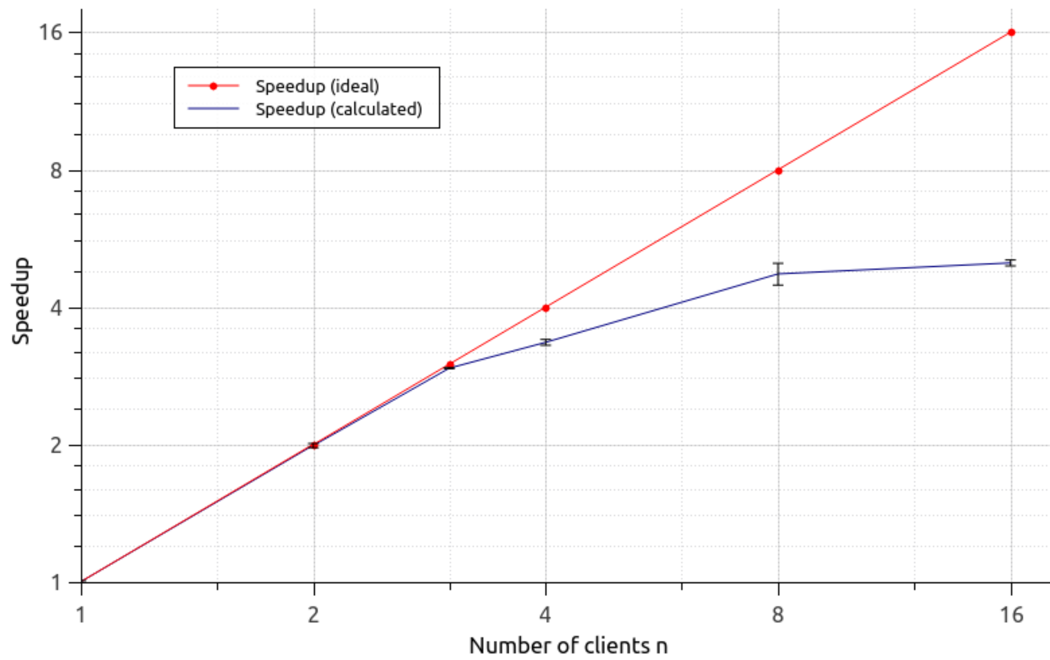


Figure 5.5: Speedup of the program against number of clients on one compute node

number of cores per compute node. This allows to write the machine-file in a form to avoid the hardware limits. The file is processed from top to bottom, which means that in the case of a total number of 9 processes, 4 processes are distributed to Client 1, 4 processes to Client 2, and 1 process to Client 3:

```
Client1 max_slots=4
Client2 max_slots=4
Client3 max_slots=4
```

In the following test case five compute nodes were used. Due to the option for the maximum number of slots per node, no computer ran more than four processes. The program runtimes as a function of the number of clients are shown in Table 5.4. The graphical evaluations can be seen in Figure 5.6 and Figure 5.7. In contrast to the test case on one compute node with the limit of four physical cores, the graphical analysis of this test case shows a good evolution of the speedup with an increasing number of clients.

Table 5.4: Run time measurement of integration example on cluster

n	Number of clients
t_n	Program runtime of n-th run
t_par	Mean runtime, Equation (5.4)
Δt_par	Standard error of mean runtime, Equation (5.5)
S_ideal	Ideal speedup of Amdahl's law, Equation (5.6)
S_calc	Calculated speedup of the program, Equation (5.7)
ΔS_calc	Error propagation of the calculated speedup, Equation (5.9)

n	t_1 /s	t_2 /s	t_3 /s	t_par /s	Δt_par /s	S_ideal	S_calc	ΔS_calc
1	66.87	67.89	66.87	67.21	0.59	1.00	1.00	0.01
2	34.13	33.99	34.48	34.20	0.25	2.00	1.97	0.02
4	17.32	16.86	17.38	17.19	0.28	4.00	3.91	0.07
8	8.80	8.77	8.87	8.81	0.06	8.00	7.63	0.05
16	4.96	4.78	4.89	4.88	0.10	16.00	13.80	0.26

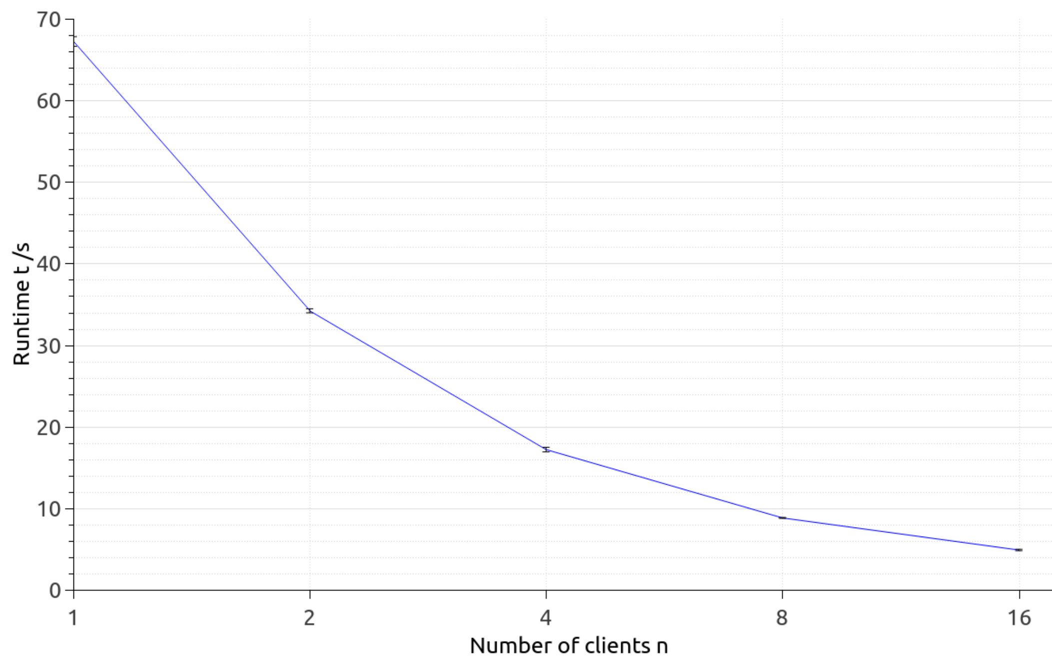


Figure 5.6: Mean runtime against number of clients on a computer cluster

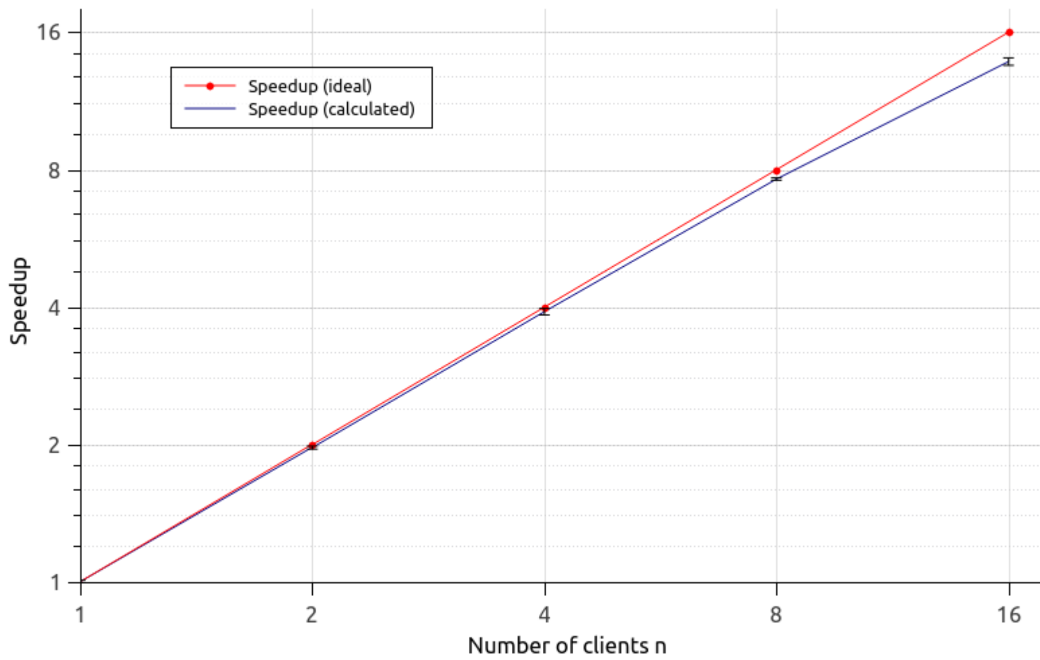


Figure 5.7: Speedup against number of clients on a computer cluster

5.2.4 Detailed runtime analysis by profiling

Profiling library

The performance measurements of the test cases above do only involve the total runtime of the code. In order to get more detailed timing information, the profiling tool MPE (MPI Parallel Environment) is used. This library, which is described in Chapter 4, supports the measurement of local operations on a particular client as well as the record of send- and receive-operations between processes [24].

While it is possible to exchange the native MPI commands to equal commands of MPE with profiling mechanisms, in the parallelization library of this thesis another option is used to enable the profiling mechanism and to leave the native MPI commands untouched. As described in Chapter 3, the major MPI commands are wrapped by methods of the class `MPIProvider`. This allows to add code to these methods to activate MPE profiling. For this issue a new class `MPELog` was created for calling MPE commands.

Graphical analysis

The profiling tool creates a file which contains all timing information of all involved processes of the parallel code. The program `jumpshot`, which is included in MPE, converts this file into a graph. The integration of MPE into the methods of the class `MPIProvider` makes it possible to record send- and receive-processes as well as local operations like the processing of work units. For such purposes user-defined events can be defined in MPE [2].

One client The evaluated profile data of the numerical integration code with one working client can be seen Figure 5.8. The horizontal axis represents the time component. The vertical axis indicate the processes which are numbered by the MPI scheme, starting with zero for the master process. The client ranks are displayed on the left side of the graphical diagram. The first row, which represents the scheduler, is mostly black. This indicates that the scheduler is waiting for messages from the clients most of the time and therefore, has enough capacities for dealing with more clients. The processing of work units is

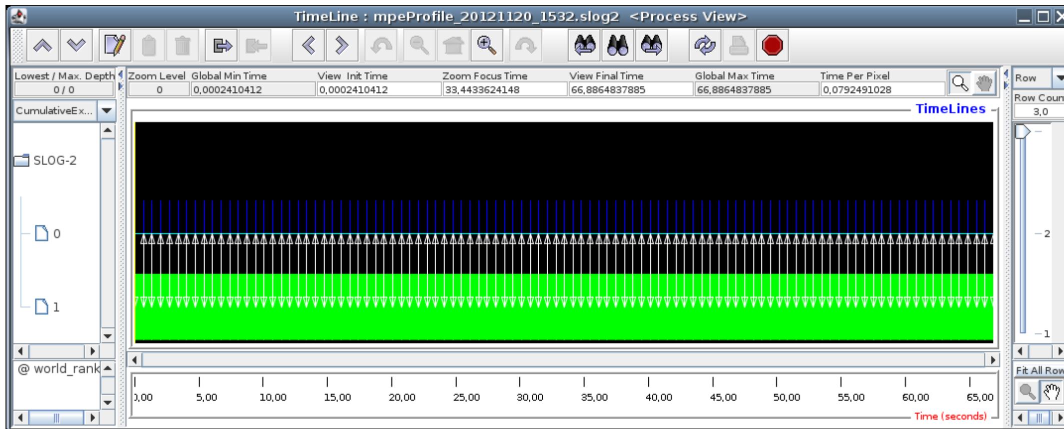


Figure 5.8: MPE profile of integration example with 1 client and 1 scheduler

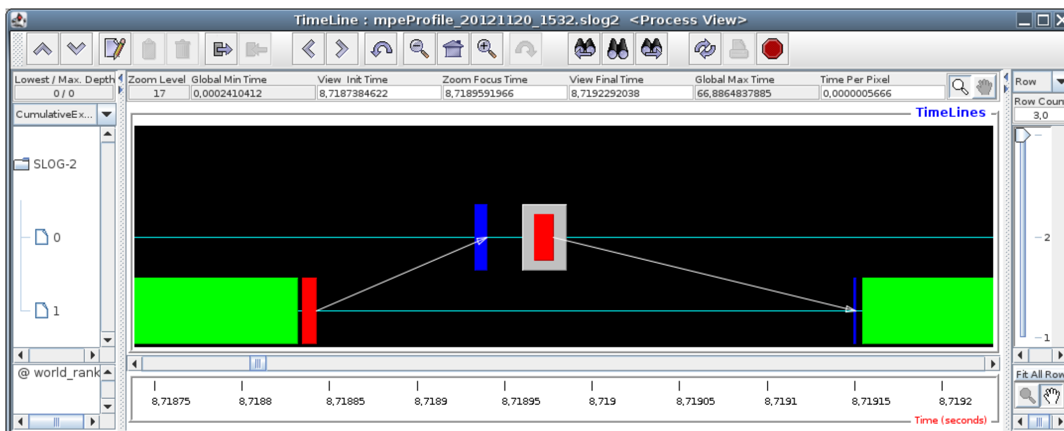


Figure 5.9: Zoomed view of Figure 5.8 to depict a communication process

marked by green blocks. The white arrows indicate Message Passing between the processes. It seems that the arrows point in both directions but a zoom to a smaller time scale shows that they are two separate ones, each pointing in the opposite direction. The client is loaded with work each time, therefore, the parallel program with one client is as fast as the sequential one.

Figure 5.9 shows a zoomed area of about 0.5 ms of Figure 5.8. It depicts the communication process between scheduler and client. As soon as the green block of Client 1 ends, the work unit has been processed completely. The following red rectangle indicates the call of the sending method of MPI, which tells the scheduler that the work unit is completed. The receiving process of the scheduler is shown by the blue block in the first row. If the message is received

from the scheduler it starts its iteration process over the waiting work units and searches the next job for the client, as shown in the flow chart of the scheduler in Figure 4.1. This process is indicated by the gray block. During this process the scheduler is busy and will not respond to other messages. Messages sent in this time are buffered by the MPI implementation and received if the scheduler calls a receiving method of MPI. If the appropriate work unit for the client is found, it is packed and sent, which is again marked by a red rectangle. The small blue vertical line indicates the fast receiving process of the working client, followed by the start of the processing of the received work unit. The analysis of such diagrams delivers information about the working principle of the developed parallelization library and proves that the code behaves as expected.

Eight clients Figure 5.10 shows the MPE diagram of the parallel code running on eight clients. It can be seen that no visible gaps between the processing operations of the work units exist. Therefore, the scheduler is fast enough to load the clients with work each time. At the end of the time scale it can be seen that not all clients finish at the same time. The reason is the different computation time of the last work units. If the mean computation time of one work unit is much smaller than the whole program's runtime, this is not significant. The scheduler tries to dispatch the work units in an effective way, but it can not predict its duration. However, if work units need too much computation time in contrast to the whole runtime, the user should modify the work units.

Figure 5.11 shows that Client 4 tells the scheduler when its last work unit is completed. After summarizing the results, the scheduler tells all clients to stop and the MPI master process quits as soon as the last process exits.

