Introduction to MPI by examples

Várady, Géza
Zaválnij, Bogdán

Introduction to MPI by examples

írta Várady, Géza és Zaválnij, Bogdán

Publication date 2015

Tartalom

Introduction to MPI by examples

1. 1 Introduction

In this book we deal with the paradigm of message passing, the standard of our days: MPI. In high performance computing the programming languages are the still used Fortran, the C and the C++ - and these are the languages, which can be used for MPI programming. Our programs will be in C++, as it is the most versatile programming language of the three but we try to constrain ourselves to the minimum of the language possibilities, to be more useful for those who would like to use MPI in C. We will also use the C flavour of MPI, although it has a C++ notation, because we feel that this is more useful. The C++ program can use the C notation freely, while it is not true the other way round.

As on most supercomputers there is some Linux or Unix system used we will present and explain our program in Linux environment and OpenMPI - and present test-runs in different supercomputing environments. Actually MPI is not really supported under windows, so for those who use windows only systems we propose to try out a Linux if not other way than in virtualized environment. One can also use Cygwin for which OpenMPI is available.

1.1. 1.1 Who do we recommend the chapters

The book can be used for courses relating to parallelization at different level of education. Chapter 1-4, 14 and some of Chapter 5 are recommended at BSc level courses. Part II includes practical examples, where different levels of programming and engineering skills are required. Most of these chapters are meant for MSc level courses (E.g. the Raytracing program or the FFT and Graph coloring) or even could be used as part of a PhD semester. For this latter, the project works are strongly suggested, which give new problems and new ideas to the implemented examples. All basic, moderate and independent examples include examples from the field of engineering and science. Although the book makes references, it could be a good starter material by itself as well.

1.2. 1.2 The need for parallel computation

There is an increasing demand for computational power. One core could not manage this need for over a decade. The speed of one core is limited. The computers of today reached a maximum frequency of 3-5GHz, which is quite flat in the last 10 years. Intel introduced its 3.8GHz Pentium4 processor exactly 10 years ago. http://ark.intel.com/products/27475/Intel-Pentium-4-Processor-570J-supporting-HT-Technology-1M-Cache-3_80-GHz-800-MHz-FSB

There is a need of parallel computing, for multiprocessor systems. Later we used multi core processors and distributed systems. Today, we use literally millions of computing cores (*www.top500.org*). We can see, that we can have exponentially increasing computational capacities.

Our modern problems are: Heat dissipation and energy cost of supercomputers. The 3-5 years cost of energy supersedes the cost of the system itself.

We have also the problem of too many cores. Many algorithms do not scale properly over hundred thousands of cores.

1.3. 1.3 Parallel architectures

Today's computer architectures vary in many ways. After we have seen the urge of computer power for large computations, we shall show the classification of nowadays' computers.

The PCs, the standalone computers at home or at the office are multi-core machines now. They have one CPU, but the CPU has more cores, typically 2 or 4, but there are processors with 10-12 cores, as well.

The next scale are bigger computers - these are the servers, usually multiprocessor systems. This later means they have more processors, usually 2 or 4. (Obviously these processors are multi-core, so such systems can have even 48 cores present.)

For HPC, High Performance Computing much much bigger computers or computer systems are used. These provide thousands or millions of cores to their users. The SMP (Symmetric Multiprocessor) systems are shared memory systems, where the programs can access the whole memory. These systems nowadays are so called ccNUMA systems, which means that only part of the memory is close to the processor, while other - bigger - part of the memory is further away. This means more memory latency which is balanced by the usage of cache, that is why the system is cc, cache coherent. The programming of such systems is usually made by openMP programs, but bigger systems can run MPI as well.

The biggest supercomputers of our time are distributed systems or clustered computers. They consist of separate computers connected by a fast interconnect system, These are programed with MPI language extensions. In this book the reader is being introduced to the MPI programming with C++.

Other architectures of today's computers are video cards, which can be programed for different purposes and even used in supercomputers. This paradigm is called the General Programming GPU, the GPGPU, and programed with languages of for example CUDA or OpenCL.

1.4. 1.4 Problem of decomposition

When we need to construct a parallel algorithm we usually want to decompose the problem into sub-problems. For some cases this can be done straightforward, as the main problem itself consists of independent tasks or the algorithm deals with a set of data which can be split and processed independently. While there are still some interesting questions of submitting the split problems to different processes - an interesting one of these questions is the example of Mandelbrot set parallelization in chapter 12 -, this case counts as a good example for parallelization.

But this is not always possible. In case of quite a few problems we cannot divide the whole problem into independent sub-problems, as they would not be independent. Many discrete algorithms fall into this category.

With some luck we can still deal with this case if we assign sub-problems to parallel tasks, and combine the solution of sub-problems by a sequential thread. One clear example can be the problem of sorting. We can assign a sub-domain of the elements to the parallel threads for sorting, and at the end combine these already sorted subsets into the final solution. This can be done by merging, so this scheme would be the parallel merge sort. By other means we can first divide the whole set into subsets such as the value of elements in each set is greater than in the previous set. In other words divide roughly the elements by value into baskets of different magnitude. After sorting these elements parallelly and routing back the solution to the master thread we are done, as each subset follows the previous one, and is sorted by itself. This scheme could be called the parallel Quicksort. We will see, that dividing the whole set into subsets raises a special problem, so a tuned version of subdivision is needed. By this subdivision is called this algorithm Samplesort.

1.5. 1.5 Problem of parallelization

When we write a parallel program we obviously would like to achive faster computation. But apart from being fast it is an interesting question how fast it is. One would like to measure the goodness of the parallelization. Obviously, for different purposes different measures can be done. For some cases even the slightes reduction in running time may be important as a problem may be of a kind that it must be solved in some time limit. Others may look at the question from economical point of view and compare the income from faster computation with the invested money (and perhaps time). There cannot be an overall perfect measurement, but still there is one which is accepted the widest. This is called the speed-up, and it is calculated the following way. For a given problem we measure the running time of the best known sequential program and the running time of our parallel program. The ratio of them will give us the speed-up number, which by this definition will be dependent of processors or computers we use. One would like to develop and use a parallel program, that will achive a speed-up of n with n processors, or in other words with twice as many processors half the running time.

In most cases, because the problem cannot be divided into independent sub-problems, we should find an alternative algorithm, which is more complex than the original one. This algorithm would run on one thread slower than the original one. So adding more processors we speed up the modified algorithm, which means we are far from reaching speed-up of number of processors compared to the original algorithm, namely we are unable to reach linear speed-up. Still, there can be a good usage of such algorithms, as even with sub-linear speed-up and with the aim of many computers (or a bigger cluster or supercomputer) we can solve much bigger problems than with the original sequential program. One may say, that gaining speed-up of 1.9 with twice as many processors is a satisfying result.

1.5.1. 1.5.1 Amdahl's law and Gustavson's law

At this point we need to mention two historical notes on the problem of speed-up and its limits. These notes called the Amdahl's law and the Gustavson's law.

The law by Gene Amdahl was formed in the late 60's, and states, that every parallel program has a limit in speed-up independently of the number of processors used. This phenomena is caused by the fact, that every program has a part which cannot be done in parallel. This usually includes the start-up of the program and reading the initial values; the starting of parallelization; the collecting the data by the main thread; and the ending of parallelization. These parts cannot be done in parallel, so no matter how many processors we use, and speed-up the parallelizable part, this part will run the same time. Say 1% of the program is like this, and we use infinite number of processors to speed up the remaining 99%, which will finish so immediately. The running time of the whole program will be $1/100$-th of the original time, the running time of the non-parallelizable part, so we gain a total speed-up of 100. No bigger speed-up is possible in this case.

We also should note, that in real problems we need communication between the different threads or processes. The 'cost' of this communication is not independent of the size of the cluster or the supercomputer. Namely, if we use more and more processors the communication will take more and more time. Also, the load balancing between different processors may be unequal causing longer execution time as some processors may be unoccupied for some time. So in real world the problem is even more pronounced, as adding more and more processors the program cannot even reach the theoretical speed-up, but will produce slower running time because of the communication taking more and more time and unbalanced problems becoming more and more unbalanced. We will show real world running time examples in our chapter 12 about the Mandelbrot set parallelization.

Although not denying the law of Amdahl John L. Gustafson and Edwin H. Barsis noted that the fraction which cannot be parallelized becoming less and less if we increase the size of the problem itself. Because the main interest is always solving problems and not finding theoretical speed-up limits, this means that with bigger computers and computer clusters we can assign them and solve bigger problems because we can achieve greater speed-up in the end. With the case of communication overhead and unbalanced decomposition we can assume mostly the same.

1.5.2. 1.5.2 Superlinear speed-up

The theoretical expected speed-up of a parallel program is the number of processors we use. But there can be an exception, where we gain more speed than the added number of processors. This phenomenon is called the super-linear speed-up, which occurs rarely, but not extremely rarely. There may be two causes for such speed-up. One is the accumulation of the cache or memory in the system. If the whole problem cannot fit into memory (or cache) it slows down. But using so many processors that the chunk of the data can fit into the memory (or cache) the problem will speed up extremely. So the speed-up at this very point will be super-linear. The reader may see an example of such behaviour in the chapter 7 about finding the shortest paths in a graph.

Other possible occurrence of super-linear speed-up may be observed in backtracking algorithms, where the result of one branch may give information to cut other branches.

2. 2 Fundamental computational problems

2.1. 2.1 Fundamental hard to solve problems – computational problems

We can divide the set of problems in two main groups. Problems which cannot be solved by computers and problems which can. The first group includes typical problems involving feelings, subjective parameters and all sorts of matters impossible to model or describe. The second group includes problems which can be described in a way that computers can handle. This means typically, that the informal description of the problem can be converted to an algorithm. An algorithm is a finite, step-by-step set of instructions what represents a function. It has well defined inputs and outputs. If a problem can be solved by such algorithms, we say, it is solvable by a computer. By using algorithms, it is a basic expectation that the algorithm not only gives a solution, but preferably, gives it fast. Since speed depends on the amount of inputs, it's better to classify algorithms according to their response to input size. This classification is noted by Big-O (Landau's symbol, part of the Landau notation) and symbolizes the order of the function. For given functions $f(x)$ and $g(x)$, we say that $f(x) = O(g(x))$ if and only if there exists a positive real number M and a real number x_0 , where $|f(x)| \leq M|g(x)|$ for all $x > x_0$. Hence, after a given x_0 point, $f(x)$ will not grow faster than $g(x)$. Let c be a constant. If, for example, $g(x)$ is cx , the order is linear. If $g(x)$ is x^c , the order is polynomial. This latter order is confine. Until this order, step numbers of algorithms are acceptable in general. If $g(x)$ is c^x , required steps for an algorithm increase fast with inputs, and it is a very long time to wait until a solution. Thus, we say that exponential algorithms are bad algorithms.

Of course, the input size is also an important factor.

According to this, we have to cope with two factors:

1. Enormous amount of data ("big data" problem),

2. Exponential time solution for the problem.

Problems connected to these or to one of the circumstances are hard to solve. The big data problem can be handled by linear scaling parallelism. The more data, the more computing power. The exponential time solution problem could be handled by exponential scaling parallelism what is, in practice, unmanageable.

2.2. 2.2 Engineering, mathematical, economical and physics computation

Computational engineering problems are most commonly simulation tasks. Aerospace engineering uses simulation in spacecraft and aircraft design, where forces between components and environment,the movement of air (aerodynamics) and the mechanism of the engine are examined. Extending this to other transportation crafts, we talk about transportation design.

Building and structure design deals with similar problems, but use different materials, forces and has different objectives. Buildings are measured according to being made high and beautiful or made robust and safe. In case of structures like bridges, tunnels and other ossatures, again, different qualities are praised.

Most of the problems are modelling problems, where simulation and visualization play the main role. We want to see whether a structure is capable of holding some weight, or capable of withstanding impacts like wind or earthquakes. We have to make a lot of calculations with lots of parameters, we have to visualize the results, we might want to change different parameters to see whether it has an effect or not. We can wait for the results for a while, but the most effective design is real-time, where we get an instant feedback to our actions.

Calculating the physics of the problems above involves numerical methods, like solving non-linear equations, curve fitting, interpolation, differentiation, integration and other computationally intensive tasks.

Mathematical problems in their complete form are hard to handle. It is common to reduce the problem to a sequence of linear and less complex problems. Numerical methods are also called approximate methods. The main simplification in numerical methods is that one searches for a plausible approximate solution in reasonable

time. The process often involves iteration, where the number of iterations, and thus the time it takes, depends on the required precision. Let us examine the simple problem of finding the roots of a polynomial equation $\sin(x) - x_2 = 0$.

A graphical solution would be to plot the graph and look where the function crosses the x -axle. This is an approximation as well, since the crossing is not exactly at a given whole number.

