Allgemeine Relativitätstheorie mit dem Computer

ZOOM ONLINE MEETING
JOHANN WOLFGANG GOETHE UNIVERSITÄT
28. MAI, 2021

Aufgrund der Corona Krise findet die Vorlesung und die Übungstermine auch in diesem Semester nur Online statt.

MATTHIAS HANAUSKE

FRANKFURT INSTITUTE FOR ADVANCED STUDIES
JOHANN WOLFGANG GOETHE UNIVERSITÄT
INSTITUT FÜR THEORETISCHE PHYSIK
ARBEITSGRUPPE RELATIVISTISCHE ASTROPHYSIK
D-60438 FRANKFURT AM MAIN
GERMANY

7. Vorlesung

Wiederholung: 6. Vorlesung

Vorlesung 6

Die im rechten Panel dieser Vorlesung dargestellten TOV-Gleichungen beschreiben das Druck- und Energiedichte-Verhalten innerhalb eines sphärisch symmetrischen Objektes und geben außerdem die Innenraum-Metrik des Objektes an. Außerhalb des Objektes ist die Raumzeit durch die Schwarzschildmetrik definiert, welche aufgrund des Birkhoff-Theorems die einzige sphärisch symmetrische Lösung der Einsteingleichung im Vakuum ($T^{\mu\nu} \equiv 0$) ist. Mittels der TOV-Gleichungen können die unterschiedlichsten Objekte in guter Näherung beschrieben werden. So kann man mit ihnen sowohl das Innere der Sonne, als auch die Eigenschaften von Weißen Zwergen und Neutronensternen gut beschreiben. Nach einer kurzen Einführung in die Astrophysik der Weißen Zwerge, Neutronensternen und Quarksterne werden in einem Python Jupyter Notebook, die TOV-Gleichungen analytisch hergeleitet und ihre nummerische Lösung, unter Verwendung einer polytropen Form der Zustandsgleichung der Neutronenstern-Materie behandelt.

Weiße Zwerge, Neutronensterne und Quarksterne

Neutronensterne sind neben weißen Zwergen und stellaren schwarzen Löchern die möglichen Endzustände des Evolutionsprozesses einer Sonne. Neutronensterne werden in gewaltigen Supernova-Explosionen geboren und sie stellen den letzten stabilen Zustand der Materie dar, bevor sie zu einem schwarzen Loch kollabiert. Diese faszinierenden stellaren Objekte besitzen lediglich einen Durchmesser von 20 Kilometern, vereinen dort jedoch auf engstem Raum eine Masse von 500 000 Erdmassen. Von den etwa 100 Millionen Neutronensternen die es in unserer Galaxie, der Milchstraße, vermutlich gibt, sind ca. 3000 als Pulsare bekannt. Pulsare sind schnell rotierende Neutronensterne mit einem starken Magnetfeld (bis zu 10^{11} Tesla), die bevorzugt entlang der Pole elektromagnetische Strahlung aussenden. Einige dieser Neutronensterne, die sogenannten Millisekunden-Pulsare rotieren so schnell, dass sie pro Sekunde mehrere hundert Umdrehungen schaffen. Für diese Millisekunden-Pulsare ist unser sphärisch symmetrische Ansatz der Metrik nicht mehr erfüllt. Unter den bekannten Neutronensternen gibt es auch einige, die sich in binären Systemen befinden, wobei ihr Begleiter entweder ein normaler Stern, ein Planet, ein weißer Zwerg oder auch wieder ein Neutronenstern sein kann. Diejenigen Neutronensterne, die in Zweiersystem umeinander kreisen, verringern ihren Abstand im Laufe der Zeit, da sie Energie durch Aussendung von Gravitationswellen abgeben. Das derzeit beeindruckendste Binärsystem ist der sogenannte Doppelpulsar: PSR J0737-3039A/B, welches im Jahre 2003 entdeckt wurde. Kollidieren zwei Neutronensterne miteinander wird eine enorme Energie in Form von Gravitationswellen frei gesetzt und eine solche Neutronenstern-Kollision konnte im Jahre 2017 mittels der Gravitationswellen-Detektoren beobachtet werden (siehe GW170817). Da die Dichte im inneren Bereich des Neutronensterns den mehrfachen Wert der normalen nuklearen Kerndichte erreichen kann, ist es wahrscheinlich, dass ein Phasenübergang zu Quarkmaterie stattfindet - diese Sterne werden als sogenannte hybride, bzw. Quarksterne bezeichnet

Vorlesung 6

Bis zu dieser Vorlesung hatten wir die raumzeitliche Struktur der Metrik als gegeben vorausgesetzt (Schwarzschild-Metrik bzw. Kerr-Metrik) und die Bewegungen von Probekörpern, im sonst materiefreien Raum, mittels der Geodätengleichung studiert. In dieser Vorlesung betrachten wir den umgekehrten Fall: Wie kann man anhand einer speziellen Materie/Energieverteilung im Raum zu der zugehörigen raumzeitliche Struktur gelangen. Wir betrachten im Folgenden ein statisches, sphärisch symmetrisches Objekt (in Näherung z.B. die Erde, Sonne oder ein Neutronenstern) und setzen die Metrik im Inneren wie folgt an:

$$g_{\mu\nu} = \begin{pmatrix} e^{2\Phi(r)} & 0 & 0 & 0 \\ 0 & -\left(1 - \frac{2m(r)}{r}\right)^{-1} & 0 & 0 \\ 0 & 0 & -r^2 & 0 \\ 0 & 0 & 0 & -r^2 \sin^2(\theta) \end{pmatrix} \quad ,$$

wobei die Funktionen $\phi(r)$ und $m(r)$ an dieser Stelle noch unbekannt sind, später aber eine physikalische Bedeutung erhalten. Die Materie setzen wir als eine ideale Flüssigkeit mit folgendem Energie-Impuls Tensor an:

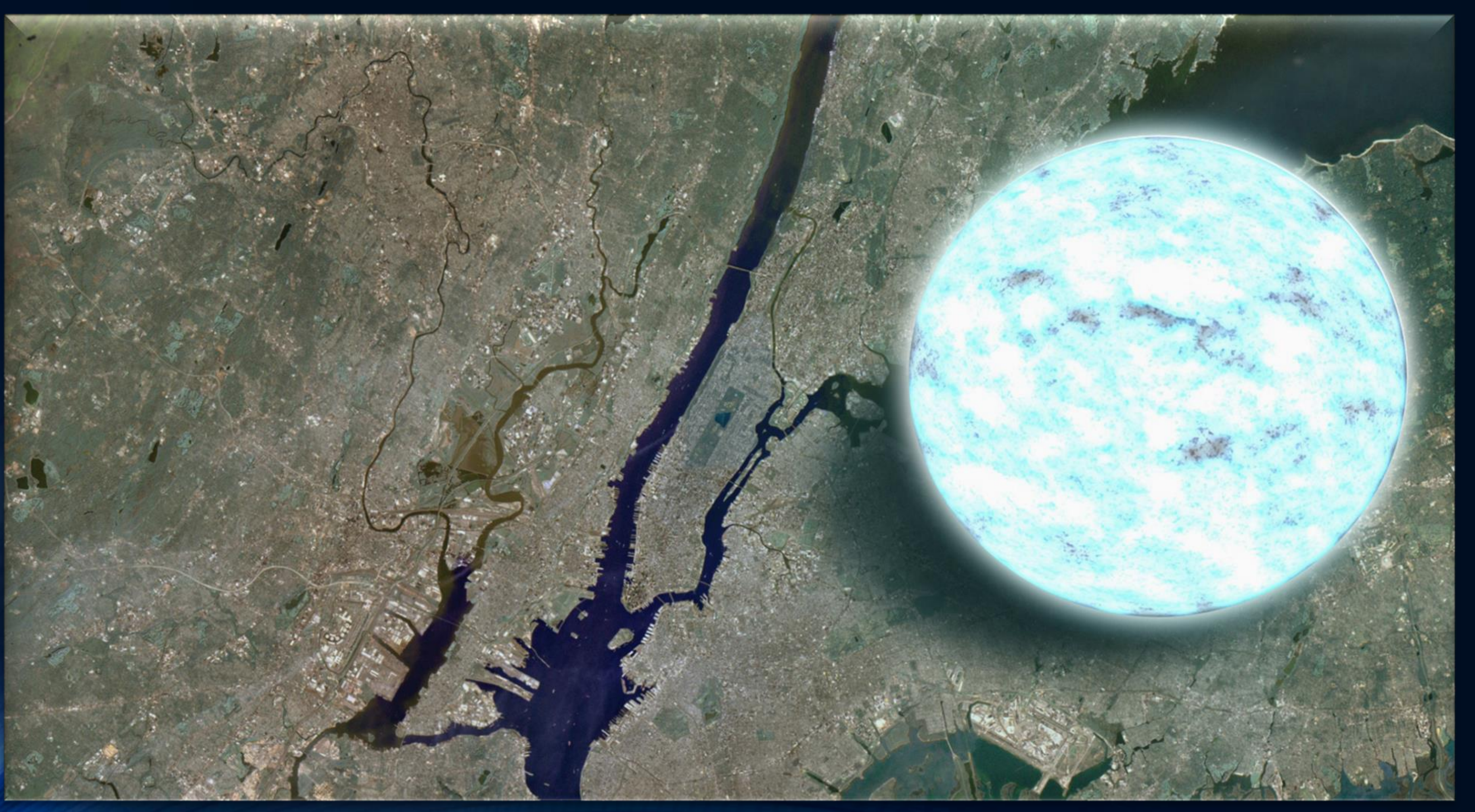
$$T^\mu{}_\nu = \begin{pmatrix} e(r) & 0 & 0 & 0 \\ 0 & -p(r) & 0 & 0 \\ 0 & 0 & -p(r) & 0 \\ 0 & 0 & 0 & -p(r) \end{pmatrix} \quad ,$$

wobei die Funktionen $e(r)$ und $p(r)$ die Energiedichte und den Druck der Materie darstellen. Die raumzeitliche Struktur im Inneren der Materie erhält man mittels der Einstein Gleichung