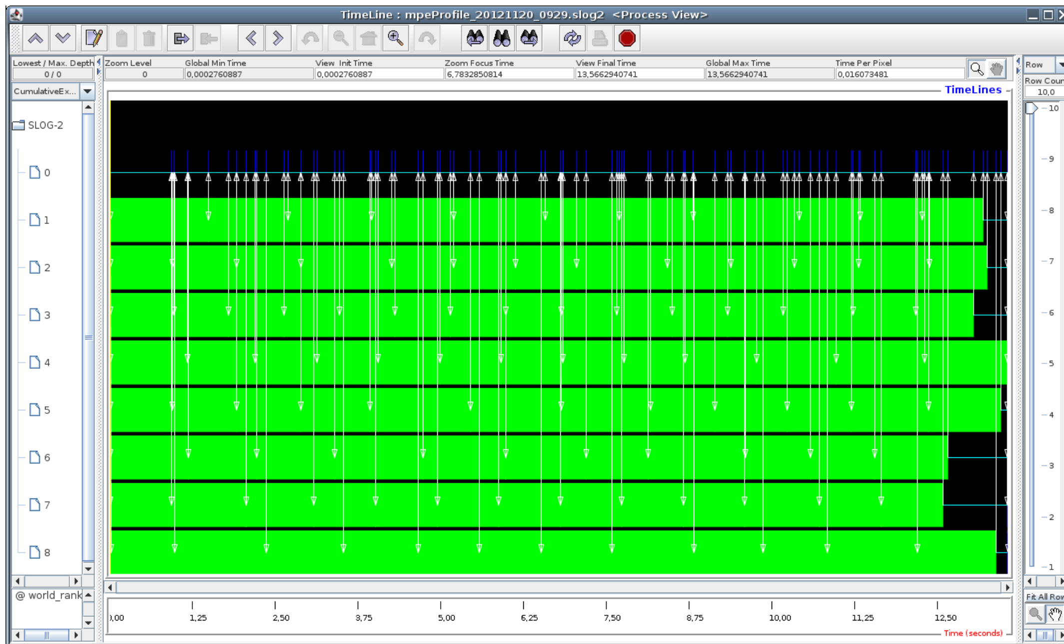


Figure 5.10: MPE profile of integration example with 8 clients and 1 scheduler

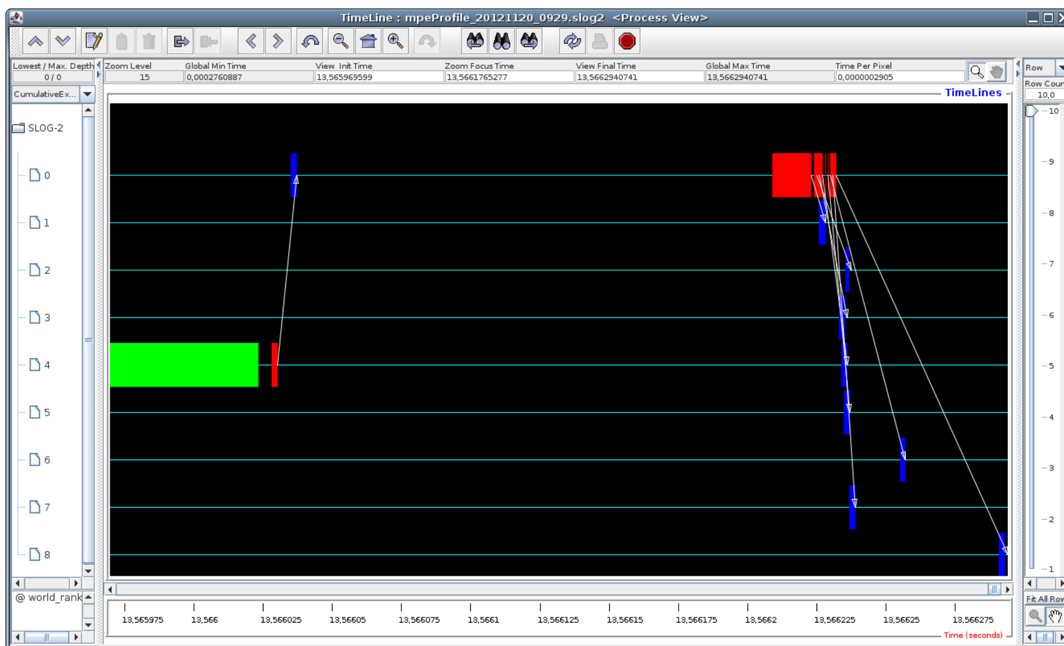


Figure 5.11: MPE profile of the end of the scheduling process

5.3 Matrix chain multiplication

The matrix multiplication fulfills the algebraic group properties. The multiplication of two matrices of one set results in a new matrix, which is again an element of the same set [12]. This is similar to the joining of two propagators within NEO-2 and is therefore used as another simplified test case for the library.

5.3.1 Problem analysis

In contrast to the numerical integration test case, there are a few significant differences:

- The integrity of the order of the matrices has to be preserved, because the results of the multiplications can not be combined in an arbitrary order.
- Scheduler and clients have to deal with a large amount of required memory, because large matrices can be simulated.
- Because of the associativity law, the results of multiplications can be required by other multiplication processes. Therefore, data between the clients have to be exchanged.
- In order to simulate the NEO-2 problem, at first the matrices have to be created on the clients. This simulates the solving of the propagators before they can be joined. If the computation of two neighboring matrices is accomplished, then the scheduler should react accordingly and create a work unit to join them.

5.3.2 Adaption of the parallelization library

Work units

The UML class diagram of this problem is depicted in Figure 5.12. One work unit represents the task of multiplying two matrices. This special work unit is called `WuMatrixMultiplication`. In order to preserve the sequence of the resulting matrices, another work unit, called `MergeWorkunit`, is created. It

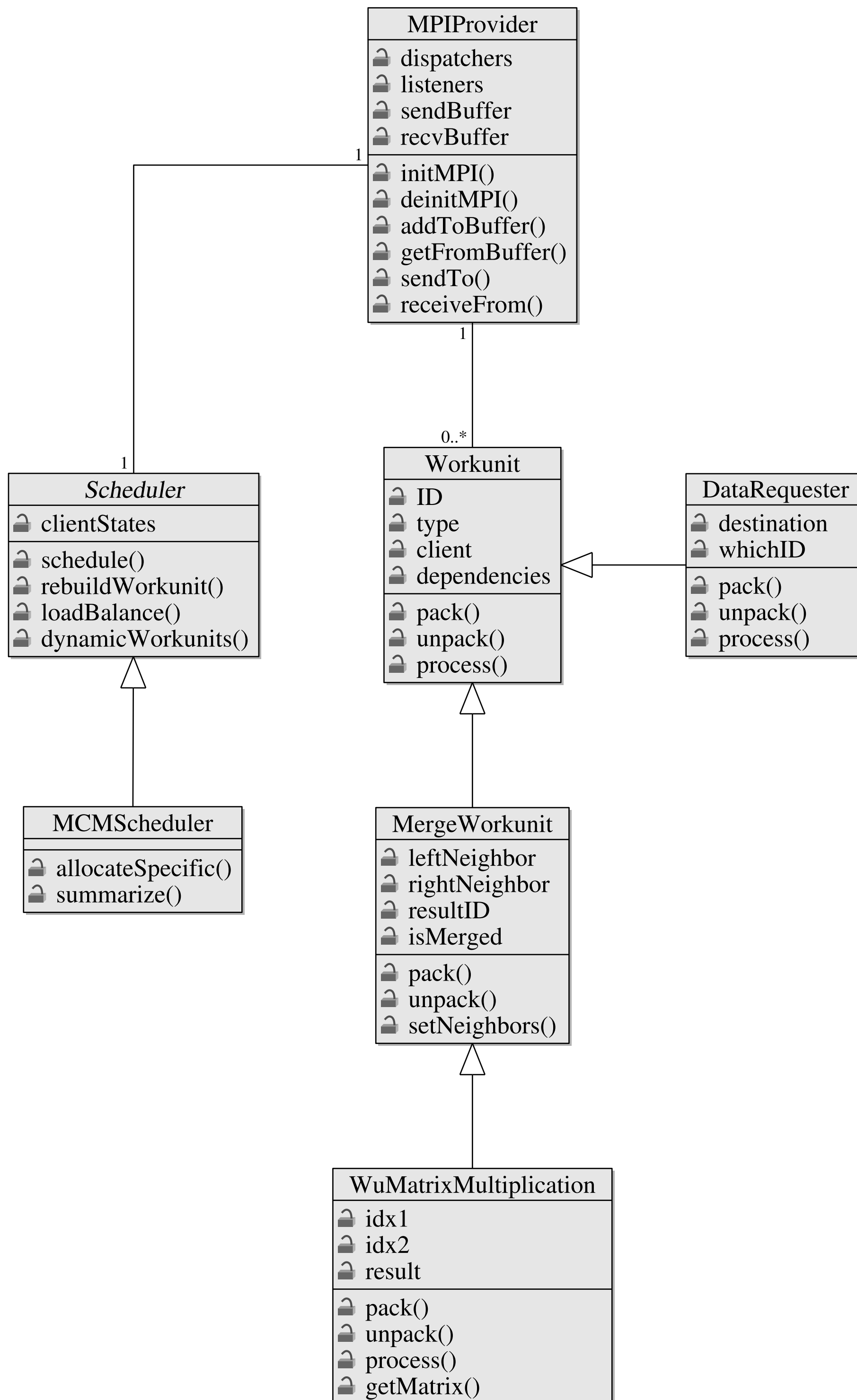


Figure 5.12: Class diagram of parallel matrix chain multiplication

is the parent class of `WuMatrixMultiplication` and provides attributes and methods to indicate its neighboring matrices to the left and to the right.

Scheduler

The main issue when developing a scheduler for this kind of problem is not the multiplication itself, which can simply be done by the intrinsic Fortran function `matmul`. It is rather the correct joining of the single matrices. In the former integration test case, the scheduler could distribute the work units in an arbitrary order. The results were simple scalars, which had to be summed up at the end to obtain the full solution. Therefore, the order of the work units was trivial and the clients did not have to exchange processed work units. For the data transmission between clients, the already described special work unit `DataRequester` is used.

Because of the associativity of the matrix multiplication, the solving process can be visualized by a tree structure. In the case of eight matrices to be multiplied, the principle of the joining processes of the matrices look like shown in Figure 5.13. According to the propagator example of Figure 2.3, the gray matrices are not solved yet and the yellow matrices are ready to be joined. Work units are indicated by the blue structures.

In the first row of the figure, the matrix elements are computed by the Work units 1 to 8. In the second row, it can be seen that each of the following work units require results of other ones. Because of the generic design of the scheduler, the user has to create all work units to solve the whole problem. However, a new scheduler can be inherited from the generic scheduler to create work units automatically. The special scheduler of this problem is called `MCMScheduler` (MCM = Matrix Chain Multiplication). The method `dynamicWorkunits` of the generic scheduler is overwritten to create new work units during the solving process in order to react on different client performances.

Parallel solving on two clients

One possible principle of solving the matrix chain multiplication on two clients is shown in Figure 5.14. Both clients are computing until they have one matrix left. At the end of the whole process it comes to the point, where the situation

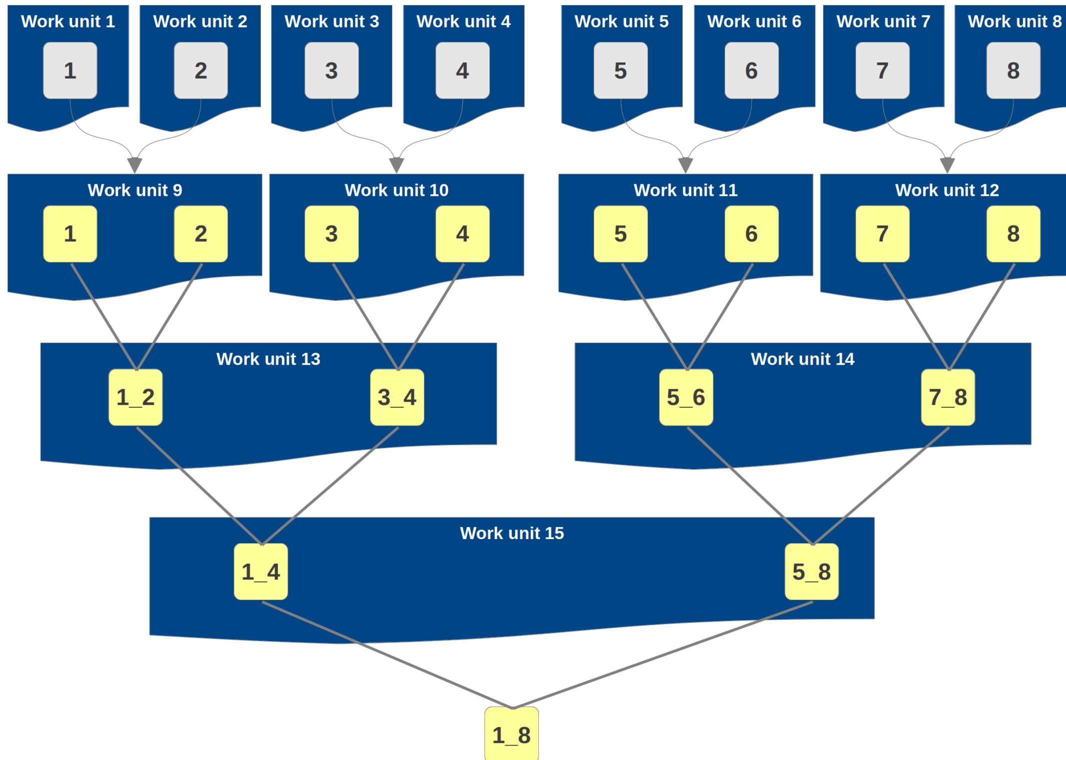


Figure 5.13: Solution of the matrix chain multiplication problem by a tree structure

arises in which one client has to send its last work unit to another remaining client. The scheduler has information about which work units were processed on which client, therefore, it detects if work units have to be exchanged between clients to continue the solving process. As explained in Section 5.2.3, a work unit of the type `DataRequester` can be created by the scheduler for transmitting work units between the clients. The `DataRequester` is sent to Client 2 and initiates the sending process of Work unit 15 to Client 1. Work unit 15 contains the matrix which is required by Work unit 16 on Client 1 to compute the last multiplication. This transmission process is indicated by the red arrow. As soon as Client 1 receives the already processed work unit, it is unpacked and stored locally for further computations. Each work unit contains an attribute which describes the state of the work unit (processed or unprocessed) for such cases. Most of the work units which are received from the scheduler are unprocessed ones and work units exchanged between clients are processed ones.