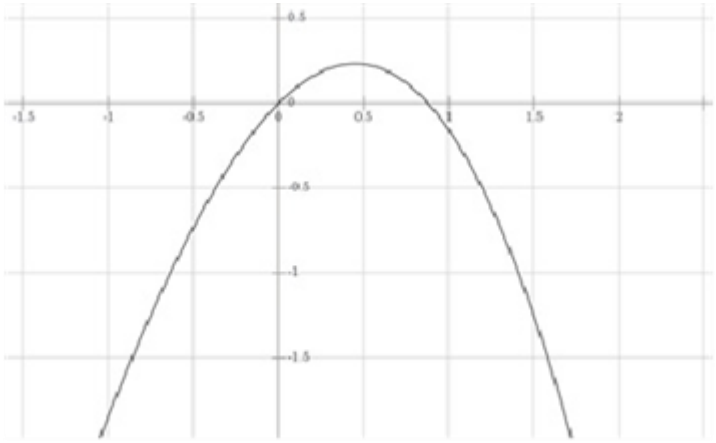

Figure 1: Example equation: $\sin(x) - x^2 = 0$ (fooplot.com)

A simple example of numerical solution would be to search within a range. The possible solution should be within this. It involves some analytical pre-processing, since we have to guess the initial and end values between which a simple iteration should run. Let's take a look at the figure and say, we are searching between 0 and 1. The 0 is a trivial solution for the above equation. A simple, albeit ineffective iterative solution would be to go through this interval with a given resolution. Let's take the resolution of 0.1. Since zero is a trivial solution, we can start at 0.1. Let's substitute x with 0.1 and check whether the equation is sufficed. If not, let's go further and check 0.2, and so on. If the result changes the sign, we run over the solution, so let's go back one step and refine the resolution, hence, go ahead with smaller steps. The resolution could be then e.g. 0.01. We could do this for a very long time, theoretically forever, and we would approximate the solution slowly. However, there could be an approximate solution, where a given precision is enough for our purpose. This is a key fact among numerical methods – most of the time, we are searching for a close enough solution. This saves us time and effort. Still we do have to define "close enough". One definition could be given by the maximum number of steps of the iteration. This could prevent us from staying in an infinite loop. The other definition would be the order of error, thus, how close the real solution is. But how could we define the latter, if we don't know the solution itself? One trick could be to monitor the order of change between iteration steps. If the change is under a predefined value, we can consider the actual solution "close enough".

There are several methods to increase the speed of approximation.

One method is The Bisection Method (binary search method). If our polynomial function is continuous and the signs of the value in the starting point and in the ending point of our interval are the opposites, we can use the method of bisection. The basic idea is simple: we divide the interval in two and choose one half to investigate further. We choose the part with opposite starting and end points and divide it into two again. The dividing goes on until we narrow our interval where the starting point value and end value of the function are closer than a given value.

One further method is The Newton Method (Newton-Raphson Method or Newton-Fourier Method). If the above conditions are met, an other method could lead to a faster solution. The idea of the Newton Method is to improve an arbitrary x_n solution in the next step, x_{n+1} by setting the value of x_{n+1} with the help of the derivative function of $f(x_n)$. This method gives a fast approximation if started not too far away from the solution. It is a good idea to use the more robust method of bisection at the beginning and to switch to the Newton method after a certain precision for faster convergence.

The method benefits from the derivative of a function being at a certain point equal to the tangent of the function. Based on this, $f'(x_n) = f(x_n)/(x_n - x_{n+1})$. This can be rearranged to the form:

$x_{n+1} = x_n - f(x_n)/f'(x_n)$. By calculating the new intersection point of the tangent line and x -axis, x_{n+1} we can have a new $f()$ value to calculate a new tangent line. Iterating this process until a given error value is achieved leads to an acceptable solution.

One other common task in applied mathematics is to calculate the integral of a definite function. This can be done in some cases analytically, but there are several cases where the analytical solution is not possible or would take too much time to find.

An obvious solution would be to do it numerically, in other words, using a method where we approximate the integral.

The basic problem can be formulated as follows. We have to find the area covered by a function at a given interval. One method is to split the interval into several sub-intervals and substitute the function in this sub-intervals with a constant. This constant should be the center value of the function of this sub-interval. With this method, one can approximate the area of function with rectangular columns. The approximate integral value will be the sum of the areas of these rectangles. The more narrow the sub-intervals are, the more precise the approximation will be. This simple method is called the rectangle rule, also considered the basic quadrature rule.

A more complex method would be to use trapezoids instead of rectangles (trapezoidal rule).

For these methods, we can refine the sub-intervals on the cost of required computational power.

A more sophisticated approximation is to use interpolation of higher degree polynomials. This yields the Newton-Cotes formulas and rises some problems of oscillation. One solution is to stay at quadrature rules with fine sub-intervals.

2.3. 2.3 Complex models

The main reason for using parallel systems is to share big amount of work among processing units and let them work separately. With this method, we can save time, hence, the work will be done faster. Since the size and - consequently - the execution time of the work units are decrease as we increase the number of processors, parallel computation gets more and more cost effective. Some trivial uses of parallel systems deal with a big amount of independent data, where the share of work is straightforward. Depending on the task, we can partition the data rather easily, although we have to pay attention to possible asymmetric load (e.g. searching for prime numbers in a given range). Parallel computing is a handy tool for modelling and solving complex problems in engineering and science, where data is dependent and several conditions are to be tested during work. In nature, there are phenomena which involve several interrelated events. Some examples for this could be the movement and orbits of planets, comets and moons. This latter can be calculated according to Kepler's laws and each element has an effect on all the other elements. In this complex system, the movement of artificial objects like rockets and satellites with actuation is a further step towards complexity, and because of the nature of the problem, workload is getting higher and higher. Further good examples are climate calculations, traffic modelling, electronic structure modelling, plasma dynamics.

2.4. 2.4 Real-time computing

By speeding up, we can achieve calculations on data in less time. This time can be so short that the reaction could seem prompt. One impact of the speed-up is to be able to create real-time systems. Real-time, by definition is not exactly prompt, but reactions on actions are done in a preliminary guaranteed time period. This has several effects. The system will most likely to be interactive, with no lags and delays, the interface will work and react, the actions we indicate will happen in a short time. This indicates better visualization, "real-time" change of output, by changing input parameters. There are, of course, other advantages of a real-time system. Such systems can control live, real-time procedures, can interfere, moreover, predictions and preventive actions can be taken. Real-time parallelism can occur at operating system or even hardware level.

One standard of real-time operations is the MPI/RT (Message Passing Interface / Real-Time). With this standard, platform independency and better porting potential is supported. The standard adds quality of service (QoS) to the MPI functionality. Many High-Performance (HPC) applications require timing guarantees.

For further information on MPI/RT we suggest reading related papers as referred[Kan1998].

3. 3 The logic of the MPI environment

First, the reader must understand how an MPI program runs, as it is different from the usual programs. There are minor differences between the program itself and a usual C++ program. There is an included mpi.h in the front, and some really few function calls in the program itself, all of them starting with MPI_. (The C++ notation uses name-space MPI:: to differentiate the MPI function calls from others. As C notation can be easily used in any programs, to avoid confusion the MPI version 3.0 declared this notation obsolete.)

The program should be compiled for which a compiler wrapper is used. For C++ one usually use mpic++ or mpicxx wrapper program - depending on the MPI programs installed, which will initiate the compiler that is present in the system: for example the g++ or other compiler like icc or pgc++. Obviously we can use the same switches for the compiler, for instance the -O3 for biggest optimization, or the -o for naming the output runnable program. To give the reader an example a "Hello World" program compilation would be like this:

```
$mpic++ -O3 hello.cpp -o hello
```

After we compiled successfully our program, we need to run it with another wrapper program the mpirun with a switch -np of the number of instances we want the program to be runned. The mpirun does nothing more, than runs the same program in multiple instances. You can try it out with running some usual Linux system programs as date or hostname - the later is quite useful for testing the actual working of the mpirun in case of multiple computers. To give an example of the previous program to be runned:

```
$mpirun -np 8 ./hello
```

With more computers we need to set up ssh communication between them and insted of -np switch use the -hostfile switch to tell the names of the separate computers we would use. In case of a supercomputer where some queueing system is presented, after compiling the program with mpic++ one must use the queue submission system, for which you need to consult with the actual documentation of the system. We give examples of these usage in the Appendix.

If you're new to MPI and want to try it out first without much installation, than usually a few packages whose name start with openmpi for your Linux distribution is sufficient (usually the openmpi and the openmpi-dev), and you can use the -np switch with an integer parameter to tell the mpirun the number of instances the program must run. In this case the same program will run on your local computer in multiple instances, as many of them as you gave the -np switch. Detailed installation description can be found in the Appendix.

3.1. 3.1 The programmers view of MPI

After our program started in multiple instances with the use of the MPI function calls we can get information about our MPI 'world', and complete information exchanges between the program instances. Let us see our first MPI program example, which is the usual version of the famous "Hello World!" program in other languages.

```
//Hello World! program

#include <iostream>
#include <mpi.h>
using namespace std;
int main (int argc, char **argv) {
  int id, nproc;
  // Initialize MPI:
  MPI_Init(&argc, &argv);
  // Get my rank:
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  // Get the total number of processors:
```

```
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  cout<<"Process "<<id<<" of "<<nproc<<": Hello World!"<<endl;
  // Terminate MPI:
  MPI_Finalize();
}
```

One can see a little difference from a tradicional sequential program. Obviously we need the mpi.h header included. Then, the most important functions are the MPI_Init and the MPI_Finalize which denote the start of the cooperating parts and the end of it. As a rule of the thumb, we must begin our program (the main function) with the first, and end it with the second. The arguments of the MPI_Init are usually the pointers to the arguments of the main function, so that all the program instances will know exactly the same starting parameters. The formal definition of this function is:

```
int MPI_Init(
    int *argc,
    char ***argv
    )
```

The MPI_Comm_rank and the MPI_Comm_size functions used to get informations about the running programs. The size is the number of program instances running, and the rank is a non-negative number unique for all the instances from 0 to $size - 1$. The formal definition of these functions are:

```
int MPI_Comm_rank(
    MPI_Comm comm,
        //the MPI communicator
    int *rank
        //the return value of the unique id
    )



int MPI_Comm_size(
    MPI_Comm comm,
        //the MPI communicator
    int *size
        //the return value of the MPI world size
    )
```

The MPI communicator is a unique identifier, which is present in almost all MPI functions. The default communicator we used here is 'created' by the MPI_Init, and denoted by MPI_COMM_WORLD. It allows the MPI program to call the other instances of the program. It is possible to construct different communicators as well for narrowing the set of processes which would like to communicate for special algorithmic use. An examle of this method will be presented for the parallelization of the FFT problem.

These are essential functions to gain information otherwise not presented in the program, as running can be different from case to case, and usually managed by the user starting the program. For example, if the user started the program as mpirun -np 4 ./hello, the size (variable nproc) will be equal to 4, and the variable id will be equal to 0, 1, 2 and 3 through the running program instances. One possible outcome of this program could be:

```
$ mpirun -np 4 ./hello
Process 0 of 4: Hello World!
Process 1 of 4: Hello World!
Process 3 of 4: Hello World!
Process 2 of 4: Hello World!
```

The reader must note, that the sequence is arbitrary (process 3 preceeding 2), and must remember that one can never be assured of the running sequence of the programs. We must prepare our algorithms so that any running sequence can occur, and if we specifically need synchronization, we must put it into our program.

To demonstrate a little more complex program, we show some communications between the running program instances, modifying the previous program:

```
//Hello World! program with minimal communications

#include <iostream>
#include <mpi.h>
using namespace std;
int main (int argc, char **argv) {
  int id, nproc, id_from;
  MPI_Status status;
  // Initialize MPI:
  MPI_Init(&argc, &argv);
  // Get my rank:
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  // Get the total number of processors:
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  if(id != 0){
    // Slave processes sending greetings:
    cout<<"Process id="<<id<<" sending greetings!"<<endl;
    MPI_Send(&id, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
  }
  else{
    // Master process receiving greetings:
    cout<<"Master process (id=0) receving greetings"<<endl;
    for(int i=1;i<nproc;++i){
      MPI_Recv(&id_from, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
        &status);
      cout<<"Greetings from process "<<id_from<<"!"<<endl;
    }
  }
  // Terminate MPI:
  MPI_Finalize();
}
```

Here the program instances are divided to a master (the program which id is equal to 0), and the slaves (all other instances). The division is made through a simple if-else statement, setting apart the program code for the master from the program code for the slaves. The slaves send their greetings to the master, and the master receives the greetings - in this example the id of the sending process, which is stored in the variable id_from. The communication functions are the MPI_Send and the MPI_Recv which we will describe later in detail. The master receives $nproc - 1$ greetings in a for loop, as it does not receive a greeting from himself.