Neutronensterne: Sehr klein und sehr schwer

Radius ~ 10 km, Masse ~ 1-2 Sonnenmassen

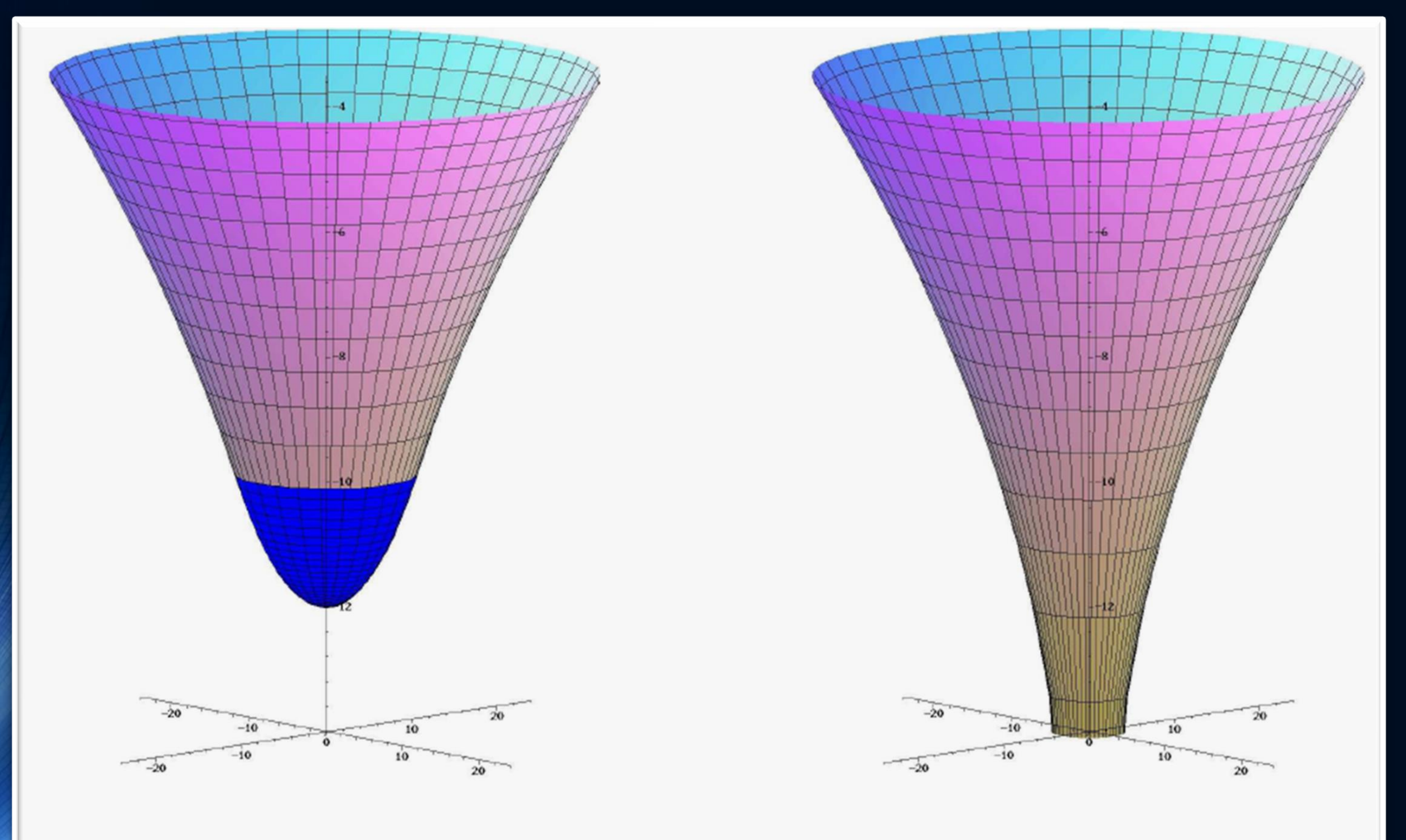
Riesige Magnetfelder ~ 10^{11} Tesla, schnell rotierend (bis zu 716 Hz)



NASA/Goddard Space Flight Center

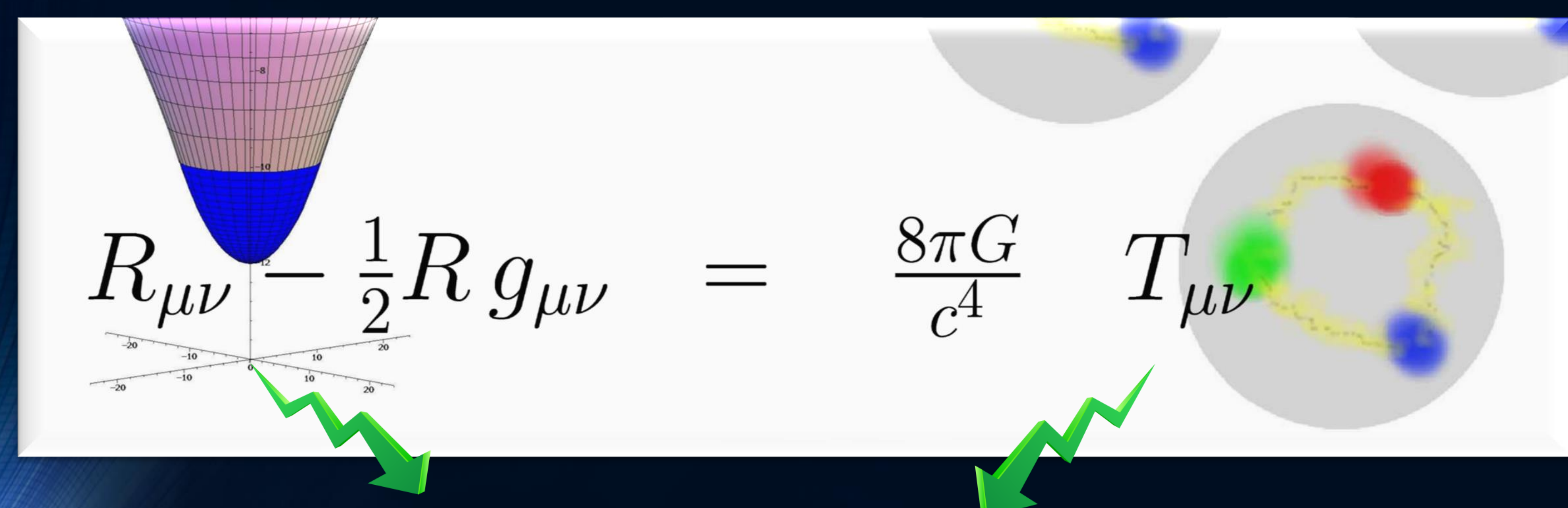
Neutronenstern

Schwarzes Loch



Die Einstein Gleichung

Vor etwa 100 Jahren präsentierte Albert Einstein die Grundgleichung der Allgemeinen Relativitätstheorie – die sogenannte **Einstein-Gleichung**:

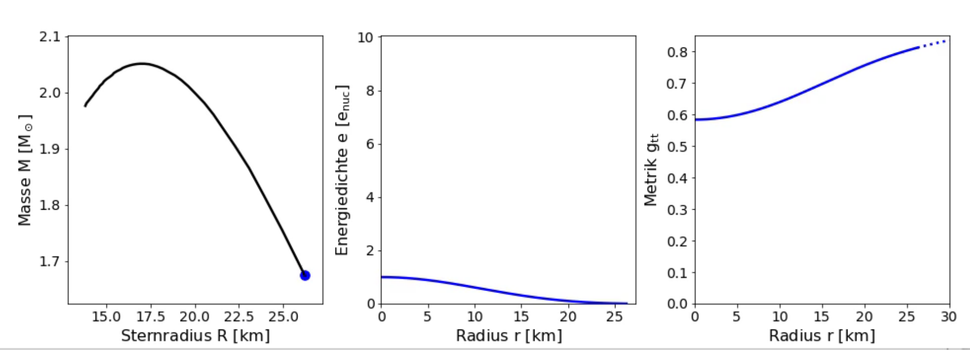


Raumzeitkrümmung
Eigenschaften der Metrik der Raumzeit

Masse, Energie und Impuls des Systems
Zustandsgleichung der Materie
Druck (Dichte , Temperatur)

Jupyter Notebook

Die Tolman-Oppenheimer-Volkoff (TOV) Gleichung



Allgemeine Re

General Theor

Vorlesung gehal
2021)

von Dr.phil.nat. Dr.rer.pol. Matthias Hanauske

Frankfurt am Main 04.04.2021

Erster Vorlesungsteil: Allgemeine Relativitätstheorie mit Python

Die Tolman-Oppenheimer-Volkoff (TOV) Gleichung

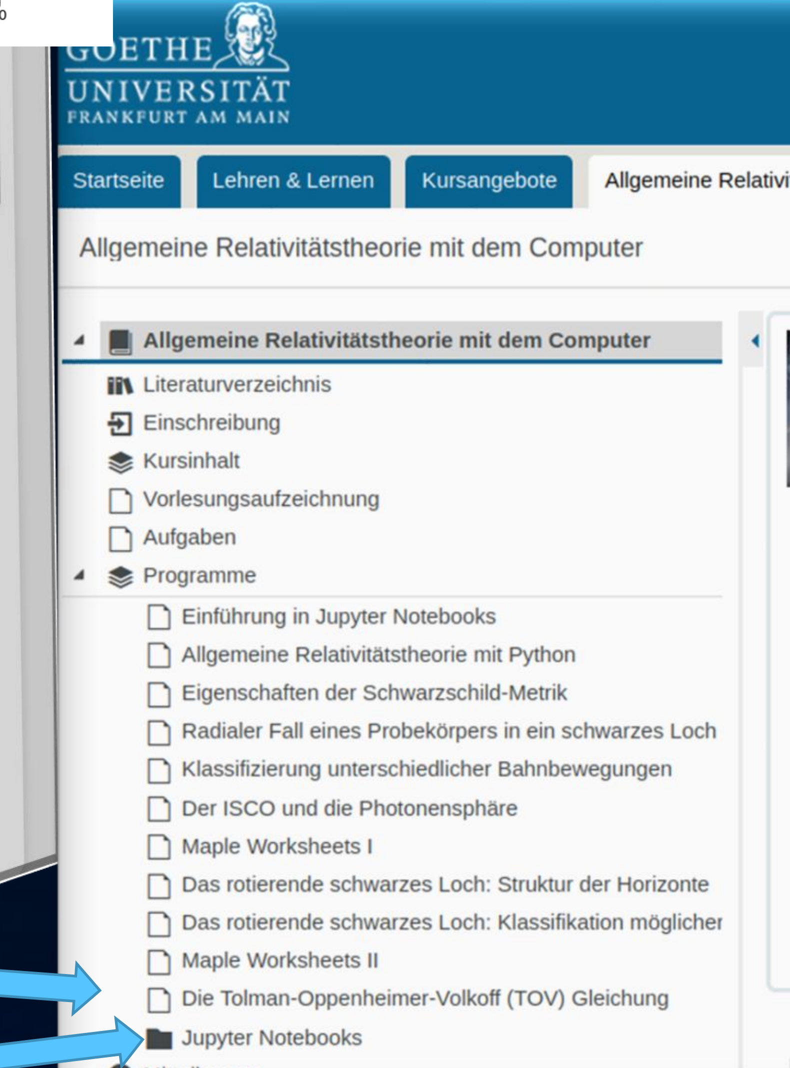
Innenraum-Lösung eines sphärisch symmetrischen, statischen Objektes (z.B. Erde, Neutronenstern)

