Allgemeine Relativitätstheorie mit dem Computer

PC-POOL RAUM 01.120
JOHANN WOLFGANG GOETHE UNIVERSITÄT
07. JUNI, 2019

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

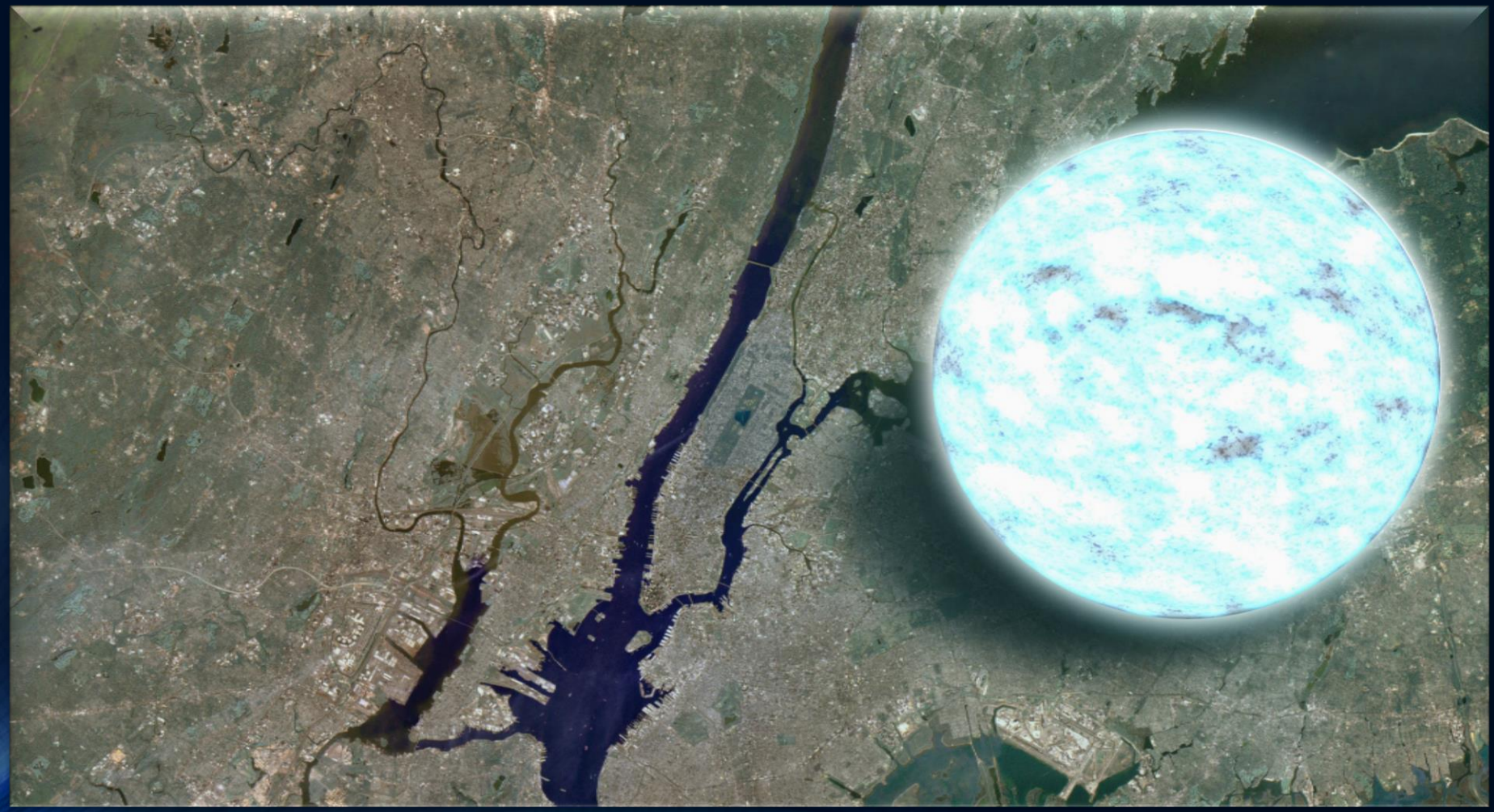
Allgemeines zur Vorlesung

- Ort und Zeit:
  PC-Pool Raum 01.120, immer freitags von 15.00 bis 17.00 Uhr
- Übungstermine:
  Zusätzlich zur Vorlesung werden ab dem 03.05.2019 freiwillige Übungstermine eingerichtet, die jeweils freitags, eine Stunde vor der Vorlesung im PC-Pool 01.120 stattfinden (Fr. 14-15.00 Uhr).
- Vorlesungs-Materialien:
  http://th.physik.uni-frankfurt.de/~hanauske/VARTC/
- Kurs auf der Online-Lernplatform Lon Capa:
  http://lon-capa.server.uni-frankfurt.de/
- Heute: Teil II: Numerisches Lösen der TOV-Gleichungen, Parallele Programmierung mit OpenMP und MPI, das paralleleC++ Programm zum Berechnender Tolman-OppenheimerVolkoff (TOV)

Neutronensterne: Sehr klein und sehr schwer

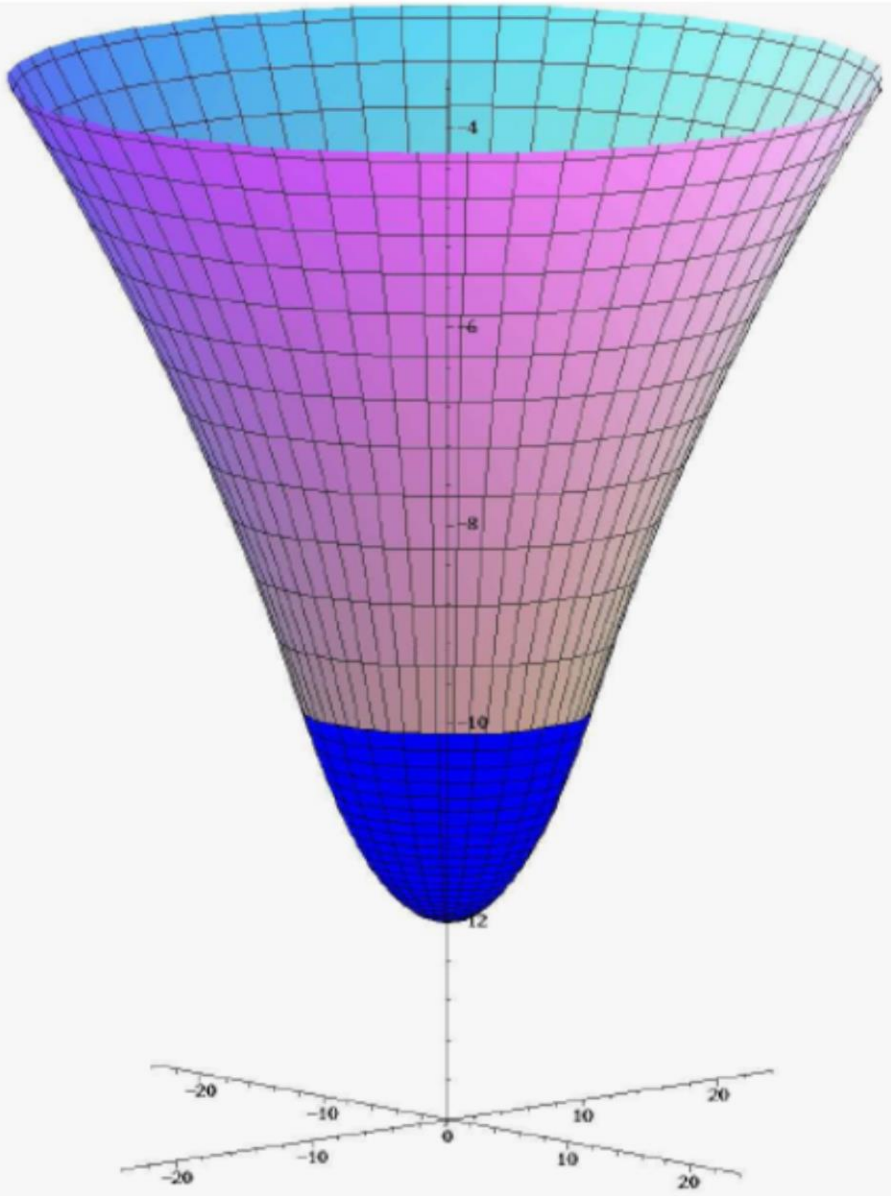
Radius ~ 10 km, Masse ~ 1-2 Sonnenmassen

