Parallel Programming

Lecture 12

Lecture course: Computational methods in Meso-Bio-Nano Science

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

Contents

Introduction

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 .

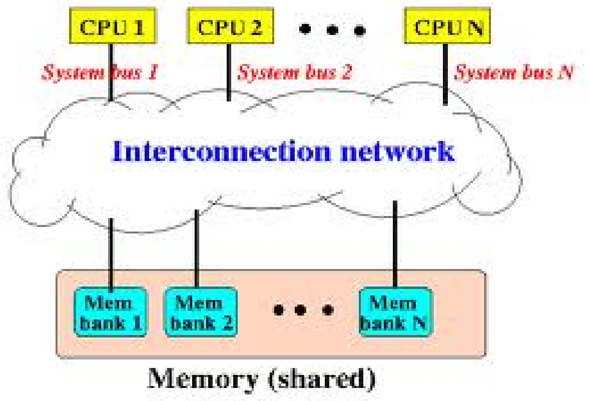
Parallel computers

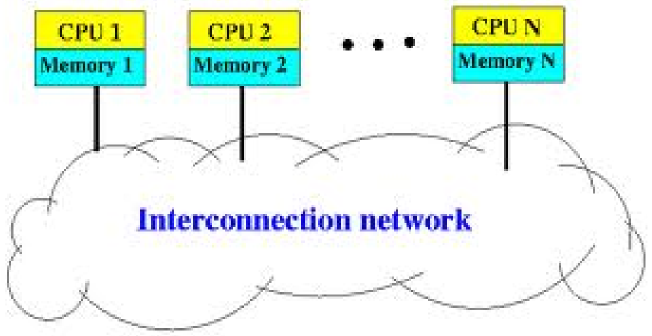
Parallel computer architectures:

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:

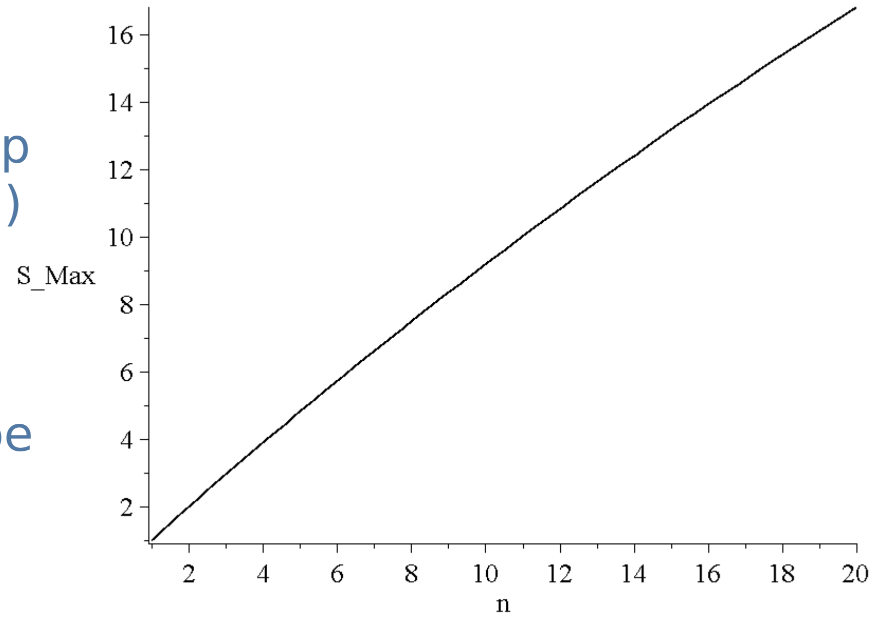
T(n): Time needed to run the programm on n processing elements (e.g. CPU's, computer nodes).

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

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.

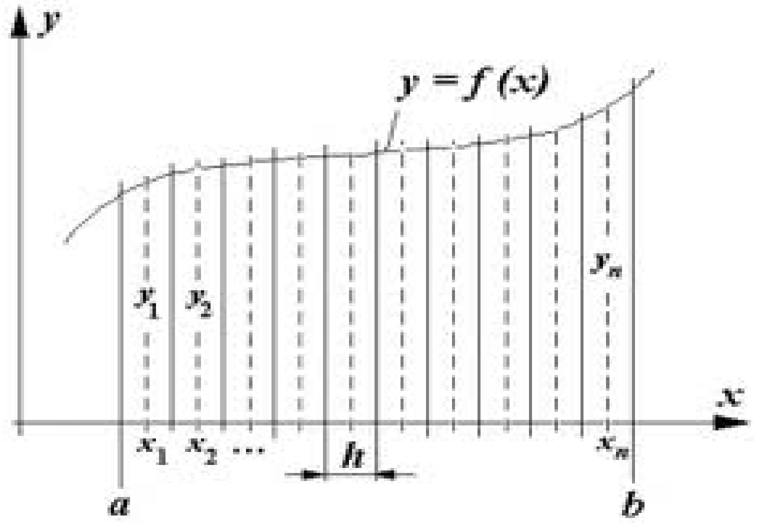
OpenMP has the advantage that common **sequential codes can easily be changed** by simply adding some OpenMP directives. Another feature of OpenMP is that the program runs also properly (but then sequentially, using only one thread) even if the compiler does not know OpenMP.

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).