Von der Einstein Gleichung zur TOV Gleichung

In den vorigen Vorlesungen wurde die Geodätengleichung in vorgegebener Schwarzschild und Kerr Raumzeit für unterschiedliche Anfangsbedingungen numerisch analysiert. Die raumzeitliche Struktur, die Metrik, wurde hierbei als gegeben vorausgesetzt. In der folgenden Vorlesung betrachteten wir nun wie man die Metrik bei vorgegebener Materieverteilung berechnet. Die zugrundeliegende Gleichung, die es hier zu lösen gilt, ist die Einstein Gleichung (in kontra-kovarianter Darstellung)

$$G^\mu{}_\nu = R^\mu{}_\nu - \frac{1}{2} g^\mu{}_\nu R = 8\pi T^\mu{}_\nu \quad .$$

Wir betrachten im Folgenden ein statisches, sph... ...trisches Objekt (in Näherung z.B. die Erde, Sonne oder ein Neutronenstern) und setzen die Metrik im Inneren wie folgt an:



Auf der OLAT Seite des Kurses finden Sie die Jupyter Notebooks zum Ansehen und zum herunterladen

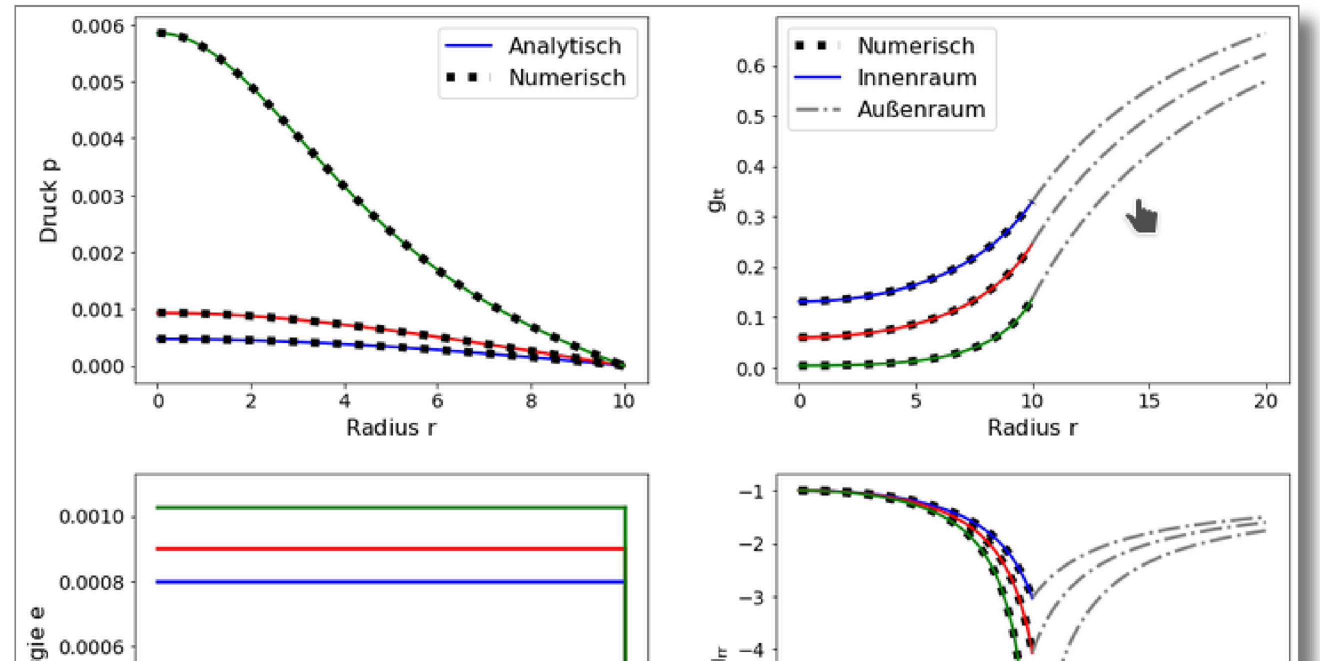
7. Vorlesung

Vorlesung 7

Im ersten Teil dieser Vorlesung werden wir in einem Jupyter Notebook drei zusätzliche Teilaspekte der TOV-Gleichungen behandelt. Zunächst wird der Spezialfall des Gravitationsfeldes einer Kugel konstante Dichte analytisch hergeleitet und die Ergebnisse numerisch überprüft. Dannach visualisieren wir uns die gekrümmte Raumzeit eines Neutronensterns in einem eingebetteten Diagramm. Als Drittes besprechen wir das numerische Euler-Verfahren zum iterativen Lösen der TOV-Differentialgleichungen. Der zweite Teil dieser Vorlesung gibt eine Einführung in die parallele Programmierung mit C++ und OpenMP (Open Multi-Processing) / MPI (Message Passing Interface).

Gravitationsfeld einer Kugel aus inkompressibler Flüssigkeit (konstante Dichte)

Die TOV-Gleichungen besitzen für den Spezialfall einer sphärisch symmetrischen Materieverteilung konstanter Dichte eine analytische Lösung ("Innere Schwarzschild Lösung", siehe rechtes Panel dieser Vorlesung). Die unteren Abbildungen zeigen die Eigenschaften von drei Körpern mit einem Radius R von 10 km und unterschiedlicher konstanter Dichte e_0.



Vorlesung 7

Die in der vorigen Vorlesung besprochenen TOV-Gleichungen wurden im Jahre 1939 in den folgenden zwei Arbeiten publiziert: Tolman, Richard C. "Static solutions of Einstein's field equations for spheres of fluid." Physical Review 55.4 (1939): 364. und Oppenheimer, J. Robert, and George M. Volkoff. "On massive neutron cores." Physical Review 55.4 (1939): 374.. Es ist beachtlich, dass Karl Schwarzschild bereits im Jahre 1916 einen wichtigen Spezialfall der TOV-Gleichungen analytisch berechnete. Herr Schwarzschild betrachtete einen sphärisch symmetrischen Körper mit konstanter Dichte (siehe Karl Schwarzschild, "Gravitationsfeld einer Kugel aus inkompressibler Flüssigkeit", Sitzungsberichte der Königlich-Preussischen Akademie der Wissenschaften. Reimer, Berlin 1916, S:424-434) und berechnete die Eigenschaften dieses Körpers in der Einsteins allgemeiner Relativitätstheorie. In seiner Arbeit zeigte er, dass eine solche Kugel nicht kleiner als $\frac{9}{8}$ seines Schwarzschildradius werden kann ($R > \frac{9}{8} R_S$), da sonst der Druck im Zentrum des Körpers unendlich wird. Hans Adolf Buchdahl gelang es dann im Jahre 1959 zu zeigen, das dieser Wert eine absolute Grenze der Stabilität von Körpern/Sternen darstellt, die sogenannte "Buchdahl Grenze" (siehe Buchdahl, Hans A. "General relativistic fluid spheres." Physical Review 116.4 (1959): 1027.). Die einzige Ausnahme sind die sogenannten "Gravastars", die auch kleiner als diese Grenze werden können. Diese besitzen jedoch einen Bereich im Stern, der eine negative Energiedichte aufweist. Im linken Panel dieser Vorlesung wird die von Schwarzschild gefundene Lösung analytisch hergeleitet und die Ergebnisse numerisch überprüft.

Die numerische Lösung der TOV-Gleichungen hatten wir bisher stets mittels der in Python vordefinierten Funktion "odeint()" berechnet. Man kann die numerische Lösung jedoch auch mittels eines einfachen

Jupyter Notebook

Die TOV-Gleichung: Zusätzliche Betrachtungen

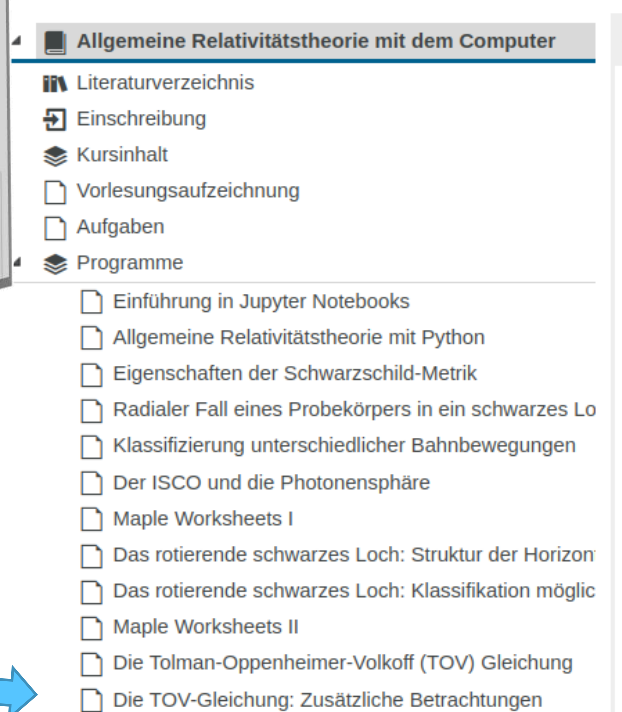
Spezialfall: Gravitationsfeld einer Kugel aus inkompressibler Flüssigkeit (konstante Dichte)

Die TOV-Gleichungen wurden im Jahre 1939 gefunden und gründen auf den folgenden zwei Arbeiten: Tolman, Richard C. "Static solutions of Einstein's field equations for spheres of fluid." Physical Review 55.4 (1939): 364. und Oppenheimer, J. Robert, and George M. Volkoff. "On massive neutron cores." Physical Review 55.4 (1939): 374.. Es ist beachtlich, dass bereits im Jahre 1916, lange bevor die TOV-Gleichungen publiziert wurden, Karl Schwarzschild schon einen wichtigen Spezialfall der TOV-Gleichungen analytisch berechnete. Herr Schwarzschild betrachtete einen sphärisch symmetrischen Körper mit konstanter Dichte (siehe Karl Schwarzschild, "Gravitationsfeld einer Kugel aus inkompressibler Flüssigkeit", Sitzungsberichte der Königlich-Preussischen Akademie der Wissenschaften. Reimer, Berlin 1916, S:424-434) und berechnete die Eigenschaften dieses Körpers in der Einsteins allgemeiner Relativitätstheorie.