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

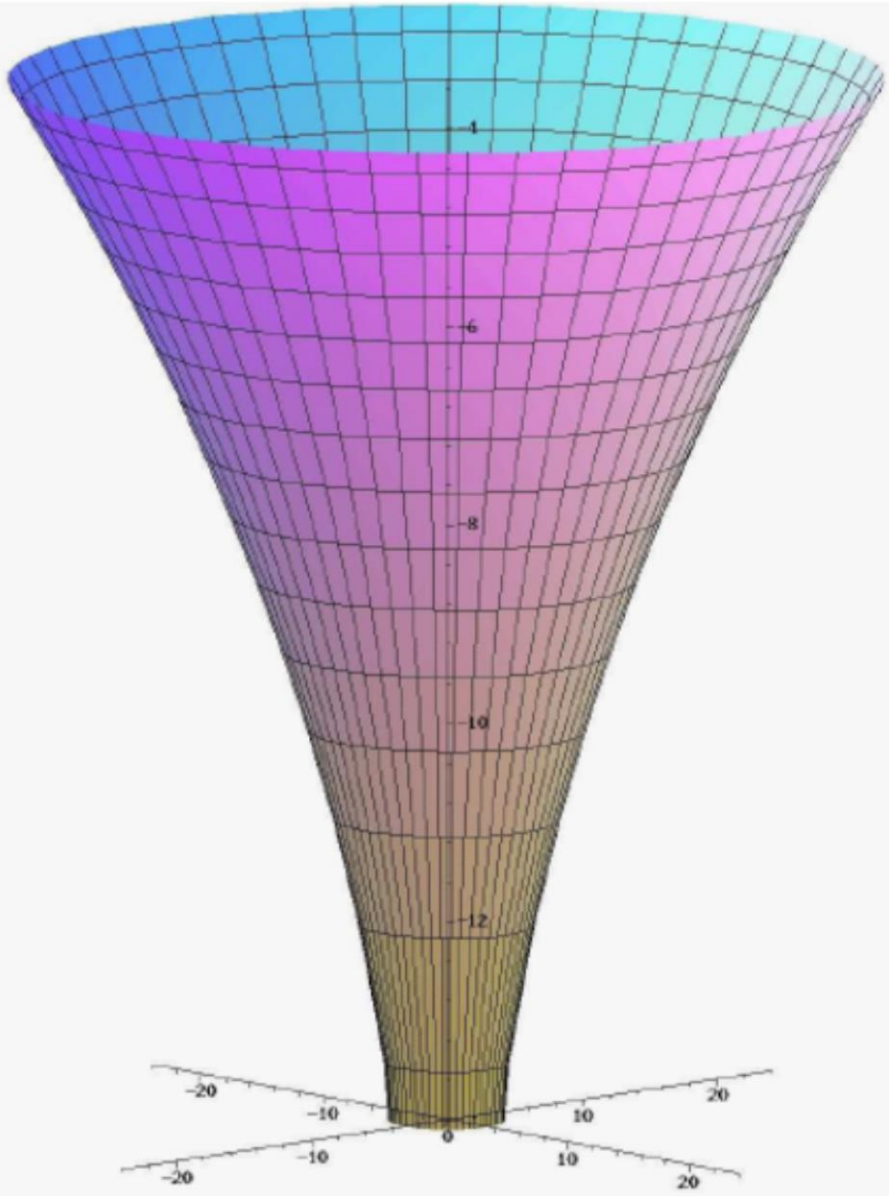


NASA/Goddard Space Flight Center

Neutronenstern



Schwarzes Loch



From the Einstein equation to the TOV equation

$$g_{\mu\nu} = \begin{pmatrix} e^{\nu(r)} & 0 & 0 & 0 \\ 0 & -e^{\lambda(r)} & 0 & 0 \\ 0 & 0 & -r^2 & 0 \\ 0 & 0 & 0 & -r^2\sin^2\theta \end{pmatrix} . \tag{2.45}$$

Das Einsetzen dieses Ansatzes der Metrik in die Einsteingleichung

$$G^{\mu}{}_{\nu} = R^{\mu}{}_{\nu} - \frac{1}{2} R\, g^{\mu}{}_{\nu} = 8\pi\kappa T^{\mu}{}_{\nu} \tag{2.46}$$

liefert das folgende System von Differentialgleichungen:

$$\begin{aligned} G^{t}{}_{t} &= -e^{-\lambda}\left(\tfrac{1}{r^2} - \tfrac{\lambda|}{r}\right) + \tfrac{1}{r^2} &&= 8\pi\kappa T^{t}{}_{t} \\ G^{r}{}_{r} &= -e^{-\lambda}\left(\tfrac{1}{r^2} + \tfrac{\nu|}{r}\right) + \tfrac{1}{r^2} &&= 8\pi\kappa T^{r}{}_{r} \\ G^{\theta}{}_{\theta} &= -\tfrac{e^{-\lambda}}{2}\left(\nu|| - \tfrac{\lambda|\nu|}{2} + \tfrac{(\nu|)^2}{2} + \tfrac{\nu|-\lambda|}{r}\right) &&= 8\pi\kappa T^{\theta}{}_{\theta} \\ G^{\phi}{}_{\phi} &= G^{\theta}{}_{\theta} &&= 8\pi\kappa T^{\phi}{}_{\phi} \end{aligned} \tag{2.47}$$

Der Energie-Impuls Tensor

$1..3, i \neq j$) vernachlässigen. Der Energieimpulstensor $T^{\mu\nu}$ einer solchen idealen Flüssigkeit, lokal betrachtet an seinem Ort, kann wie folgt geschrieben werden

$$T^{\mu\nu} = (\epsilon + P)u^\mu u^\nu - g^{\mu\nu} P \quad \text{mit: } u^\mu = \frac{dx^\mu}{d\tau} \quad , \tag{2.48}$$

wobei u^μ die 4er Geschwindigkeit der Materie ist, τ die lokale Eigenzeit an einem betrachteten Materiepunkt beschreibt ($d\tau = \sqrt{ds^2} = \sqrt{g_{tt}}\, dt$, t ist die Koordinatenzeit eines unendlich entfernten Beobachters), ϵ die Energiedichte und P der Druck der Materie ist.

Die Tolman-Oppenheimer-Volkoff Gleichung

als die **Tollman-Oppenheimer-Volkoff (TOV) Gleichungen**

$$\begin{aligned} \frac{dP}{dr} &= -\frac{(\epsilon + P)4\pi r^3 + m}{r(r - 2\,m)} \\ m(r) &= \int_0^r 4\pi \tilde{r}^2 \epsilon(\tilde{r})\, d\tilde{r} \\ \frac{d\nu}{dr} &= \frac{8\pi P r^3 + 2\,m}{r(r - 2\,m)} \quad , \end{aligned} \tag{2.61}$$

wobei die raumzeitliche Struktur durch die folgenden Ausdrücke bestimmt ist

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

$$ds^2 = e^{\nu(r)} dt^2 - \left(1 - \frac{2\,m(r)}{r}\right)^{-1} dr^2 - r^2 \left(d\theta^2 + \sin^2\theta d\phi^2\right) .$$

Die innere Schwarzschildlösung eines sphärisch symetrischen, statischen Objektes (z.B. Erde, Neutronenstern)

Im folgenden wird die Einsteingleichung einer sphärisch symetrischen und statischen Matrieverteilung betrachtet. Die Matrie wird hierbei als ideale Flüssigkeit angesetzt.

Von der Einstein Gleichung zur Tolman-Oppenheimer-Volkoff Gleichung (TOV)

```
>    restart:
  with( tensor ):
```

Wir definieren einen sphärisch symetrischen und statischen Ansatz der Metrik:

$$g_{\mu\nu} = \begin{pmatrix} e^{2\phi(r)} & 0 & 0 & 0 \\ 0 & -\left(1-\frac{2m(r)}{r}\right)^{-1} & 0 & 0 \\ 0 & 0 & -r^2 & 0 \\ 0 & 0 & 0 & -r^2\sin^2(\theta) \end{pmatrix} \qquad \text{mit:} \quad x^\mu = (t, r, \theta, \phi) \quad ,$$

wobei die Funktionen $\phi(r)$ und $m(r)$ an dieser Stelle noch unbekannt sind und keine physikalische Bedeutung besitzen.

```
>          coord := [t, r, theta, phi]:
  g_compts := array(symmetric,sparse, 1..4, 1..4):
           g_compts[1,1] := exp(2*phi(r)):
        #g_compts[2,2] := exp(2*lambda(r)):
```

Der Energie-Impuls Tensor (rechte Seite der Einsteingleichung) wird als ideale Flüssigkeit angesetzt:

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

wobei die Funktionen $e(r)$ und $p(r)$ die Energiedichte und den Druck der Neutronensternmaterie darstellen, die ihrerseits über die Zustandsgleichung $p(e)$ miteinander verknüpft sind.

```
> T:=create([1,-1], array([[e(r),0,0,0],[0,-p(r),0,0],[0,0,-p(r),0],[0,0,0,-p(r)]]));
                                    Tl:=lower(g,T,1);
                                  Tu:=raise(ginv,T,2):
                                prod(ginv, Tl, [2, 1]):
                                  contract(T, [1, 2]):
```

$$T := table\left(\left[compts = \begin{bmatrix} e(r) & 0 & 0 & 0 \\ 0 & -p(r) & 0 & 0 \\ 0 & 0 & -p(r) & 0 \\ 0 & 0 & 0 & -p(r) \end{bmatrix}, index\_char = [1, -1]\right]\right)$$

$$Tl := table\left(\left[compts = \begin{bmatrix} e^{2\Phi(r)}\, e(r) & 0 & 0 & 0 \\ 0 & -\frac{r\, p(r)}{-r + 2\, m(r)} & 0 & 0 \\ 0 & 0 & r^2\, p(r) & 0 \\ 0 & 0 & 0 & r^2 \sin(\theta)^2\, p(r) \end{bmatrix}, index\_char = [-1, -1]\right]\right)$$

(2.1.4)

Aus der Einsteingleichung folgt die Erhaltung des Energie-Impulses. Diese sogenannten hydrodynamischen Gleichungen (kovariante Erhaltung des Energie-Impulses) sind durch die folgenden vier Gleichungen definiert (Bemerke: in der Literatur wird die kovariante Ableitung mit unterschiedlichen Symbolen bezeichnet):

$$\nabla_\mu G^\mu{}_\nu = D_\mu G^\mu{}_\nu = G^\mu{}_{\nu\,||\mu} = 0 \quad \rightarrow \quad \nabla_\mu T^\mu{}_\nu = 0 \quad .$$

wobei die kovariante Ableitung eines Tensors zweiter Stufe wie folgt definiert ist:

$$\nabla_\alpha T^\mu{}_\nu = \partial_\alpha T^\mu{}_\nu + \Gamma^\mu_{\alpha\rho} T^\rho{}_\nu - \Gamma^\rho_{\alpha\nu} T^\mu{}_\rho \quad ,$$

```
>       DT:=cov_diff(T, coord, Cf2):
 DTa:=get_compts(contract(DT, [1, 3]))[2]=0;
```

$$DTa := -\left(\frac{\mathrm{d}}{\mathrm{d}r}\phi(r)\right) p(r) - \left(\frac{\mathrm{d}}{\mathrm{d}r}\phi(r)\right) e(r) - \left(\frac{\mathrm{d}}{\mathrm{d}r} p(r)\right) = 0 \qquad (2.1.7)$$

und nach $\frac{dp}{dr}$ aufgelöst ergibt sich das folgende:

$$TOV1 := \frac{d}{dr} p(r) = \frac{\left(m(r) + 4\,\pi\, r^3\, p(r)\right)\,\left(e(r) + p(r)\right)}{r\,\left(-r + 2\,m(r)\right)}$$

$$TOV2 := \frac{d}{dr} m(r) = 4\,\pi\, e(r)\, r^2$$

$$TOV3 := \frac{d}{dr} \Phi(r) = -\frac{m(r) + 4\,\pi\, r^3\, p(r)}{r\,\left(-r + 2\,m(r)\right)} \qquad (2.1.10)$$

Das oben abgebildete System aus drei gekoppelten Differentialgleichungen bezeichnet man als die Tolman-Oppenheimer-Volkoff Gleichungen (TOV-Gleichung). Bemerkung: In manchen Büchern werden auch lediglich die ersten beiden Gleichungen als TOV-Gleichungen bezeichnet.

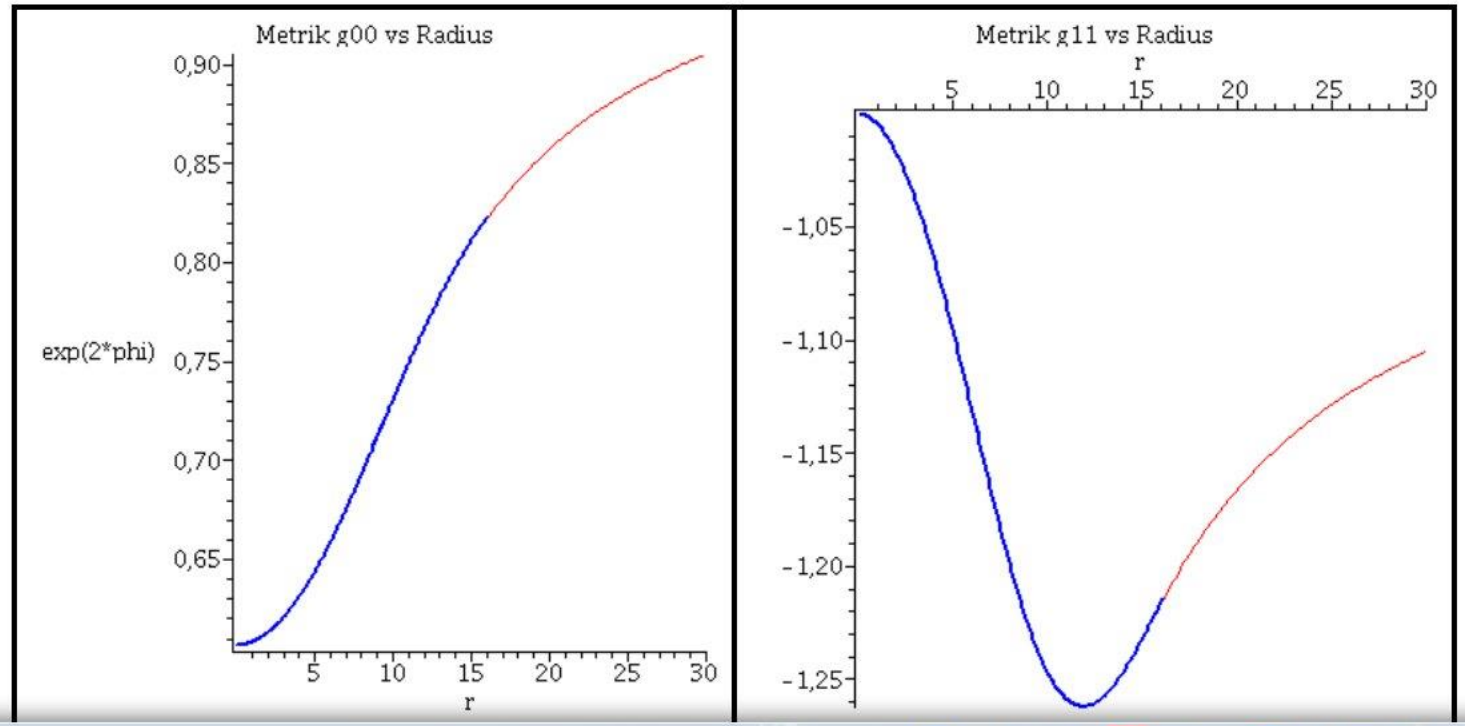
Numerische Lösung der TOV-Gleichungen

Im folgenden werden die TOV-Gleichungen numerisch gelöst, indem wir einerseits eine Zustandsgleichung der Materie (eine Funktion p(e)) festlegen und von einem Startwert der zentralen Energiedichte im Inneren des sphärisch symetrischen Objektes nach Außen integrieren.

```
> a:=10;
  b:=5/3:
  p(r):=a*(e(r))^b;
  W3:=plot(a*x^b,x=0..1,color=blue):
  TOV1:=solve(DTa,diff(phi(r), r))*(e(r)+p(r))*(-1)=rhs(Einstein2)*
  (e(r)+p(r))*(-1);
  TOV2:=Einstein1;
  TOV3:=Einstein2;
```

Veranschaulichung der g_{00}-Komponente (linke Abbildung) und g_{11}-Komponente (rechte Abbildung) der Innenraummetrik (blaue Kurven) und Außenraummetrik (rote Kurven).

```
> ranf:=10^(-1):
Plot3:=odeplot(Loes1,[r,exp(2*phi(r))],ranf..rend,numpoints=200,color=blue,thickness=2,title="Metrik g00 vs Radius"):
Plot4:=odeplot(Loes1,[r,-1/(1-2*m(r)/r)],ranf..rend,numpoints=200,color=blue,thickness=2,title="Metrik g11 vs Radius"):
Plot5:=plot(1-2*M/r,r=rend..30,color=red):
Plot6:=plot(-1/(1-2*M/r),r=rend..30,color=red):
display(Matrix(1,2,[[display(Plot3,Plot5),display(Plot4,Plot6)]]));
```