A possible running output would be:

```
$ mpirun -np 4 ./hello2
Master process (id=0) receving greetings
Process id=1 sending greetings!
Process id=2 sending greetings!
Greetings from process 2!
Process id=3 sending greetings!
Greetings from process 3!
Greetings from process 1!
```

Another run may produce a different output:

```
$ mpirun -np 4 ./hello2
Master process (id=0) receving greetings
Greetings from process 3!
Greetings from process 1!
Greetings from process 2!
Process id=1 sending greetings!
Process id=2 sending greetings!
Process id=3 sending greetings!
```

Again you can see the strange sequence of the output. We can only be sure that the sending of the message precedes the receiving of that very message, but even the output before sending may arrive after the output of the receiving. The only certainty is that the line "Master process..." is preceding the lines "Greetings...", because these lines are the output of the master process, and this program on itself is a serial program. So the outputs of one particular program will follow each other as this program would run in the usual sequential way.

Detailed examples of master-slave algorithmization will be presented in the following chapters.

3.2. 3.2 Send and recieve

The MPI_Send and MPI_Recv are the two most comonly used functions. The first one sends one ore more values while the other recieves them.

3.2.1. 3.2.0.1 The formal definition of send is:

```
int MPI_Send(
    void *buffer,
        //Address of send buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int dest,
        //Rank of destination
    int tag,
        //Message tag
    MPI_Comm comm
        //Communicator
    )
```

We give a pointer of the variable that will be sent (the addres of one single variable or an adress of an array), the number of elements, the data type. The later is one of the MPI datatypes (described in details in the Appendix) for compatibility reason, usually MPI_INT or MPI_DOUBLE. We must provide the exact destination, as this is point-to-point communication. The message tag is the label of the envelope, and serves the purpose to differentiate among similar messages. The communicator is denoting the MPI world.

The most commonly used data types are:

MPI_CHAR	char
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

3.2.2. 3.2.0.2 The formal definition of recieve is:

```
int MPI_Recv(
    void *buffer,
        //Address of recieve buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int source,
        //Rank of source
    int tag,
```

```
        //Message tag
    MPI_Comm comm,
        //Communicator
    MPI_Status *status
        //Status object
    )
```

The pointer is showing the variable or array where the message data will be stored. The count and data type is the same as in the case of sending. The source indicates the exact rank of the sending process, although here we can recieve message from an unknown sender by using the wildcard MPI_ANY_SOURCE. The tag must mach the tag of the sended message, or again, a wildcard MPI_ANY_TAG can be used, although we do not recommend it, as in our opinion the message should always be definite. The communicator is indicating the MPI world. The status object is a structure that contains the following fields:

- MPI_SOURCE - id of processor sending the message
- MPI_TAG - the message tag
- MPI_ERROR - error status.

It also contains other informations as well, which can be requested by MPI_Get_count and MPI_Probe or MPI_Iprobe.

3.2.3. 3.2.1 Deadlock problem

The usual MPI_Send() and MPI_Recv() functions are blocking point-to-point functions. This means that the sending function will return to the program only after the sending of all data was completed, and recieving returns when all data recieved. This can cause some problems if we are not cautious enough, namely a deadlock problem. Let us see a simple example program, where both processes set value to a variable and send it over to the other process.

```
//wrong sequence of send/recieve -> may cause deadlock
int a,b;
if(id == 0){
  a=1;
  MPI_Send(&a, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Recv(&b, 1, MPI_INT, 1, 2, MPI_COMM_WORLD, &status);
}else{ //id == 1
  b=2;
  MPI_Send(&b, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Recv(&a, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
}
```

Although the program seams to be well formed it can lead to a deadlock situation. As sending is blocking, the process of $id = 0$ will send the value of a, and wait to it be recieved. It will not initiate the recieving until the sending is complete. The other process of $id = 1$ also initiates a sending the variable b, and waits for it to be recieved, but not starts the recieving. It is clear, that these processes will hang at this point, and this situation is called the deadlock.

A correct message passing program should always consider the sequence of sending and recieving at both parties. It should be ordered like this:

```
//right sequence of send/recieve -> may not cause deadlock
int a,b;
if(id == 0){
  a=1;
  MPI_Send(&a, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Recv(&b, 1, MPI_INT, 1, 2, MPI_COMM_WORLD, &status);
}else{ //id == 1
  b=2;
  MPI_Recv(&a, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
```

```
  MPI_Send(&b, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
}
```

Actually there are different send and recieving functions in MPI, of which the reader may read about more in the Appendix. The usual MPI_Send() and MPI_Recv() functions are mixed mode ones. They try to buffer the sending, and so the send function is returned after the buffering is complete. So if the message is short - as in previous example - the deadlock may not occur at all. But this is not ensured, so we must alert the reader to always think through the deadlock problem. It is considered as a good practice to ensure that there is no possible deadlock situation even if the functions may not blocking, and reorganize the send/recieve functions in apropriate sequence.

3.3. 3.3 Broadcast

We need to mention one special send-recieve function in addition to the previous, the broadcasting function. It is a quite usual need to send a certain data to all of the processes from one master process. In other words in this case all the processes will work on the same dataset. The most common place is the initialization of the algorithms in the beginning of the program. For this a special function of broadcasting stands in MPI.

3.3.1. 3.3.0.1 The formal definition of broadcast is:

```
int MPI_Bcast(
    void *buffer,
        //Address of send-recieve buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int root,
        //Rank of root process
    MPI_Comm comm
        //Communicator
    )
```

It is a symmetric function, which means all the processes, the sending one and the recieving ones are calling this function the same mode exactly at the same time. The rank of the root indicates the sender. The reader will see detailed usage in the examples of the second part of the book.

4. 4 First MPI programs

4.1. 4.1 Summation of elements

We start with a simple problem. We would like to sum the numbers from 1 to 10 000. The sequential program, which needs no explanation would be:

```
// the sum from 1 to 10000

#include<iostream>
using namespace std;

int main(int argc, char ** argv){
  int sum;
  sum = 0; // zero sum for accumulation
  for(int i=1;i<=10000;++i)
    sum = sum + i;
  cout << "The sum from 1 to 10000 is: " << sum << endl;
}
```

If we want to parallelize this procedure, we have to divide the process into sub-processes, in this case we assign different numbers to be summed to different processes, and sum the sub-sums.

```
// parallel calculation of the sum
// from 1 to 10000

#include<iostream>
#include<mpi.h>
using namespace std;

int main(int argc, char ** argv){
  int id, nproc;
  int sum,startval,endval,accum;
  MPI_Status status;
  MPI_Init(&argc,&argv);
  // get number of total nodes:
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  // get id of mynode:
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  sum = 0; // zero sum for accumulation
  startval = 10000*id/nproc+1;
  endval =   10000*(id+1)/nproc;
  for(int i=startval;i<=endval;++i)
    sum = sum + i;
  cout<<"I am the node "<< id;
  cout<< "; the partial sum is: "<< sum<<endl;
  if(id!=0) //the slaves sending back the partial sums
    MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
  else //id==0! the master recieves the partial sums
    for(int j=1;j<nproc;j=j+1){
      MPI Recv(&accum, 1, MPI INT, j, 1, MPI COMM WORLD, &status);
      sum = sum + accum;
      cout<<"The sum yet is: "<<sum<<endl;
    }
  if(id == 0)
    cout << "The sum from 1 to 10000 is: " << sum << endl;
  MPI Finalize();
}


$ mpirun -np 4 ./sum1
I am the node 0; the partial sum is: 3126250
The sum yet is: 12502500
The sum yet is: 28128750
The sum yet is: 50005000
The sum from 1 to 10000 is: 50005000
I am the node 1; the partial sum is: 9376250
I am the node 2; the partial sum is: 15626250
I am the node 3; the partial sum is: 21876250
```

4.2. 4.2 Calculation of the value of π

In our book we would like to present not only programming examples, but some useful hints for algoritmization as well. So our next example will present a good and useful technique for parallelization. It is a technique which uses a randomization approach, the Monte Carlo method. It can be used where we look for some approximation and do not need the exact result. It divides the field of the problem to very small partitions and randomly "measures" some but not all of them. By the means of "measuring" more and more points, we hope to get a more precise answer to the problem in question. This is a useful method for many problems including engineering and scientific problem solving.

Also, we must note, that although we hope to get more precise answer by the means of more measurement points, it is not so simple. The answer first converges fast, but then it will slow down. Actually this method displays $1/\sqrt{N}$ convergence, so quadrupling the number of sampled points halves the error. And then, after some point, we won't be able to get more precise answer by adding more points simply because of the used

number precision, the round-off errors and that fact that we use a pseudo random number generator instead of real random numbers. The parallelization of the Monte Carlo method is straightforward: we can assign different "measure points" to the parallelly running programs and add up the results in the end in one master program.

The second example is a program which calculates the value of π. The Monte Carlo method is the following. Imagine a dart board of size 1×1 meter, and a circle of 1 meter diameter on it. Then imagine that we throw numerous darts in the board. Now, we ask, what is the possibility that a dart hitting the square dartboard is also hitting the circle. (This is the same as one minus the probability that the dart hitting the dartboard not hitting the circle.) The answer is easy: as the dart hitting is proportional to the surface, we should calculate the ratio of the dart board ($1m^2$) and the circle ($r^2 \times \pi = \pi/4$). We can also get the answer in the empirical way, calculating the actual hits of the dartboard and the circle. These two values should be close enough.

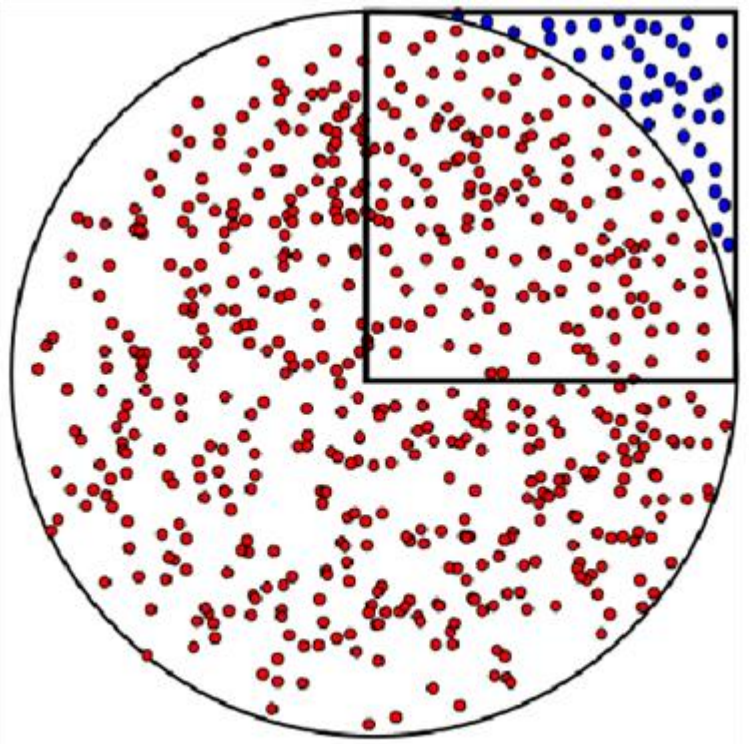

Figure 2: Simulating a dart shot

If we calculate the red and the blue dots on the Figure 2 in the smaller square, we would get 140 and 40 - by our calculation. So altogether 180 darts was thrown at the smaller square, from which 140 hit the quoter circle. The calculated value would be quite close to the actual value of π:

$$\pi \approx \frac{140}{180} \times 4 = 3.11$$

So the Monte Carlo method for calculating the value of π generates "dart shoots", and calculates the ratio of hits in the circle. For easier programming we usually use a quarter board of a 2×2 board with a quarter circle with a radius of 1. Then two coordinates are generated (x and y of values from 0 to 1), and the distance of the origo is calculated ($\sqrt{x^2 + y^2}$). If the distance is smaller or equal than 1, it is a hit in the circle, if the distance is bigger, than it is not a hit in the circle.