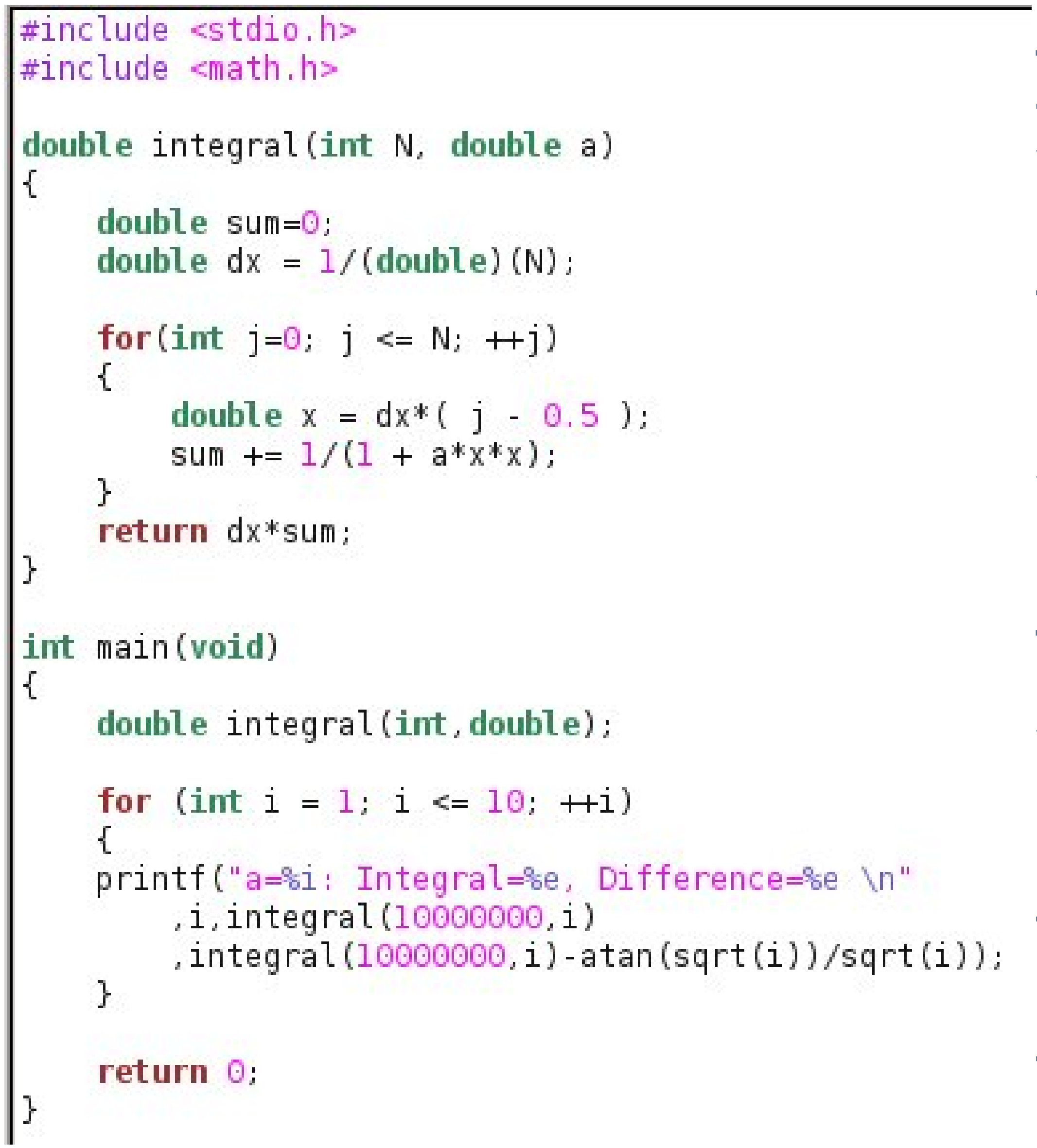
$$\int_0^1 \frac{1}{1+a\,x^2}\,\mathrm{d}x = \frac{\arctan(\sqrt{a})}{\sqrt{a}}$$

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



```
#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;
}
```

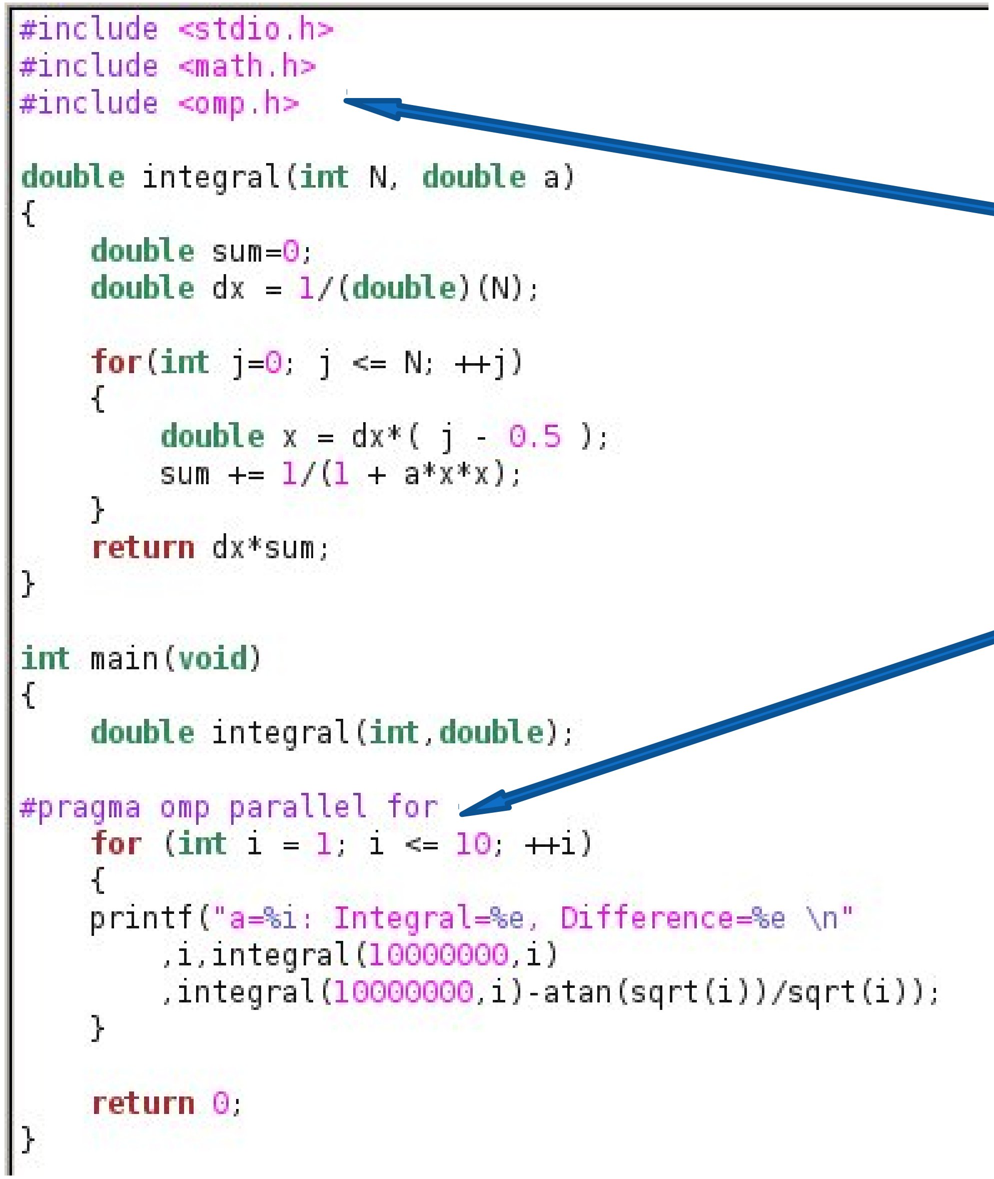
The integral is defined as a function which depends on two variables (N and a). 'N' is the number of integration points (integration segments) and 'a' is the parameter defined within the example. With the use of a 'for-loop', the total area of the N-rectangles are summed up. The value of the integral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

FIAS Frankfurt Institute for Advanced Studies

Parallel Code No.1

MesoBioNano Science



```
#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;
}
```