C++ Grundgerüst und Variablen

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

Beginn des Hauptprogramms

Ausgabe eines Strings

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

Variablen bekommen einen festen Zahlenwert (Initialisierung)

Ausgabe des Wertes der Variablen

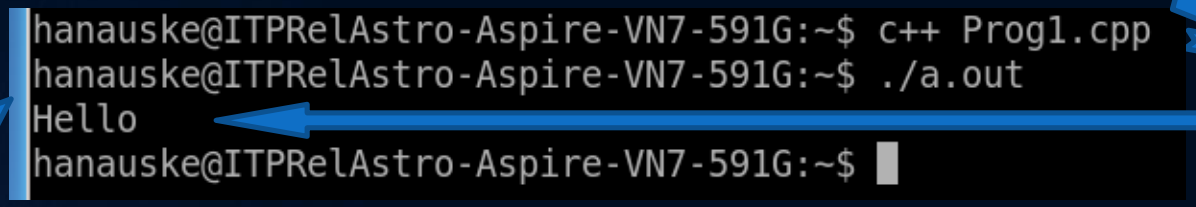
```
#include <iostream>                    //Ein-/Ausgabe (Include-Dateien)
using namespace std;                   //Fuer cout

main(void)  //Hauptprogramm
  {

    cout<<"Hello \n";

  }
```


```
#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 compiliert 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



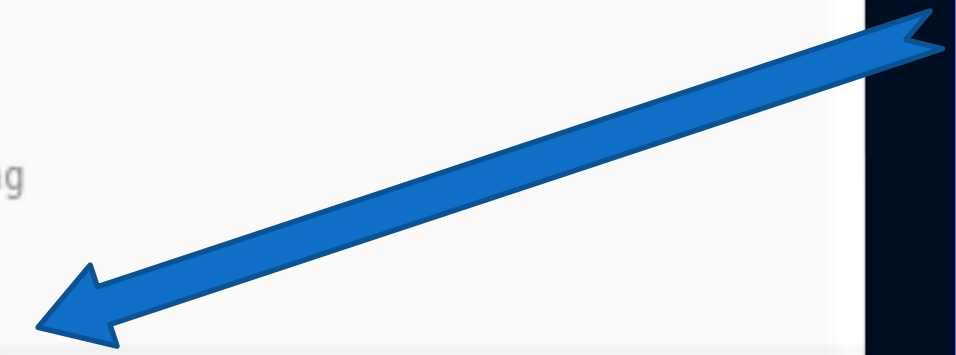
Das Programm wird gestartet und erzeugt im Terminal die Ausgabe “Hello”

Beim Compilierungsprozess 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 Interation dürchführen soll (i<=10) und die Angabe um wieviel 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;).

Einführung in die Parallele Programmierung

fias.uni-frankfurt.de/~hanauske/VARTC/T2/intro/Hanauske_ParallelizationTut.odp
fias.uni-frankfurt.de/~hanauske/VARTC/T2/intro/Hanauske_ParallelizationTut.pdf

Introduction

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

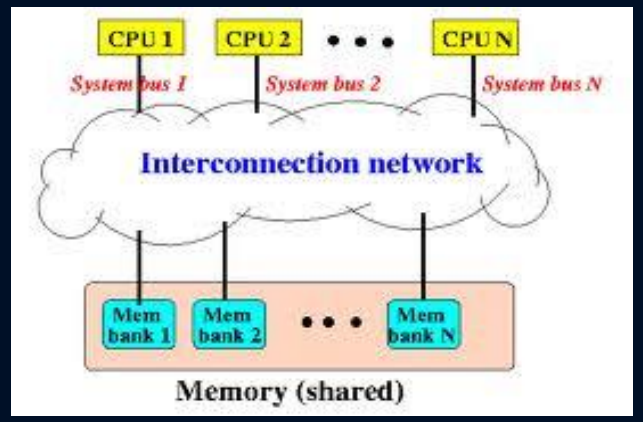
Depending on the architecture of the parallel computer (or computer cluster) the **Parallel Programming Framework** (OpenMP, MPI, Cuda, OpenCL, ...) has to be choosen .

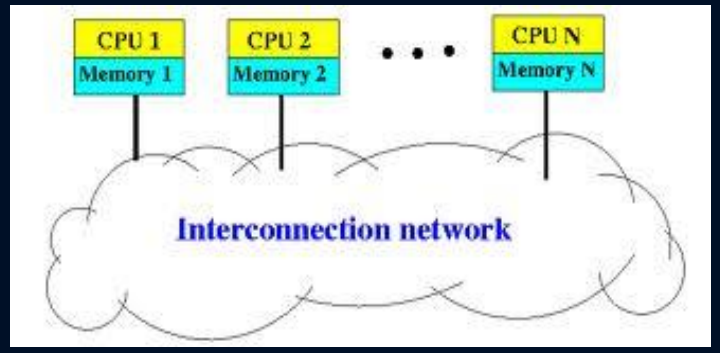
Parallel computers

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

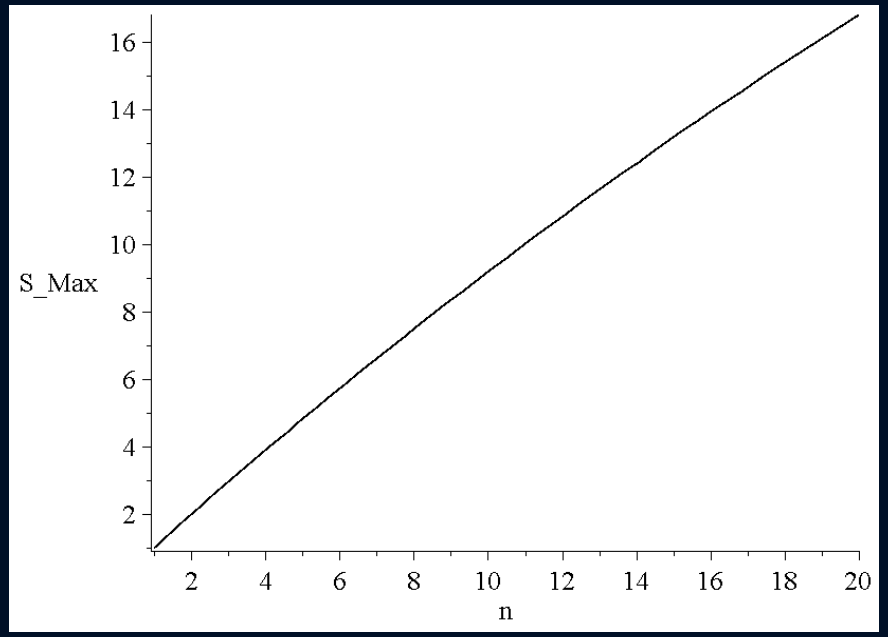
The **performance of a parallel computer code** can be measured using the following characteristic values:

T(n): Time needed to run the programm 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 diveded in two parts (T(n)=Ts+Tp(n)), where Ts is the time needed for the non-parallizable part of the programm and Tp(n) is the parallizable part, which can be executed concurrently by different processors. A(n):=Max(S(n))

A(n)=1/(a+(1-a)/n), with a:=Ts/T(1)



Amdahl's law with a=[0.01,0.4]

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

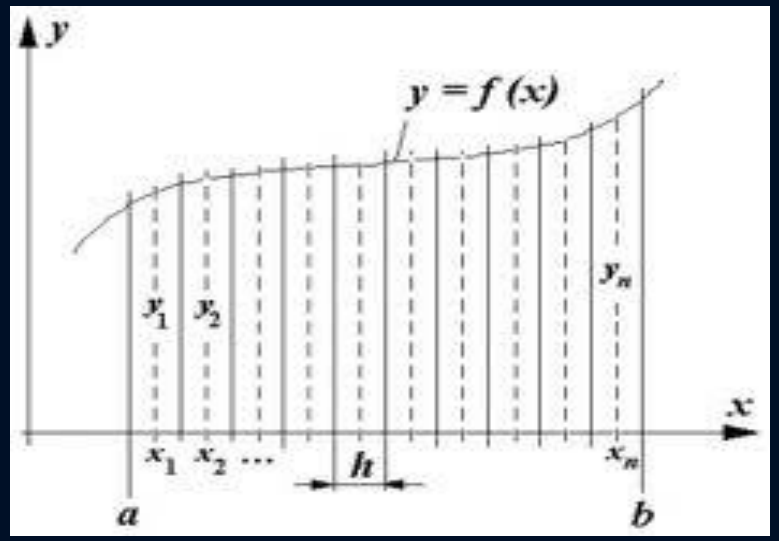
OpenMP has the advantage that common **sequential codes can easily be changed** by simply adding some OpenMP directives. Another feature of OpenMP is that the program runs also properly (but then sequentially, using only one thread) even if the compiler does not know OpenMP.

Example

The simple computation problem used in the following is the nummerical 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 diveded into N pieces. The value of the integration function is taken at the middle of each integration segment (Gauss method).



Gauss'schen integration method.

Sequential Code