Im Folgenden werden wir auch diesen Spezialfall betrachten und die analytische Lösung mit unseren numerischen Berechnungen vergleichen. Wir starten hingegen nicht, wie Schwarzschild es damals, von der Einsteingleichung, sondern nehmen die TOV-Gleichungen als gegeben an.

```
In [1]: import numpy as np
        from sympy import *
        init_printing()
        from einsteinpy.symbolic import *
```

Auf der OLAT Seite des Kurses finden Sie die Jupyter Notebooks zum Ansehen und zum herunterladen



Python-Programm

```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/python/TOVPython$ python TOV-Sequence-plot2021.py
i = 0
Neutronensternradius [km]          = 16.125959999558688
Neutronensternmasse [Sonnenmassen]  = 0.9650804371812499
00-Metrikkomponente im Sternzentrum = 0.6065692130701107
```

TOV-Sequence-plot2021.py | VARTC2021_old8.html

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import math
import matplotlib.gridspec as gridspec

# Definition der Zustandsgleichung
K=10.0
Gamma=5.0/3.0
def eos(p):
    e=math.pow(p/K,1.0/Gamma)
    return e

# plot settings
params = {
    'figure.figsize'    : [10, 7.5],
    'text.usetex'       : True,
}
matplotlib.rcParams.update(params)

# Gitter zum Plotten von sechs unterschiedliche Diagramme definie
plt.figure(0)
gs = gridspec.GridSpec(2, 3, width_ratios=[1,1,1], wspace=0.3, hs
ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])
ax3 = plt.subplot(gs[2])
ax4 = plt.subplot(gs[3])
ax5 = plt.subplot(gs[4])
ax6 = plt.subplot(gs[5])
props = dict(boxstyle='round', facecolor='white', alpha=0.92)
```



```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/python/TOVPython$ ffmpeg -framerate 1 -i './img-%03d.jpg' -s 2000x1500 TOV-Seq.mp4
```

TOVPython : bash

Einführung in die Parallele Programmierung

Einführung in die Parallele Programmierung

Introduction

1. Parallelization on shared memory systems using OpenMP
2. Parallelization on distributed memory systems using MPI
3. Further resources

Paralleles Programmieren mit C++ und OpenMP/MPI

Computersimulationen von realistischen, komplizierten Problemen in der Allgemeinen Relativitätstheorie (z.B. die Simulation einer Neutronenstern Kollision) erfordern, sogar auf "Supercomputern", eine enorme Rechenzeit. Bei der Konzeption der Simulationsprogramme ist es deshalb erforderlich, dass die Rechenleistung des Computers stets voll ausgelastet ist und separate, voneinander unabhängige Teilaufgaben innerhalb der Programme gleichzeitig (parallel) berechnet werden. Dieser Unterpunkt im Teil II der Vorlesung gibt eine Einführung in die parallele Programmierung mit C++ und OpenMP (Open Multi-Processing) / MPI (Message Passing Interface). Die Folien der Präsentation sind unter den folgenden Links einsehbar: Einführung in die parallele Programmierung (LibreOffice Datei , PDF Datei. Am Beispiel eines einfachen numerischen Problems (der Integration einer Funktion), wird das Programmierparadigma der parallelen Programmierung erläutert. Zunächst wird ein einfaches sequentielles C++-Programm erstellt, das die Integration der Funktion $f(x) = \frac{1}{1+a\,x^2}$ in den Grenzen [0,1] und den Werten $a \in [0, 10]$ mit dem Gauß'schen Integrationsverfahren numerisch berechnet (siehe sequentielle Version 1 und sequentielle Version 2). In dem Programm wird der Wert $W(a)$ des Integrals $\int_0^1 \frac{1}{1+a\,x^2}\,dx$ für die Parameterwerte $a \in [0, 10]$ ausgegeben und mit dem analytischen Ergebnis $W(a) = \frac{\arctan(\sqrt{a})}{\sqrt{a}}$ verglichen. Die Parallelisierung dieses sequentiellen Programms wird zunächst mit OpenMP (siehe OpenMP Version 1 , OpenMP Version 2 , OpenMP Version 3 , OpenMP Version 4 und OpenMP Version 5) und danach mit MPI (siehe MPI Version 1 , MPI Version 2) durchgeführt.

Contents

1. Introduction
2. Parallelization on shared memory systems using OpenMP
3. Parallelization on distributed memory systems using MPI
4. Further resources

Introduction

1. Introduction
   a) What is parallelization?
   b) When and where can it be used?
   c) Parallel architectures of computer clusters.
   d) Different parallelization languages.
2. Parallelization on shared memory systems using OpenMP
3. Parallelization on distributed memory systems using MPI
4. Further resources

Introduction

Parallel Programming is a programming paradigm (a fundamental style of computer programming).

Within a parallel computer code a single computation problem is separated in different portions that may be executed concurrently by different processors.

Parallel Programming is a construction of a computer code that allows its execution on a parallel computer (multi-processor computer) in order to reduce the time needed for a single computation problem.

Depending on the architecture of the parallel computer (or computer cluster) the Parallel Programming Framework (OpenMP, MPI, Cuda, OpenCL, …) has to be chosen .

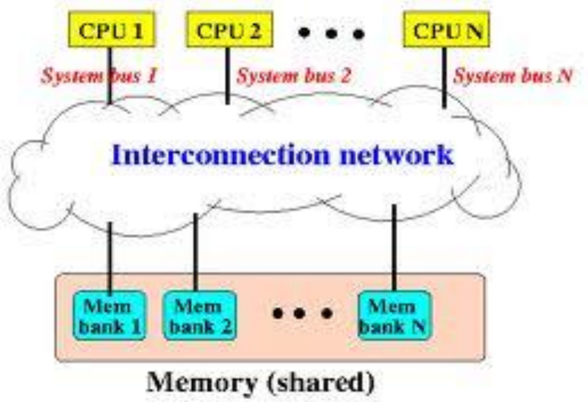
Parallel computers

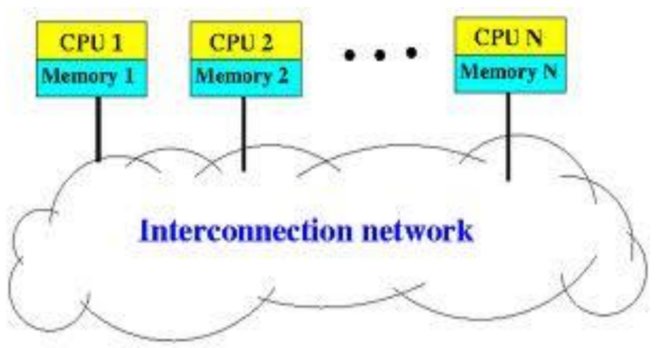
Parallel computer architectures:

SIMD (Single Instruction, Multiple Data)

Example: Parallel computers with graphics processing units (GPU's)
using the Cuda or OpenCL language.

MIMD (Multiple Instruction, Multiple Data)





Shared Memory
(OpenMP, OpenCL, MPI)

Distributed Memory
(MPI, (Shell programming))

Performance

The **performance of a parallel computer code** can be measured using the following characteristic values:

T(n): Time needed to run the program on n processing elements (e.g. CPU's, computer nodes).

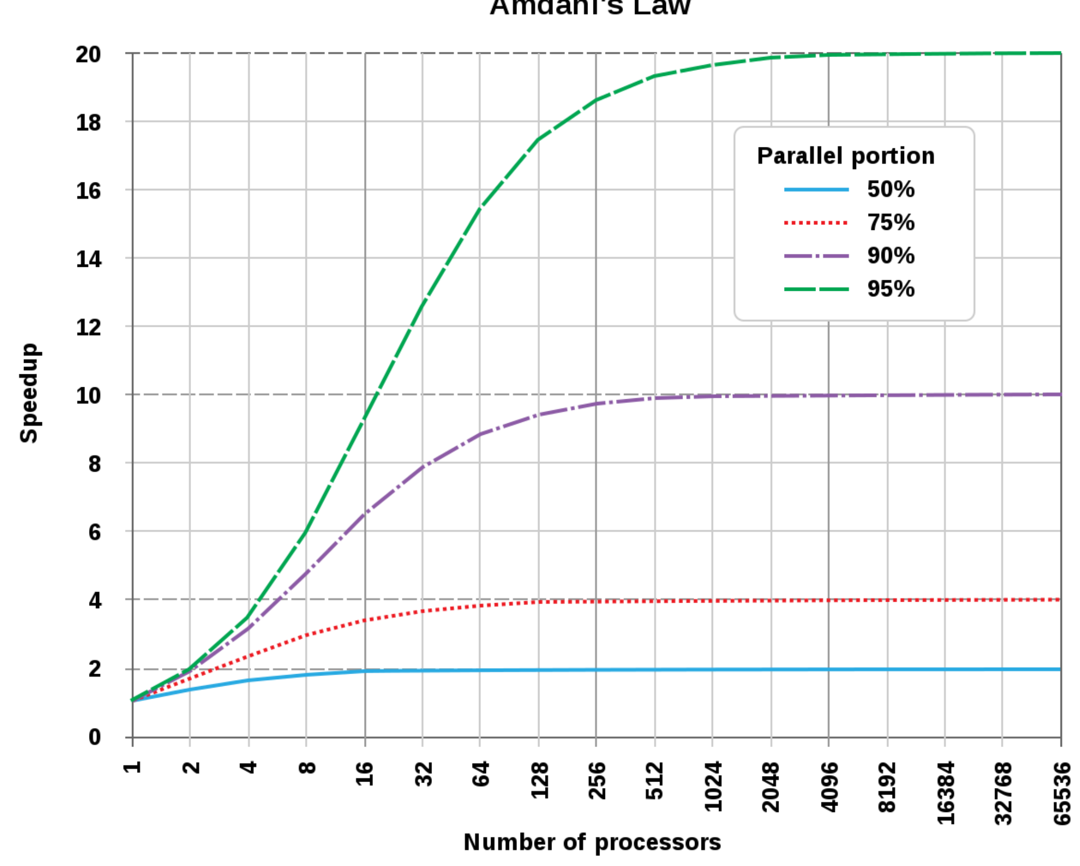
Speedup: S(n):=T(1)/T(n) , **Efficiency**: E(n):=S(n)/n

Amdahl's law:

The "Amdahl's law" describes the speedup of an optimal parallel computer code. T(n) is divided in two parts (T(n)=Ts+Tp(n)), where Ts is the time needed for the non-parallelized part of the program and Tp(n) is the parallelized part, which can be executed concurrently by different processors.

A(n):=Max(S(n))

A(n)=1/(a+(1-a)/n), with 'Parallel portion' a:=Ts/T(1)



Shared Memory and OpenMP

1. Introduction
2. Parallelization on shared memory systems using OpenMP

a) Introduction to OpenMP

b) Example

c) Further OpenMP directives

d) Additional material

3. Parallelization on distributed memory systems using MPI
4. Further resources

OpenMP

The **parallel computer language "OpenMP** (Open Multi-Processing)" supports multi-platform shared-memory parallel programming in C/C++ and Fortran. It is a collaborative developed parallel language which has its origin in 1997.

OpenMP separates the parallelized part of the program into several **'Threads',** where each thread can be executed on a different processing element (CPU) using shared memory.

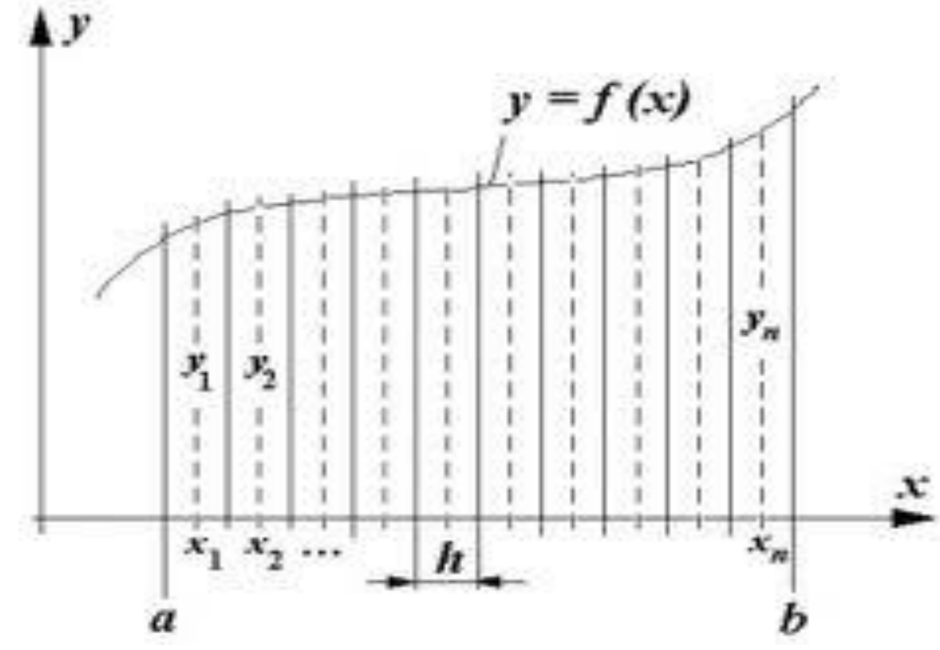
OpenMP has the advantage that common **sequential codes can easily be changed** by simply adding some OpenMP directives.
Another feature of OpenMP is that the program runs also properly
(but then sequentially, using only one thread)
even if the compiler does not know OpenMP.

Example

The simple computation problem used in the following is the numerical integration of an integral using the Gauß integration method. The following integral should be calculated for 10 different values (a=1,2,..,10).

$$\int_0^1 \frac{1}{1 + a\,x^2}\,dx = \frac{\arctan\left(\sqrt{a}\right)}{\sqrt{a}}$$

The integration interval [0,1] is divided into N pieces. The value of the integration function is taken at the middle of each integration segment (Gauß method).



Gauß'sche integration method

C++ Einführung

Grundgerüst und Variablen

Einlesen von Header-Files (Definition nötiger C++ Funktionen)

Beginn des Hauptprogramms

Ausgabe eines Strings