4.2.1. 4.2.1 The sequential program

The main part of the program is self explanatory. We make random x and y values - do not forget to cast to double, as integer division is different from floating point division! -, and calculate the (square) distance from the origin, and see if it is smaller than 1. We need not use squere root, as taking the square of both sides the

calculation is more easy. Counting those value pairs that closer to the origo than 1 and dividing it with the value of all pairs we get $\pi/4$.

```
const long long iternum=1000000000;
long long sum=0;

srand((unsigned)time(0));
for(long long i=0;i<iternum;++i){
  x=(double)rand()/RAND_MAX;
  y=(double)rand()/RAND_MAX;
  if(x*x+y*y<1) ++sum;
 }

Pi=(4.0*sum)/iternum;
```

4.2.2. 4.2.2 The parallel program

The parallel version of the previous program is quite simple. As generating random number pairs and counting those, which represent points on the circle totally independent tasks, we can assign them independently to different processes. At the end the slave processes send their local sum value to the master process, which collects these local sums and sum them up. We need to be cautious only at the last calculation of π itself: insted of iternum we must use iternum*nproc, as each process made iternum number of value pairs.

```
//Calculation of the constant Pi by Monte Carlo method

#include <cstdlib>
#include <ctime>
#include <iostream>
#include <math.h>
#include <mpi.h>
using namespace std;

int main(int argc, char **argv){
  int id, nproc;
  MPI_Status status;
  double x,y, Pi, error;
  long long allsum;
  const long long iternum=1000000000;

  // Initialize MPI:
  MPI_Init(&argc, &argv);
  // Get my rank:
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  // Get the total number of processors:
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  srand((unsigned)time(0));
  cout.precision(12);

  long long sum=0;
  for(long long i=0;i<iternum;++i){
    x=(double)rand()/RAND_MAX;
    y=(double)rand()/RAND_MAX;
    if(x*x+y*y<1) ++sum;
  }

  //Slave:
  if(id!=0){
    MPI_Send(&sum, 1, MPI_LONG_LONG, 0, 1, MPI_COMM_WORLD);
  }

  //Master:
  else{
    allsum=sum;
    for(int i=1;i<nproc;++i){
      MPI_Recv(&sum, 1, MPI_LONG_LONG, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
        &status);
```

```
      allsum+=sum;
    }

    //calculate Pi, compare to the Pi in math.h
    Pi=(4.0*allsum)/(iternum*nproc);
    error = fabs( Pi-M_PI );
    cout<<"Pi: \t\t"<<M_PI<<endl;
    cout<<"Pi by MC: \t"<<Pi<<endl;
    cout<<"Error: \t\t"<<fixed<<error<<endl;
  }

  // Terminate MPI:
  MPI_Finalize();
  return 0;
}
```

4.2.3. 4.2.3 Reduction operation

As collecting some partially ready data is more than quite common in parallel programming there also exists a special reduction function in MPI for this. Reduction means that we collect the data and perform on it some operation (reduce it). Similarly to the broadcast function we name the root process, which collects the data, and we name the operator, which will be performed by this root on all data.

4.2.3.1. 4.2.3.1 The formal definition of reduction is:

```
int MPI_Reduce(
    void *sendbuf,
        //Address of send buffer
    void *recvbuf,
        //Address of receive buffer (significant only at root)
    int count,
        //Number of elements in send buffer
    MPI_Datatype datatype,
        //Data type of elements of send buffer
    MPI_Op op,
        //Reduce operation
    int root,
        //Rank of root process
    MPI_Comm comm
        //Communicator
    )
```

As the data collected may be in conflict with the sending data on the root process we must provide different buffers for sending and recieving, although only for the root the recieve buffer is used. The rank of the root, the count of the send buffer, the MPI data type, and the communicator are all the same as in the previously described dending, recieving and broadcasting functions. As reduction is performed the recieve buffer count is always 1, so it is not needed to be named. The only new parameter is the operator of the reduction. Again, as with data types, for compatibility reasons MPI defines its own operators. The detailed operator list is:

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

4.2.3.2. 4.2.3.2 Allreduce

A variant of reduction operator is MPI_Allreduce which - after the reduction of the local elements - sends back the result to all processes. The syntax is almost the same as MPI_Reduce except we do not need to name the root process. An example of usage will be shown below.

4.2.4. 4.2.4 π by Monte Carlo and reduction

With the described reduction operation we can simplify the previously showed example. The sum variable is the local sum of the in-circle dartshots, the allsum variable is the collected and summed value of the local sums. The MPI operator we use is the MPI_SUM.

```
//Calculation of the constant Pi by Monte Carlo method

#include <cstdlib>
#include <ctime>
#include <iostream>
#include <math.h>
#include <mpi.h>
using namespace std;

int main(int argc, char **argv){
  int id, nproc;
  MPI_Status status;
  double x,y, Pi, error;
  long long allsum;
  const long long iternum=1000000000;

  // Initialize MPI:
  MPI_Init(&argc, &argv);
  // Get my rank:
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  // Get the total number of processors:
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  srand((unsigned)time(0));
  cout.precision(12);

  long long sum=0;
  for(long long i=0;i<iternum;++i){
    x=(double)rand()/RAND_MAX;
    y=(double)rand()/RAND_MAX;
    if(x*x+y*y<1) ++sum;
  }

  //sum the local sum-s into the Master's allsum
  MPI_Reduce(&sum, &allsum, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
  //Master writes out the calculated Pi
  if(id==0){
    //calculate Pi, compare to the Pi in math.h
    Pi=(4.0*allsum)/(iternum*nproc);
    error = fabs( Pi-M_PI );
    cout<<"Pi: \t\t"<<M_PI<<endl;
    cout<<"Pi by MC: \t"<<Pi<<endl;
    cout<<"Error: \t\t"<<fixed<<error<<endl;
  }

  // Terminate MPI:
  MPI_Finalize();
  return 0;
}
```

4.2.5. 4.2.5 Minimum and maximum location reduction

There are two special reduction operators which perform on double values. The MPI_MAXLOC and MPI_MINLOC The firs value is treated as values which should be compared through the distributed processes. The second value is treated as an index. The function chooses the minimum (or the maximum) value, and returns this value with its index.

The function MPI_MAXLOC defined as:

$$\begin{bmatrix} u \\ i \end{bmatrix} \star \begin{bmatrix} v \\ j \end{bmatrix} = \begin{bmatrix} w \\ k \end{bmatrix},$$

where $w = max(u, v)$, and

$$k = \begin{cases} i & \text{if } u > v \\ min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

The function MPI_MINLOC defined similary:

$$\begin{bmatrix} u \\ i \end{bmatrix} \star \begin{bmatrix} v \\ j \end{bmatrix} = \begin{bmatrix} w \\ k \end{bmatrix},$$

where $w = min(u, v)$, and

$$k = \begin{cases} i & \text{if } u < v \\ min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

4.2.5.1. 4.2.5.1 Special double data types

For the above desribed MPI_MAXLOC and MPI_MINLOC functions one should use double values. The table below shows the built in double value types of MPI.

MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	pair of ints
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

To use them one should first construct a structure as in the example in the Dijkstras's Algorithm for shortest path in a graph. The p will be the send buffer, the p_tmp the recieve buffer for the MPI_Allreduce function.

```
struct{
      int dd;
      int xx;
    } p, tmp_p;
    p.dd=tmp; p.xx=x;

    MPI_Allreduce(&p, &tmp_p, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);

    x=tmp_p.xx;
    D[x]=tmp_p.dd;
```

4.3. 4.3 Exercises

In the followings we present some simple exercises for students. They can be done as class work or homework as well. The presented material should be enough for solving these problems. First try to solve these problems in parallel without any efficiency questions in mind. Always compare the output with a sequential program to check whether the parallel program did any miscalculations. It is extremely usual to fail at first even with the simplest programs, as parallel programming needs a completely different thinking scheme.

4.3.1. 4.3.1 Matrix-Vector multiplication

The reader may try to parallelize the matrix-vector multiplication. We suggest to create a simple sequential program first, with a small sized matrix and vector. Then try to write a parallel program distributing the elements of the vector like in the element summation problem.

The definition of matrix-vector multiplication is:

$$\mathbf{Ax} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}$$

4.3.2. 4.3.2 Matrix-Matrix multiplication

The second programming excercise will be the implementation of a parallel matrix-matrix multiplication program.

For multiplying matrices, the elements of the rows in the first matrix are to multiplied with the corresponding columns in the second matrix.

We have two matrices of size $n \times m$ and $m \times p$:

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{bmatrix}$$

where the number of columns in A necessarily equals the number of rows in B, in this case m. The matrix product $\mathbf{AB}$ is to be the $n \times p$ matrix:

$$\mathbf{AB} = \begin{bmatrix} (AB)_{11} & (AB)_{12} & \cdots & (AB)_{1p} \\ (AB)_{21} & (AB)_{22} & \cdots & (AB)_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (AB)_{n1} & (AB)_{n2} & \cdots & (AB)_{np} \end{bmatrix}$$

where $\mathbf{AB}$ has entries defined by:

$$(AB)_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}.$$

4.3.3. 4.3.3 Numerical integration

A useful exercise may be the programming of the numerical integration with different methods. Again, we propose to first create a sequential program and try it with different inputs. Look out for the good distribution of inputs, starting from a constant function, for example $f(x) = 1$, moving to more complex functions. Try out different endpoints and number of inner points. If the results seems to be good, then try to parallelize the program. Parallellization should be quite straightforward, as each method sums independently calculated values, so the only problem is to distribute the sections evenly.

The described methods can be used for calculating the definite integral of one dimensional functions.

4.3.3.1. 4.3.3.1 Rectangle method

In this method, we partition the integration interval of the function to n sections, and substitute the function with a rectangle of height of the function value at the beginning of this subinterval. The figure 3 shows this method.

Namely:

$$\int_a^b f(x)\,dx \approx h \sum_{n=0}^{N-1} f(x_n)$$

where $h = (b-a)/N$ and $x_n = a + nh$.

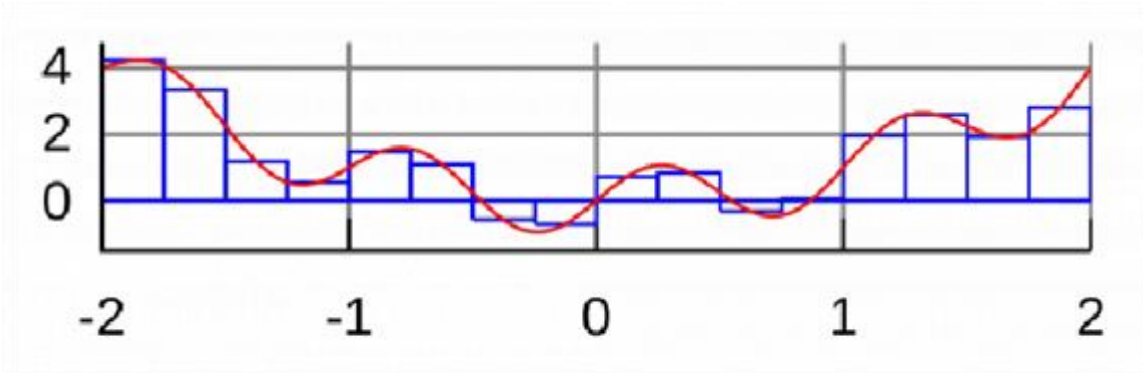

Figure 3: Rectangle integration

4.3.3.2. 4.3.3.2 Trapezoid method

This method is like the previous one, but instead of rectangles we use trapezoids, where the beginning height is the value of the function in the beginning of the section, and the end height is the value of the function at the end of the section. The figure 4 shows this method.

The trapezoid rule for one section is:

$$\int_a^b f(x)\,dx \approx (b-a)\left[\frac{f(a)+f(b)}{2}\right].$$

You should add up the trapezoids to get the whole integral.

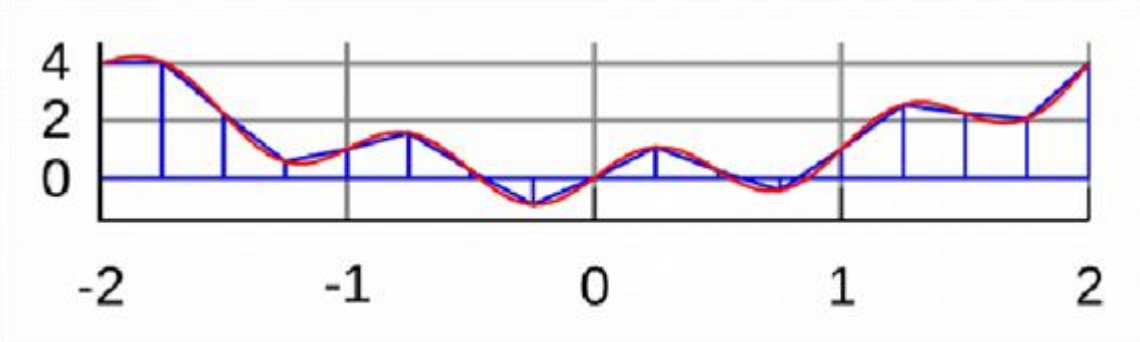

Figure 4: Trapezoid integration

4.3.3.3. 4.3.3.3 Simpson's rule

Again, we partition the function to sections, but it calculates with polynomials with an order of 2. The figure 5 shows this method. The Simpson's rule for a section is:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right].$$

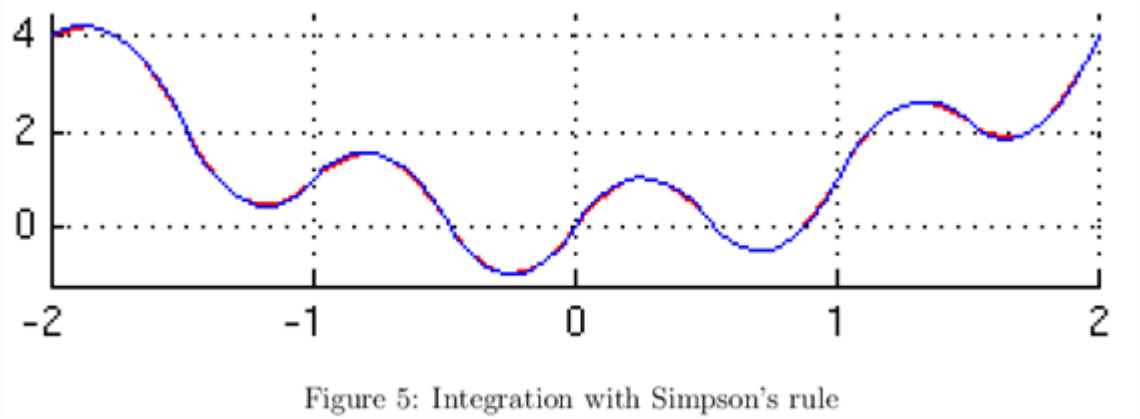

Figure 5: Integration with Simpson's rule

5. 5 Basic parallel techniques

The main challenge of problem parallelization is not exactly the algorithm part, but first, the decomposition of the problem into sub-problems. There are several methods and principles regading this question, but in our book we will be able to discuss this topic only briefly. Our goal is to present the reader with an introduction to MPI programming. For more information and deeper discussion of this truly large topic one may read the books in our bibliography, especially [Gram2003] and [Matt2005]. All in all, there is no royal road to this problem, but each and every case must be dealt differently and with precaution.