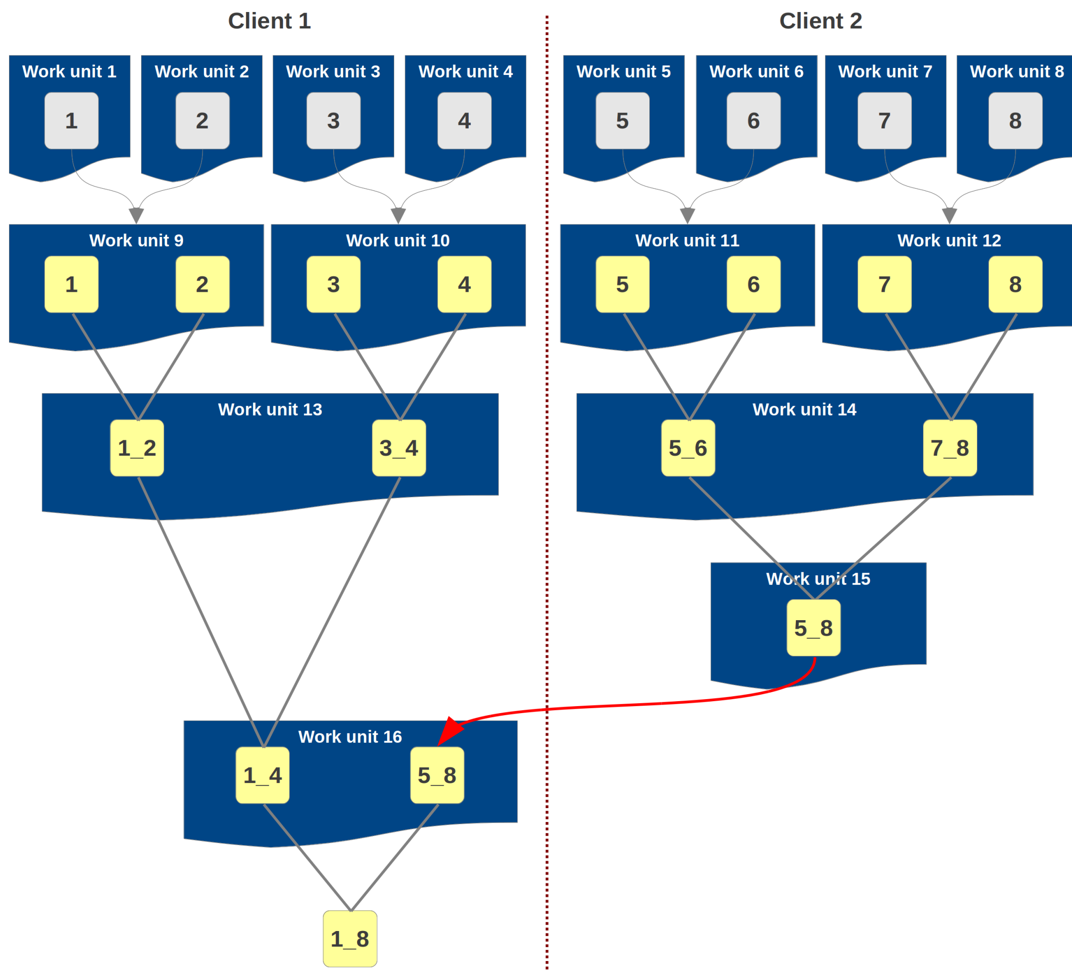


Figure 5.14: Solution of the matrix chain multiplication problem on two clients

5.3.3 Testing

For testing purposes, matrices with different dimensions to be multiplied are created by the test program. Due to this approach two advantages are achieved:

- Different computation time per work unit
  The multiplication process of two large matrices takes longer than the multiplication of two small matrices. This is a contrast to the numerical integration example where the computation times of the work units were very similar.
- Integrity check of multiplication order
  Two matrices can only be multiplied if the number of columns of the left matrix equals the number of rows of the right one [12]. A violation causes the intrinsic matrix multiplication routine of Fortran to crash, which is an useful method to check the correctness of the multiplication order.

Investigation of a slow scheduler

This section describes the operation principle of the specialized scheduler for this kind of problem. As depicted in Figure 5.12 it is inherited from the generic scheduler. If the program starts, it builds the whole multiplication tree, as can be seen in Figure 5.14, at first, without solving the matrices. This ensures the correct multiplication order for the further solving process and enables the scheduler to distribute parts of the solution tree equally to all processes.

Various test scenarios of this scheduler delivered MPE diagrams as depicted in Figure 5.15. It can be seen that during the runtime of the code, the scheduler reacts too slow to clients requesting new work units. This causes the clients to wait longer for a work unit than it takes to compute them. The waiting time for new jobs of the clients is marked by yellow blocks.

A waiting client is not useful in a parallel environment, therefore, the scheduler has to be optimized. One optimization step is to exchange the standard send-command of MPI, which is blocking, to the non-blocking version. The difference between blocking and non-blocking methods is explained in Section 1.3.2. Due to this exchange, the scheduler can continue its iteration process while a work unit is sent to a client.

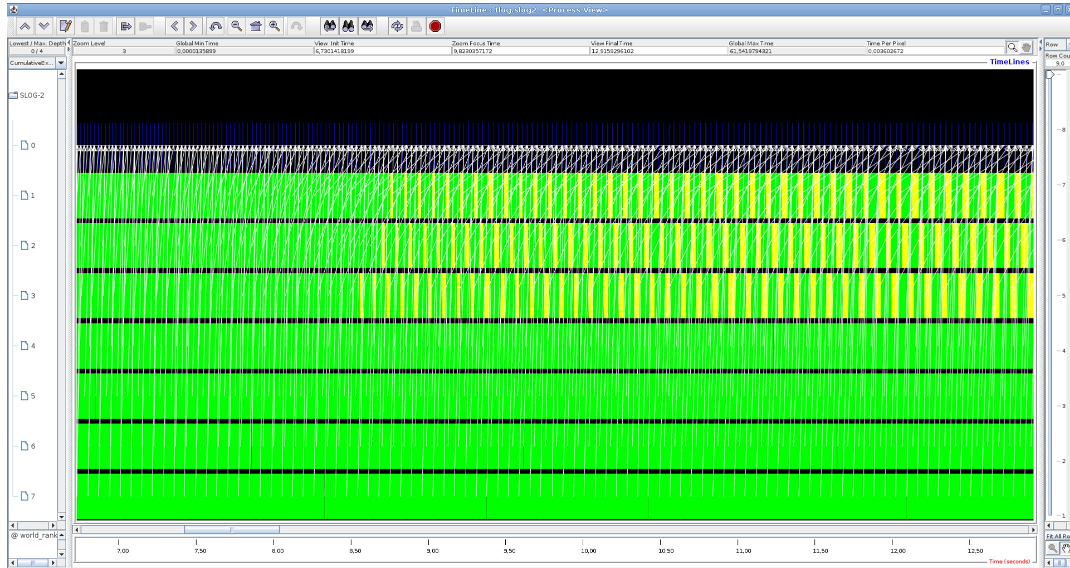


Figure 5.15: MPE diagram of a slow scheduler

Simulation of long computation

Compared to the mean time of the solving and joining of propagators, the simulated matrix multiplications are too fast to simulate NEO-2. In order to simulate work units which require more computation time, a random sleep is added to the processing routine. While this only simulates faster and slower work units, it does not consider faster and slower clients. Therefore, an additional random number for each client is created at the program start to simulate its processing power. These values can be configured by a configuration file to test the program with different values. The matrix multiplication itself is kept in the program because it can be used to check for the correct multiplication order.

The program runtime of a simulated long computation against the number of cores is plotted in Figure 5.16. The speedup of the code is shown in Figure 5.17. It can be seen that the speedup does not result in the ideal speedup with increasing number of clients. The text box in the diagram gives information about the runtime properties of the test program. It can be seen that 1000 matrices with dimensions between 400x400 and 800x800 of random double precision numbers were multiplied. The property for the maximum chunk size indicates that the scheduler was allowed to bind at maximum eight work units

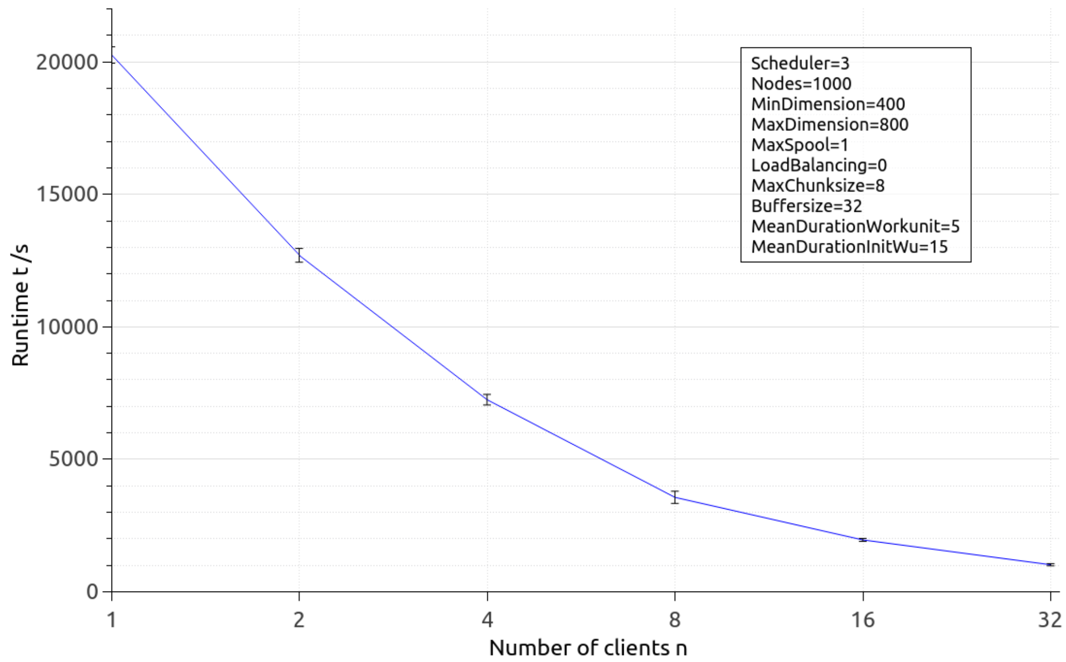


Figure 5.16: Program runtime of matrix chain multiplication example plotted against the number of clients

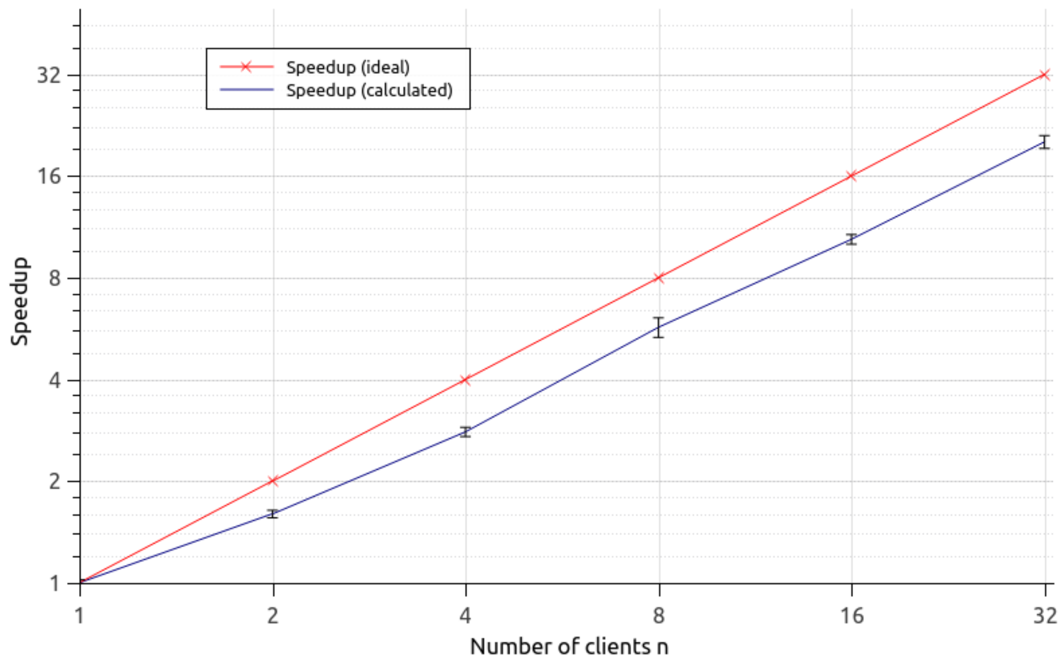


Figure 5.17: Speedup of matrix chain multiplication example plotted against the number of clients

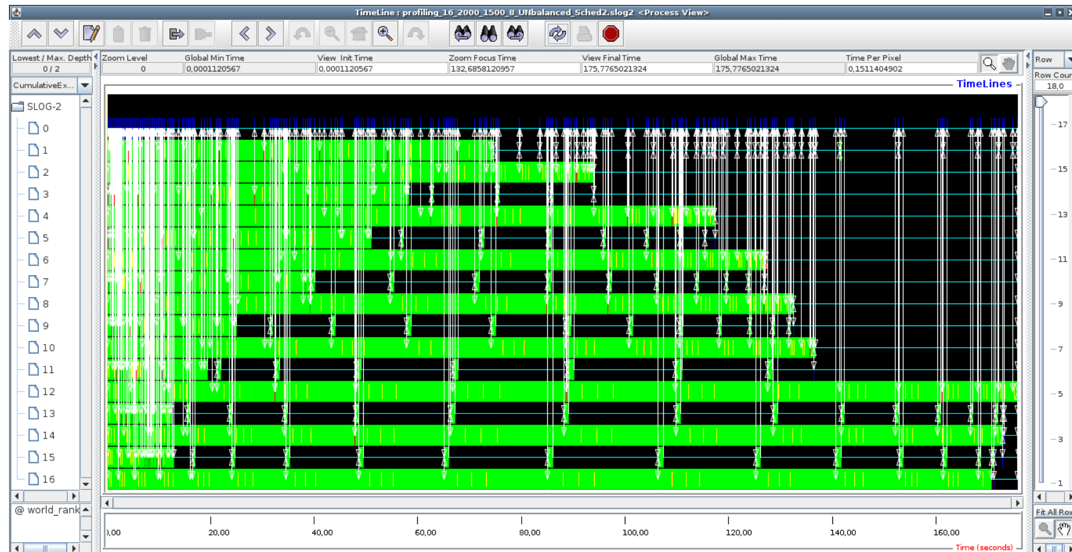


Figure 5.18: MPE diagram with no load-balancing enabled

to one package before sending them. This decreases the number of communication processes but increases the mean time for processing the work units. The simulated mean time for multiplying two matrices was set to 5 seconds and the creation time for a matrix to 15 seconds.

An example MPE diagram of the scheduler is shown in Figure 5.18. It can be seen clearly that clients with different processing power and work units with different computation times cause many clients to wait (black areas in the diagram). Therefore, a load-balancing routine is developed and added to the scheduler, which detects free clients and sends work units to them which were intended for other clients.

Load-balancing algorithm

The load-balancing routine checks after every scheduler iteration, if there are no jobs left for a certain client. If such a situation is detected, the scheduler moves one work unit, which was indented to be solved on a particular client to a free client. This process makes it more complicated to obtain the correct multiplication order, however, it has positive influence on the runtime, which is evident from the nearly ideal speedup curve in Figures 5.19 and 5.20.