```
#include <iostream>          //Ein-/Ausgabe (Include-Dateien)
using namespace std;         //Fuer cout

main(void)   //Hauptprogramm
  {

     cout<<"Hello \n";

  }
```

Prog1.cpp

Deklaration einer Integer (natürliche Zahl) und einer double (reelle Zahl) Variable

Variablen bekommen einen festen Zahlenwert (Initialisierung)

Ausgabe des Wertes der Variablen

```
#include <iostream>          //Ein-/Ausgabe (Include-Dateien)
using namespace std;         //Fuer cout

main(void)   //Hauptprogramm
  {
    //Variablendeklarationen
    int i;
    double a;

    //Variableninitialisierung
    i=3;
    a=1.435553;

    cout<<"i="<<i<<"\n";
    cout<<"a="<<a<<"\n";
  }
```

Vom Quellcode zum ausführbaren Programm

Der Quellcode (z.B. Prog1.cpp) muss kompiliert werden um ein ausführbares Programm (a.out) zu erzeugen. Man öffnet hierzu in dem Verzeichnis, in dem sich der Quellcode befindet, ein Terminal und führt das folgende Kommando aus:

c++ Prog1.cpp

```
hanauske@ITPRelAstro-Aspire-VN7-591G:~$ c++ Prog1.cpp
hanauske@ITPRelAstro-Aspire-VN7-591G:~$ ./a.out
Hello
hanauske@ITPRelAstro-Aspire-VN7-591G:~$
```

Das Programm wird gestartet und erzeugt im Terminal die Ausgabe “Hello”

Beim Kompilierung-Prozess wird eine Datei (a.out) erzeugt, die man dann mittels des folgenden Kommandos ausführen kann:

./a.out

C++ Die for-Schleife

```
#include <iostream>                    //Ein-/Ausgabe (Include-Dateien)
using namespace std;                   //Fuer cout

main(void)  //Hauptprogramm
  {
    //Variablendeklarationen
    int i;
    double a;

    //Variableninitialisierung
    a=1.435553;

    //for Schleife
    for (i = 1; i <= 10; ++i)
      {
        cout<<"i="<<i<<"\n";
        cout<<"i mal a ="<<i*a<<"\n";
      }
  }
```

Mittels einer for-Schleife können iterative Aufgaben im Programm implementiert werden. Die for-Schleife benötigt einen Anfangswert (i=0), die Angabe wie lange sie die Integration durchführen soll (i<=10) und die Angabe um wie viel sie die Variable in jedem Schritt verändern soll (++i). “++i” bzw. “i++” ist nur eine Kurzschreibweise von i=i+1.

C++ Die do-Schleife

```
#include <iostream>                   //Ein-/Ausgabe (Include-Dateien)
using namespace std;                  //Fuer cout

main(void)  //Hauptprogramm
  {
    //Variablendeklarationen
    int i;
    double a;

    //Variableninitialisierung
    i=1;
    a=1.435553;

    //do Schleife
    do
      {
        cout<<"i="<<i<<"\n";
        cout<<"i mal a ="<<i*a<<"\n";
        i++;
      }
     while(i <= 10);
  }
```

Mittels einer do-Schleife können iterative Aufgaben im Programm implementiert werden. Die do-Schleife benötigt lediglich eine Abbruchbedingung (while(i<=10);) wobei im Inneren der Schleife die Variable i in jedem Schritt verändert werden muss (i++;). Die Variable I muss jedoch zunächst außerhalb der Schleife initialisiert werden (i=1;).

Sequential Code

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

double integral(int N, double a)
{
    double sum=0;
    double dx = 1/(double)(N);

    for(int j=0; j < N; ++j)
    {
        double x = dx*( j + 0.5 );
        sum += 1/(1 + a*x*x);
    }
    return dx*sum;
}

int main(void)
{
    double integral(int,double);
    double ergebnis;

    for (int i = 1; i <= 10; ++i)
    {
        ergebnis=integral(100000000,i);
        printf("a=%i: Integral=%e, Difference=%e \n",
                  i,ergebnis,ergebnis-atan(sqrt(i))/sqrt(i));
    }

    return 0;
}
```

The integral is defined as a function which depends on two variables (N and a). 'N' is the number of integration points (integration segments) and 'a' is the parameter defined within the example. With the use of a 'for-loop', the total area of the N-rectangles are summed up. The value of the integral is then returned (dx*sum).

To calculate and output the value of the integral for different values of 'a' (a=1,..,10), the main function of the program contains also a 'for-loop'. The output contains the value of 'a', the value of the calculated integral (N=10 million) and the difference of the calculation with the 'analytic' result.