```c
#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);

    for (int i = 1; i <= 10; ++i)
    {
    printf("a=%i: Integral=%e, Difference=%e \n"
        ,i,integral(10000000,i)
        ,integral(10000000,i)-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

```c
#include <stdio.h>
#include <math.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)
{
    double integral(int,double);

#pragma omp parallel for
    for (int i = 1; i <= 10; ++i)
    {
    printf("a=%i: Integral=%e, Difference=%e \n"
        ,i,integral(10000000,i)
        ,integral(10000000,i)-atan(sqrt(i))/sqrt(i));
    }

    return 0;
}
```

To parallize 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 parallized 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

```c
#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);

#pragma omp parallel for
    for (int i = 1; i <= 10; ++i)
    {
    int id = omp_get_thread_num();
    printf("a=%i: Integral=%e, Difference=%e, Thread No:%i \n"
        ,i,integral(10000000,i)
        ,integral(100000000,i)-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 parallized 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.

```
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ c++ sequential_time.cpp
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ ./a.out
a=1: Integral=7.853983e-01, Difference=1.000016e-08
a=2: Integral=6.755110e-01, Difference=9.999904e-09
a=3: Integral=6.045999e-01, Difference=9.999895e-09
a=4: Integral=5.535745e-01, Difference=9.999747e-09
a=5: Integral=5.144129e-01, Difference=1.000001e-08
a=6: Integral=4.830393e-01, Difference=1.000005e-08
a=7: Integral=4.571214e-01, Difference=9.999836e-09
a=8: Integral=4.352100e-01, Difference=9.999864e-09
a=9: Integral=4.163487e-01, Difference=1.000005e-08
a=10: Integral=3.998761e-01, Difference=1.000015e-08
Time needed: 22 seconds
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$
```

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

```
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ c++ -fopenmp parallel_omp_1_time_id.cpp
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ ./a.out
a=6: Integral=4.830393e-01, Difference=1.000005e-08, Thread No:1
a=1: Integral=7.853983e-01, Difference=1.000016e-08, Thread No:0
a=7: Integral=4.571214e-01, Difference=9.999836e-09, Thread No:1
a=2: Integral=6.755110e-01, Difference=9.999904e-09, Thread No:0
a=8: Integral=4.352100e-01, Difference=9.999864e-09, Thread No:1
a=3: Integral=6.045999e-01, Difference=9.999895e-09, Thread No:0
a=9: Integral=4.163487e-01, Difference=1.000005e-08, Thread No:1
a=4: Integral=5.535745e-01, Difference=9.999747e-09, Thread No:0
a=10: Integral=3.998761e-01, Difference=1.000015e-08, Thread No:1
a=5: Integral=5.144129e-01, Difference=1.000001e-08, Thread No:0
Time needed: 10 seconds
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$
```

Parallel Code No.2

(wrong!)

```c
#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
    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);

    for (int i = 1; i <= 10; ++i)
    {
    printf("a=%i: Integral=%e, Difference=%e \n"
        ,i,integral(10000000,i)
        ,integral(100000000,i)-atan(sqrt(i))/sqrt(i));
    }

    printf("Time needed: %i seconds\n"
        ,(int)(time(NULL)-startTime));
    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.

```
hanauske@green:~/Vortraege/Tutorial_Parallelization/Ope
a=1: Integral=2.899848e-01, Difference=-4.724802e-01
a=2: Integral=2.628621e-01, Difference=-4.165062e-01
a=3: Integral=2.221546e-01, Difference=-3.745245e-01
a=4: Integral=2.037079e-01, Difference=-3.450925e-01
a=5: Integral=1.909233e-01, Difference=-3.232404e-01
a=6: Integral=1.833300e-01, Difference=-3.038624e-01
a=7: Integral=1.679277e-01, Difference=-2.861134e-01
a=8: Integral=1.638385e-01, Difference=-2.715731e-01
a=9: Integral=1.523053e-01, Difference=-2.583911e-01
a=10: Integral=1.486715e-01, Difference=-2.531299e-01
Time needed: 25 seconds
hanauske@green:~/Vortraege/Tutorial_Parallelization/Ope
```

Two problems arise when executing the program:

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

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

Parallel Code No.2

```c
#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);

    for (int i = 1; i <= 10; ++i)
    {
    printf("a=%i: Integral=%e, Difference=%e \n"
        ,i,integral(10000000,i)
        ,integral(100000000,i)-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 comunication 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 adress 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 slowdown of execution.

This 'race condition' can be solved using the syncronisation directive

reduction(+:sum)

Running the code

Sequential version:

```
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ c++ sequential_time.cpp
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ ./a.out
a=1: Integral=7.853983e-01, Difference=1.000016e-08
a=2: Integral=6.755110e-01, Difference=9.999904e-09
a=3: Integral=6.045999e-01, Difference=9.999895e-09
a=4: Integral=5.535745e-01, Difference=9.999747e-09
a=5: Integral=5.144129e-01, Difference=1.000001e-08
a=6: Integral=4.830393e-01, Difference=1.000005e-08
a=7: Integral=4.571214e-01, Difference=9.999836e-09
a=8: Integral=4.352100e-01, Difference=9.999864e-09
a=9: Integral=4.163487e-01, Difference=1.000005e-08
a=10: Integral=3.998761e-01, Difference=1.000015e-08
Time needed: 22 seconds
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$
```

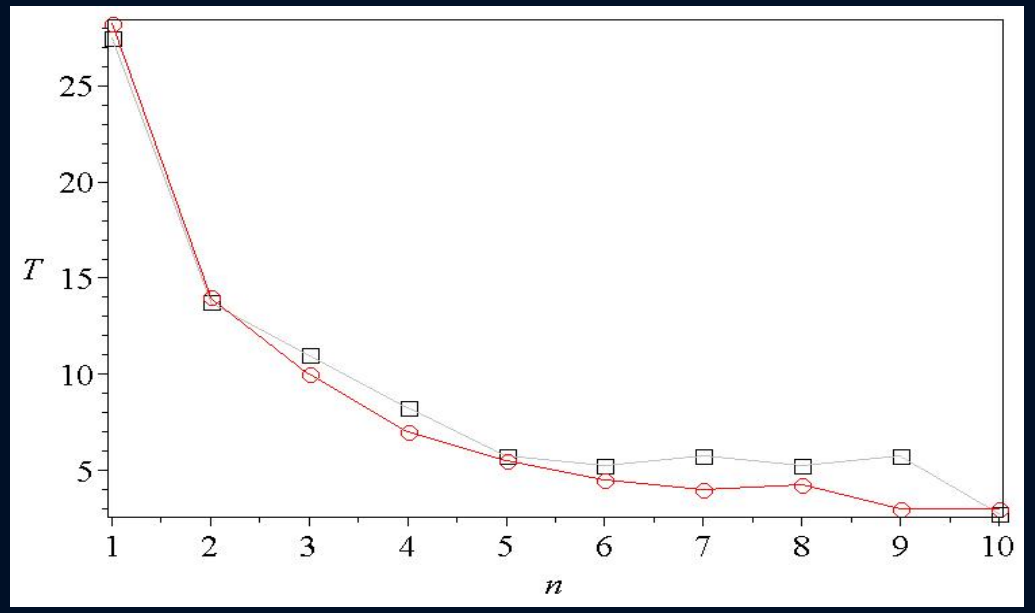
Parallel version (No.2): Due to the non-sequential summation of the different parts of the integral, a different rounding error occurs within the parallel version. This is the reason that the calculated integrals are not exactly the same as in the sequential version.

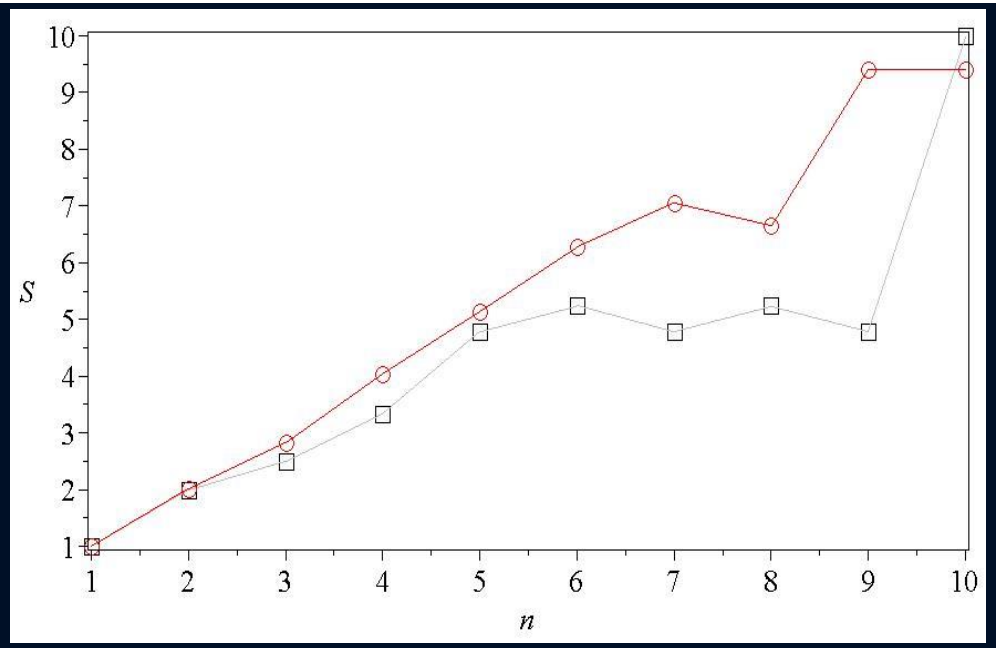
```
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ c++ -fopenmp parallel_omp_2_time.cpp
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ ./a.out
a=1: Integral=7.853983e-01, Difference=1.000003e-08
a=2: Integral=6.755110e-01, Difference=1.000009e-08
a=3: Integral=6.045999e-01, Difference=9.999958e-09
a=4: Integral=5.535745e-01, Difference=9.999924e-09
a=5: Integral=5.144129e-01, Difference=9.999987e-09
a=6: Integral=4.830393e-01, Difference=9.999908e-09
a=7: Integral=4.571214e-01, Difference=1.000000e-08
a=8: Integral=4.352100e-01, Difference=1.000000e-08
a=9: Integral=4.163487e-01, Difference=1.000012e-08
a=10: Integral=3.998761e-01, Difference=9.999997e-09
Time needed: 11 seconds
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$
```

Performance

The following calculations where performed on 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 programm 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);
    int startTime = time(NULL);

    for (int i = id+1; i <= 10; i = i + p)
    {
        printf("a=%i: Integral=%e, Difference=%e \n"
        ,i,integral(10000000,i)
        ,integral(100000000,i)-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 diveded 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).

The first run on the right hand side was performed by only using one process (sequential version).

The second run was much faster and has used two processes.