The main goals of the parallelization is to achieve more speed-up. In order to achieve this we need to take care of three separate problems.

The first one is to distribute the work among the processes (running on different processors) more evenly. If the distribution is uneven, then this will cause some processes run shorter time while the whole running time will be dominated by the slower processes. Let us look at an example. Take a problem which is decomposed into 100 sub-problems, and we will use 10 processors and so 10 processes. If the sub-problems are even and we give each process 10 sub-problems, that is one tenth of the whole problem, then each process will finish its work in one tenth of the original time. (We do not consider here other tasks connected to parallelization yet.) The speed-up is 10. Now, if we give one process 20 sub-problems and the others only 9 or 8, then the processes given less tasks will roughly finish in less then half time as the one given 20 sub-problems. The whole system will have a running time of the longest process, and that would lead to a speed-up of 5, which is clearly a worse case. Obviously, if we know that the sub-problems are equal we will assign them to processes evenly. But in case of uneven sub-problems we cannot do so, and we either must correctly guess the size of the sub-problems to assign them evenly, or still end at an uneven distribution. In this chapter we deal with this question in details.

The second problem is the locality of the data. For shared memory systems this question is a serious one, as the possibility of reading any data at any time hides away the real problem of how far the data is. Failing to notice this problem may cause a shared memory algorithm run slower by magnitudes! Also, there is a huge problem of cache usage, as nowadays multiprocessor computers are cache coherent, but achieving cache coherency may take quite a good time from the computation. For detailed analysis on cache coherency the reader may see the book [Sima1997]. For the question of algorithmic design considering cache usage of the book [Braw1989] produces very good examples.

In the case of distributed systems with message passing communication this problem is easier and harder at the same time. It is easier, because there is no possible way of directly reading data from the other computer, so the algorithmic design must always take this into consideration. But it is a harder problem, for the same reason, as there is no easy way of getting the data. One must always design it, consider it, and reconsider several times. And the main drawback is not the hardness of writing correct programs, but the communication overhead. If we are sending back and forth data many times, our program may be slower than the sequential version! At the end, we will always face a special problem, when adding more processors does not make the program faster, but even slows it down. In this case the execution time of the program is dominated by the communication time. So, when designing our problem decomposition we must take care of data locality, hence minimising the intercommunication between different processes and make them work mostly on their local data.

The third problem is the overhead of the algorithm. This one consists of that part which must be done in serial way and the cost of the parallelization itself. The serial part cannot be overridden. There will be always some parts - mostly setting up the problem, distributing the original data, collecting the sub-results, making tests on correctness -, which cannot be distributed. Moreover, we may add to these the changed start-up time of an MPI system. Starting several hundreds of processes on a distributed supercomputer may take some seconds. If the problem itself is solvable on a PC in some minutes, there is no reason for using a supercomputer, which will be slower, clumsier and overloaded most of the time.

5.1. 5.1 Possible problem decomposition methods

As we have seen one of the main components of parallel algorithm design is the decomposition of the problem. We are given a large problem and our task is to split it up into parts so that these parts can be assigned to different computing resources. This task is not straightforward, and we can only outline some methods that can be helpful.

First, we should note, that one problem of the decomposition is the actual number of parts we would like to split the problem into. Actually we do not know the actual number of processes in advance, as algorithms can be applied to different cases. It seems that choosing the same number of parts as processes available is desired, but it is not so. If the parts cannot be constructed in equally computational expensive way then using the same number of parts will lead to unequal load balance. So actually using more sub-problems as processes is more useful, as we can deal better with inequalities. Obviously using much-much more sub-problems is also undesirable, as accounting them will cost us resources comparable to those we use to do useful work. This means that using more sub-problems as processes but not too much more is usually the goal.

When we divide the problem, sometimes the problem itself consists of independent tasks. In this case we can speak about task parallelism, and use it as the base of our parallelization. Ray-trace or molecular dynamics problems are the examples of these kinds, where a ray or a molecule is an independent task we calculate.

Another case arises, when the sequential algorithm itself decomposes the problem during run. This is the divide-and-conquer case, as for example in merge sort or quick sort algorithms. If this is the case we can try to assign these sub-tasks to different processes.

In case of some problems the structure of input data itself can help us to find a method to split it up. In the case of many engineering or scientific modelling problems the data is bound to real world topology, so geometric decomposition is possible. In these cases usually computing is made at local points and some communication between neighbouring points will be accomplished. These tasks do the work on different input data independently, at least for a while, so we can speak about decomposition by data.

Sometimes decomposition can be made not on spatial but temporal base. It means that we need to do some tasks on the data in sequence. Then we can assign different tasks to processes and traverse the data from one process to another to be the next task made on it. This is a pipeline structure. Signal processing or batch graphical processing are good examples of this kind.

The reader can see examples in the second part of our book. The shortest path problem, graph coloring and ray-trace are good examples of task parallelism. The plain heating is a geometric decomposition, and the linear equality is a decomposition by data.

After we made the decision about the decomposition itself in theory, we still have to implement it. The next few sections will show some possible methods for doing so. Also we need to decide whether the task assignment will be made statically a priori, or dynamically during the run of the program. There can be advantages in both methods.

5.2. 5.2 Loop-splitting

The possible simplest method of decomposing the data is the loop splitting method. The data is considered as an array, and each process out of nproc takes every nproc-th data item out of the array. Namely:

```
//N: number of elements
//id: the rank of the process
//nproc: the number of all processes
//a[]: the array of work descriptors
for(int i=id;i<N;i+=nproc)
  do_work(a[i]);
```

Here each process starts to process the elements of the array at the point noted by its rank, the id, and takes steps of nproc magnitude.

To demonstrate this method with a complete and small program we show the problem of summation of elements from the previous chapter. The program with loop splitting would be:

```
// parallel calculation of a sum
// from 1 to 10000
// by loop splitting

#include<iostream>
#include<mpi.h>
using namespace std;

int main(int argc, char ** argv){
  int id, nproc;
  int sum,accum,startval,endval;
  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);  // get number of total nodes
  MPI_Comm_rank(MPI_COMM_WORLD, &id);     // get id of mynode
  sum = 0; // zero sum for accumulation
  //startval = 10000*id/nproc+1;
  //endval =   10000*(id+1)/nproc;
  for(int i=1+id;i<=10000;i+=nproc) // loop splitting
    sum = sum + i;
  cout<<"I am the node "<<id<<"; the partial sum is: "<<sum<<endl;
```

```
  if(id!=0)
    MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
  else
    for(int j=1;j<nproc;j=j+1){
      //MPI_Recv(&accum,1,MPI_INT,j,1,MPI_COMM_WORLD, &status);
      MPI_Recv(&accum,1,MPI_INT,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,
  &status);
      // we do not wait for particular slave: recieve answer from anybody
      sum = sum + accum;
      cout<<"The sum yet is: "<<sum<<endl;
    }
  if(id == 0)
    cout << "The sum is: " << sum << endl;
  MPI_Finalize();
}
```

The usage of loops splitting is highly encouraged. First, it is simple to write the loop, so usually minimal modification is needed to the program. Second, as it takes every nproc-th data element it often makes more even distribution. Usually the decomposed sub-problems differ in complexity more in different parts of the decomposition. Let's say, at the beginning there are easier problems present, which become more and more complex at the end. In this case the loop splitting makes quite a good job. But be aware, there can be counter examples as well, say, if even numbered problems are much harder as the odd ones, then in case of an even number of processors the work distribution will be very uneven.

5.3. 5.3 Block scheduling

In contrast to loop splitting there is another method, more natural for most programmers, the block scheduling. In this case we separate the problem in blocks. For N elements and $nproc$ processes the first $N/nproc$ elements will be assigned to the first process, the second part to the second and so on. With loops we first must assign a startvalue and an endvalue. The partial program would be like:

```
//N: number of elements
//id: the rank of the process
//nproc: the number of all processes
//a[]: the array of work descriptors
int startval = N*id/nproc;
int endval =   N*(id+1)/nproc;
for(int i=startval;i<endval;++i)
  do_work(a[i]);
```

To demonstrate the method with a complete program we show again the summation of elements problem:

```
// parallel calculation of the sum
// from 1 to 10000

#include<iostream>
#include<mpi.h>
using namespace std;

int main(int argc, char ** argv){
  int id, nproc;
  int sum,startval,endval,accum;
  MPI_Status status;
  MPI_Init(&argc,&argv);
  // get number of total nodes:
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  // get id of mynode:
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  sum = 0; // zero sum for accumulation
  startval = 10000*id/nproc+1;
  endval =   10000*(id+1)/nproc;
  for(int i=startval;i<=endval;++i)
    sum = sum + i;
  cout<<"I am the node "<< id;
  cout<< "; the partial sum is: "<< sum<<endl;
```

```
  if(id!=0) //the slaves sending back the partial sums
    MPI_Send(&sum,1,MPI_INT,0,1,MPI_COMM_WORLD);
  else //id==0! the master recieves the partial sums
    for(int j=1;j<nproc;j=j+1){
      MPI_Recv(&accum, 1, MPI_INT, j, 1, MPI_COMM_WORLD, &status);
      sum = sum + accum;
      cout<<"The sum yet is: "<<sum<<endl;
    }
  if(id == 0)
    cout << "The sum from 1 to 10000 is: " << sum << endl;
  MPI_Finalize();
}
```

The difference between the loop splitting and block scheduling is an interesting question. We already noted, that loop splitting may produce more even distribution. On the other hand, the loop which increment one by one may run a bit faster. However, this will be unnoticeable in case of most problems.

Still, there is a case when block scheduling is a must. If the sub-problems are connected to each other and there is some necessary communication between the sub-problems in order to maintain consecutive numbering, than we cannot use loop splitting without making too much communication overhead. The geometric decomposition of many engineering problems fall into this category. In our later examples the heating of a plate will show this problem in details.

5.4. 5.4 Self scheduling

In contrast to the previous work assigning methods, where the assignment was done statically a priori, we can assign the work to processes dynamically as well.

One best known method to do this is the master-slave work processing or in other words post office parallelization.

There is a Master process, who takes care of and accounts for the work to be done. The Slave processes ask for a job to do, and report back when they are ready. After sending the results, they ask for another job. When the job pool is empty, in other words no job (to be done) has been left, the Master process tells the Slaves to terminate.

The basic - but yet incomplete - program part would be like this:

```
//N: number of elements
//id: the rank of the process
//a[]: the array of work descriptors

if(id==0){//Master process
  int id_from;
  unit_result_t ANSWER;
  for(int i=0;i<N;++i){
    //recive from anybody, tag=1:
    MPI_Rec(&id_from, 1, MPI_INT, MPI_ANY_SOURCE, 1,...);
    //recieve answer, tag=2:
    MPI_Rec(&ANSWER, 1, MPI_datatype, id_from, 2,...);
    //send to slave who asked for job, tag=3:
    MPI_Send(&a[i], 1, MPI_datatype, id_from, 3, ...);
  }
 }else{//Slave process
  unit_result_t ANSWER;
  unit of work t QUESTION;
  while(true){
    //send our id
    MPI_Send(&id, 1, MPI_INT, 0, 1, ...);
    //send the answer
    MPI_Send(&ANSWER, 1, MPI_datatype, 0, 2, ...);
    //calculate the answer and ask for the next question
    ANSWER=do_work(QUESTION);
    MPI_Rec(&QUESTION, 1, MPI_datatype, 0, 3,...);
  }
 }
```

We can notice the incompleteness, as there is no start and no end in this question-answer conversation. These parts must be written separately. Also the reader must note the strict ordering of send and receive commands. If they would be in other order deadlock may occur, as discussed earlier in chapter three.