Parallel Code No.1

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>

double integral(int N, double a)
{
    double sum=0;
    double dx = 1/(double)(N);

    for(int j=0; j < N; ++j)
    {
        double x = dx*( j + 0.5 );
        sum += 1/(1 + a*x*x);
    }
    return dx*sum;
}

int main(void)
{
    int startTime = time(NULL);
    double integral(int,double);
    double ergebnis;

#pragma omp parallel for
    for (int i = 1; i <= 10; ++i)
    {
    ergebnis=integral(100000000,i);
    int id = omp_get_thread_num();
    printf("a=%i: Integral=%e, Difference=%e, Thread No:%i \n"
        ,i,ergebnis,ergebnis-atan(sqrt(i))/sqrt(i),id);
    }

    printf("Time needed: %i seconds\n"
        ,(int)(time(NULL)-startTime));
    return 0;
}
```

To parallelize the code with OpenMP, only two minor changes are necessary:

1) The OpenMP-Header file (omp.h) need to be included

2) The OpenMP-Pragma (#pragma omp parallel for) should be inserted just right before the loop that we want to be calculated concurrently.

During the execution of the program (when entering the parallelized loop), several threads are created. The number of threads is not specified; it depends on the number of available processors and the size of the loop.

Parallel Code No.1a

parallel_omp_1_time_id.cpp

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>

double integral(int N, double a)
{
    double sum=0;
    double dx = 1/(double)(N);

    for(int j=0; j < N; ++j)
    {
        double x = dx*( j + 0.5 );
        sum += 1/(1 + a*x*x);
    }
    return dx*sum;
}

int main(void)
{
    int startTime = time(NULL);
    double integral(int,double);
    double ergebnis;

#pragma omp parallel for
    for (int i = 1; i <= 10; ++i)
    {
    ergebnis=integral(100000000,i);
    int id = omp_get_thread_num();
    printf("a=%i: Integral=%e, Difference=%e, Thread No:%i \n"
        ,i,ergebnis,ergebnis-atan(sqrt(i))/sqrt(i),id);
    }

    printf("Time needed: %i seconds\n"
        ,(int)(time(NULL)-startTime));
    return 0;
}
```

In respect to the ongoing calculation, this version of the parallelized code does not differ at all from the previous one. Nevertheless, two changes have been made:

To compare the performance of the parallel version of the code with its sequential counterpart, the time needed for the calculation is also printed out.

To understand the ‘Thread-based’ calculation, the id-number of each Thread is additionally printed out.

Running the code

To run the sequential version of the code under Linux, the executable file (a.out) has been created using the c++ compiler.

```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$ c++ sequential_time.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$ ./a.out
a=1: Integral=7.853982e-01, Difference=1.583178e-13
a=2: Integral=6.755109e-01, Difference=-9.625634e-14
a=3: Integral=6.045998e-01, Difference=-1.046940e-13
a=4: Integral=5.535744e-01, Difference=-2.531308e-13
a=5: Integral=5.144128e-01, Difference=7.993606e-15
a=6: Integral=4.830392e-01, Difference=5.168088e-14
a=7: Integral=4.571213e-01, Difference=-1.637024e-13
a=8: Integral=4.352099e-01, Difference=-1.355027e-13
a=9: Integral=4.163486e-01, Difference=4.873879e-14
a=10: Integral=3.998760e-01, Difference=1.537104e-13
Time needed: 11 seconds
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$
```

The parallel version (No.1a) has been created using c++ with the option '-fopenmp'.
The program was executed on a system with eight CPU's.

```
Time needed: 11 seconds
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$ c++ -fopenmp parallel_omp_1_time_id.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$ ./a.out
a=3: Integral=6.045998e-01, Difference=-1.046940e-13, Thread No:2
a=1: Integral=7.853982e-01, Difference=1.583178e-13, Thread No:0
a=8: Integral=4.352099e-01, Difference=-1.355027e-13, Thread No:7
a=5: Integral=5.144128e-01, Difference=7.993606e-15, Thread No:4
a=9: Integral=4.163486e-01, Difference=4.873879e-14, Thread No:8
a=4: Integral=5.535744e-01, Difference=-2.531308e-13, Thread No:3
a=2: Integral=6.755109e-01, Difference=-9.625634e-14, Thread No:1
a=10: Integral=3.998760e-01, Difference=1.537104e-13, Thread No:9
a=6: Integral=4.830392e-01, Difference=5.168088e-14, Thread No:5
a=7: Integral=4.571213e-01, Difference=-1.637024e-13, Thread No:6
Time needed: 1 seconds
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$
```

Parallel Code No.2

(wrong!)

parallel_omp_2_wrong.cpp

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

double integral(int N, double a)
{
    double sum=0;
    double dx = 1/(double)(N);

#pragma omp parallel for
    for(int j=0; j < N; ++j)
    {
        double x = dx*( j + 0.5 );
        sum += 1/(1 + a*x*x);
    }
    return dx*sum;
}

int main(void)
{
    double integral(int,double);
    double ergebnis;

    for (int i = 1; i <= 10; ++i)
    {
    ergebnis=integral(100000000,i);
    printf("a=%i: Integral=%e, Difference=%e \n"
        ,i,ergebnis,ergebnis-atan(sqrt(i))/sqrt(i));
    }

    return 0;
}
```

The OpenMP-Pragma (#pragma omp parallel for) has been inserted just right before the loop that is inside the function which calculates the integral.

```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Int
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Int
a=1: Integral=7.835347e-02, Difference=-7.070447e-01
a=2: Integral=7.535383e-02, Difference=-6.001570e-01
a=3: Integral=6.476837e-02, Difference=-5.398314e-01
a=4: Integral=6.365077e-02, Difference=-4.899236e-01
a=5: Integral=4.804519e-02, Difference=-4.663676e-01
a=6: Integral=4.704121e-02, Difference=-4.359980e-01
a=7: Integral=5.212998e-02, Difference=-4.049913e-01
a=8: Integral=4.157262e-02, Difference=-3.936373e-01
a=9: Integral=5.484395e-02, Difference=-3.615046e-01
a=10: Integral=4.570275e-02, Difference=-3.541733e-01
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Int
```

Two problems arise when executing the program:

1) The parallel program needs even more time than the sequential version.

2) The integrals are calculated wrong (see huge difference to the analytic result).

Parallel Code No.2

parallel_omp_2_time.cpp

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <omp.h>

double integral(int N, double a)
{
    double sum=0;
    double dx = 1/(double)(N);

#pragma omp parallel for reduction(+:sum)
    for(int j=0; j < N; ++j)
    {
        double x = dx*( j + 0.5 );
        sum += 1/(1 + a*x*x);
    }
    return dx*sum;
}

int main(void)
{
    int startTime = time(NULL);
    double integral(int,double);
    double ergebnis;

    for (int i = 1; i <= 10; ++i)
    {
    ergebnis=integral(100000000,i);
    printf("a=%i: Integral=%e, Difference=%e \n"
        ,i,ergebnis,ergebnis-atan(sqrt(i))/sqrt(i));
    }

    printf("Time needed: %i seconds\n"
        ,(int)(time(NULL)-startTime));
    return 0;
}
```

The problem of the previous code is due to a wrong communication and interference between the threads at the code line

sum += 1/(1+a*x*x);

During the execution, this line is actually separated in several steps:

1) The values of sum, a and x are read.

2) The value of '1/(1+a*x*x)' is calculated and added up with the value of 'sum'.

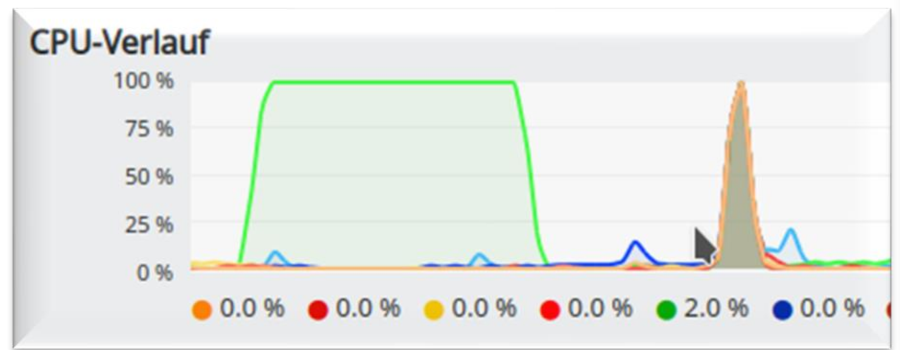
3) The result of 2) is written as the new value of 'sum' to the address of the variable 'sum'.

When several threads are created inside the loop, it is possible that while one thread (A) is at stage 2), another thread (B) begins at stage 1). If A writes its new value at stage 3), B is at stage 2). When B finally writes its new value at stage 3), the integration increment of A is lost. This leads to wrong results and a slowdown of the code.