```
hanauske@green:~/Vortraege/Tutorial_Parallelization/MPI$ mpic++ parallel_mpi_time.cpp
hanauske@green:~/Vortraege/Tutorial_Parallelization/MPI$ mpirun -np 1 ./a.out
a=1: Integral=7.853983e-01, Difference=1.000016e-08
a=2: Integral=6.755110e-01, Difference=9.999904e-09
a=3: Integral=6.045999e-01, Difference=9.999895e-09
a=4: Integral=5.535745e-01, Difference=9.999747e-09
a=5: Integral=5.144129e-01, Difference=1.000001e-08
a=6: Integral=4.830393e-01, Difference=1.000005e-08
a=7: Integral=4.571214e-01, Difference=9.999836e-09
a=8: Integral=4.352100e-01, Difference=9.999864e-09
a=9: Integral=4.163487e-01, Difference=1.000005e-08
a=10: Integral=3.998761e-01, Difference=1.000015e-08
Time needed: 21 seconds
hanauske@green:~/Vortraege/Tutorial_Parallelization/MPI$ mpirun -np 2 ./a.out
a=2: Integral=6.755110e-01, Difference=9.999904e-09
a=1: Integral=7.853983e-01, Difference=1.000016e-08
a=4: Integral=5.535745e-01, Difference=9.999747e-09
a=3: Integral=6.045999e-01, Difference=9.999895e-09
a=6: Integral=4.830393e-01, Difference=1.000005e-08
a=5: Integral=5.144129e-01, Difference=1.000001e-08
a=8: Integral=4.352100e-01, Difference=9.999864e-09
a=7: Integral=4.571214e-01, Difference=9.999836e-09
a=10: Integral=3.998761e-01, Difference=1.000015e-08
a=9: Integral=4.163487e-01, Difference=1.000005e-08
Time needed: 10 seconds
```

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(500000000,1,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=500 million). The loop that performs these 500 milion iterations is diveded among different processes.

As every process nows only the part that it has calculated, the processes need to comunicate in order to calculate the value of the whole integral. Within MPI, several way of communications are possible. Within this version a point-to-point comunication has 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 on the right hand side was performed by only using one process (sequential version). The second run was much faster and has used two processes.

```
hanauske@green:~/Vortraege/Tutorial_Parallelization/MPI$ mpic++ parallel_mpi_2_time.cpp
hanauske@green:~/Vortraege/Tutorial_Parallelization/MPI$ mpirun -np 1 ./a.out
a=1: Integral=7.853982e-01, Difference=2.000005e-09
Time needed: 10 seconds
hanauske@green:~/Vortraege/Tutorial_Parallelization/MPI$ mpirun -np 2 ./a.out
a=1: Integral=7.853982e-01, Difference=1.999957e-09
Time needed: 5 seconds
hanauske@green:~/Vortraege/Tutorial_Parallelization/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/

Die TOV Gleichungen

Allgemeine Relativiätstheorie mit dem Computer: Teil II

Grundlagen zur numerischen Lösung der Tolman-Oppenheimer-Volkoff Gleichung (einfaches Euler Verfahren)

Das Differentialgleichungssystem der Tolman-Oppenheimer-Volkoff (TOV) Gleichung besitzt das folgende Aussehen

$$\frac{dp}{dr} = -\frac{(p+e)\,(m+4\pi r^3 p)}{r\,(r-2m)} \tag{1}$$

$$\frac{dm}{dr} = 4\pi r^2 e \tag{2}$$

$$\frac{d\phi}{dr} = \frac{m+4\pi r^3 p}{r\,(r-2m)} \quad , \tag{3}$$

wobei $p = p(r)$ und $e = e(r)$ der Druck und die Energiedichte der Materie darstellen, $m = m(r)$ die radiusabhängige gravitative Masse ist und die Funktion $\phi = \phi(r)$ die 00- bzw. tt-Komponente der Metrik bestimmt ($g_{00} = e^{2\phi}$; hier bezeichnet e die Eulersche Zahl!).

TOV-Gleichungen: Numerisches Vorgehen

Eine numerische Lösung der Sterneigenschaften benötigt lediglich Gleichung (1) und (2) und geht im einfachsten Fall (einfaches Euler Verfahren) nach folgendem Schema vor:

- Man definiert die Zustandsgleichung (EOS) der Sternmaterie als eine Funktion e(p).
- Man startet im Sternzentrum $r = r_0$ und legt den Wert des zentralen Druckes $p = p_0 := p(r_0)$, der zentralen Energiedichte $e = e_0 := e(r_0)$ und der Masse $m = m_0 := m(r_0) = 0$ fest. Da die TOV Gleichung (1) bei $r_0 = 0$ singulär wird, wählt man hier einen sehr, sehr kleinen Wert für r_0 (z.B. $r_0 = 10^{-14}$).

$$r = 10^{-14}\,, \quad p = p_0\,, \quad e = e_0\,, \quad m = 0 \tag{4}$$

- Die TOV Gleichungen werden als Differenzengleichungen umgeschrieben und eine kleine Schrittweite $dr = \Delta r << 1$ wird festgelegt. In einer Schleife wird dann in jedem Radiusschritt die Druck- und Massenänderung berechnet und die jeweiligen Größen beim nächsten Schritt um diesen Faktor erhöht bzw. verringert:

$$dp = -\frac{(p+e)\,(m+4\pi r^3 p)}{r\,(r-2m)}\,dr$$

TOV-Gleichungen: Numerisches Vorgehen

- Die TOV Gleichungen werden als Differenzengleichungen umgeschrieben und eine kleine Schrittweite $dr = \Delta r << 1$ wird festgelegt. In einer Schleife wird dann in jedem Radiusschritt die Druck- und Massenänderung berechnet und die jeweiligen Größen beim nächsten Schritt um diesen Faktor erhöht bzw. verringert:

$$
\begin{aligned}
dp &= -\frac{(p+e)\,(m+4\pi r^3 p)}{r\,(r-2m)}\,dr \\
dm &= 4\pi r^2 e\,dr \\
p &= p + dp \\
m &= m + dm \\
r &= r + dr
\end{aligned}
$$

- Im Laufe der iterativen Lösung verringert sich der Druck ständig. Die Schleife wird solange ausgeführt bis der Wert des Druckes gleich Null bzw. negativ wird (Abruchbedingung: $p \leq 0$), da an der Sternoberfläche der Druck verschwindet.