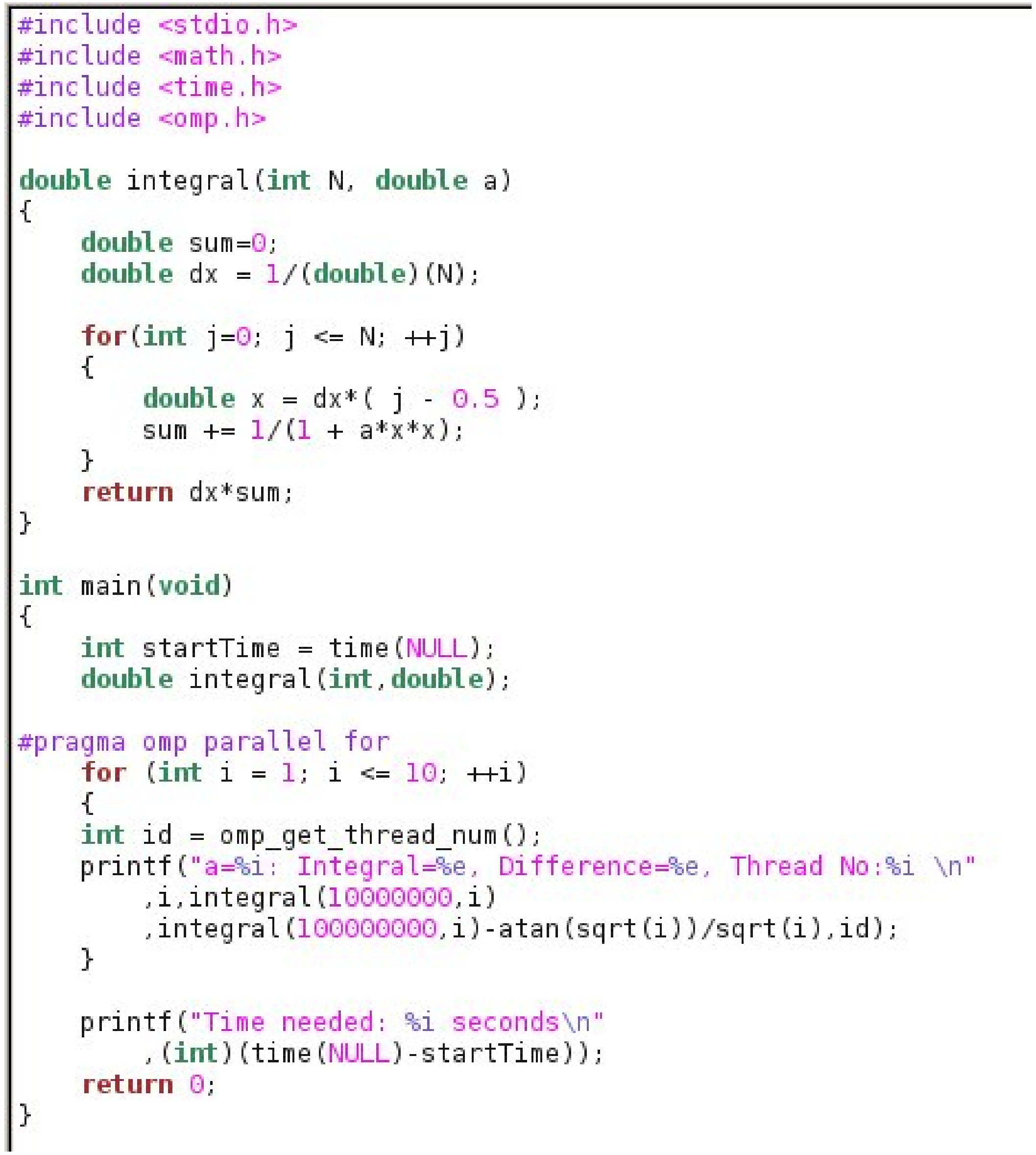
To parallize the code with OpenMP, only two minor changes are necessary:

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;
}
```

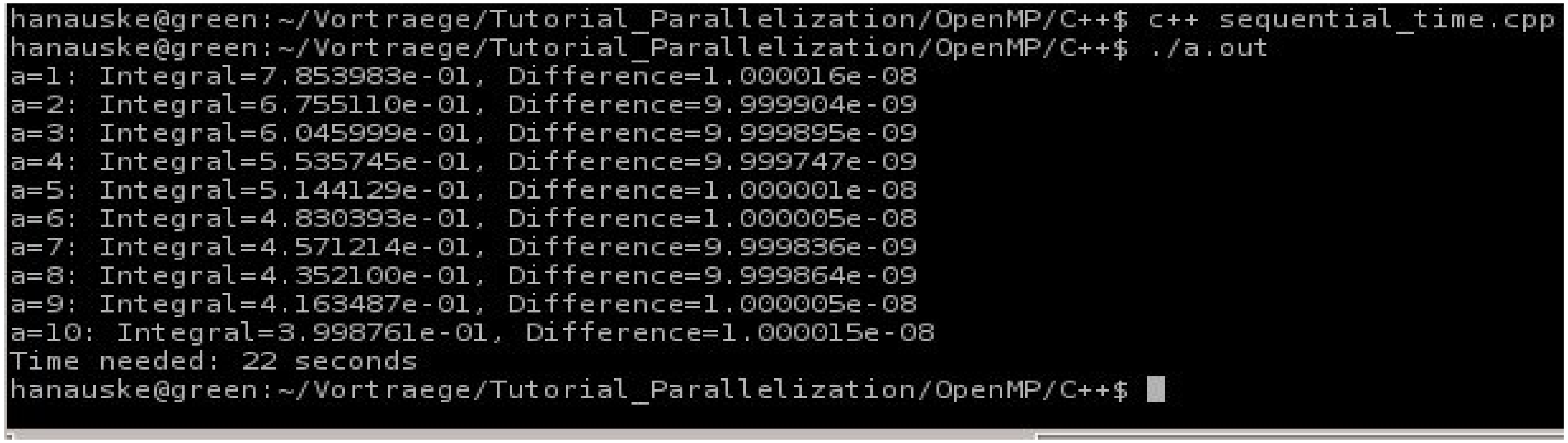
In respect to the ongoing calculation, this version of the parallized code does not differ at all from the previous one. Nevertheless two changes have been made:

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.

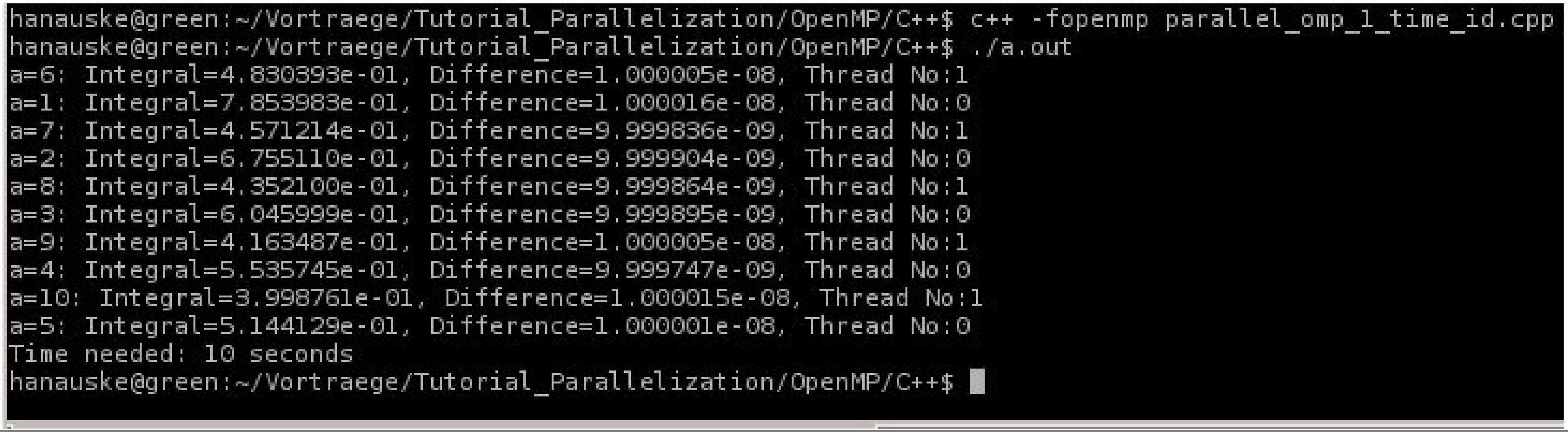
Running the code

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++$
```