It is also important to make distinction between the different message types, and this is done by assigning different tags to different types of messages. It is also important in order to make the program error free.

To demonstrate the method completely, we will show a simple but complete frame program that performes this master-slave job distribution without the actual jobs.

The header of the program. We define the communication tags, which means the work tag commands the slaves to do the job, the die tag tells them to terminate. We also declare the functions we use.

```
1: // http://www.lam-mpi.org/tutorials/one-step/ezstart.php
  2: #include <iosream>
  3: #include <mpi.h>
  4:
  5: using namespace std;
  6:
  7: const int WORKTAG=1;
  8: const int DIETAG=2;
  9: typedef double unit_result_t;
 10: typedef double unit_of_work_t;
 11:
 12: /* Local functions */
 13:
 14: void master(void);
 15: void slave(void);
 16: unit_of_work_t get_next_work_item(void);
 17: void process_results(unit_result_t result);
 18: unit_result_t do_work(unit_of_work_t work);
 19:
```

Te actual main() function, which starts and finishes at the end the MPI_COMM_WORLD, and makes the distinction of the Master and Slave processes. The Master process calls the master() function, the Slave calls the slave() function.

```
20:
 21: int main(int argc, char **argv){
 22:   int myrank;
 23:
 24:   /* Initialize MPI */
 25:
 26:   MPI_Init(&argc, &argv);
 27:
 28:   /* Find out my identity in the default communicator */
 29:
 30:   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 31:   if (myrank == 0) {
 32:     master();
 33:   } else {
 34:     slave();
 35:   }
 36:
 37:   /* Shut down MPI */
 38:
 39:   MPI_Finalize();
 40:   return 0;
 41: }
 42:
```

The master() function is rather a long one, so we decided to present it in three pieces. In the first part the Master asks for the size of the MPI_COMM_WORLD, that is how many jobs are running altogether. After, he calculates the next job: work=get_next_work_item();, and sends the first jobs to each of the Slaves.

```
43:
```

```
 44: void master(void){
 45:   int ntasks, rank;
 46:   unit_of_work_t work;
 47:   unit_result_t result;
 48:   MPI_Status status;
 49:
 50:   /* Find out how many processes there are in the default
 51:      communicator */
 52:
 53:   MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
 54:
 55:   /* Seed the slaves; send one unit of work to each slave. */
 56:
 57:   for (rank = 1; rank < ntasks; ++rank) {
 58:
 59:     /* Find the next item of work to do */
 60:
 61:     work = get_next_work_item();
 62:
 63:     /* Send it to each rank */
 64:
 65:     MPI_Send(&work,             /* message buffer */
 66:              1,                 /* one data item */
 67:              MPI_INT,           /* data item is an integer */
 68:              rank,              /* destination process rank */
 69:              WORKTAG,           /* user chosen message tag */
 70:              MPI_COMM_WORLD);   /* default communicator */
 71:   }
 72:
```

The main circle of the Master process is first to receive the answers from the slaves after they finished with the assigned job. And then calculate the next job (while there is a job in the job pool), and send it to the Slave who just sent an answer.

```
 73:   /* Loop over getting new work requests until there is no more work
 74:      to be done */
 75:
 76:   work = get_next_work_item();
 77:   while (work != NULL) {
 78:
 79:     /* Receive results from a slave */
 80:
 81:     MPI_Recv(&result,           /* message buffer */
 82:              1,                 /* one data item */
 83:              MPI_DOUBLE,        /* of type double real */
 84:              MPI_ANY_SOURCE,    /* receive from any sender */
 85:              MPI_ANY_TAG,       /* any type of message */
 86:              MPI_COMM_WORLD,    /* default communicator */
 87:              &status);          /* info about the received message */
 88:
 89:     /* Send the slave a new work unit */
 90:
 91:     MPI_Send(&work,             /* message buffer */
 92:              1,                 /* one data item */
 93:              MPI_INT,           /* data item is an integer */
 94:              status.MPI_SOURCE, /* to who we just received from */
 95:              WORKTAG,           /* user chosen message tag */
 96:              MPI_COMM_WORLD);   /* default communicator */
 97:
 98:     /* Get the next unit of work to be done */
 99:
100:     work = get_next_work_item();
101:   }
102:
```

If there is no job left in the job pool, the Master receives the last answer from the Slave, and then sends him a message with a die tag, which is a request of termination.

```
103:   /* There's no more work to be done, so receive all the outstanding
```

```
104:      results from the slaves. */
105:
106:   for (rank = 1; rank < ntasks; ++rank) {
107:     MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
108:              MPI_ANY_TAG, MPI_COMM_WORLD, &status);
109:   }
110:
111:   /* Tell all the slaves to exit by sending an empty message with the
112:      DIETAG. */
113:
114:   for (rank = 1; rank < ntasks; ++rank) {
115:     MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
116:   }
117: }
118:
```

The Slave function is simple. In an infinite loop the Slave receives the next job, calculates it, and sends back the answer. On the other hand, if the Slave receives a die tag it terminates, which means, it returns to the calling main().

```
119:
120: void slave(void){
121:   unit_of_work_t work;
122:   unit_result_t result;
123:   MPI_Status status;
124:
125:   while (1) {
126:
127:     /* Receive a message from the master */
128:
129:     MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
130:              MPI_COMM_WORLD, &status);
131:
132:     /* Check the tag of the received message. */
133:
134:     if (status.MPI_TAG == DIETAG) {
135:       return;
136:     }
137:
138:     /* Do the work */
139:
140:     result = do_work(work);
141:
142:     /* Send the result back */
143:
144:     MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
145:   }
146: }
147:
```

The functions of get_next_work_item(), process_results(unit_result_t result), do_work(unit_of_work_t work) are empty, because this is only a frame for structuring a master-slave communication. However the reader may fill in the functions with the required processing functionality and will get a good parallel program if the problem in question is parallelizable in this way.

```
148:
149: unit_of_work_t get_next_work_item(void){
150:   /* Fill in with whatever is relevant to obtain a new unit of work
151:      suitable to be given to a slave. */
152: }
153:
154:
155: void process_results(unit_result_t result){
156:   /* Fill in with whatever is relevant to process the results returned
157:      by the slave */
158: }
159:
160:
161: unit_result_t do_work(unit_of_work_t work){
```

```
162:   /* Fill in with whatever is necessary to process the work and
163:      generate a result */
164: }
```

5.5. 5.5 Other dynamic load balancing

In our opinion the work sharing by self scheduling is a good dynamic method. Other literature give other examples as well. On of them is called Work Stealing. It propose a static distribution in the beginning. When a process runs out of its assigned jobs, then it steals unstarted jobs from other, more occupied processes.

While the static distribution is the beginning may have real advantages, the more complex programing of the stealing makes this method harder to program. For learning purposes we strongly suggest the self scheduling method.

5.6. 5.6 Meshes for computation

At many problems, computational partition is already given (E.g. matrices). For some problems, this is not given, we have to partition the objects into structures for ourselves. The main idea is to decompose complex surfaces, shapes and volumes to small parts, thus, we can build a complex equation system with the help of many small and simple equations. Science today is unimaginable without massive calculations. These calculations often involve partial differential equations. The analytical methods for exact solutions of these equations are not very usable for real-life problems. The most common way is to approximate the solution with numerical methods. In case of differential equations, the finite element method is a good numerical approximation. The finite element method (FEM) is useful to simulate fluid flow, heat transfer, mechanical force simulation or even electromagnetic wave propagation. The main concern of this method is still a huge topic: how to do the decomposition? One obvious but rather ineffective way would be the point-wise decomposition, where a large group of n-dimensional points represent the whole scene. In this case, several questions would arise: resolution, infinite element number and necessity. We would cause an overwhelming and mostly needless computational load by doing this. A better idea is to decompose the given object to a finite number of sub-objects with given size or extent. We can do this also with varying sizes, so the more interesting parts could stand for smaller elements, while parts of no interest can stand for some or only one element.

The partitioning is done by creating sub-meshes that construct the form of our original object. Our mesh or grid in 2D consists of triangles or rectangles, or in 3D tetrahedra or rectangular prisms. Let's stay at 2D in further. According to the basic problem, the triangles have to have special geometry which can be achieved by different mesh generating algorithms. The generating procedure can be way more time consuming than the solution of the FEM modelling itself, so it's obvious, that parallelism would help the most at this stage.

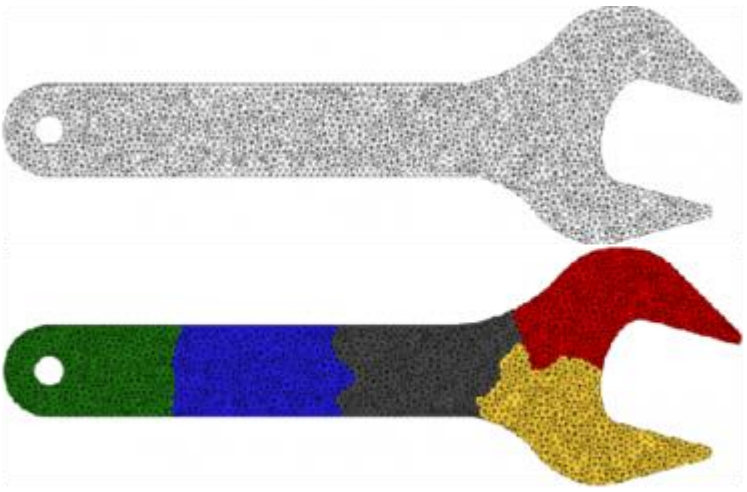

Figure 6: Equally distributed mesh and partitioning (Iványi P, Radó J [Ivanyi2013])

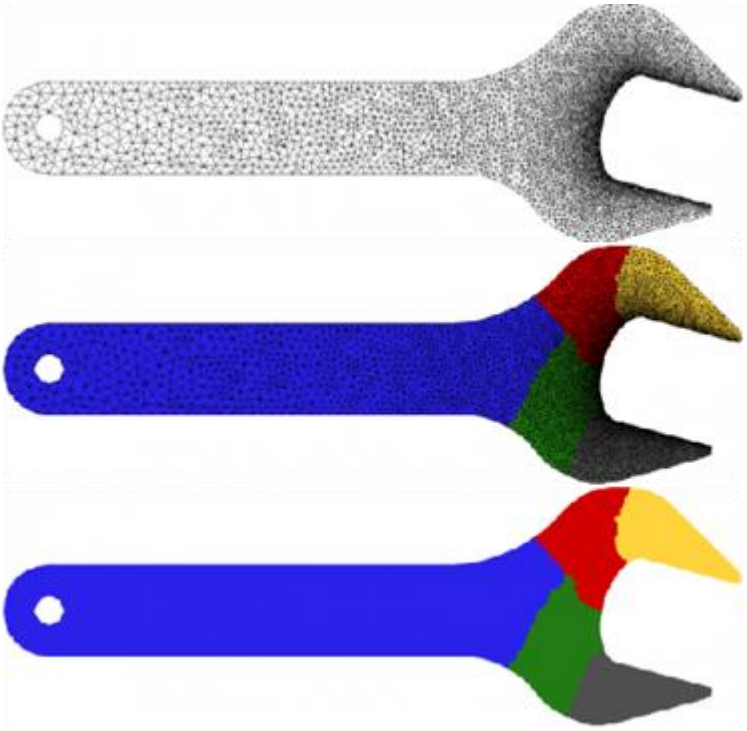

Figure 7: Adaptive mesh, partitioning and domains (Iványi P, Radó J [Ivanyi2013])

There are several mesh generating strategies. We can separate two types based on the starting state. One kind of meshing starts from a set of points, where these points have to be connected in a way that uniformly spread triangles or rectangles build up a mesh, including every starting point. A typical algorithm for doing this is the Delaunay-triangulation. Another kind of meshing is without any inner points. Our starting geometry is a silhouette, a boundary form, which we would like to fill up with triangles with given size and parameters. A simple meshing strategy for that would be the "Advancing Front" method.

5.6.1. 5.6.1 Short description of the Delaunay-triangulation

[Ivanyi2013]

The method starts with a set of points. These points have to be connected with edges in a way, that the result will be a mesh with non-intersecting faces and more-or-less similar triangles. The exact rule is that for every triangle no other point can be inside the circumcircle of that triangle. This rule maximizes the triangle's minimum angles, thus gives a computationally suitable form.