A change of the structure of the created multiplication tree by the scheduler

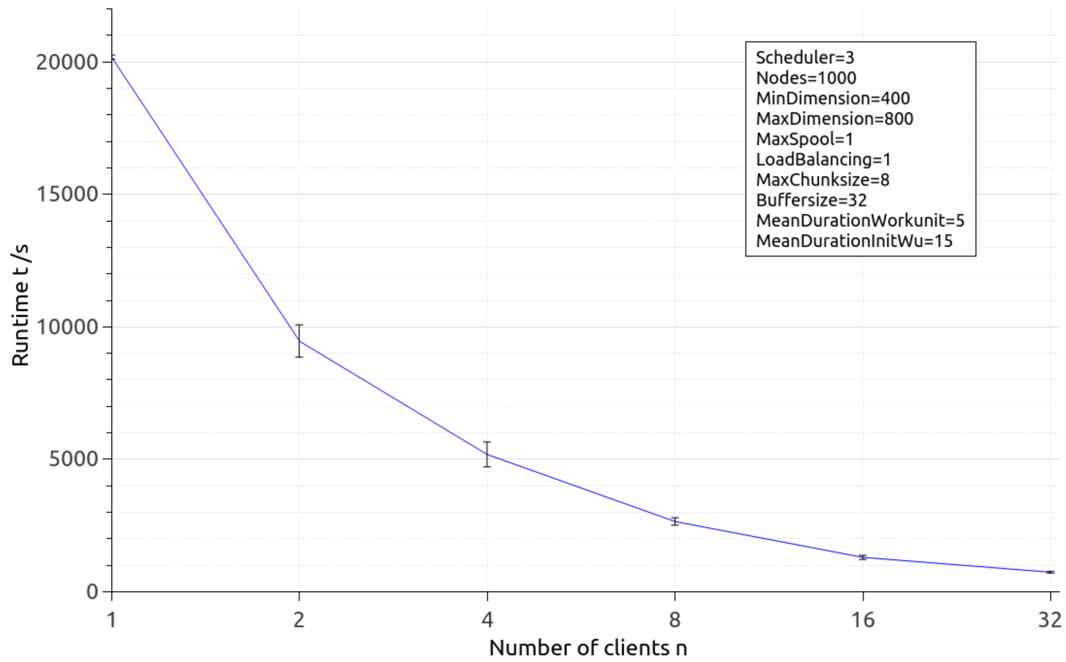


Figure 5.19: Program runtime with load-balancing of matrix chain multiplication example plotted against the number of clients

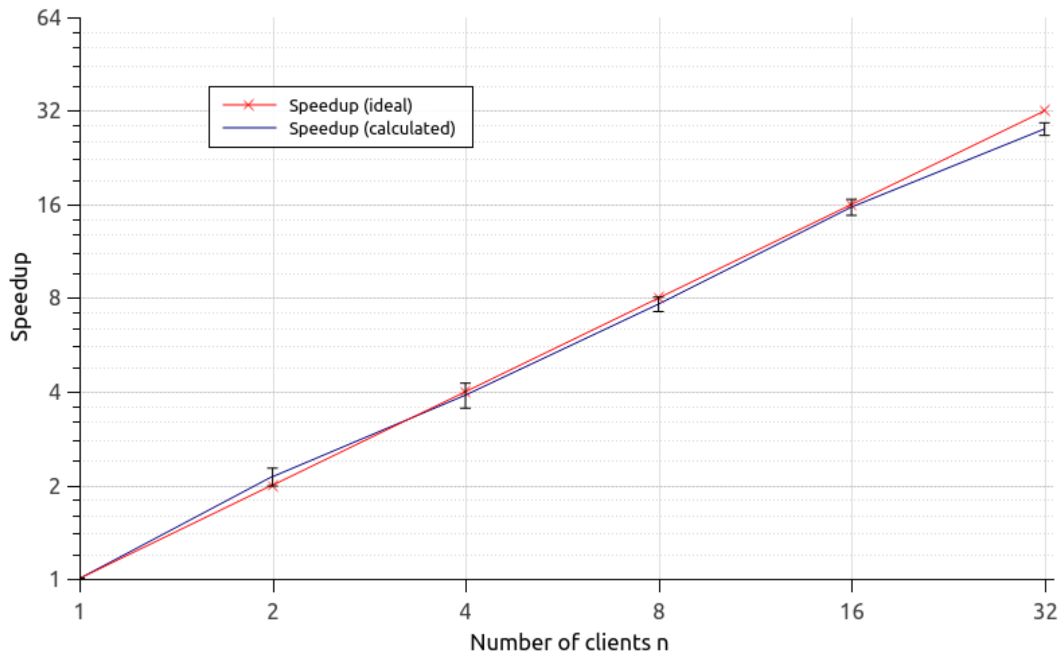


Figure 5.20: Speedup with load-balancing of matrix chain multiplication example plotted against the number of clients

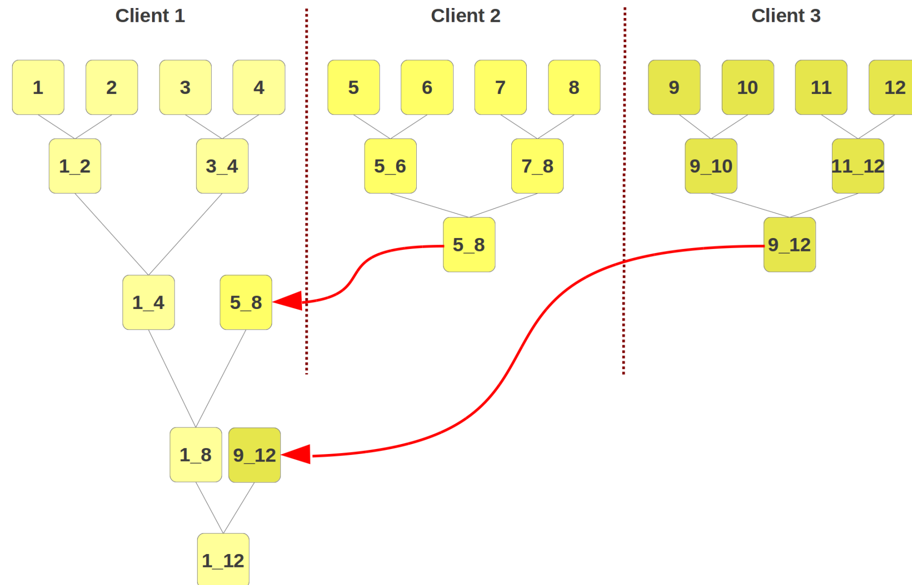


Figure 5.21: Ideal case of the matrix chain multiplication problem on three clients

leads to more communication processes between the clients. An ideal solving work flow is depicted by Figure 5.21. For the clarity only matrices, not the work units, are shown. As can be seen, the optimal behavior of the program running on three clients is that only two communication processes are needed. This requires the clients to have the same performance as well as work units with the same computation time.

A more realistic case is shown in Figure 5.22. Client 1 is simulated to be slower and thus, the scheduler links the work unit to solve the Matrices 3 and 4 to Client 3, which is faster. As can be seen the whole process gets more complicated than the ideal case and one communication process more is necessary.

The significant difference between the schedulers with and without load-balancing is shown by the MPE diagrams of the Figures 5.18 and 5.23. If load-balancing is enabled, all clients are loaded with work, but there are more transmission processes between the clients required as explained with the help of Figure 5.22. This can be easily observed because there exists a large number of sending operations at the end of the program (red blocks), however, the program speedup is close to the ideal behavior, which justifies the use of the load-balancing algorithm.

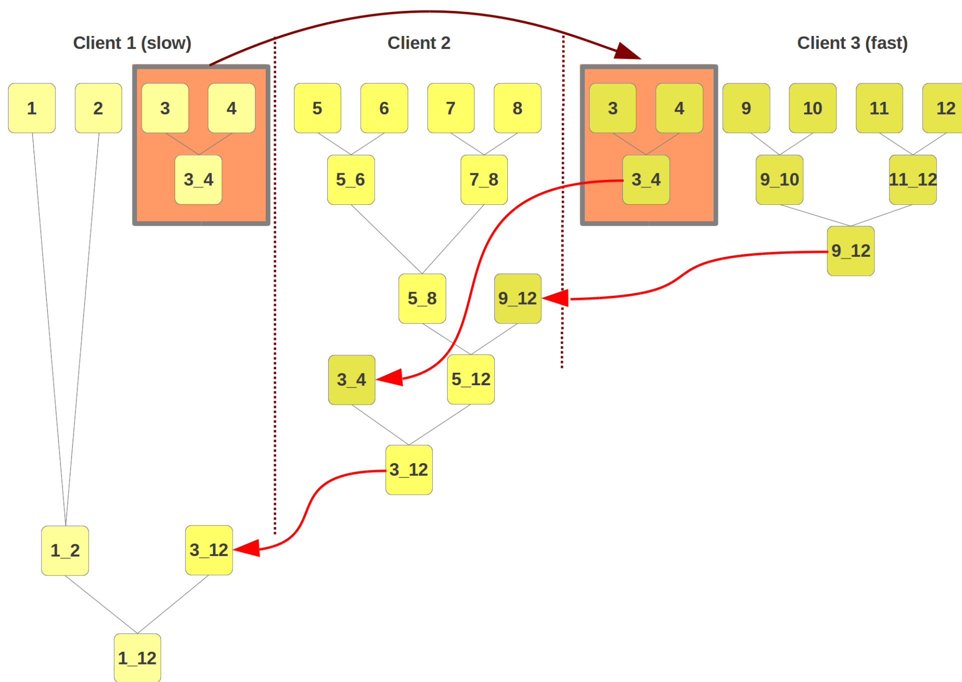


Figure 5.22: Load-balanced case of the matrix chain multiplication problem on three clients

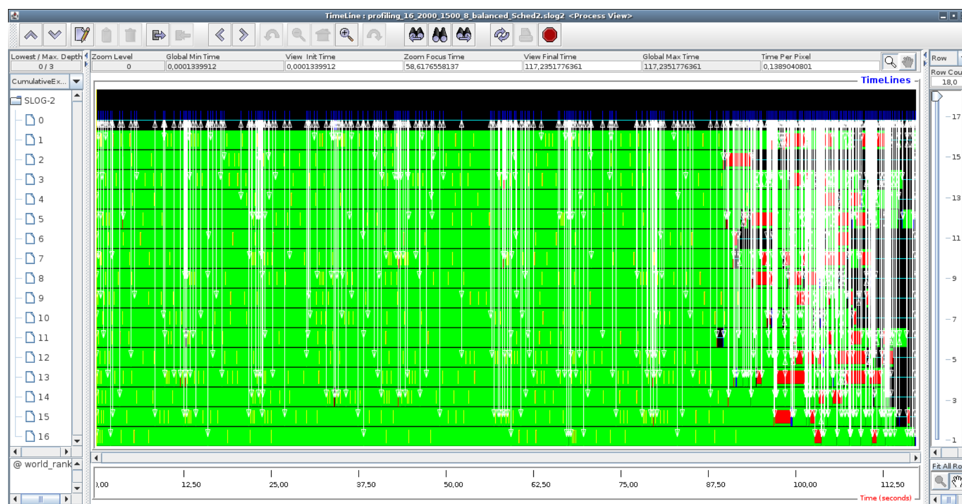


Figure 5.23: MPE diagram of the same problem as in Figure 5.18 with enabled load-balancing

6 Parallelization of NEO-2

6.1 Introduction

This chapter describes the first real physical problem to which the parallelization library is applied. This is the parallelization of NEO-2. As described in Chapter 5 the required tasks to adapt the library to a special problem are to identify work units and to adapt the scheduler. Two different program tests have to be done after the parallelization. The first one is to verify the correctness of the results of the parallelized code with the help of reference values and the second one is to evaluate the program's runtime.

6.2 Analysis of the computational problem

The integration of the library into NEO-2 has to be well-planned in order not to corrupt an algorithm, e.g., due to logical errors which cause the code to produce wrong results after the parallelization. The operation principle of NEO-2 has already been explained in Chapter 2. Therefore, the sequential and parallel parts are already defined. For reasons of clarity Figure 2.3 of Chapter 2 describing the solving and joining of propagators is shown again in Figure 6.1.

An adapted parallelized form, according to the mechanism acquired during the development of the matrix chain multiplication example, is depicted in Figure 6.2. As can be seen, for the parallel solution of the NEO-2 problem, all clients process the same initial data. As soon as these initial data have been computed, each client has all physical information to solve all the propagators. In order to obtain an increase of performance, all clients process different propagators.