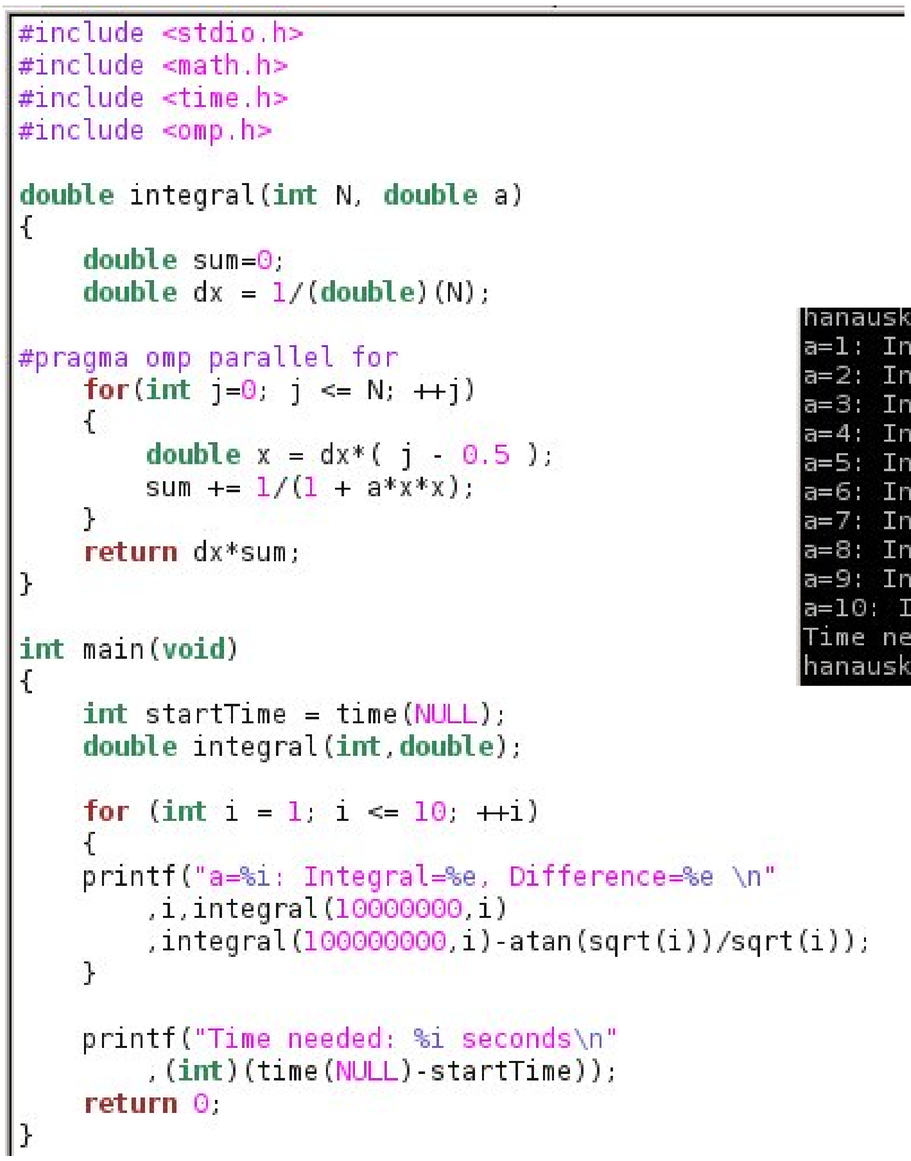
The parallel version (No.1a) has been created using c++ with the option '-fopenmp'. The program was executed on a system with two CPU's.



```
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++$
```