5.6.2. 5.6.2 Short description of the "Advancing Front" method

[Ivanyi2013]

The method starts with only a boundary form consisting of points and connecting lines. This form should be filled with triangles generating a mesh on the inner side of the boundary form. The idea of this method is to place triangles on the inner side of the bounding faces, connect the far end edge of them with each other and start the procedure over on these new connecting edges. This will generate stripes in the inside of the form and finally, the stripes will knit and we can finish the procedure with some final connecting edges in the middle.

5.6.3. 5.6.3 Parallel mesh generation

The FEM (Finite Element Method) also suggests the possibility of effective parallelization, since we have to deal with a high number of elements. We can distribute the constituent elements amongst working nodes and by

this, we can gain speed-up. Using meshes and the FEM is suitable for parallelization, though the problem is not trivial. For effective work, proper partition is needed first. After the sub-meshes are generated, they have to be merged. At this stage, inappropriate triangles or polynomials can arise which have to be handled. That often affect surrounding nodes as well and makes the procedure complex.

For further details on mesh generation, partition and preparing parallel mesh calculations, the reader is referred to the book by P. Iványi and J. Radó - "Előfeldolgozás párhuzamos számí tásokhoz" [Ivanyi2013] suggested.

5.7. 5.7 Monte-Carlo and Las Vegas methods

There are several problems for which a closed-form expression or a deterministic algorithm cannot be constructed or is not practical. For these cases, there could be a good idea to use repeatedly gathered samples with which a statistically approached numerical approximation can be done. These methods construct a set of results based on random probes, imitating a casino situation, hence the name. We can presume that these kinds of approximations will tend to have less and less errors with increasing number of probes. Since we need mostly exponentially growing computing power for more accurate calculations, here, parallelization seems appropriate again. A big advantage of the method is that the probes are independent from each other, hence, intercommunication of computing nodes in a parallel solution will be minimal.

For discrete problems sometimes the Las Vegas probabilistic method can be used. In this method we always get the right answer, but we do not know the running time. This method can be used for more complex problems of NP-hard type, but that is beyond the scope of this book.

5.7.1. 5.7.1 π by Monte-Carlo

One good example for the Monte-Carlo method would be a graphical calculation of the value of π. This could happen by using a 2×2 square and an inscribed circle of radius 1. By throwing virtual darts to this table, assuming that every throw will be a random hit, we can generate hits inside and outside the circle. The ratio of the number of points inside the circle and inside the square is proportional to their surface: $2 \times 2 = 4$ and $1^2 \times \pi$, so it will approximate the value of $\pi/4$. We showed an exapmle program of this calculation in the previuos chapter.

5.8. 5.8 Excercises

The reader may go back to the previous chapter's exercises, and think it over, if the solution is efficient or not. What are the main problems with those solutions if any, and can they be improved? What is the expected speed-up of the programs?

The reader also can try out to measure running times of different implementations and compare them with the expected ones.

6. 6 Practical Examples

6.1. Introduction

In this part we would like to show some practical problems. These problems usually considered hard problems, and the solution takes a long time even using today's computers. So it is clear, that parallelization will be useful.

Obviously we try to make our examples more readable and so we made some simplifications to the programs and sometimes to the problems itself. But still, the presented examples can show the usefulness of the parallelization, and can demonstrate a few tricks for designing parallel programmes with MPI.

Our examples will demonstrate the possibilities of parallel programming with MPI on some discrete mathematical problems, as finding shortest paths in graphs, graph colouring and sorting of numbers. Also we will demonstrate some engineering problems as solving linear equations, Fast Fourier Transformation and a practical example of simulating the heat distribution in time. Also we will show examples more picturesque as calculating the Mandelbrot set and a parallel version of a simple raytrace program.

6.2. Test runs

In our demonstration we tried to make the examples more vivid. In our opinion this should include the demonstration of the speed-up of the algorithms and implementations. So for some problems we demonstrate the running times in different HPC computing environments - two supercomputers in Hungary. Both of them allows the researchers to run programs up to 512 processes, so our examples will scale up to that point. Also, because of the architecture of the second computer we chose to scale up from 12 processes by multiplying the number of processes by 2. So our examples will (in some cases) demonstrate 12×2^k processors together with 512 processes runs, which means usage of 1, 12, 24, 48, 96, 192, 384 or 512 cores.

One of them - to which we will refer later in our book as computer A - is an SMP computer, which means that the interprocess communication is fast. This computer is part of the Hungarian supercomputer infrastructure and located at the University of Pecs. It is an SGI UltraViolet 1000 computer, which is a ccNUMA architecture. The peak performance of this 1152 core computer is 10 TFlops. http://www.niif.hu/node/660

The second one - to which we will refer later in our book as computer B - is a clustered supercomputer, with more processors, but slower fat tree intercommunication. It is located at the University of Szeged, and consists of 24 core node blades. The peak performance of this 2304 core computer is 14 TFlops. http://www.niif.hu/node/661

The reason we choose these very computers is to demonstrate the effect of the interconnect itself on the running time of the algorithms. Actually the cores of the computer A are faster than the cores in the computer B, so the "ratio" of the computing power and the cost of communication is much better for the computer A, the SMP computer, as in the case of the clustered computer B. On the other hand the computer B has more computing power if the algorithm can explore its capabilities. Also, we should note, that there exist quite big problems for building large SMP computers, so our example computer is nearly the biggest available nowadays. In contrast clustered supercomputers are able to scale up to much bigger sizes, so our clustered example should be considered as a smaller type. Which means that for bigger computing capacity an SMP computer is not an option.

The test runs produced some interesting timing results, so the authors note, that the numbers presented here should only point at some interesting phenomenon, and show trends for comparison. First, the problems are too small for real testing. Second, the timings varied greatly from one run to an other. Sometimes differences of a double ratio in time were noticed. We did not made several runs and computed averages. In real life one does not calculate the same problem several times, so it is unrealistic. With bigger problems the variation of runs would disappear. And we could still see those points we would like to emphasise.

7. 7 Shortest path

The shortest path problem is widely used to solve different optimization problems. In this problem we construct a directed weighted graph where the nodes of the graph connect to directed edges if there is a path in that direction, and we also indicate the weight of this path. Given two nodes: u and v, we are looking for the shortest path between these two nodes.

7.1. 7.1 Dijkstra's algorithm

Edsger W. Dijkstra constructed a simple but powerful algorithm to solve the problem indicated in the case where the weights are non-negative.[Dijk1959] It is a greedy algorithm, which finds the shortest path from a given (s) node to all other nodes (algorithmically this will cost us the same). The algorithm uses a D (distance) array, where the yet shortest paths are saved from s. It takes the closest node to s from the not ready

nodes, and marks it as ready. Then looks for alternative paths through this node, and if finds one, then upgrades the appropriate D array value. Formally we show it in Algorithm 7.1.

Algorithm 1 Dijkstra's algorithm for shortest paths problem

Require: N as size of the graph

```
1: function DIJKSTRA(s)
2:     OK[s] ← true
3:     for all i do
4:         D[i] ← (s, i)
5:     loop
6:         x ← i where D[i] is minimum for OK[i] is false
7:         OK[x] ← true
8:         for all i where OK[i] is false do
9:             if D[i] > D[x] + (x, i) then
10:                D[i] ← D[x] + (x, i)
11:                Indicate that we came to i from x
```

If we construct a program from this algorithm, the main stress point is to find the minimum of the D array. There are different methods depending on the structure of the graph - whether it is dense or not. For non dense graphs some heap structure is the best method (binary, binomial, Fibonacci), for dense graphs a simple loop can be used. As we are concentrating later on parallelization, we will use the more simple approach. Also we consider that the adjacency matrix of the graph is given ($G[N][N]$), so the (u, v) distance can be read from it. The sequential program for Dijkstra's shortest path algorithm is:

```
const int INFINITY=9999;
// let INFINITY be a very big number, bigger than any other

// G[][] is the adjacency matrix
// D[] is the distance array
// path[] is indicating where from we came to that node

void dijkstra(int s){
  int i,j;
  int tmp, x;

  //initial values:
  for(i=0;i<N;i++){
    D[i]=G[s][i];
    OK[i]=false;
    path[i]=s;
  }
  OK[s]=true;
  path[s]=-1;

  //there are N-1 steps in the greedy algorithm:
  for(j=1;j<N;j++){

    //finding maimimum of D[]:
    tmp=INFINITY;
    for(i=0;i<N;i++){
      if(!OK[i] && D[i]<tmp){
        x=i;
        tmp=D[i];
      }
    }
    OK[x]=true;

    //alternating paths:
    for(i=0;i<N;i++){
      if(!OK[i] && D[i]>D[x]+G[x][i]){
        D[i]=D[x]+G[x][i];
        path[i]=x;
      }
```

```
    }
  }
}
```

7.2. 7.2 Parallel version

The parallelization of the program should start with a decomposition of the problem space into sub-problems. It is not trivial to do it in this case, as no clear independent parts can be noted. First, we make the division between the nodes. This means, that in each part of our program where a loop for all nodes is given, we rewrite it to consider only a subset of the nodes. We can use a block scheduling or even a loop splitting. We used the latter , so in our program each loop looks like:

```
for(i=id;i<N;i+=nproc)
```

As there is a part where the closest of the nodes is searched, and there is a part where the alternating paths are calculated for each node. The latter can be clearly calculated independently on all the nodes. But the former is problematic: we search for global minimum, not for a minimum between some nodes.

This can be done in a parallel way as follows. We search for the minimum distance of a subset of nodes, then collect the minimums by the master, which calculates the global minimum, and sends it back to the slaves.

Thus, after calculating the local x and D[x] we construct a pair (pair[2]) of these two. The slaves send their pair to the master. The master receives the pair, and by each receiving calculates if the received value is smaller than the saved one. If so, the master interchanges the saved value with the received one. At the end the minimum values are broadcasted back, and each process updates the x and D[x] value by the global minimum:

```
pair[0]=x;
    pair[1]=tmp;

    if(id!=0){
      MPI_Send(pair,2,MPI_INT,0,1,MPI_COMM_WORLD);
    }else{ // id==0
      for(i=1;i<nproc;++i){
        MPI_Recv(tmp_pair,2,MPI_INT,i,1,MPI_COMM_WORLD, &status);
        if(tmp_pair[1]<pair[1]){
          pair[0]=tmp pair[0];
          pair[1]=tmp_pair[1];
        }
      }
    }
    MPI_Bcast(pair,2,MPI_INT,0,MPI_COMM_WORLD);
    x=pair[0];
    D[x]=pair[1];
```

We could have used the MPI_ANY_SOURCE at the receiving, but it would not speed up the program. In any case, we must wait for all the slaves to send their values, and the broadcast will act as a barrier.

Was it a good idea, to split the nodes by loop splitting? In any case, there will be inequalities in the sub-parts of the nodes, as the nodes will be ready one by one and we ignore them in the future computation. But it is impossible to predict how this will act during the execution, so we can split the nodes by our will, as no better method is possible.

To see the strength of our little example, we run the parallel program on two different supercomputers. One of them (A) is an SMP computer with a fast interprocess communication. The second one (B) is a clustered supercomputer with more processors but slower communication. We constructed an artificial graph of 30 000 and 300 000 nodes (denoted by 30k and 300k) and run the program with different numbers of active processors. In the table below we indicate the running time of the sequential program and the running time of the parallel program with different numbers of processors. We also indicate the speed-up from the sequential program.

nproc	A-30k		B-30k		A-300k		B-300k	
1	80.0s		169s		9841s		20099s	
12	10.8s	7.4x	49s	3.5x	2302s	4.3x	2896s	6.9x
24	5.7s	14.0x	34s	5.0x	1320s	7.5x	1201s	16.8x
48	3.2s	25.0x	11s	15.6x	447s	22.0x	807s	24.9x
96	2.4s	33.0x	55s	3.0x	226s	43.5x	433s	46.4x
192	3.5s	23.0x			151s	65.2x	360s	55.8x
384	11.0s	7.3x			129s	76.3x	580s	34.7x
512					152s	64.7x		

As one can see, the presented program served its purpose. We can notice quite good speed-up times, while the problem itself is not an easily parallelizable one. We managed it by carefully distributing the work. This meant that we assigned a part of the nodes of a graph to the distributed processes . They took care of calculating the partial minimum, and the update of their part with alternating paths if there are better ones than previously found.

The communication left is only the synchronization of the minimum nodes. It consists of the collection of partial minimums, calculating the global minimum, and sending it back to every process.

The speed-up is limited by the ratio of the work done by each process and the frequency of the communication.

7.3. 7.3 Examination of results

Carefully examining the presented table we can find out some interesting consequences. By these we can point to some well known phenomena as well.

First, we can clearly see, that the speed-up is somehow limited. At first as we add more and more processes the program accelerates rapidly. Then, as we add more, the acceleration slows down, and even stops. At the end we can even see some points where adding more processors slows down the computation.