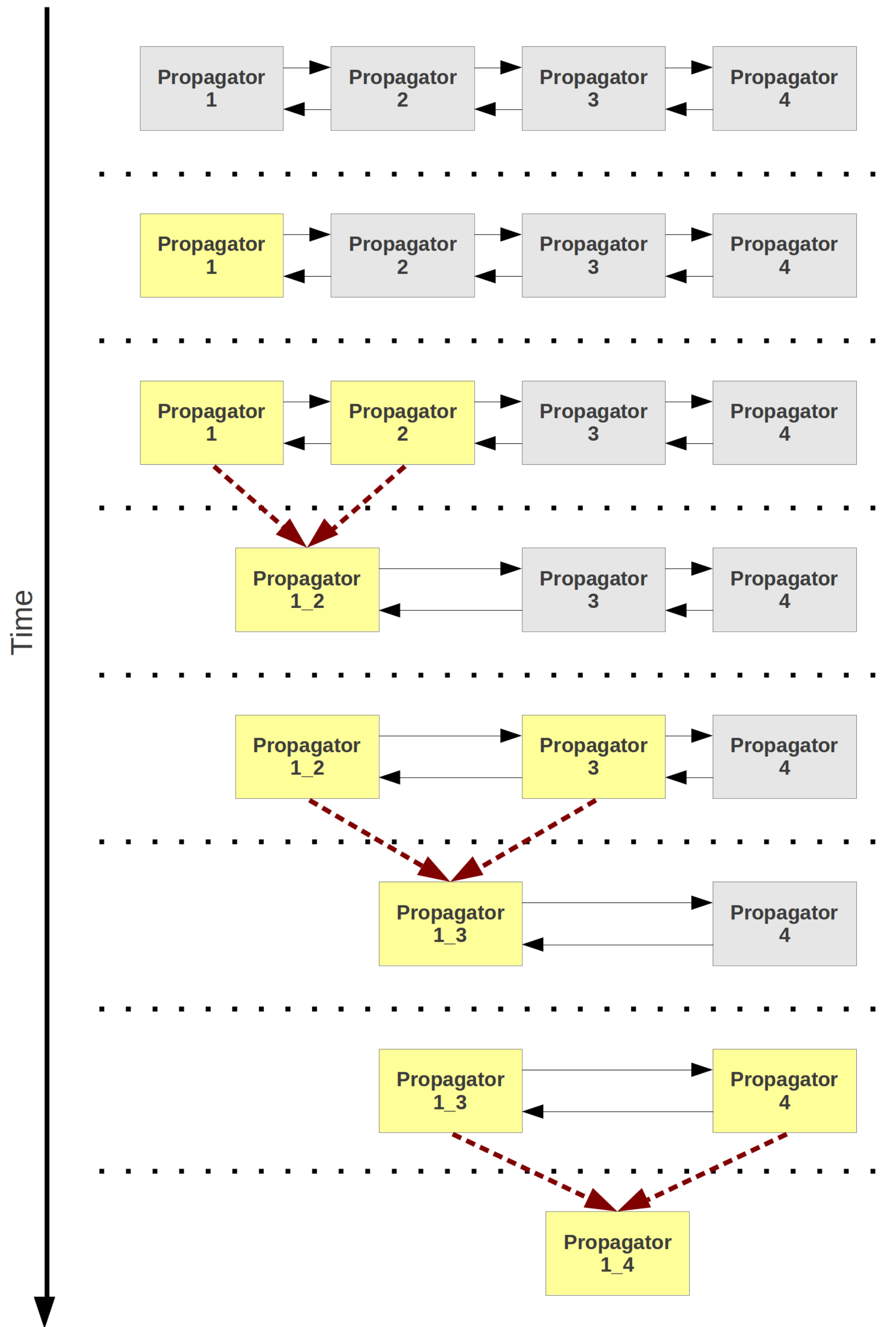


Figure 6.1: Concept of solving and joining propagators

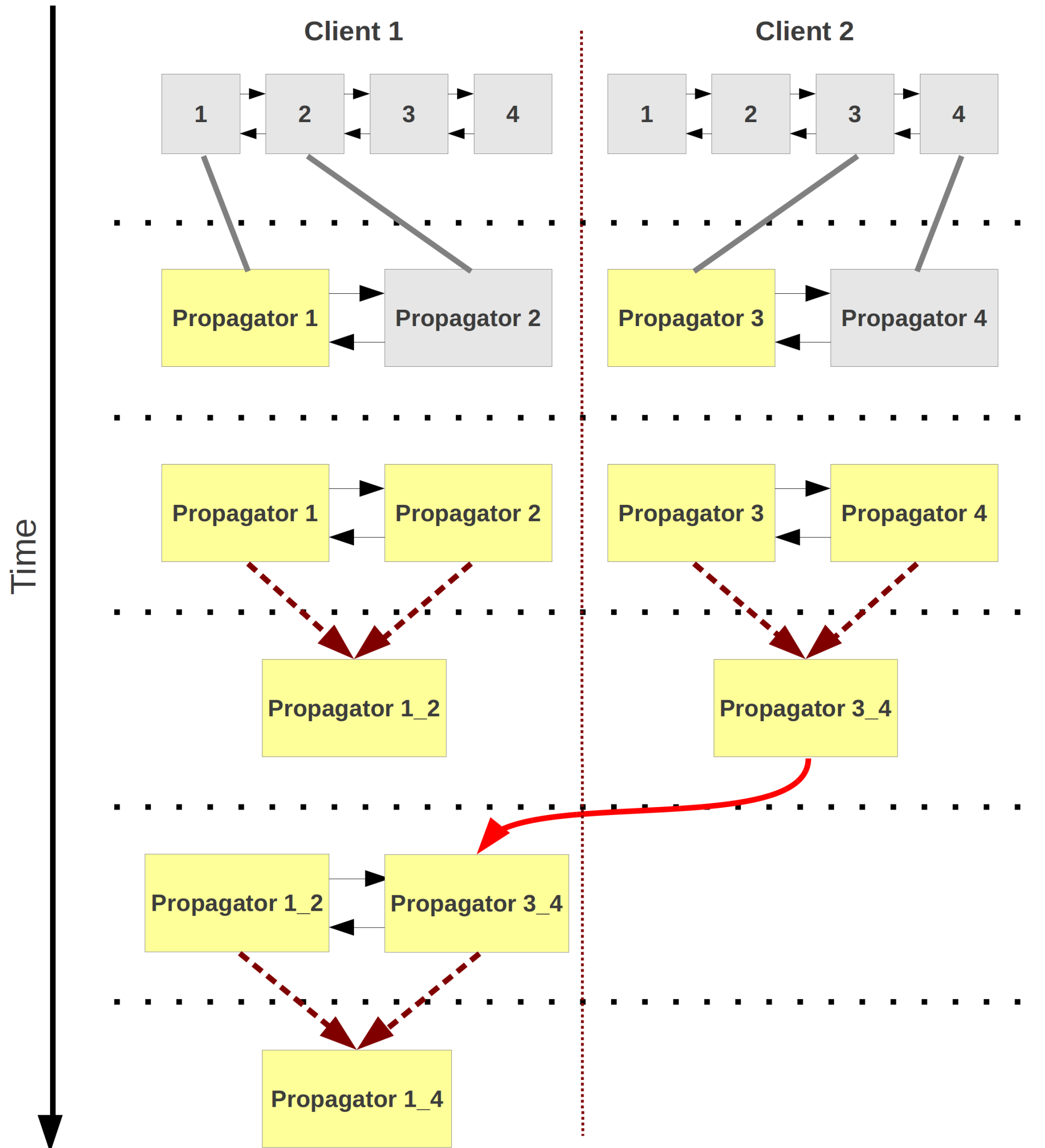


Figure 6.2: Concept of solving and joining propagators on two clients

6.3 Adaption of the library

The parallelization library can be adapted to parallelize NEO-2 in the same way as the matrix chain multiplication, because of the similarity between those problems. At first, the work units have to be defined to separate the problem into sub-problems. Two new work units are required to solve the problem:

- One to compute a propagator and to store it locally,
- and another one to join two neighboring, already computed, propagators in order to create a resulting propagator which has all relevant physical information of its origins.

6.3.1 Definition of the work units

The approach is to design work units, which call methods of NEO-2 instead of moving functionalities of NEO-2 into the work units. Thereby, it is possible to reduce code changes in the complex code of NEO-2. The two required work units have to be developed just to call appropriate functions of NEO-2 and to provide logistical functionality in order to store the resulting propagators. This approach makes it possible to run the code in both, sequential and parallel mode after the parallelization, because the base of NEO-2 is not changed.

The UML class diagram for this problem is shown in Figure 6.3. Both new work units (`Neo2SolvePropagator`, `Neo2JoinPropagators`) have exactly one propagator as result. The main issue is to pack and unpack the resulting propagators. In order to stay within the object-oriented concept, a third work unit acting as a parent class of all further classes which have to pack and unpack propagators is created. This parent class `Neo2GenericWorkunit` provides features which are required by the children classes:

- The storage of the resulting propagator,
- and the appropriate packing and unpacking algorithms.

The work unit `Neo2SolvePropagator` is designed to compute all propagators with tags between the attributes `prop_tag_start` and `prop_tag_end`. It is required that these propagators belong to the same field period in order to

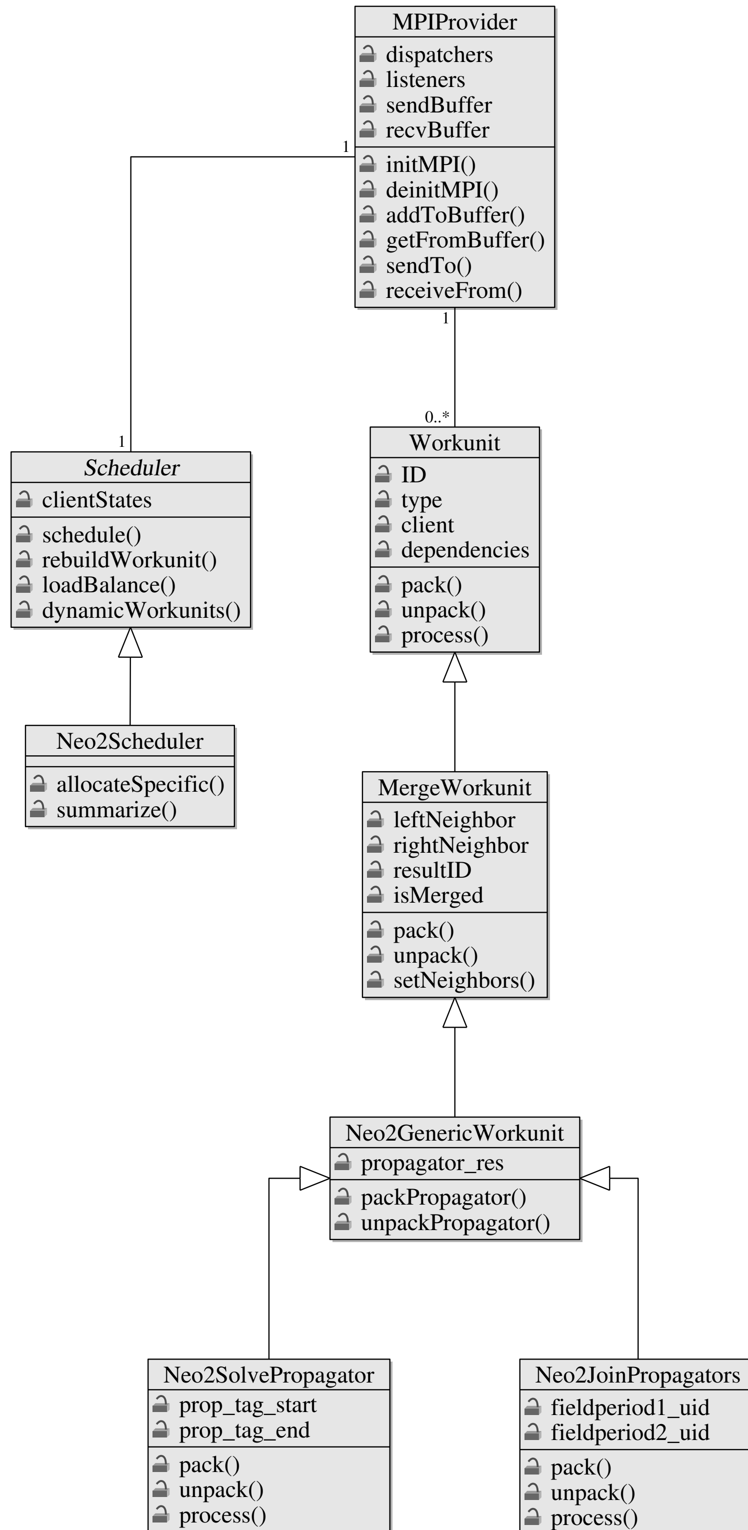


Figure 6.3: UML class diagram of parallel NEO-2

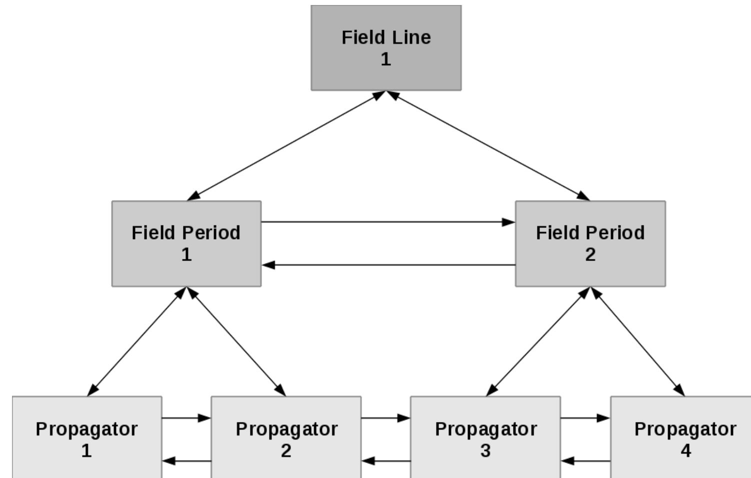


Figure 6.4: Principle of the initial data structure of NEO-2

simplify the joining process. All neighboring work units are joined automatically by NEO-2 during this process, therefore, each work unit only has one propagator as result. The work unit `Neo2JoinPropagators` joins the resulting work units of `Neo2SolvePropagator` of two neighboring field periods. The result is a new propagator which can again be joined with a neighboring one. The data structure of NEO-2, which has been analyzed in Chapter 2, is shown in Figure 6.4 again.

The load-balancing and the method to dynamically create new work units of the code for the matrix chain multiplication can be used because of the similarity between the problems. By comparing the right neighbor of one propagator with the left neighbor of another propagator, the scheduler evaluates the propagators ability to be joined and creates a work unit of the type `Neo2JoinPropagators`.

6.3.2 Implementation

The package NEO-2 provides a standard make-file for different compilers to build an executable file. In order to compile NEO-2 with the library, which is written to fulfill the Fortran 2003 standard, the GFortran part of the standard make-file was converted to a CMake file. The build system CMake is described in Chapter 4. To ensure backward compatibility, all parallelized parts of the program can be disabled by two compiler directives. These have been defined

during the implementation process:

- `MPI_SUPPORT`
  Enables the compilation and activation of all parallel parts.
- `MPE_SUPPORT`
  Enables the compilation of the classes for MPE profiling.

If the code is compiled using the standard make-file, these directives are disabled and, therefore, the code runs in sequential mode.

6.4 Performance measurements

The parallel version of NEO-2 is tested on the Vienna Scientific Cluster - 2 (VSC-2). The official website of the cluster provider [21] states that among others, the compilers GFortran and Intel® Composer are provided. Because of a GFortran version which does not provide Fortran 2003 yet, the compiler Intel® Composer XE 2013 was used to compile the code. NEO-2 uses some external libraries which also have to be compiled on the cluster to ensure good performance.

6.4.1 Definition of the test case

In order to reproduce the runtime tests of this section, a test case is defined in Table 6.1. The full configuration file `neo2.in`, which is read by NEO-2 as soon as the program is started, is printed in Appendix B.

Table 6.1: NEO-2 test case

Parameter	Value
Device	W7-X
Collisionality ν^*	$5.58 \cdot 10^{-4}$

6.4.2 Evaluation of the performance

The parallelized NEO-2 is the first test case in which the sequential part of a program has significant influence on the runtime, therefore, three different

runtimes have to be measured. The measurement of these time values is already provided by the parallelization library, as described in Section 5.2.3:

- Total runtime
- Sequential runtime (Time before scheduling)
- Parallel runtime (Scheduler runtime)

The performance analysis of the parallelized NEO-2 on 32 cores is given in Program Output 5. As can be seen, the number of data requester objects on the processing nodes varies because of the load-balancing algorithm. The time to send a processed propagator is about one millisecond, while the packing time is significantly smaller because the output of the packing times shows zero seconds. Therefore, it is assumed that the send- and pack-methods of the MPI implementation offer very good performance.

Plotting the different runtime values against the number of clients results in Figure 6.5. The error bars are based on the investigation of the calculated errors of the runtime measurements in the integration example (Table 5.3). When evaluating the behavior of the sequential runtime while the number of clients is increased, it can be seen that the time for the sequential part also increases, although expected to be constant. For further evaluation of this effect, Figure 6.6 has been created to show that the sequential runtime increases sharply between 8 and 16 processes. Since it is only possible to reserve whole compute nodes on VSC-2, there were no data points recorded between 8 and 16 processes [33].

In order to give an explanation for this behavior, the hardware specifications of VSC-2 have to be considered. Each node consists of 2 multi-core processors with 8 physical cores each [21]. In Figure 6.6 it can be seen that the sequential runtime increases when the number of clients equals the number of physical cores of one single multi-core processor. Therefore, it is assumed that this indicates a hardware issue. In the paper of Diamond et al. [34] it is stated that memory bottlenecks may occur in multi-core machines if algorithms or source codes are not optimally optimized for such environments. Since the computation process of the propagators uses external libraries, this issue is not easy to resolve and not further investigated in this thesis.

Program Output 5 NEO-2 test run on VSC-2 with 32 processing cores (with simplified compute node names)

```
Scheduler needed      364.672 s for processing all workunits.
Scheduler spent       364.494 s for waiting.
Balanced workunits:        25

----- PERFORMANCE ANALYSIS (Client) -----
Client          Workunits   Time[s]   Total[s]    DataRequesters   Time[s]    Packtime[s]
 1 on Node2            39     8.985    350.407                 1     0.001          0.000
 5 on Node2            42     8.595    360.986                 3     0.001          0.000
 2 on Node2            32    11.190    358.073                 3     0.001          0.000
 3 on Node2            30    11.588    347.647                 2     0.001          0.000
 4 on Node2            38     9.289    352.980                 2     0.001          0.000
 6 on Node2            38     9.362    355.756                 2     0.001          0.000
 7 on Node2            30    11.855    355.652                 2     0.001          0.000
 9 on Node2            24    14.970    359.280                 2     0.001          0.000
10 on Node2            28    12.965    363.024                 2     0.001          0.000
11 on Node2            28    12.599    352.761                 1     0.001          0.000
12 on Node2            26    13.358    347.316                 1     0.001          0.000
13 on Node2            28    12.484    349.559                 1     0.001          0.000
18 on Node1            30    12.007    360.202                 2     0.000          0.000
14 on Node2            36    10.019    360.682                 3     0.001          0.000
16 on Node1            32    11.105    355.361                 2     0.001          0.000
15 on Node2            44     8.024    353.059                 2     0.001          0.000
17 on Node1            28    12.671    354.794                 1     0.001          0.000
19 on Node1            28    12.737    356.649                 2     0.001          0.000
 8 on Node2            28    12.601    352.818                 1     0.001          0.000
20 on Node1            26    13.540    352.034                 1     0.001          0.000
21 on Node1            34    10.588    359.977                 3     0.001          0.000
22 on Node1            30    11.901    357.034                 2     0.001          0.000
24 on Node1            32    11.244    359.815                 1     0.001          0.000
28 on Node1            28    12.538    351.052                 1     0.001          0.000
23 on Node1            26    13.654    355.015                 1     0.001          0.000
25 on Node1            24    14.961    359.068                 1     0.001          0.000
26 on Node1            30    12.035    361.060                 1     0.001          0.000
27 on Node1            24    14.441    346.582                 1     0.001          0.000
29 on Node1            38     9.310    353.768                 4     0.001          0.000
30 on Node1            38     9.224    350.530                 2     0.001          0.000
31 on Node1            34    10.278    349.466                 3     0.001          0.000

Runtime analysis
----------------
Complete runtime:        1286.769 s
Scheduler runtime:        374.586 s
Time before scheduling:   912.183 s
Time after scheduling:      0.001 s
```