This 'race condition' can be solved using the synchronization directive reduction(+:sum)

Running the code

Sequential version (green):




```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$ c++ sequential_time.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$ ./a.out
a=1: Integral=7.853982e-01, Difference=1.583178e-13
a=2: Integral=6.755109e-01, Difference=-9.625634e-14
a=3: Integral=6.045998e-01, Difference=-1.046940e-13
a=4: Integral=5.535744e-01, Difference=-2.531308e-13
a=5: Integral=5.144128e-01, Difference=7.993606e-15
a=6: Integral=4.830392e-01, Difference=5.168088e-14
a=7: Integral=4.571213e-01, Difference=-1.637024e-13
a=8: Integral=4.352099e-01, Difference=-1.355027e-13
a=9: Integral=4.163486e-01, Difference=4.873879e-14
a=10: Integral=3.998760e-01, Difference=1.537104e-13
Time needed: 11 seconds
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$
```

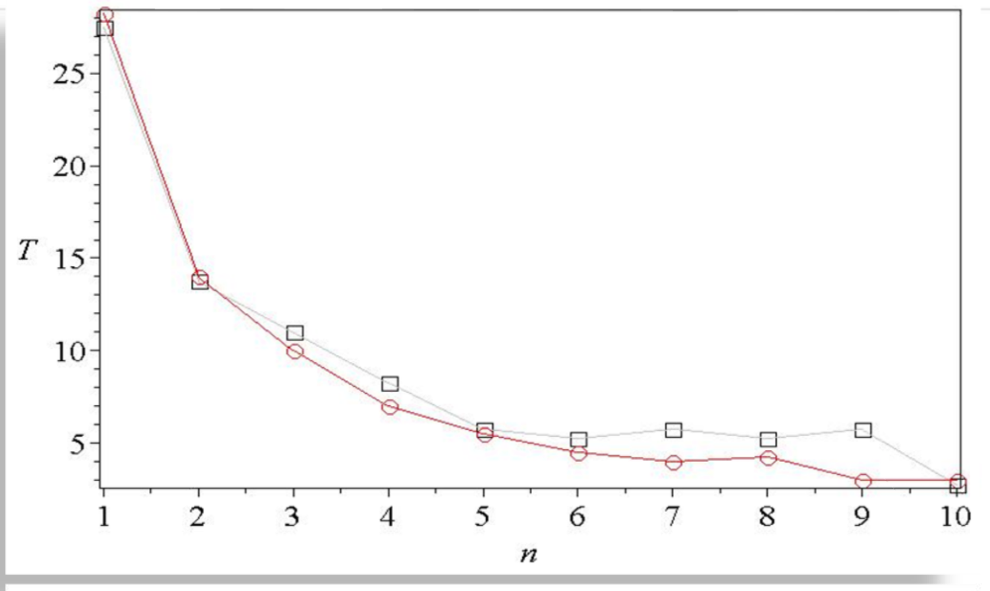
Parallel version (No.2): Due to the non-sequential summation of the different parts of the integral, a different rounding error occurs within the parallel version. This is the reason that the calculated integrals are not exactly the same as in the sequential version.

```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$ c++ -fopenmp parallel_omp_2_time.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$ ./a.out
a=1: Integral=7.853982e-01, Difference=8.770762e-15
a=2: Integral=6.755109e-01, Difference=-5.107026e-15
a=3: Integral=6.045998e-01, Difference=2.708944e-14
a=4: Integral=5.535744e-01, Difference=-2.464695e-14
a=5: Integral=5.144128e-01, Difference=-7.549517e-15
a=6: Integral=4.830392e-01, Difference=-2.237099e-14
a=7: Integral=4.571213e-01, Difference=6.605827e-15
a=8: Integral=4.352099e-01, Difference=-7.882583e-15
a=9: Integral=4.163486e-01, Difference=1.404432e-14
a=10: Integral=3.998760e-01, Difference=1.276756e-15
Time needed: 1 seconds
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/openmp$
```

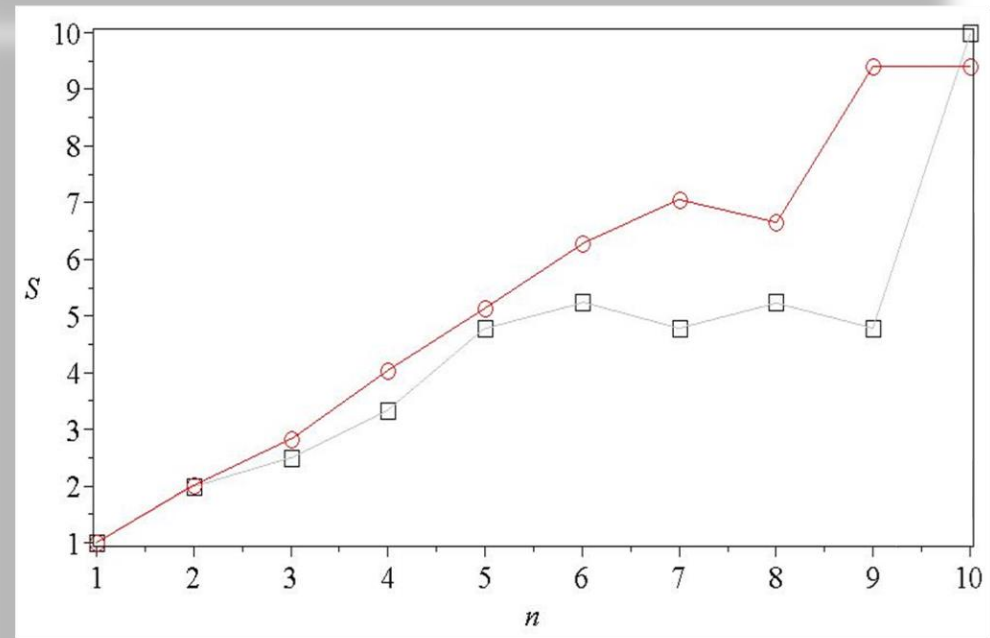
Performance

The following calculations were performed in the Center for Scientific Computing (CSC) of the Goethe University Frankfurt using the FUCHS-CPU-Cluster.

The upper picture shows the time needed (T(n)) to run the program using n processing elements (respectively threads)



and the lower picture shows the speedup S(n). The black curve indicates the performance of the parallel code No.1 and the red curve shows the results of code No.2.



Further OpenMP directives

Loop parallelization(#pragma omp parallel for …):

- Access to variables (shared(), private(), firstprivate(), reduction())
- Syncronisation (atomic, critical)
- Locking (omp_init_lock(),omp_destroy_lock(),omp_set_lock(),_unset_lock())
- Barriers (#pragma omp barrier)
- Conditional parallelization (if(…))
- Number of threads (omp_set_num_threads())
- Loop workflow (shedule())

Additional material:

The OpenMP® API specification for parallel programming: http://openmp.org/

The Community of OpenMP: http://www.compunity.org/

OpenMP-Tutorial: https://computing.llnl.gov/tutorials/openMP/

Book: Parallel Programming in C with MPI and OpenMP, by Michael J. Quinn.

Book: Patterns for Parallel Programming, by Timothy G. Mattson, et.al.

Book: Using OpenMP, by Chapman,et.al.

Book: OpenMP by Hoffmann, Lienhart

Tutorium Examples: http://fias.uni-frankfurt.de/~hanauske/new/parallel/openmp/

Distributed Memory and MPI

1. Introduction
2. Parallelization on shared memory systems using OpenMP
3. Parallelization on distributed memory systems using MPI

a) Introduction to MPI

b) Example

c) Additional material

4. Further resources

MPI

The **parallel computer language "MPI** (Message Passing Interface)" supports multi-platform shared- amd distributed-memory parallel programming in C/C++ and Fortran. The MPI standard was firstly presented at the "Supercomputing '93"-conference in 1993.

With MPI, the whole computation problem is separated in different Tasks (processes). Each process can run on a different computer nodes within a computer cluster. In contrast to OpenMP, MPI is designed to run on distributed-memory parallel computers.

As each process has its own memory, the result of the whole computation problem has to be combined by using both point-to-point and collective communication between the processes.

Parallel Code No.1

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <mpi.h>

double integral(int N, double a)
{
    double sum=0;
    double dx = 1/(double)(N);

    for(int j=0; j < N; ++j)
    {
        double x = dx*( j + 0.5 );
        sum += 1/(1 + a*x*x);
    }
    return dx*sum;
}

int main( int nArguments, char **arguments )
{
    MPI::Status status;
    MPI::Init ( nArguments, arguments );
    int p = MPI::COMM_WORLD.Get_size ( );
    int id = MPI::COMM_WORLD.Get_rank ( );

    double integral(int,double);
    double ergebnis;
    int startTime = time(NULL);

    for (int i = id+1; i <= 10; i = i + p)
    {
        ergebnis=integral(100000000,i);
        printf("a=%i: Integral=%e, Difference=%e \n"
        ,i,ergebnis,ergebnis-atan(sqrt(i))/sqrt(i));
    }

    if (id == 0)
    {
        printf("Time needed: %i seconds\n"
        ,(int)(time(NULL)-startTime));
    }

    MPI::Finalize ( );
    return 0;
}
```

To parallize the code with MPI, some changes are necessary:

1) The MPI-Header file (mpi.h) need to be included.

2) Arguments has to be included within the main function.

3) Several other things need to be specified

At the beginning of the execution of the program, a specified number of processes (p) are created. Each process has its own id-number, and it can be executed on different nodes within a computer cluster or on different processors of one node. Within this version of the parallel program the loop which goes over different values of 'a' (a=1,..,10) is divided among different processes.

Running the code No.1

To build the executable file (a.out) of the parallel version of the MPI-program under Linux, a mpi-compiler (mpic++, mpicxx, mpicc,…) has to be used. To run the program, on needs to use the command "mpirun" and specify the number of processes (e.g. -np 2).

```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$ mpic++ parallel_mpi_time.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$ mpirun -np 1 ./a.out
a=1: Integral=7.853982e-01, Difference=1.583178e-13
a=2: Integral=6.755109e-01, Difference=-9.625634e-14
a=3: Integral=6.045998e-01, Difference=-1.046940e-13
a=4: Integral=5.535744e-01, Difference=-2.531308e-13
a=5: Integral=5.144128e-01, Difference=7.993606e-15
a=6: Integral=4.830392e-01, Difference=5.168088e-14
a=7: Integral=4.571213e-01, Difference=-1.637024e-13
a=8: Integral=4.352099e-01, Difference=-1.355027e-13
a=9: Integral=4.163486e-01, Difference=4.873879e-14
a=10: Integral=3.998760e-01, Difference=1.537104e-13
Time needed: 11 seconds
```

The first run was performed by only using one process (sequential version).

```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$ mpirun -np 2 ./a.out
a=1: Integral=7.853982e-01, Difference=1.583178e-13
a=2: Integral=6.755109e-01, Difference=-9.625634e-14
a=3: Integral=6.045998e-01, Difference=-1.046940e-13
a=4: Integral=5.535744e-01, Difference=-2.531308e-13
a=5: Integral=5.144128e-01, Difference=7.993606e-15
a=6: Integral=4.830392e-01, Difference=5.168088e-14
a=7: Integral=4.571213e-01, Difference=-1.637024e-13
a=8: Integral=4.352099e-01, Difference=-1.355027e-13
a=9: Integral=4.163486e-01, Difference=4.873879e-14
Time needed: 6 seconds
```

This run has used two processes.