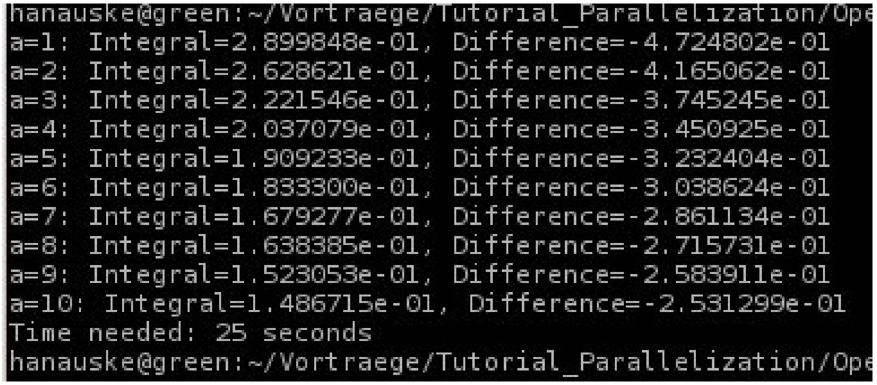
Parallel Code No.2

(wrong!)



```
#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.



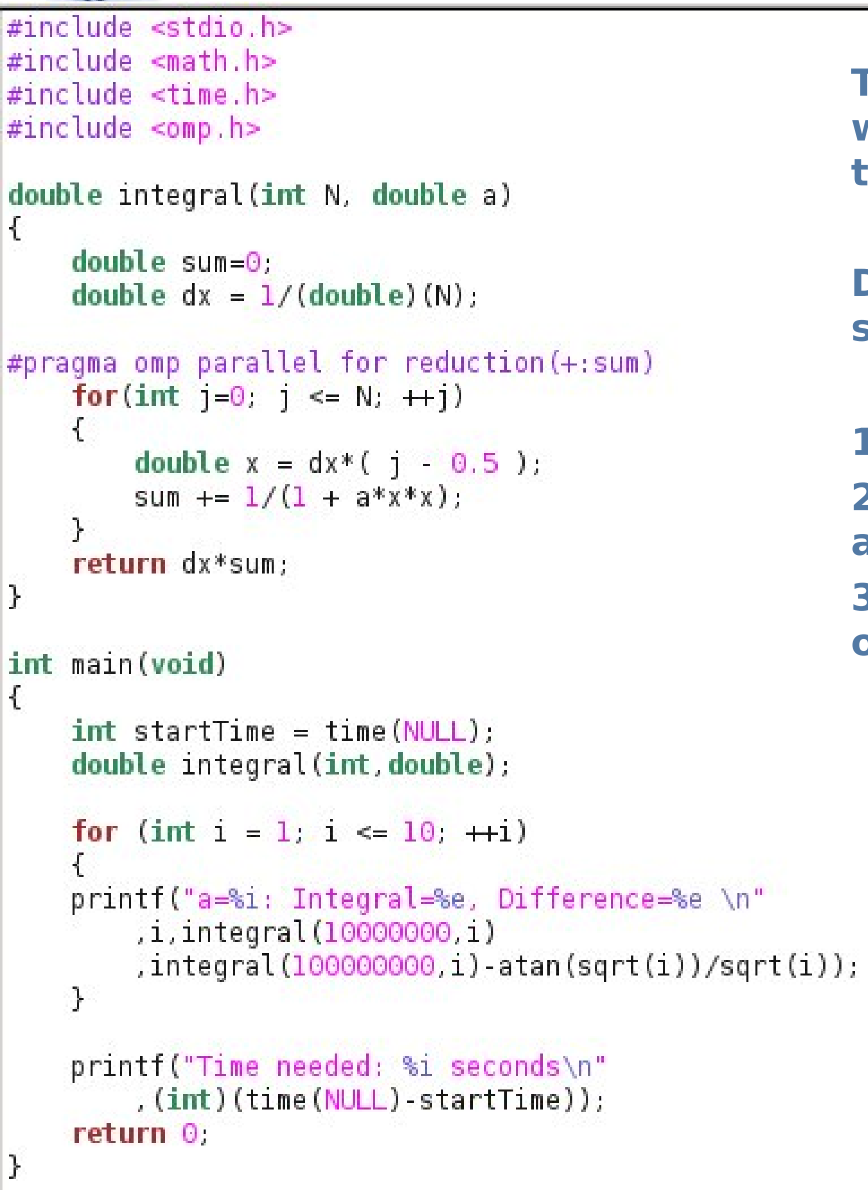
```
hanauske@green:~/Vortraege/Tutorial_Parallelization/Ope
a=1: Integral=2.899848e-01, Difference=-4.724802e-01
a=2: Integral=2.628621e-01, Difference=-4.165062e-01
a=3: Integral=2.221546e-01, Difference=-3.745245e-01
a=4: Integral=2.037079e-01, Difference=-3.450925e-01
a=5: Integral=1.909233e-01, Difference=-3.232404e-01
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



```
#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'.

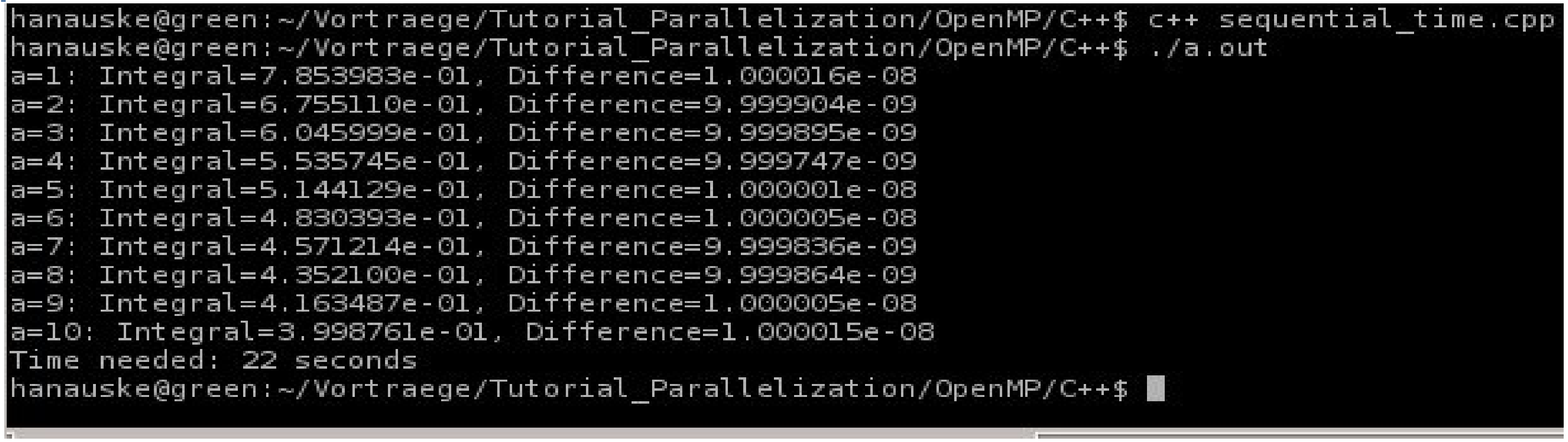
When several threads are created inside the loop, it is possible that while one thread (A) is at stage 2), another thread (B) begins at stage 1). If A writes its new value at stage 3), B is at stage 2). When B finally writes its new value at stage 3), the integration increment of A is lost. This leads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

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

reduction(+:sum)

Running the code

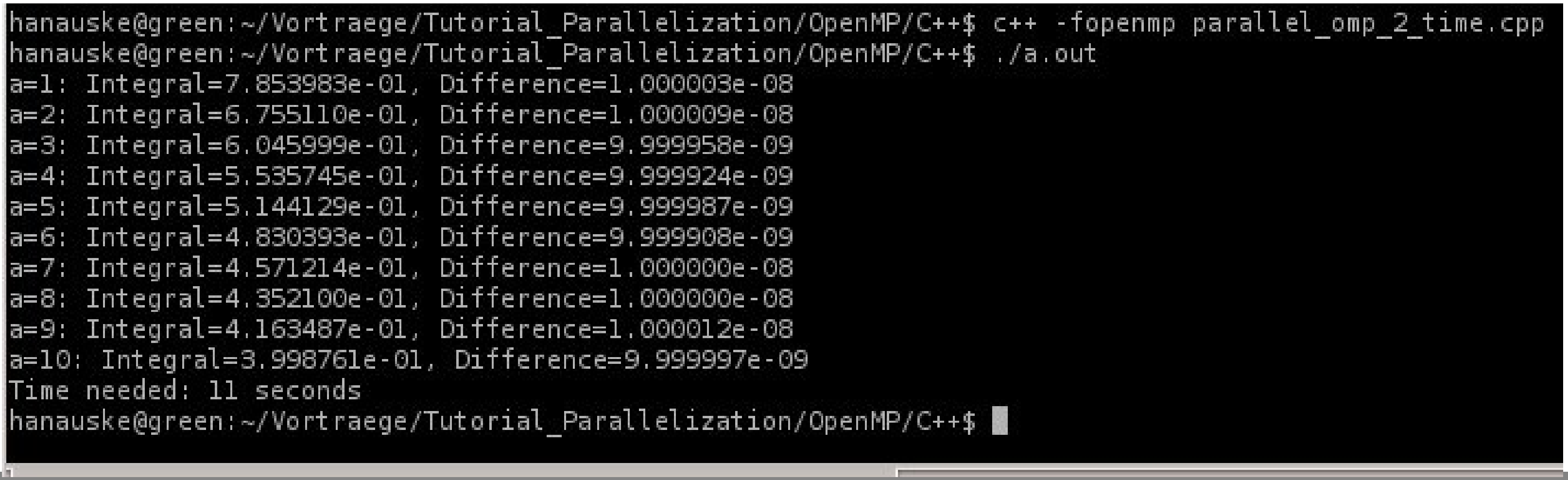
Sequential version:



```
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.

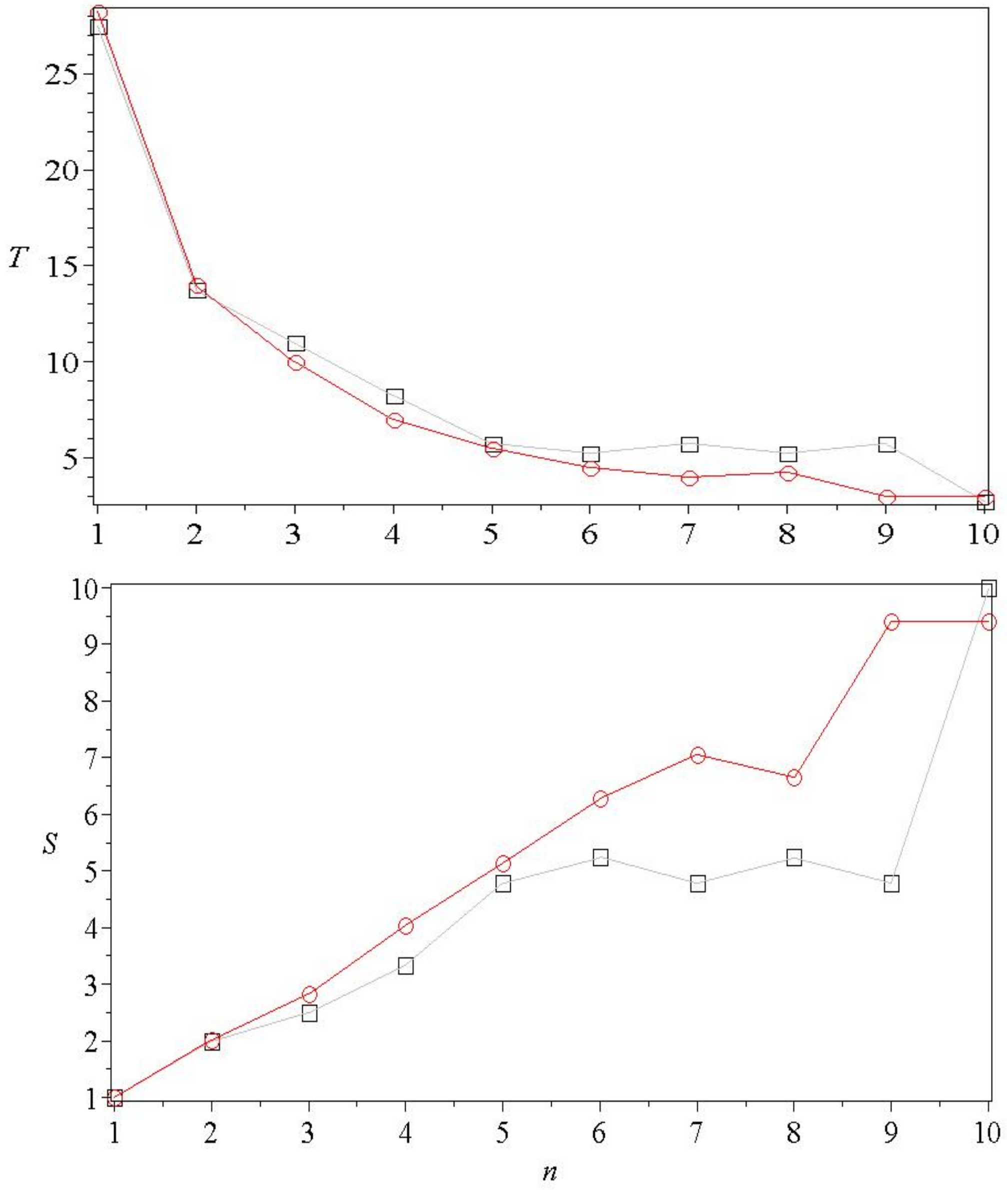


```
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ c++ -fopenmp parallel_omp_2_time.cpp
hanauske@green:~/Vortraege/Tutorial_Parallelization/OpenMP/C++$ ./a.out
a=1: Integral=7.853983e-01, Difference=1.000003e-08
a=2: Integral=6.755110e-01, Difference=1.000009e-08
a=3: Integral=6.045999e-01, 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

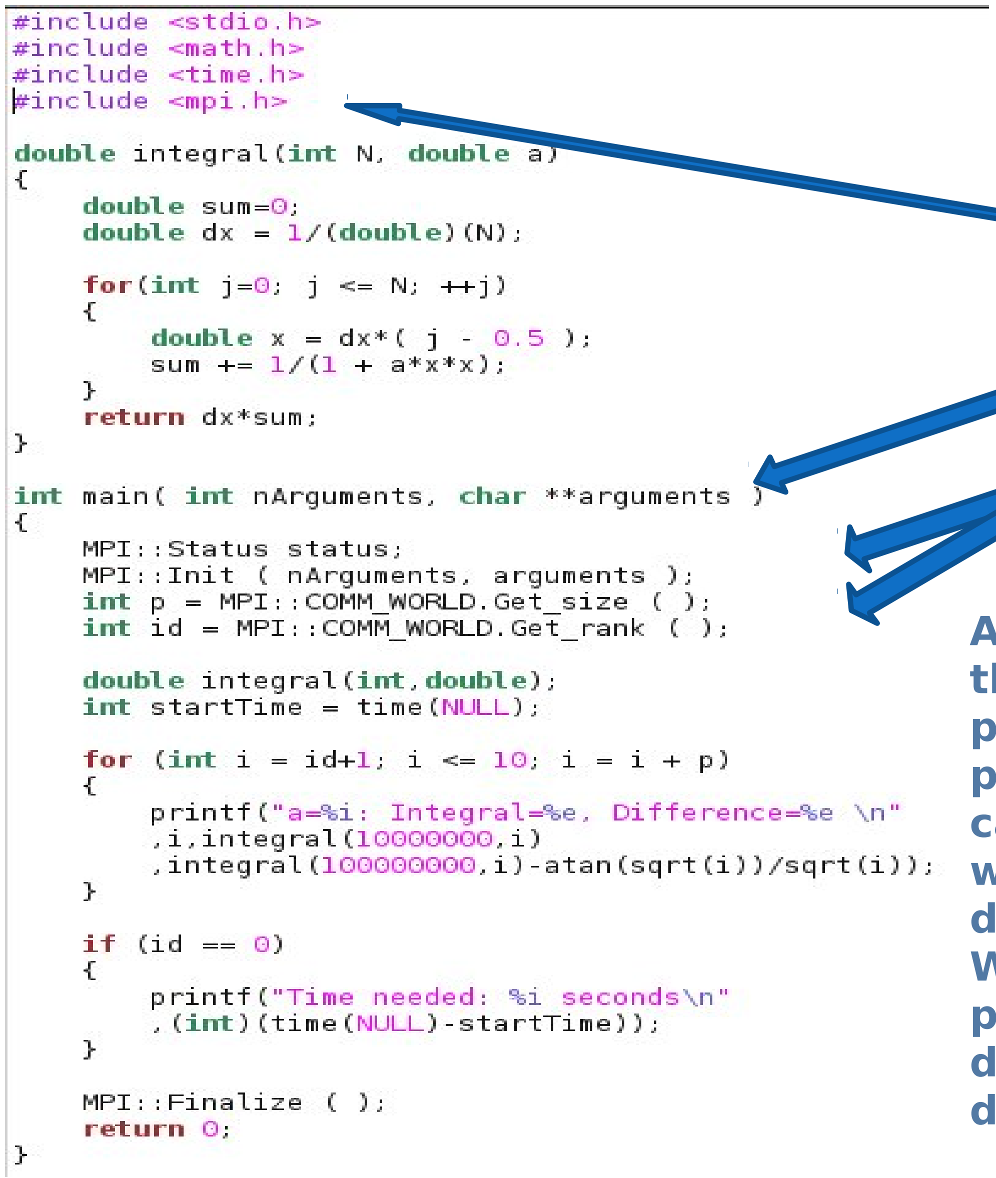
MPI