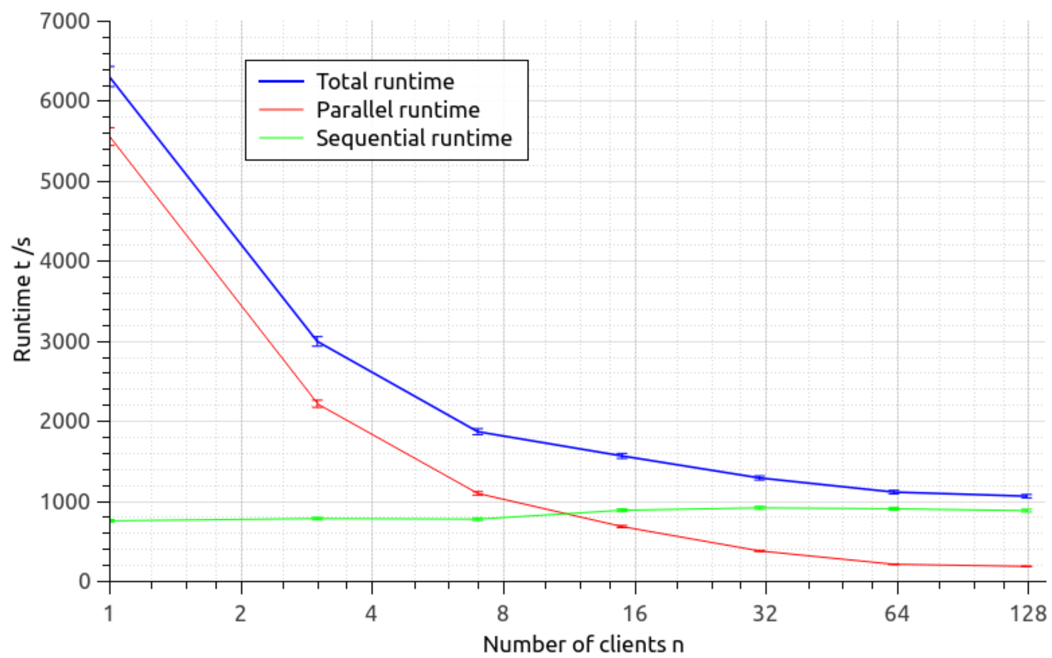


Figure 6.5: Total, parallel and sequential runtimes of a NEO-2 test case

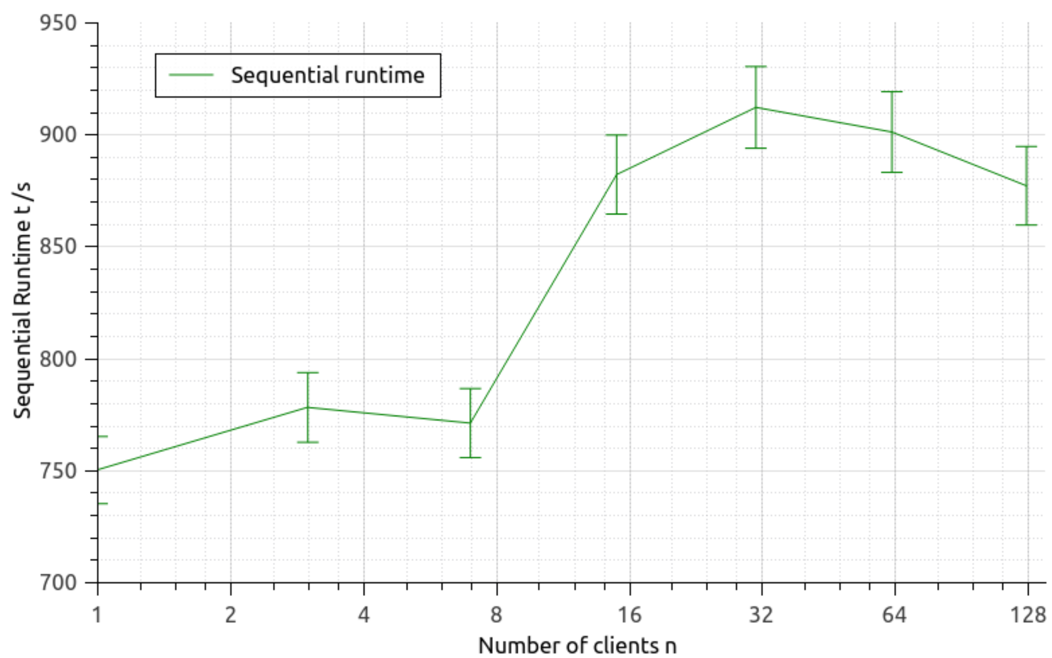


Figure 6.6: Sequential runtime of a NEO-2 test case

6.4.3 Further analysis of the speedup

The evolution of the program's speedup is shown in Figure 6.7. The calculated speedup is evaluated by the ratio of the runtime of a single client to the runtime of many clients as expressed in Formula (5.4). The uncertainties are calculated by the propagation of uncertainty as in Equation (5.9) in Section 5.2.3. In order to compare the calculated speedup curve to Amdahl's law, a non-linear fit with the fit function of Equation (1.1) is performed. Amdahl's law is designed to describe problems with constant f [4], therefore, the fit in Figure 6.7 does not perfectly fit the curve because of the varying sequential runtime.

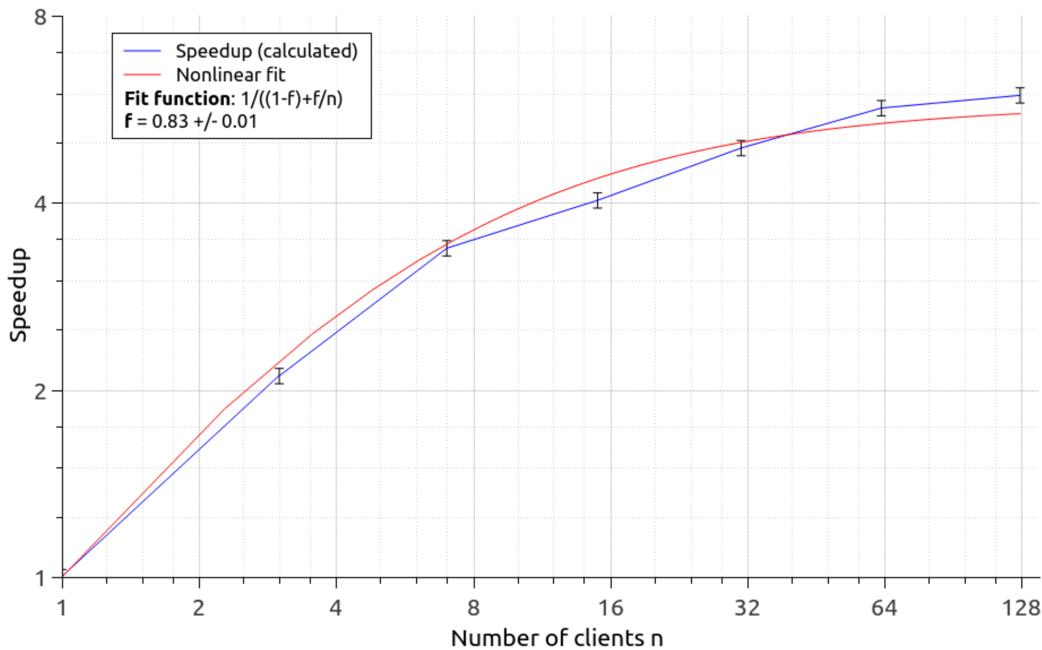


Figure 6.7: Calculated speedup of a NEO-2 test case and fitted speedup by Amdahl's law

The result of the non-linear fit is

$$f = (0.83 \pm 0.01),$$

which states that about 17 percent of this test case of NEO-2 are not paralleliz-

able. Using Equation (1.2) delivers the following result:

$$S_{\max} = \lim_{n\to\infty} \frac{1}{(1-f)+\frac{f}{n}} = \frac{1}{1-f} = (5.9 \pm 0.4) \tag{6.1}$$

The uncertainty of 0.4 is evaluated by the propagation of uncertainty from the book of Bartsch [29]:

$$\Delta S_{\max} = \sqrt{\left(\frac{dS_{\max}}{df}\right)^2} = \left|\frac{\Delta f}{(1-f)^2}\right| = 0.4 \tag{6.2}$$

6.4.4 Conclusion of the performance analysis

The parallelization of this test case for NEO-2 is a success, because the code shows a good speedup evaluation on a computer cluster. The non-linear fit of Figure 6.7 delivers an estimation for the maximum speedup $S_{\max} = (5.9 \pm 0.4)$, which means that the code can run about 5 to 6 times faster than the sequential version at maximum. This limit is caused by the sequential part of the code (about 17 percent), which has significant influence on Amdahl's law. In Figure 6.7 it also can be seen that the maximum speedup is nearly reached with 32 processing cores, therefore, it would make no sense to run the code on a higher number than 32 cores. Such a case would just lead to a waste of energy without a win of performance.

However, these performance analysis refers to one special configuration of NEO-2 (W7-X device, collisionality $\nu^* = 5.58 \cdot 10^{-4}$) and the values for the relative parallel part f and the maximum speedup $S_{\max}$ may change radically for other configurations and other collisionalities.

6.5 Verification of the results

A plasma which is confined by a toroidal magnetic field, e.g., a Tokamak or a Stellarator, looses energy and particles due to transport phenomena. In addition to classical transport because of collisions, the neoclassical transport considers the toroidal geometry of the confinement. Instead of computing the exact orbits of the particles which gyrate around the magnetic field lines, in

the neoclassical theory the drift of the guiding center perpendicular to the field lines is investigated. The magnetic mirror effect causes trapped particles in so-called banana-orbits, which have high influence on the diffusivity of the plasma [35].

In order to ensure that an algorithm does not become corrupted due to the parallelization, the results produced by the parallel code have to be verified by comparison with reference values of the sequential code. In the paper of Beidler et al. [36] it is stated that three mono-energetic transport coefficients are required to describe the neoclassical transport in a plasma which is confined by a Stellarator:

- D_{11} for the description of the radial transport.
- D_{33} for the description of the parallel transport.
- D_{31} for the description of the bootstrap current.

These coefficients can be obtained by the results of NEO-2 and are plotted against the collisionality ν^*. The collisionality parameter and the device can be set as input argument to NEO-2 by configuration files.

To reproduce the plots of the mentioned paper [36], the mono-energetic transport coefficients have been computed for collisionalities between $\nu^* = 10^{-5}$ and $\nu^* = 10^2$ for a W7-X device. The results are shown in Figures 6.8, 6.9, and 6.10. The results from the parallel runs are the same as those stated in the paper [36], however, they were computed in a shorter time.

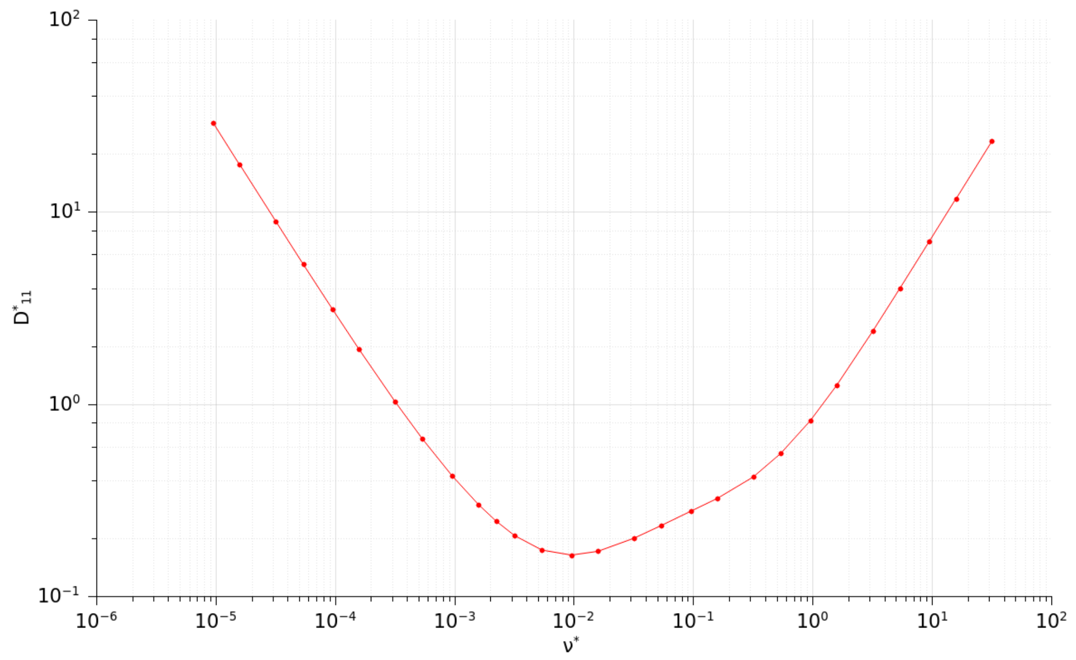


Figure 6.8: Normalized mono-energetic radial transport coefficient as a function of the collisionality

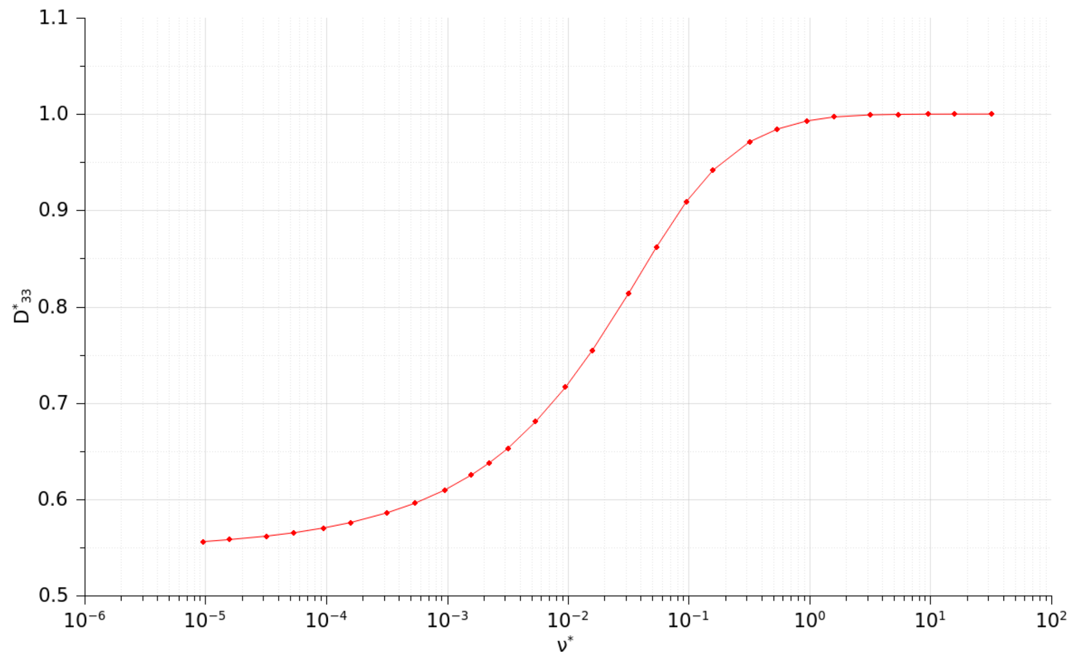


Figure 6.9: Normalized mono-energetic parallel transport coefficient as a function of the collisionality

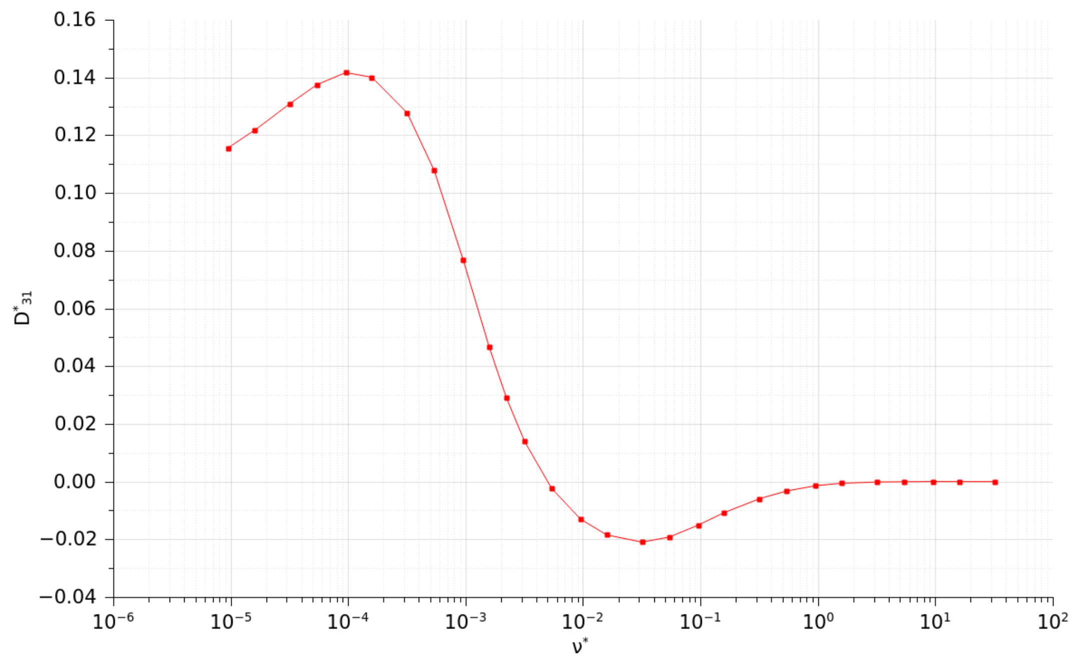


Figure 6.10: Normalized mono-energetic bootstrap current coefficient as a function of the collisionality

7 Use of the library

7.1 Introduction

In order to ease the use of the library for more complex tasks, this chapter provides a manual for adapting the library to solve different problems. In Appendix A the commented source code of the numerical integration example is given to improve the clarity of the integration process of the library to existing code.