C++ Lösen der TOV-Gleichung

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

//Definition der Zustandsgleichung
double eos(double p)
  {
    double e;
    e=pow(p/10,3.0/5);
    return e;
  }

main(void)  //Hauptprogramm
  {
    //Variablendeklarationen
    double M,p,e,r,dM,dp,de,dr;
    double eos(double);

    //Variableninitialisierung
    M=0;
    r=pow(10,-14);
    p=10*pow(0.0005,5.0/3);
    dr=0.000001;

    //do-while Schleife (Numerische Lösung der TOV-Gleichung)
    do
      {
        e=eos(p);                                  //Wert der Energiedichte bei momentanen Druck
        dM=4*M_PI*e*r*r*dr;                        //Massenzunahme bei momentanem r und Schrittweite dr
        dp=-(p+e)*(M+4*M_PI*r*r*r*p)/(r*(r-2*M))*dr;  //Druckzunahme bei momentanem r und Schrittweite dr (TOV-Gleichung)
        r=r+dr;                                    //momentaner Radius des Neutronensterns
        M=M+dM;                                    //momentane Masse des Neutronensterns innerhalb des Radius r
        p=p+dp;                                    //momentaner Druck des Neutronensterns innerhalb des Radius r
      }
      while(p>0);

    //Ausgabe der Masse und des Radius auf dem Bildschirm
    cout<<"Neutronensternradius [km]           = "<<r<<"\n";
    cout<<"Neutronensternmasse [Sonnenmassen] = "<<M/1.4766<<"\n";