```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$ mpirun -np 9 ./a.out
a=4: Integral=5.535744e-01, Difference=-2.531308e-13
a=6: Integral=4.830392e-01, Difference=5.168088e-14
a=1: Integral=7.853982e-01, Difference=1.583178e-13
a=7: Integral=4.571213e-01, Difference=-1.637024e-13
a=2: Integral=6.755109e-01, Difference=-9.625634e-14
a=5: Integral=5.144128e-01, Difference=7.993606e-15
a=8: Integral=4.352099e-01, Difference=-1.355027e-13
a=9: Integral=4.163486e-01, Difference=4.873879e-14
a=3: Integral=6.045998e-01, Difference=-1.046940e-13
a=10: Integral=3.998760e-01, Difference=1.537104e-13
Time needed: 2 seconds
```

This run has used nine processes.

Parallel Code No.2

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <mpi.h>

double integral(int N, double a, int id, int p)
{
    double sum=0;
    double dx = 1/(double)(N);

    for(int j=id; j < N; j = j + p)
    {
        double x = dx*( j + 0.5 );
        sum += 1/(1 + a*x*x);
    }
    return dx*sum;
}

int main( int nArguments, char **arguments )
{
    MPI::Status status;
    MPI::Init ( nArguments, arguments );
    int p = MPI::COMM_WORLD.Get_size ( );
    int id = MPI::COMM_WORLD.Get_rank ( );

    double integral(int,double,int,int);
    double ergebnis[24];
    int startTime = time(NULL);

    ergebnis[id]=integral(1000000000,1,id,p);
    if(id != 0){MPI::COMM_WORLD.Send( &ergebnis[id], 1, MPI::DOUBLE, 0, 1);}

    if (id == 0)
    {
        for (int q = 1; q <= p - 1; q++ )
        {
            MPI::COMM_WORLD.Recv( &ergebnis[q], 1, MPI::DOUBLE, q, 1, status );
            ergebnis[0]=ergebnis[0]+ergebnis[q];
        }
        printf("a=1: Integral=%e, Difference=%e \n"
        ,ergebnis[0],ergebnis[0]-atan(sqrt(1))/sqrt(1));
        printf("Time needed: %i seconds\n",(int)(time(NULL)-startTime));
    }

    MPI::Finalize ( );
    return 0;
}
```

Within this parallel version only one integral was calculated (a=1), but the accuracy of the performed numerical calculation has been increased (N=1000 million). The loop that performs these 1000 million iterations is divided among different processes.

As every process knows only the part that it has calculated, the processes need to communicate in order to calculate the value of the whole integral. Within MPI, several ways of communications are possible. Within this version a point-to-point communicationhas been used.

The MPI function “Send” was used by every process (except process 0) to send its value to process 0.

Process 0 then receives all the different values, makes a sum and prints the final result out.

One can use collective operation “MPI_Reduce” for this purpose.

Running the code No.2

To build the executable file (a.out) of the parallel version of the MPI-program under Linux, a mpi-compiler (mpic++, mpicxx, mpicc,…) has to be used.

To run the program, on needs to use the command "mpirun" and specify the number of processes (e.g. -np 2). The first run was performed by only using one process (-np 1 , sequential version). The second run was much faster and has used ten processes.

```
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$ mpic++ parallel_mpi_2_time.cpp
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$ mpirun -np 1 ./a.out
a=1: Integral=7.853982e-01, Difference=4.440892e-14
Time needed: 11 seconds
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$ mpirun -np 10 ./a.out
a=1: Integral=7.853982e-01, Difference=1.465494e-14
Time needed: 1 seconds
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$ mpirun -np 2 ./a.out
a=1: Integral=7.853982e-01, Difference=5.351275e-14
Time needed: 6 seconds
(base) hanauske@hanauske-Aspire-A717-72G:~/VARTC/CoronaCampus2021/c++/Intro/mpi$
```

Further MPI Functions

Point-to-point message-passing:

- Int MPI-Send(buff, count, MPI_type, dest, tag)
- e.g. MPI::COMM_WORLD.Send(&ergebnis[id], 1, MPI::DOUBLE, 0, 1);
- Int MPI-Recv(buff, count, MPI_type, source, tag, stat)
- e.g. MPI::COMM_WORLD.Recv(&ergebnis[q], 1, MPI::DOUBLE, q, 1, status);

- Collective Communication: MPI_Bcast
- Barriers: MPI_Barrier
- ....

Additional material:

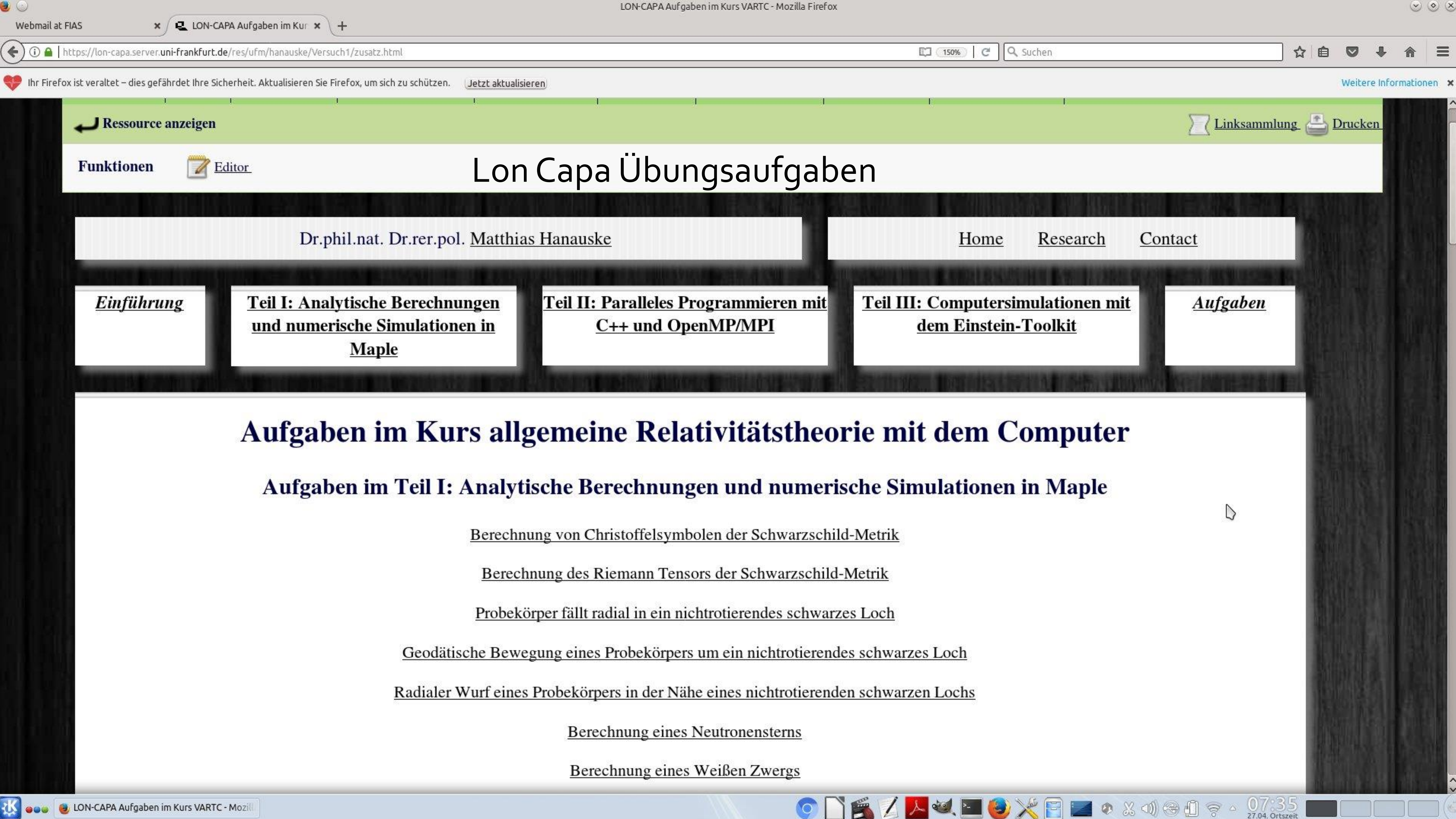
MPI-Tutorial: https://computing.llnl.gov/tutorials/mpi/

Book: Parallel Programming in C with MPI and OpenMP, by Michael J. Quinn.

Book: Patterns for Parallel Programming, by Timothy G. Mattson, et.al.

MPI-Examples: http://people.sc.fsu.edu/~jburkardt/cpp_src/mpi/mpi.html

Tutorium Examples: http://fias.uni-frankfurt.de/~hanauske/new/parallel/mpi/



LON-CAPA Aufgaben im Kurs VARTC - Mozilla Firefox
Webmail at FIAS
LON-CAPA Aufgaben im Kur
https://lon-capa.server.uni-frankfurt.de/res/ufm/hanauske/Versuch1/zusatz.html
Ihr Firefox ist veraltet – dies gefährdet Ihre Sicherheit. Aktualisieren Sie Firefox, um sich zu schützen.
Jetzt aktualisieren
Weitere Informationen
Ressource anzeigen
Linksammlung
Drucken
Funktionen
Editor
Lon Capa Übungsaufgaben
Dr.phil.nat. Dr.rer.pol. Matthias Hanauske
Home
Research
Contact
Einführung
Teil I: Analytische Berechnungen und numerische Simulationen in Maple
Teil II: Paralleles Programmieren mit C++ und OpenMP/MPI
Teil III: Computersimulationen mit dem Einstein-Toolkit
Aufgaben
Aufgaben im Kurs allgemeine Relativitätstheorie mit dem Computer
Aufgaben im Teil I: Analytische Berechnungen und numerische Simulationen in Maple
Berechnung von Christoffelsymbolen der Schwarzschild-Metrik
Berechnung des Riemann Tensors der Schwarzschild-Metrik
Probekörper fällt radial in ein nichtrotierendes schwarzes Loch
Geodätische Bewegung eines Probekörpers um ein nichtrotierendes schwarzes Loch
Radialer Wurf eines Probekörpers in der Nähe eines nichtrotierenden schwarzen Lochs
Berechnung eines Neutronensterns
Berechnung eines Weißen Zwergs
LON-CAPA Aufgaben im Kurs VARTC - Mozil
07:35