7.2 Technical information

This section provides some technical information to compile the library and its example programs.

7.2.1 Compilation of the library

As already mentioned, the library implementation fulfills the Fortran 2003 standard. Therefore, during the development of this work the compiler GFortran in version 4.6 and later was used to have support for the newest standards. However, the test machine with the operating system Debian Linux 6.0 only provided version 4.4.5 of GFortran [37]. It is not possible to link the preinstalled MPI module of the test system to Fortran 2003 code, because the compiler complains about a version conflict of the module files.

A solution to this issue is to install the newest release of OpenMPI locally on the test machine. OpenMPI can be obtained from its official website [8]. This package can be compiled with GFortran version 4.6 to obtain a module-file which can be linked to the Fortran 2003 code of the library.

7.2.2 Source code

Due to steady further development of the library and its code size, the source code is not added to the printed form of this thesis. The developed library, including the example programs, is located on the server of the Institute of Theoretical and Computational Physics of the Graz University of Technology. The compilation process creates a library, which has to be linked to the program to be parallelized.

In order not to overfill this chapter, the full source code of the integration example is found in the Appendix A. The documented source files for the main program, the module-file for the numerical integration, the specialized work unit, and the specialized scheduler are provided.

7.3 Adaption to a specific problem

The parallelization library provides a generic work unit with an empty processing method. In order to define the task of a work unit, it has to be inherited and adapted, as explained in the former test cases about numerical integration and matrix multiplications in Chapter 5.

7.3.1 Work units

First of all, the problem to be parallelized has to be separated into work units. As stated in Section 1.2.1, there are three major options to parallelize code:

- Physical options
- Mathematical options
- Programmatic options

The questions of the following sections help to find the necessary information to design effective new work units.

Which job should be fulfilled by the work unit?

As already mentioned, the problem has to be separated into smaller sub-problems, which can be solved independently from each other. The smaller the sub-problems, the more flexible the scheduler can distribute them. However, in the case of too small units, the clients may compute them in shorter time than the scheduler needs for an iteration process over all work units. This causes longer waiting times for the clients, as evaluated by the simulation of a slow scheduler in Section 5.3.3. Otherwise, in a situation with only a few work units and many clients, the scheduler may not be able to provide jobs to all clients.

Example In the testing scenarios work units have been created which integrate parts of functions, multiply matrices, or send and receive other work units.

Which data does the work unit need to process?

This task defines the input arguments of the work unit to process its job. In the parallelization library all clients run the same executable file, which makes it possible to run commands to compute initial conditions before the clients begin to wait for scheduler commands. These initial data are available on all processes without the need of an initial work unit to compute them.

The packing and unpacking processes of work units are rather a programmatic problem than a physical or mathematical one. It should be considered that some data structures may cause issues when being packed, e.g., linked lists because of their pointers. The pointer addresses are only valid in the memory of the client defining it, therefore, it can not be packed and sent to another client like an integer number. One possible solution to transmit a linked list is to convert it into an array and to rebuild it into a linked list on the receiving side.

Due to an attribute for required work units, it is possible to define dependencies. However, the scheduler has to wait for all work units to be processed until a particular one, which depends on the others, can be sent to a client. This may cause a situation in which clients have to wait several iteration cycles of the scheduler to receive new work units because of an ineffective set of dependencies.

Example A work unit to integrate a part of a function requires parameters which define the integration domain and the function to be integrated.

Does the work unit require data from other, already processed, work units?

The predefined work unit `DataRequester` enables the clients to exchange data. However, depending on the kind of problem, the user has to create these objects accordingly to solve the problem. Therefore, the scheduler is inherited and adapted as described later in Section 7.3.2.

Example The problem of exchanging work units between clients can be explained by the matrix chain multiplication test case. Assuming that one work unit multiplies two matrices and stores the resulting matrix locally, a following work unit requires this resulting matrix to process the next result. This situation is depicted in Figure 5.14.

Which results does the work unit have?

The result of the sub-problem represented by a work unit can be a set of variables (new attributes of the inherited class), which are stored on the client until the scheduler requests them. The parallelization library provides a special kind of predefined work unit, called send-back work unit. If the send-back attribute of a generic work unit is set to the value true, the client is forced to send back the processed work unit to the scheduler instead of storing it locally.

Example This is a comfortable way for problems such as the integration test case. In such problems there is no need to exchange data between clients, because the results of the work units can be combined in an arbitrary order. The summarization process is done by the scheduler as soon as all units are completed.

7.3.2 Scheduler

If the work units required to solve the parallel part of the problem are designed, the next task is to define how they are distributed among the clients. The

implemented library of this master thesis provides a generic scheduler which is designed to be adapted to obtain a scheduler specialized to the kind of problem. Similar to the design process of new work units there are a few questions to be answered to design a new scheduler.

Are initial conditions required on each client?

Most of the physical problems need initial conditions which have to be equal for all the clients. A possible solution for this kind of problem is to compute all required initial data on the master process and then broadcast them to all clients. Though, this method seems to be a clean way to solve this problem, it has the disadvantage that the clients will just wait and may waste resources. Another disadvantage of this method occurs when a large amount of initial data is needed, this may cause the transmission process to take more time than the calculation itself.

Therefore, instead of computing the initial data on the master process and sending them to the waiting clients, all clients can compute them at the same time. This method is used to parallelize NEO-2 as described in Chapter 6.

Example Before the parallelized NEO-2 can start the distribution of the propagators among the clients of the parallel environment, a large amount of initial data has to be computed to obtain the total number and the structures of the propagators.

Are dynamically created work units required?

The scheduler of the library supports the dynamic creation of new work units during the scheduling process. After each iteration process, which is depicted in Figure 4.1, the scheduler calls a method which has to be overwritten in the inherited scheduler class to create new tasks. These new work units have to be added to the list of waiting work units, so that the new work unit is taken into account by the scheduler at the next iteration over this list.

Example The dynamic creation of work units is needed to solve the matrix chain multiplication in the pertinent example. In contrast to the integration

example, where all work units were defined before the scheduling process was started, the different ways to get the results of a chain multiplication cause the need for appropriate reactions on client responses to keep the solving process working. Before running the program it is not possible to predict which client will actually compute which work units because of the load-balancing mechanism.

Is a summarization process needed after all jobs are done?

The summarizing routine is called after the completion of the scheduling process on all processes. It may be used to print results on the master process or to free memory on the clients.

Example This function is used to sum up the results of the different subintervals of the integration example.

8 Conclusion and outlook

The aims of this master thesis, which were the development of a flexible library to solve physical problems by parallel computation and the parallelization of NEO-2 as a first physical purpose of the library, have been fulfilled. To ensure the adaptability of the developed library to many kinds of physical problems it is designed in an object-oriented way and implemented to fulfill the Fortran 2003 standard. The use of MPI (Message Passing Interface) allows to run the parallel code on multi-core systems as well as on computer clusters.

The test scenarios of the parallelized NEO-2 yielded better performance than the sequential code. However, only known physical results were computed up to now, in order to verify the program's integrity by comparison to reference values.

The developed parallelization library was designed to be easily further developed to ensure its application in future projects.

Acknowledgments

Some persons supported me during the work on this master thesis and I would like to thank them.

I am very grateful to *Ao.Univ.-Prof. Dipl.-Ing. Dr.phil. Martin Heyn* for the possibility to write this master thesis at the Institute of Theoretical and Computational Physics.

Special thanks to *Ass.Prof. Dipl.-Ing. Dr.techn. Winfried Kernbichler* for the helpful support during the last year and for listening to all kinds of problems.

I would like to thank all members of the Plasma Physics Division, especially *Andreas F. Martitsch* for many helpful discussions.

Many thanks to *Andreas Hirczy* who always supplied me with the latest software versions of the used tools.

I am very grateful to *Günter Krois* for forcing me to have some coffee breaks and for the true friendship since the beginning of this study.

Thanks to all of my friends and colleagues with whom I worked closely during the last years.

This work could not have been written without the endless support of my mother *Emmy*, my father *Franz*, my sister *Tanja*, and my girlfriend *Annemarie*. Thank you very much for being here for me and for listening to me when I tried to explain some "interesting" physical stuff.

A Integration example

A.1 Main program

```
!> Demonstration program for numerical integration
program simpleintegrate
  ! Module for generic scheduler of the parallelization library
  use scheduler_module
  ! Module for numerical integration
  use integrate_module
  ! Module for specialized work unit to integrate a subinterval
  use wuIntegrate_module
  ! Module for specialized scheduler to distribute the new work units
  use simpleIntScheduler_module

  implicit none

  ! Declaration of a new work unit
  class(wuIntegrate), pointer :: wu
  ! Declaration of the new scheduler
  type(simpleIntScheduler) :: sched
  ! Some local variables
  integer :: k, NumOfWUs, n_wu
  double precision :: a_wu, b_wu, step

  ! Initialize MPI (mpro is the singleton name of the MPIProvider)
  call mpro%init()
  ! Initialize the integration module
  call initIntegrationModule()

  ! Evaluate if program runs in parallel mode
  if (mpro%getNumProcs() == 1) then
    ! Normal sequential program execution
    ! Call integration-routine of integration module
    call integrate()
  else
    ! Parallel program execution
    ! Initialize the new scheduler
    call sched%init()

```

```
    ! Define the number of work units
39  NumOfWUs = subintervals

41  ! Define attributes of first work unit
    a_wu = a
43  step = (b-a) / NumOfWUs
    b_wu = a + step
45  n_wu = n/NumOfWUs

47  ! Only the master process (rank = 0) prepares the work units
    if (mpro%getRank() == 0) then
49
      ! Print information
51    write (*,"(A, F10.6)") "Subinterval width: ", step

53    ! Call initial work units
      call sched%prepare()
55
      ! Create the work units for integrating the subintervals
57    do k = 1, NumOfWUs

59      ! Allocate and initialize new work unit
        nullify(wu)
61      allocate(wu)
        call wu%init()
63
        ! Assign attributes to work unit
65      wu%a_wu = a_wu
        wu%b_wu = b_wu
67      wu%n_wu = n_wu

69      ! Activate sendBack to receive the result directly on the scheduler
        wu%sendBack = .true.
71
        ! Add the prepared work unit to the waiting list
73      call sched%addWorkunit(wu)

75      ! Step to the next subinterval
        a_wu = a_wu + step
77      b_wu = b_wu + step
      end do
79  end if

81  ! Call the scheduling method of the library
    call sched%schedule()
83  ! De-initialize the scheduler
    call sched%deinit()
85 end if
  ! Close all connections by the use of the MPIProvider
87 call mpro%deinit()
end program simpleintegrate
```

A.2 Integration module

```fortran
!> Module for numerical integration routines
module integrate_module
  ! Use the MPIProvider
  use mpiprovider_module

  implicit none

  ! Define parameters for the integration
  double precision, parameter :: pi = 3.141592653 !> PI definition for the test function
  double precision :: a = -4 !> Left border of full problem (Default = -4)
  double precision :: b = +4 !> Right border of full problem (Default = +4)
  integer :: n = nint(1e6) !> Number sampling points (Default = 1e6)
  integer :: subintervals = 8 !> Number of subintervals (work units)
  double precision :: res !> Result of integration

  ! Define elements of name-list for configuration file
  namelist / nmlIntegrate / a, b, n, subintervals

contains

  ! Initialize the module
  subroutine initIntegrationModule()
    integer :: f = 50
    integer :: stat

    ! Open name-list file
    open(f, file="intconfig.txt", status='old', action='read', iostat = stat)

    if (stat == 0) then
      ! Read input arguments to program (a, b, n, subintervals)
      read(f, nml=nmlIntegrate)
      close(f)
    else
      write (*,*) "No configuration file for simple integration module found, using defaults"
    end if

  end subroutine initIntegrationModule

  !> Test function to integrate
  function fun(x) result(y)
    double precision :: x
    double precision :: y
    double precision :: sig, x0

    ! Definition of the function to integrate (here normal distribution)
    x0 = 0
    sig = 1
    y = 1.0 / (sig * sqrt(2*pi)) * exp(-1.0/2.0 * ((x - x0)/(sig))**2)
```

```
50     !y = 2*x
       !y = x**2
52   end function fun

54   !> Numerical integration routine with trapezoid method
     subroutine integrate()
56     integer :: k
       double precision :: h, x
58
       ! Compute interval width
60     h = (b-a)/n

62     ! Start at the left bound of integration
       x = a
64
       ! Set the result to zero
66     res = 0

68     do k = 1,n
         ! Evaluate the trapezoid formula
70       res = res + h * fun((x + x + h)/2)
         ! Step to the next sampling point
72       x = x + h
       end do
74
       ! Write results to program output
76     write (*,"(A, F10.6, A, F10.6, A, I12, A, I3, A, F10.6)") "Integration from ", a, " to", b,
            " with ", n,&
         " points on client ", mpro%getRank(), ". Result = ", res
78
     end subroutine integrate
80
   end module integrate_module
```