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

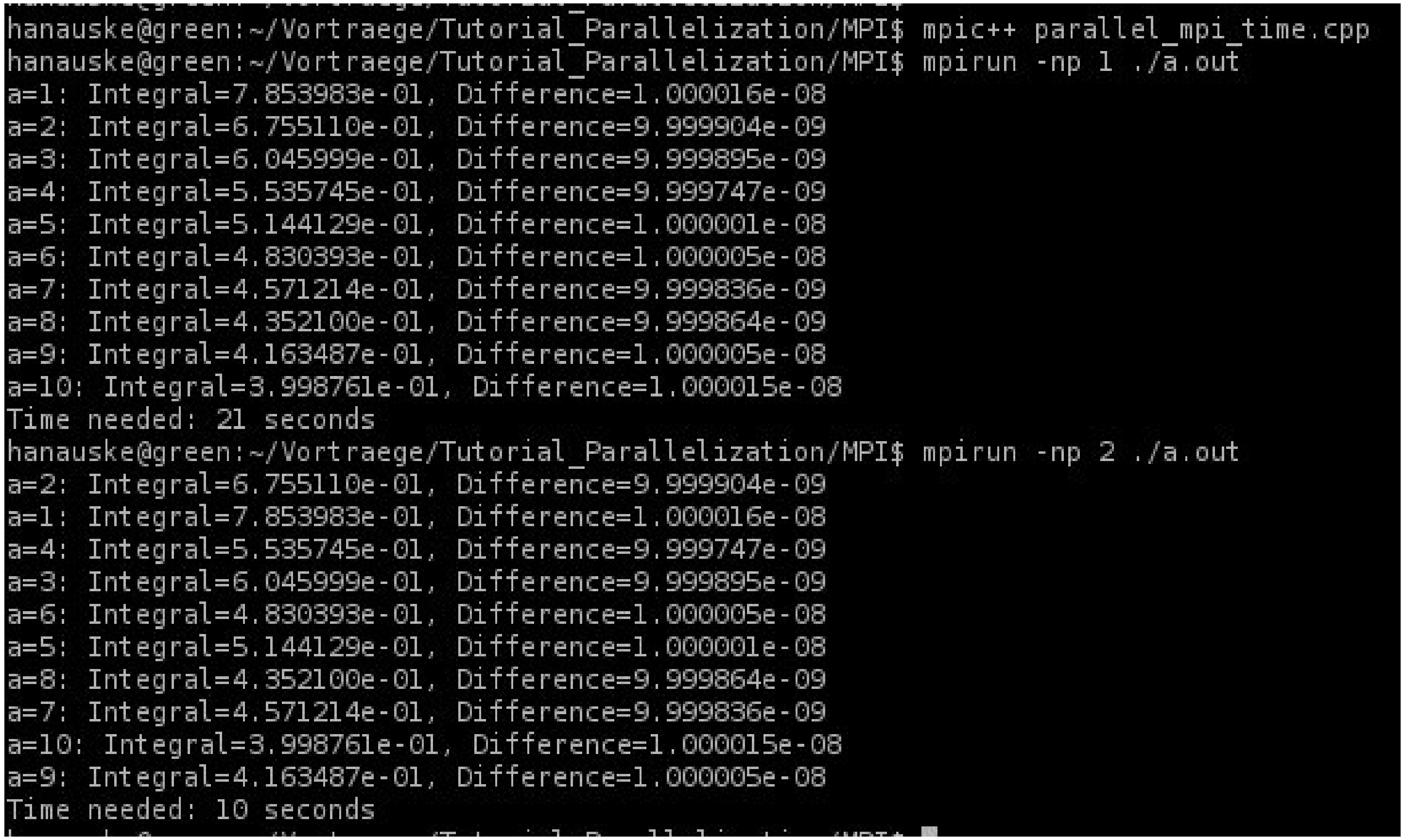
At the beginning of the execution of the program a specified number of processes (p) are created. Each process has its own id-number and it can be executed on different nodes within a computer cluster or on different processors of one node. Within this version of the parallel program the loop which goes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

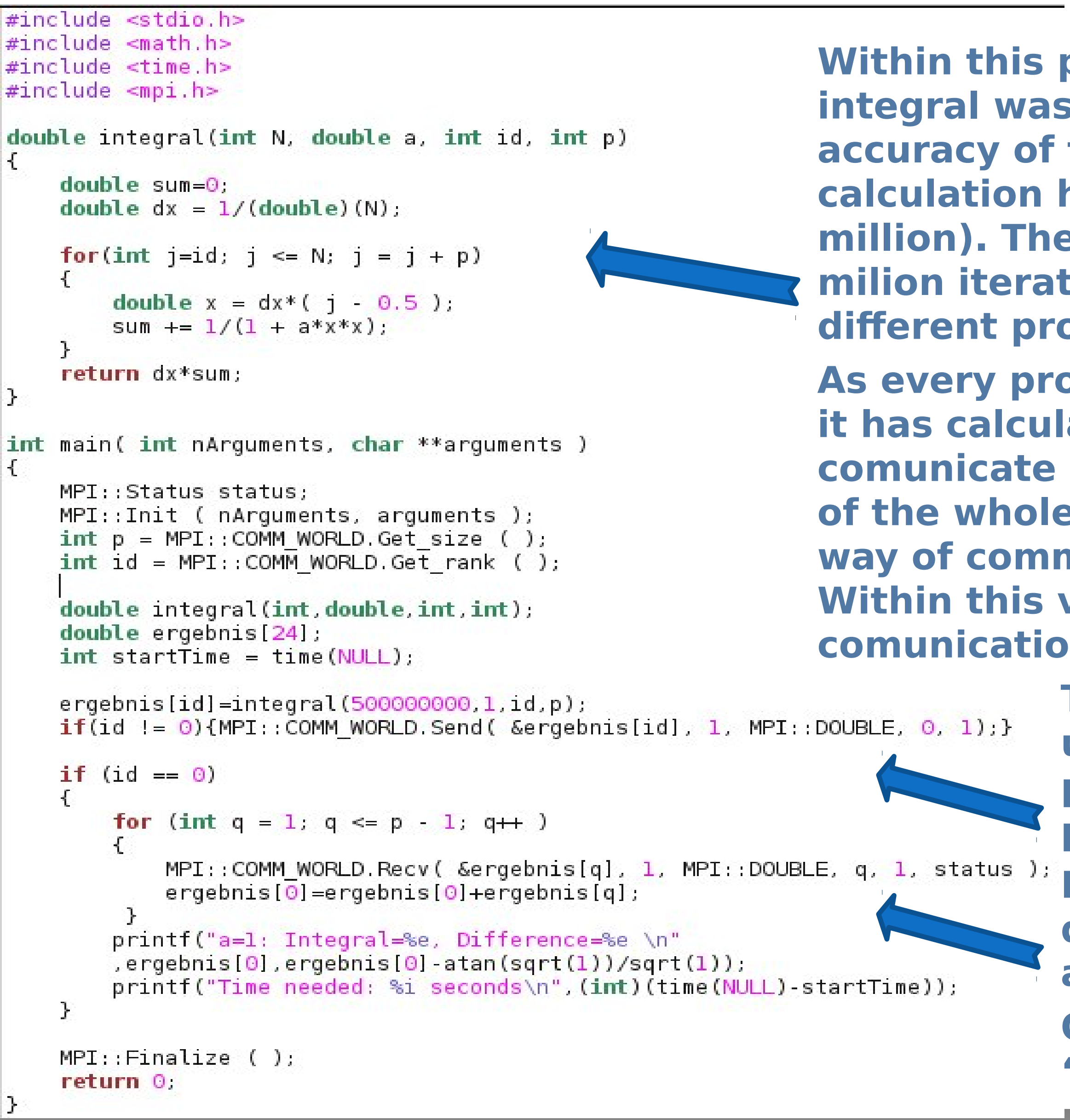