    return 0;                                      //main beenden (Programmende)
  }
```

Die polytrope **Zustandsgleichung** ist als eine Funktion außerhalb des Hauptprogramms definiert

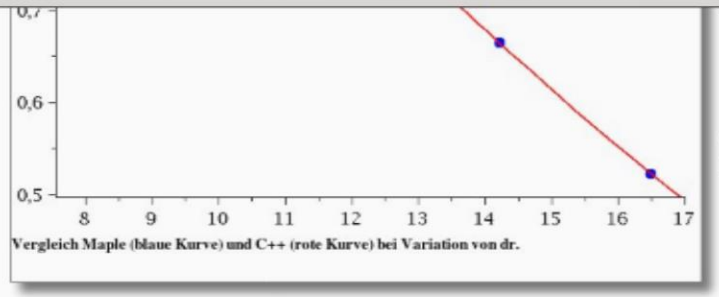
Deklaration der nötigen **Variablen** und der Zustandsgleichungsfunktion

Festlegung der **Anfangswerte** im Sternzentrum (M,r,p) und der Radiusschrittweite dr

TOV-Gleichungen

Ausgabe auf dem Bildschirm

Parallele Programme siehe Teil 2 der Internetseite der Vorlesung



Vergleich Maple (blaue Kurve) und C++ (rote Kurve) bei Variation von dr.

2.3) Parallele OpenMP-Version 1 von 2.2)

Das sequentielle Programm 2.2) wurde nun mittels OpenMP (siehe TOV OpenMP Version) parallelisiert. Hiebei wurde einfach das OpenMP-Pragma *'#pragma omp parallel for private(i,M,p,e,r,nu,dM,dp,de,dnu)'* vor die for-Schleife der unabhängigen Berechnung der einzelnen Neutronensterne geschrieben. Wichtig ist nun, dass man das Programm mit dem folgenden Befehl compiliert: 'c++ -fopenmp TOV_parallel_omp.cpp'. Führt man das Programm mit './a.out' aus, so erkennt man als erstes, dass es (in Abhängigkeit wieviele CPU-Kerne man in seinem Computer hat), viel schneller läuft. Die im Terminal ausgegebenen Werte sind jedoch nicht mehr geordnet, sondern die Berechnung der 40 Neutronensterne erfolgt parallel und ungeordnet. Die nebenstehende Abbildung zeigt die Terminalausgabe des parallelen Programms und die Auslastung der 8 CPU-Kerne meines Laptops, wobei zuerst das parallele Programm und dannach das sequentielle ausgeführt wurde.

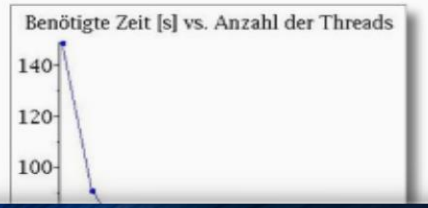
```
Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
hanauske@ITPRelAstro-Aspire-VN7-591G:~$ c++ -fopenmp TOV_parallel_omp.cpp
hanauske@ITPRelAstro-Aspire-VN7-591G:~$ ./a.out
35
Neutronensternradius [km]           = 9.13464
Neutronensternmasse [Sonnenmassen]  = 1.20785
00-Metrikkomponente im Sternzentrum = 0.198145
30
Neutronensternradius [km]           = 9.50537
Neutronensternmasse [Sonnenmassen]  = 1.21506
00-Metrikkomponente im Sternzentrum = 0.220643
25
Neutronensternradius [km]           = 9.94998
Neutronensternmasse [Sonnenmassen]  = 1.21927
00-Metrikkomponente im Sternzentrum = 0.248056
20
Neutronensternradius [km]           = 10.4976
Neutronensternmasse [Sonnenmassen]  = 1.21836
00-Metrikkomponente im Sternzentrum = 0.28223
15
Neutronensternradius [km]           = 11.1977
Neutronensternmasse [Sonnenmassen]  = 1.20841
00-Metrikkomponente im Sternzentrum = 0.326145
10
Neutronensternradius [km]           = 12.1444
```

2.4) Parallele OpenMP-Version 2 (2.3) mit geordneter Terminal-Ausgabe)

Diese Version entspricht Version 2.3) mit einer geordneten Ausgabe. Die geordnete Ausgabe wird hierbei realisiert, indem für die drei ausgegebenen, numerischen Werte (Radius, Masse, zentraler g_{00}-Wert), drei Datenfelder (Arrays) der Länge 40 eingerichtet werden. Nachdem ein OpenMP-Thread mit seiner Berechnung fertig ist, speichert er sein individuelles Ergebnis in die spezifische Position innerhalb des Arrays und berechnet den nächsten Stern. Die geordnete Ausgabe aller Werte erfolgt dann sequentiell, außerhalb der parallelisierten Schleife.

2.5) Parallele OpenMP-Version 3 (2.4) mit Ausgabe in eine Datei)

Diese Version entspricht Version 2.4) wobei die geordnete Ausgabe nun nicht mehr in dem Terminal geschieht, sondern die berechneten Werte werden in eine externe Datei (tov.txt) ausgegeben - die Ausgabedatei erfolgt im Unterordner 'output', welcher vor dem Ausführen des Programms angelegt werden muss. Der Vorteil hierbei ist, dass nachdem das Programm ausgefürt wurde, die Ergebnisse einfacher verarbeitet und dargestellt werden können. So kann man z.B. mittels Gnuplot sich das Radius-Masse Diagramm darstellen. Noch einfacher, kann man sich die einzelnen Gnuplot-Befehle zum Darstellen diverser Diagramme in ein ausführbares Shell-Script schreiben, das dann automatisch die jeweiligen Plots erzeugt (siehe Gnuplot Shell-Script).



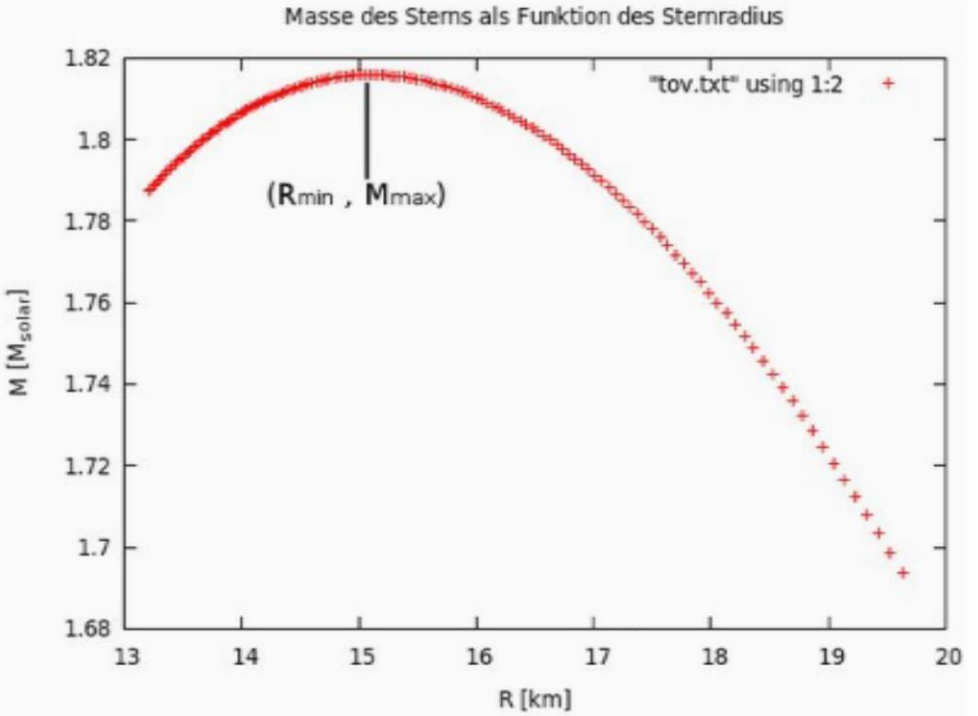
2.6) Parallele OpenMP-Version mit geordneter Ausgabe in eine Datei und variabler Zustandsgleichung

Diese Version entspricht Version 2.5), wobei die als Funktion definierte Zustandsgleichung variabler gestaltet wurde (EOS: ($e(P, K, \gamma) = (P/K)^{1/\gamma}$) für Neutronensterne und Weiße Zwerge bzw. (e(P,Bag)=3 p + 4 Bag) für Quarksterne im MIT-Bag Model).

Struktur und Performance des parallelen OpenMP - C++ Programms

Aufgaben zum 2.Teil der Vorlesung siehe E-Learning „Lon Capa“

Berechnen Sie unter Verwendung des C++ Programms aus Teil II der Vorlesung die maximale Masse M_{max} in $[\mathrm{M}_\odot]$ und den zugehörigen minimalen Radius R_{min} eines Neutronensterns in [km]. Verwenden Sie eine polytrope Zustandsgleichung der Form $p = K * e^{\gamma}$, wobei $\gamma = 5/3$ und $K = 20.25$ $\left[\mathrm{km}^{4/3}\right]$ ist.

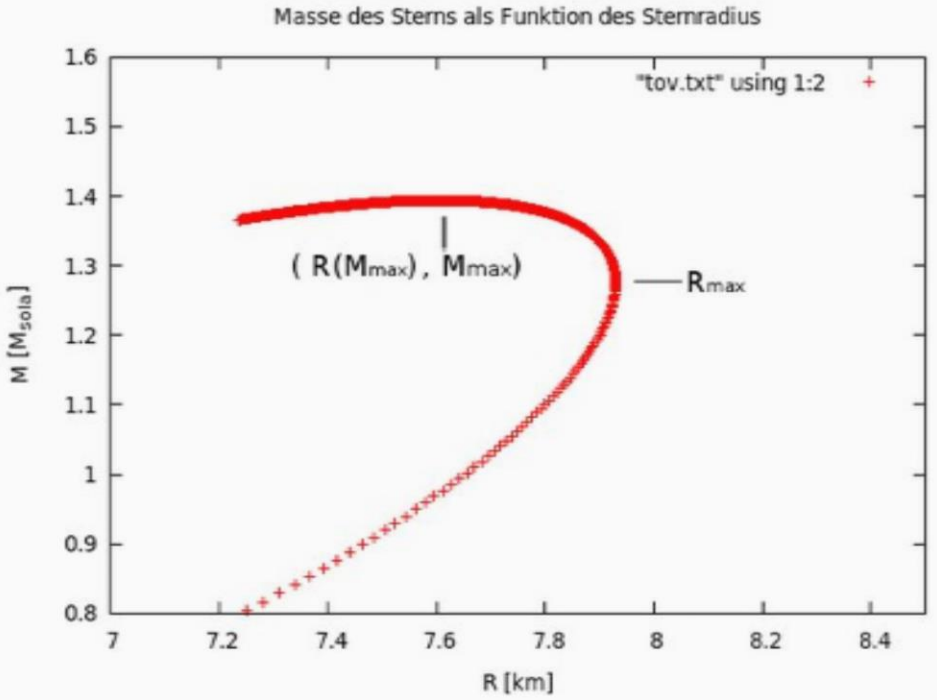


M_{max} = , R_{min} =

Antwort einreichen Versuche 0/20

Aufgaben zum 2.Teil der Vorlesung siehe E-Learning „Lon Capa“

Berechnen Sie unter Verwendung des C++ Programms aus Teil II der Vorlesung die maximale Masse M_{max} in $[\mathrm{M}_\odot]$ und den zugehörigen Radius $R(M_{max})$ eines Quarkstern Modells in [km]. Verwenden Sie die lineare Zustandsgleichung des MIT-Bag Modells $p = \frac{1}{3}\left(e - 4 * B\right)$;, wobei der Parameter B die für das Confinement nötige Bag Konstante ist; verwenden Sie B=0.000152869496944 (entspricht ungefähr $B^{1/4} =$ 194 [MeV]). Geben Sie desweiteren auch dem maximalen Radius R_{max} des Quarksternmodells an.

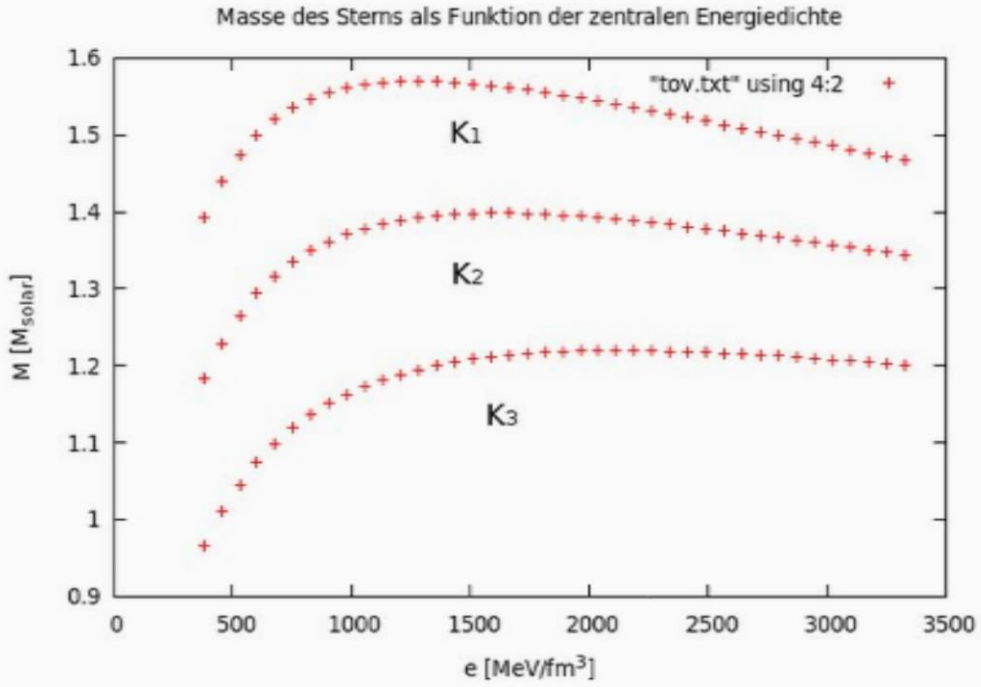


M_{max} = , $R(M_{max})$ = , R_{max} =

Antwort einreichen Versuche 0/20

Aufgaben zum 2.Teil der Vorlesung siehe E-Learning „Lon Capa“

Die maximale Masse M_{max} eines Neutronensterns sei gegeben, und die zugrunde liegende Zustandsgleichung der Neutronensternmaterie sei durch folgenden polytropen Ansatz $p = K * e^{\gamma}$ bestimmt, wobei $\gamma = 5/3$ und $K =$ eine noch zu bestimmende unbekannte Konstante ist. Bei Variation von K ändert sich das gesamte Masse-Radius, bzw. Masse-zentrale Energiedichte Profil einer Sequenz von Sternen und der Wert der maximale Masse M_{max} verschiebt sich (siehe nebenstehende Abbildung). Berechnen Sie unter Verwendung des C++ Programms aus Teil II der Vorlesung den Wert der Konstanten K in $\left[\mathrm{km}^{4/3}\right]$ und geben Sie den zugehörigen Radius des maximalen Massen Sterns ($R_{M_{max}}$) an. Der Wert der maximalen Masse beträgt $M_{max} = 1.735147\ \left[\mathrm{M}_{\odot}\right]$.



$K =$, $R_{M_{max}} =$

Antwort einreichen Versuche 0/20