A.3 Specialized work unit

```fortran
!> Module for the inherited workunit wuIntegrate
module wuIntegrate_module
  ! Use generic work unit from the parallelization library
  use genericWorkunit_module
  ! Use the MPIProvider
  use mpiprovider_module
  ! Use the numerical integration module
  use integrate_module

  implicit none

  !> Child class wuIntegrate of genericWorkunit
  type, extends(genericworkunit) :: wuIntegrate
    ! Attributes
    double precision :: a_wu, b_wu !> Bounds of integration per work unit
    integer :: n_wu !> Number of sampling points per work unit
    double precision :: res !> Integration result per work unit
  contains

    ! Methods
    procedure :: pack => pack_wuIntegrate !> Method to pack the work unit
    procedure :: unpack => unpack_wuIntegrate !> Method to unpack the work unit
    procedure :: init => init_wuIntegrate !> Method to initialize the work unit
    procedure :: process => process_wuIntegrate !> Method to process the work unit
    procedure :: print => print_wuIntegrate !> Method to print the work unit

  end type wuIntegrate

contains

  !> The initial method of the work unit
  subroutine init_wuIntegrate(this)
    class(wuIntegrate) :: this

    ! The type is required to rebuild the work unit by the specialized scheduler
    this%type = "wuIntegrate"

    !> Call the initial-routine of the scheduler of parent class
    call this%genericworkunit%init()
  end subroutine init_wuIntegrate

  !> Defines the job of the work unit, in this case, the integration process
  subroutine process_wuIntegrate(this)
    class(wuIntegrate) :: this

    ! Set the values of the module for the integration to the values of the current work unit
    a = this%a_wu
    b = this%b_wu
    n = this%n_wu
```

```
51      ! Call the integration routine of the integration module
        call integrate()
53
        ! Store the result locally
55      this%res = res

57      ! Indicate that the work unit is processed
        this%isProcessed = .true.
59    end subroutine process_wuIntegrate

61    !> Pack attributes
      subroutine pack_wuIntegrate(this)
63      class(wuIntegrate) :: this

65      ! Call pack from parent work unit to store logistical data
        call this%genericworkunit%pack()
67
        ! Call generic add-methods of MPIProvider to push the attributes into the send buffer
69      call mpro%packBuffer%add(this%a_wu)
        call mpro%packBuffer%add(this%b_wu)
71      call mpro%packBuffer%add(this%n_wu)

73      ! If processed, then add the result
        if (this%isProcessed) then
75        call mpro%packBuffer%add(this%res)
        end if
77    end subroutine pack_wuIntegrate

79    !> Unpack attributes
      subroutine unpack_wuIntegrate(this)
81      class(wuIntegrate) :: this

83      ! Call unpack of parent work unit to restore the work unit type and other logistical data
        call this%genericworkunit%unpack()
85
        ! Read attributes from buffer in the same order as they were packed
87      call mpro%packBuffer%get(this%a_wu)
        call mpro%packBuffer%get(this%b_wu)
89      call mpro%packBuffer%get(this%n_wu)

91      ! If processed, then read the result
        if (this%isProcessed) then
93        call mpro%packBuffer%get(this%res)
        else
95
        end if
97    end subroutine unpack_wuIntegrate

99    !> Print work unit information
      subroutine print_wuIntegrate(this)
```

```
101     class(wuIntegrate) :: this

103     write (*,*) "wuIntegrate: ", this%uid, this%a_wu, this%b_wu, this%n_wu, this%res
      end subroutine print_wuIntegrate
105
    end module wuIntegrate_module
```

A.4 Adapted scheduler

```
!> Module for the specialized scheduler
module simpleIntScheduler_module
  ! Use the generic scheduler of the library
  use scheduler_module
  ! Use the specialized work unit
  use wuIntegrate_module

  implicit none

  !> Class SimpleIntScheduler (Inherited from generic Scheduler)
  type, extends(scheduler) :: simpleIntScheduler
    ! Attributes
    ! No additional attributes required in this test case
  contains
    ! Methods
    procedure :: allocateSpecific => allocateSpecific_simpleIntScheduler !> Method to help
        rebuilding the work unit
    procedure :: summarize => summarize_simpleIntScheduler !> Method to summarize the results
  end type simpleIntScheduler

contains

  !> Overwritten function of the generic scheduler to allocate new work units
  function allocateSpecific_simpleIntScheduler(this, wuType) result(res)
    class(simpleIntScheduler) :: this
    character(len=maxStrLen) :: wuType
    class(workunit), pointer :: res

    ! The generic scheduler does not know the new work units and can not call the appropriate
        allocate-method.
    ! Therefore, this method has to be provided.

    ! Evaluate the type of the received work unit, allocate the specialized work unit, and
        return it to the generic scheduler
    nullify(res)
    select case (wutype)
      case ("wuIntegrate")
        allocate(wuIntegrate :: res)
    end select
  end function allocateSpecific_simpleIntScheduler

  !> Overwritten function of the generic scheduler to summarize the results
  subroutine summarize_simpleIntScheduler(this)
    class(simpleIntScheduler) :: this
    double precision :: res
    class(workunit), pointer :: selectWU => null()

    ! Only the master process summarizes the results, which it received due to the sendBack -
        work units
```

```
46    if (mpro%getRank() .eq. 0) then
        ! Initialize result
48      res = 0

50      ! Rewind list of processed work units to the start
        call mpro%storage%processedWorkunits%rewind()
52
        ! Iterate over all processed work units
54      do while (associated(mpro%storage%processedWorkunits%currentElement))
          ! Select current work unit
56        selectWU => mpro%storage%processedWorkunits%getCurrent()
          select type (wu => selectWU)
58          ! Evaluate its type in order to have access to the attributes of the inherited work
                unit
            type is (wuIntegrate)
60          ! Sum up the integration results
            res = res + wu%res
62        end select

64        ! Go to the next processed work unit
          call mpro%storage%processedWorkunits%gotoNext()
66      end do

68      ! Print the result to the program output
        write (*,*) "The integration result is ", res
70    end if
    end subroutine summarize_simpleIntScheduler
72
  end module simpleIntScheduler_module
```

B NEO-2 configuration

B.1 Content of neo2.in

```
! This is the input file for neo2
! overwritten data from default file neo2.def
!
&settings
  phimi=0.d0                ! beginning of period
  nstep=960                 ! number of integration steps per period
  nperiod=1000              ! number of periods
  mag_nperiod_min=150       ! minimum number of periods
  mag_save_memory=0         ! saving memory
  rbeg= 210d0               ! starting R
  proptag_begin=0 !1184     ! 0 take first from fieldline, otherwise begin
  proptag_final=0 !1184     ! 0 take last from fieldline, otherwise end
  mag_magfield=1            ! 0 homogeneous, 1 normal
  magnetic_device = 1       ! 0 Tokamak, 1 W7-AS
  mag_coordinates = 1       ! 0 cylindrical, 1 Boozer
  boozer_s = 0.25d0
  boozer_theta_beg = 0.0d0
  boozer_phi_beg = 0.0d0
  mag_start_special=0       ! 0 original, 1 abs max, 2 abs min, 3 proptag_begin
  mag_cycle_ripples=1       ! 0: old behaviour, 1: cycle through
  mag_close_fieldline=2     ! 1: close fieldline artificially 0: do not
  aiota_tokamak=0.35145
  eta_part_global = -1
  eta_part_globalfac = 3.0d0
  eta_part_globalfac_p = 3.0d0
  eta_part_globalfac_t = 3.0d0
  eta_part_trapped = 0
  eta_alpha_p = 4.0d0
  eta_alpha_t = 2.0d0
  solver_talk = 0          ! 0: silent, 1: talks
  mag_symmetric = .false. ! .true.
  mag_symmetric_shorten = .false.  ! .true.
  mag_dbhat_min = 5d-2
/

&collision
  conl_over_mfp = 1d-3      ! collisionality parameter
  lag = 4                   ! number of Laguerre polynomials
```

```
  leg = 3                 ! number of Legendre polynomials
  legmax = 0              ! maximum number of Legendre polynomials
  z_eff = 1.d0            ! effective charge
  isw_lorentz = 1         ! switch
/

! remarks about binsplit
!
! phi_split_mode    1: halfstep, 2: automatic
! phi_place_mode    1: only one point between automatic phi's
!                   2: odd number of points between  automatic phi's
!                      according to hphi * hphi_mult
!
! bin_split_mode    0: no binary split for eta
!                      original eta used (higher eta_part necessary)
!                   1: binary split is done
! bsfunc_modelfunc  1: Gauss
!                   2: prop to exp(-|x-x0|/sqrt(2)/sigma)
!                   3: prop to 1/(|x-x0|^2+\sigma^2)
! mag_local_sigma   0: compute sigma for eta-placement (old)
!                   1: add 2 local sigma-values within ripple

&binsplit
  eta_s_lim = 1.2d1
  eta_part = 20
  lambda_equi = 0               ! 0/1
  phi_split_mode = 2            ! [1/2]
  phi_place_mode = 2            ! [1/2]
  phi_split_min = 3             ! [1/3/5]
  max_solver_try = 30           ! how often the solver tries on error 3
  hphi_mult = 1.0d0             ! 1.0d0 or a little bit more
  bin_split_mode = 1            ! [0/1]
  bsfunc_message = 0            ! 0/1
  bsfunc_modelfunc = 1          ! 1/2/3
  bsfunc_modelfunc_num = 2      !
  bsfunc_local_err = 1.0d-2     ! 3.0d-2
  bsfunc_min_distance = 0.0d0   !
  bsfunc_max_index = 300        !
  bsfunc_max_splitlevel = 30    !
  bsfunc_sigma_mult = 1.618033988749895d0
  bsfunc_sigma_min = 1.0d-20    !
  bsfunc_local_solver = 4       ! [0/1/2/3/4]
  mag_local_sigma = 0           ! [0/1]
  bsfunc_divide = 0! 7 !0       ! [0/1]
  mag_ripple_contribution = 2   ! [1/2] 2 new sigma formula
  boundary_dist_limit_factor = 1d-2
  bsfunc_local_shield_factor = 10.0d0
  bsfunc_shield = .true.
  sigma_shield_factor = 5.0d0
  split_inflection_points = .true.
/
```

```
&propagator
  prop_diagphys = 0       ! 0/1
  prop_overwrite = 1      ! 0/1
  prop_diagnostic = 0     ! 0/1/2/3
  prop_binary = 0         ! 0/1
  prop_timing = 0         ! 0/1
  prop_join_ends = 1      ! 0/1
  prop_fluxsplitmode = 1  ! 0/1/2
  mag_talk = .FALSE.      ! .TRUE. / .FALSE.
  mag_infotalk = .FALSE.  ! .TRUE. / .FALSE.
  hphi_lim = 1.0d-6
  prop_write = 0
  prop_reconstruct = 0
  prop_ripple_plot = 0
/

! settings for plotting
&plotting
  plot_gauss = 0   ! plotting of gauss function in flint  [0/1]
  plot_prop  = 0   ! plotting of propagator info in flint [0/1]
/
```

References

[1] G.O. Leitold. *Computation of neoclassical transport coefficients and generalized Spitzer function in toroidal fusion plasmas.* PhD thesis, Graz University of Technology, 2010.

[2] W. Gropp, E. Lusk, and A. Skjellum. *MPI - Eine Einführung*. Oldenbourg, 2007.

[3] J. Liu, J. Wu, S.P. Kini, P. Wyckoff, and D.K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[4] X.H. Sun and Y. Chen. Reevaluating Amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

[5] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 2002.

[6] OpenMP Architecture Review Board. OpenMP Application Program Interface. Technical report, Version 3.1, July 2011. URL `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, Version 2.2, September 2009. URL `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.

[8] The Open MPI Project. Open MPI: Open Source High Performance Computing. Website, 2012. Available online at `http://www.open-mpi.org`; visited on 24. July 2012.

[9] University of Florida. SuiteSparse: A Suite of Sparse matrix packages. Website, 2012. Available online at `http://www.cise.ufl.edu/research/sparse/SuiteSparse`; visited on 9. December 2012.

[10] Computational Research Division. SuperLU. Website, 2012. Available online at `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/`; visited on 9. December 2012.

[11] A.F. Martitsch. Development of a conservative finite difference scheme for head transport problems in magnetized plasmas. Master's thesis, Graz University of Technology, 2012.

[12] K. Jänich. *Lineare Algebra*. Springer, 2003.

[13] W. Kernbichler, S.V. Kasilov, and G.O. Leitold. DKE Solver NEO-2: Field Line Tracing Revisited. Conference proceedings, 2004.

[14] W. Kernbichler. Private communication, 2012.

[15] W. Küchlin and A. Weber. *Einführung in die Informatik: Objektorientiert mit Java*. Springer, 2005.

[16] Object Management Group, Inc. (OMG). OMG Unified Modeling Language (OMG UML), Infrastructure. Technical report, Version 2.4.1, August 2011. URL `http://www.omg.org/spec/UML/2.4.1/Infrastructure`.

[17] M. Metcalf, J.K. Reid, and M. Cohen. *fortran 95/2003 Explained*, volume 416. Oxford University Press New York, 2004.

[18] Gnu Project. GFortran - GCC Wiki. Website, 2012. Available online at `http://gcc.gnu.org/wiki/GFortran`; visited on 23. July 2012.

[19] Gnu Project. GCC Development Mission Statement. Website, 1999. Available online at `http://gcc.gnu.org/gccmission.html`; visited on 24. July 2012.

[20] Free Software Foundation Inc. GNU General Public License. Website, 2007. Available online at `http://www.gnu.org/licenses/gpl-3.0.txt`; visited on 24. July 2012.

[21] VSC - Vienna Scientific Cluster c/o Vienna University of Technology. VSC-2. Website, 2012. Available online at `http://vsc.ac.at/about-vsc/vsc-pool/vsc-2/`; visited on 5. December 2012.

[22] The Open MPI Project. Open MPI License. Website, 2011. Available online at `http://www.open-mpi.org/community/license.php`; visited on 24. July 2012.

[23] Argonne National Laboratory. MPICH2: High-performance and Widely Portable MPI. Website, 2012. Available online at `http://www.mcs.anl.gov/research/projects/mpich2/`; visited on 12. November 2012.

[24] Argonne National Laboratory. Performance Visualization for Parallel Programs. Website, 2012. Available online at http://www.mcs.anl.gov/research/projects/perfvis/; visited on 12. November 2012.

[25] Kitware, Inc., Insight Software Consortium. CMake - Cross Platform Make - About. Website, 2012. Available online at http://www.cmake.org/cmake/project/about.html; visited on 24. July 2012.

[26] Kitware, Inc., Insight Software Consortium. CMake - License. Website, 2012. Available online at http://www.cmake.org/cmake/project/license.html; visited on 24. July 2012.

[27] Apache Software Foundation. Apache Subversion. Website, 2011. Available online at http://subversion.apache.org; visited on 24. July 2012.

[28] Apache Software Foundation. Apache License Version 2.0. Website, 2004. Available online at http://www.apache.org/licenses/LICENSE-2.0.txt; visited on 24. July 2012.

[29] H.J. Bartsch. *Taschenbuch mathematischer Formeln.* Hanser Verlag, 2007.

[30] ISO/IEC JTC1/SC22. Information technology - programming languages - fortran - enhanced module facilities. Technical report, 2004. URL ftp://ftp.nag.co.uk/sc22wg5/N1601-N1650/N1602.pdf.

[31] The MathWorks, Inc. Numerically evaluate integral, adaptive Lobatto quadrature - MATLAB - MathWorks Deutschland. Website, 2013. Available online at http://www.mathworks.de/de/help/matlab/ref/quadl.html; visited on 23. January 2013.

[32] The Open MPI Project. mpiexec(1) man page (version 1.4.5). Website, 2012. Available online at http://www.open-mpi.org/doc/v1.4/man1/mpiexec.1.php; visited on 23. January 2013.

[33] VSC - Vienna Scientific Cluster c/o Vienna University of Technology. doku:vsc2 - VSCWiki. Website, 2012. Available online at https://wiki.zserv.tuwien.ac.at/doku.php?id=doku:vsc2&rev=1350477581; visited on 15. January 2013.

[34] J. Diamond, M. Burtscher, J.D. McCalpin, B.D. Kim, S.W. Keckler, and J.C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 32–43. IEEE, 2011.

[35] A. Dinklage, T. Klinger, G. Marx, and L. Schweikhard. *Plasma Physics: Confinement, Transport and Collective Effects*, volume 670. Springer, 2005.

[36] C.D. Beidler, K. Allmaier, M.Y. Isaev, S.V. Kasilov, W. Kernbichler, G.O. Leitold, H. Maaßberg, D.R. Mikkelsen, S. Murakami, M. Schmidt, et al. Benchmarking of the mono-energetic transport coefficients—results from the International Collaboration on Neoclassical Transport in Stellarators (ICNTS). *Nuclear Fusion*, 51(7):076001, 2011.

[37] SPI Inc. Debian – Details of package gfortran in squeeze. Website, 2013. Available online at `http://packages.debian.org/en/squeeze/gfortran`; visited on 2. January 2013.