The first run on the right hand side was performed by only using one process (sequential version).

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.

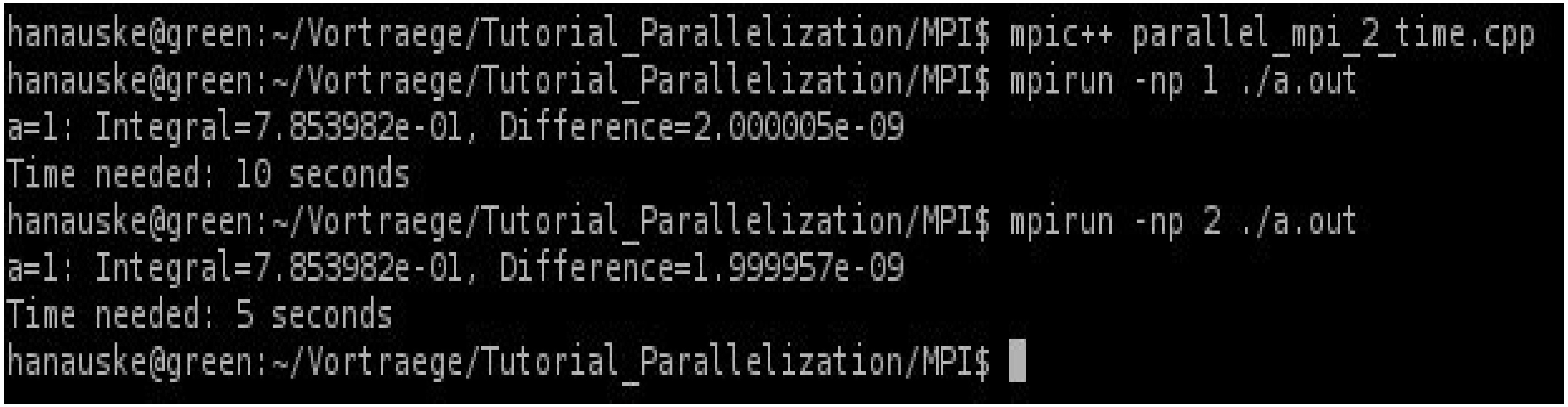
One can use collective operation "MPI_Reduce" for this porpouse.

FIAS Frankfurt Institute for Advanced Studies

Running the code No.2

MesoBioNano Science

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/

Parallel Programming

Thank you for your attention

Lecture 12

Lecture course: Computational methods in Meso-Bio-Nano Science

GOETHE UNIVERSITÄT FRANKFURT AM MAIN