Remember Amhdahl's law, which states that every parallel computation has its limit of speed-up. Amhdal pointed out that every program has a sequential part (at least the initialization), so we cannot shorten the running time below the time of running of the sequential part. We can add even more. The communication overhead can also dominate the problem, if the communication part takes more time as the actual calculation. So the running time will decline after some point where too many processes are added, which makes the sub-problems too small and the communication too frequent. The reader should check the table to find this point! Note, that it is different for the two computers, as their interconnects are quite different.

Second, we can observe that Amhdal's law starts to dominate the problem at different points for different problem sizes. This observation leads us to Gustavson's law. It states, that for bigger problems we can achieve better speed-up even if we do not reject Amhdal's low. The reader should check the table to see!

And third, there is an interesting point, where doubling the number of processes reduces the running time more than a factor of 2! The reader again should check the table to find this point! This is not a wrong measurement (we checked and rechecked it several times.) This moment is called the super-linearity. At such cases we can see more than twice of the speed-up for twice as many processors. We suspect that this speed-up depends on the actual architecture. In our case the problem size, the part of the adjacency matrix and the distance array fits into the cache of one processor after this point. This means, that the memory accesses will be much faster, so the speed-up of our algorithm gains more than the awaited factor of two.

7.4. 7.4 The program code

At the end, we would like to present the complete program code of the parallel version. In the first part the constants and the global variables are presented. Global variables are used for two purposes. First, this way they are accessible from the main function and also from all the other functiona. Second, the actual adjacency matrix

is kept on the heap this way, and with big matrices one can avoid memory problems. (Obviously dynamically allocated memory would serve the same purpose.)

Note, that variables id and nproc are also global, as the initialization takes place in main, while the communication is done by the void dijk(int) function.

The int Read_G_Adjacency_Matrix() functions here as a dummy one. The reader should replace it with one which reads in the actual adjacency matrix. Other possibility can be, when the matrix is not stored but calculated for each edge one by one. This method is used when the matrix would be too big and wouldn't fit into memory. Actually our test-runs were made this way. In this case the G[][] should be exchanged to a function in each occurrence.

```
1: #include <iostream>
 2: #include <mpi.h>
 3: using namespace std;
 4:
 5: const int SIZE=30000; //maximum size of the graph
 6: char G[SIZE][SIZE]; //adjacency matrix
 7: bool OK[SIZE]; //nodes done
 8: int D[SIZE]; //distance
 9: int path[SIZE]; //we came to this node from
10: const int INFINITY=9999; //big enough number, bigger than any possible path
11:
12: int nproc, id;
13: MPI_Status status;
14: int N;
15:
16: int Read_G_Adjacency_Matrix(){
17:   //actual G[][] adjacency matrix read in
18: }
19:
20: void dijk(int s){
```

The second part is the actual function which calculates the distances from one node. We already explained the parallelization.

```
20: void dijk(int s){
21:   int i,j;
22:   int tmp, x;
23:   int pair[2];
24:   int tmp pair[2];
25:   for(i=id;i<N;i+=nproc){
26:     D[i]=G[s][i];
27:     OK[i]=false;
28:     path[i]=s;
29:   }
30:   OK[s]=true;
31:   path[s]=-1;
32:   for(j=1;j<N;j++){
33:
34:     tmp=INFINITY;
35:     for(i=id;i<N;i+=nproc){
36:       if(!OK[i] && D[i]<tmp){
37:  x=i;
38:  tmp=D[i];
39:       }
40:     }
41:
42:     pair[0]=x;
43:     pair[1]=tmp;
44:
45:     if(id!=0){ //Slave
46:       MPI_Send(pair,2,MPI_INT,0,1,MPI_COMM_WORLD);
47:     }else{ // id==0, Master
48:       for(i=1;i<nproc;++i){
49:  MPI_Recv(tmp_pair,2,MPI_INT,i,1,MPI_COMM_WORLD, &status);
50:  if(tmp_pair[1]<pair[1]){
51:    pair[0]=tmp_pair[0];
```

```
52:    pair[1]=tmp_pair[1];
53:  }
54:       }
55:     }
56:     MPI_Bcast(pair,2,MPI_INT,0,MPI_COMM_WORLD);
57:     x=pair[0];
58:     D[x]=pair[1];
59:     OK[x]=true;
60:     for(i=id;i<N;i+=nproc){
61:       if(!OK[i] && D[i]>D[x]+G[x][i]){
62:  D[i]=D[x]+G[x][i];
63:  path[i]=x;
64:       }
65:     }
66:   }
67: }
68:
69: main(int argc, char** argv){
```

The third part is the int main() function itself. It produces the G[][] adjacency matrix, initializes the MPI communicator and measures the running time.

```
69: main(int argc, char** argv){
70:   double t1, t2;
71:
72:   MPI_Init(&argc,&argv);
73:   MPI_Comm_size(MPI_COMM_WORLD, &nproc); // get totalnodes
74:   MPI_Comm_rank(MPI_COMM_WORLD, &id);     // get mynode
75:
76:   N=Read_G_Adjacency_Matrix();
77:   //read in the G[][]
78:   //set actual size
79:
80:   if(id==0){
81:     t1=MPI_Wtime();
82:   }
83:
84:   dijk(200);
85:   //call the algorithm with the choosen node
86:
87:   if(id==0){
88:     t2=MPI_Wtime();
89:
90:     //check the results with some output from G[][] and D[]
91:
92:     cout<<"time elapsed: "<<(t2-t1)<<endl;
93:   }
94:
95:   MPI_Finalize();
96: }
```

7.5. 7.5 Examples for variants with different MPI Functions

We saw that MPI has reduction functions, and we can use one appropriately here as well. This is the MPI_Reduce and its variants. We need to find a global maximum, not only the value but the index of it as well. As we need to know which node is to be moved to the ready set, and to find alternate paths through. For this purpose we can use the MPI_MINLOC operator for the reduction, which finds the smallest value and its index. (The reader may find the detailed specification in the Appendix.) Also, we would like the output of the reduction to be presented in all processes, so we may use the MPI_Allreduce function. The part of our code involved is presented, and we can see the simplification of it.

```
struct{
      int dd;
      int xx;
    } p, tmp_p;
    p.dd=tmp; p.xx=x;
```

```
    MPI_Allreduce(&p, &tmp_p, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);

    x=tmp_p.xx;
    D[x]=tmp_p.dd;
```

Obviously we make the test-runs with this version as well, but no significant difference in running time was observed. The reduction functions are easier to write and the program is easier to read, but in most cases there will be no speed gain.

7.6. 7.6 The program code

We present here only the void dijk(int) function, as the other parts are unchanged.

```
20: void dijk(int s){
21:   int i,j;
22:   int tmp, x;
23:   int pair[2];
24:   int tmp_pair[2];
25:   for(i=id;i<N;i+=nproc){
26:     D[i]=G[s][i];
27:     OK[i]=false;
28:     path[i]=s;
29:   }
30:   OK[s]=true;
31:   path[s]=-1;
32:   for(j=1;j<N;j++){
33:
34:     tmp=INFINITY;
35:     for(i=id;i<N;i+=nproc){
36:       if(!OK[i] && D[i]<tmp){
37:  x=i;
38:  tmp=D[i];
39:       }
40:     }
41:
42:     struct{
43:       int dd;
44:       int xx;
45:     } p, tmp_p;
46:     p.dd=tmp; p.xx=x;
47:
48:     MPI_Allreduce(&p, &tmp_p, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
49:
50:     x=tmp_p.xx;
51:     D[x]=tmp_p.dd;
52:     OK[x]=true;
53:
54:      for(i=id;i<N;i+=nproc){
55:       if(!OK[i] && D[i]>D[x]+G[x][i]){
56:  D[i]=D[x]+G[x][i];
57:  path[i]=x;
58:       }
59:     }
60:   }
61: }
62:
63: main(int argc, char** argv){
```

7.7. 7.7 Different implementations

The presented algorithm prescribes two special steps: finding the minimum of all distance values for node that are not yet done, and decreasing these values if better alternating path is found. These two steps are greatly dependent on the data structure behind the scenes.[Corm2009] With dense graphs simple array of distance

values can be used effectively. Our presented solution showed this approach, and our test-runs used dense graphs. Also for teaching, this solution is the best, as it is the most simple one.

With less dense graphs other data structures should be used, as binary heap (in other name the priority queue), binomial heap or Fibonacci heap. These data structures have a Extract-Min and a Decrease-Key functions, which one would use in the implementation of the algorithm in the above mentioned places. The reader should consult the cited book for more details.

Still, a question arises. If we used an other implementation with, for example, Fibonacci heap, how would the parallel program differ from the presented one? The answer is simple: not too much. We distributed the nodes to the processes, so these processes would store only a subset of all nodes. The storage would be the same, in our case, in a Fibonacci heap. The Extract-Min algorithm obviously should be modified slightly not to extract, but to look up the minimum value first. Then the reduction to find the overall minimum element would take place the same way as in our example. The actual extraction should be made only after this. The second part of finding alternate paths would take place again on the subset of nodes given to each process, and the Decrease-Key operation will be performed on the nodes where the value of distance was needed to be changed.

8. 8 Graph coloring

Let $\Gamma = (V, E)$ be a finite simple graph, where V is the finite set of nodes and E is the set of undirected edges. There is exactly either one undirected edge or none between two nodes. The edges are not weighted.

We color the nodes of Γ so that each node receives exactly one color and that two nodes cannot have the same color if they are connected by an edge. This coloring is sometimes called legal or well coloring. A coloring of the nodes of Γ with r colors can be described more formally as a surjective map $f : V \to \{1, \ldots, r\}$. Here we identify the r colors with the numbers $1, \ldots, r$, respectively.

The level sets of f are the so-called color classes of the coloring. The i -th color class $C_i = \{v : v \in V, f(v) = i\}$ consists of all the nodes of Γ that are colored with color i. The color classes $C_1, \ldots, C_r$ form a partition of V. Obviously, the coloring is uniquely determined by the color classes $C_1, \ldots, C_r$. These partitions are also independent sets of the Γ graph, as the rule of coloring forbids any adjacent nodes to be in the same color class.

8.1. 8.1 Problem description

Coloring itself is an NP-hard problem, but there exist some good greedy algorithms. These algorithms may not find a best coloring, by which we mean coloring with the least possible colors, but can color a graph with colors close to this number. These greedy algorithms are used as auxiliary algorithms for other problems such as maximum clique problem. Also some problems are directly solved by a coloring, as some scheduling problems for example.

For clique search a coloring gives us an upper bound for the clique size, as any node in a Δ clique in the graph must be colored by different colors, as the nodes of a clique are pairwise adjacent. So any coloring by k colors gives the upper bound for maximum clique size of k. Clearly the better the coloring is, so the less colors we use, the better the upper bound will be. Also, it will speed up the clique search by many-fold times.

As a good coloring is useful in many cases, these greedy algorithms are quite useful. Because of their usefulness, different types of these algorithms are known, and mostly differ in their running time and goodness in terms of how many colors they use. Obviously one must choose between fast running time and the better coloring with less colors. All these algorithms are of good use. For example, for scheduling a much slower algorithm witch produces less color classes may be useful. For clique problems as auxiliary algorithms the faster ones are better as they may be called literally million times during a clique search.

8.2. 8.2 DSATUR algorithm of Daniel Brélaz

One well known coloring algorithm is the DSATUR algorithm of Daniel Brélaz. This greedy algorithm produces quite good coloring with moderate color classes while it runs quite fast. Not the fastest one, but close to it. Because of these properties this algorithm is used in clique search as well as for big scheduling problems of such size as other algorithms may be not feasible.

The DSATUR algorithm consists of three steps.

1. Find the optimal node to be colored:
   - has minimal freedom - that we can put this node into less suitable color classes;
   - for equal freedom nodes has maximum saturation - has many neighbours. (This step is debated in the literature.)
2. Color the chosen node:
   - put into the first free color class, or
   - open a new class if none is free. (Opening a new color class will increase the freedom of all nodes by one!)
3. Update information of the remaining nodes:
   - If the just colored node have a neighbour, then
   - It cannot be placed in the same color class, so
   - The freedom of this node will decrease by one.

Let us see the program. Beside the main() function we have a couple of other functions assigned to different sub-functions.

The function read_clq reads in the graph form a clq file. This type of file is the edge representation of a graph, stating the number of nodes and edges in the header, and enumerating the edges as pair of nodes. This type of file is described by the homepage of DIMACS Implementation Challenges: http://dimacs.rutgers.edu/Challenges/, and widely used by the scientific graph community.

The write_pbm is a non-standard output form for graph adjacency matrix, as it uses the PBM image format. This format uses a simplified text to describe the picture. It can be black and white where '1'-s and '0'-s indicate the white and the black dots - we use this one in this example, which makes the picture format explicitly equivalent to the adjacency matrix of the graph. The grayscale PBM uses a number between '0' and '255' for gray shade - and we will use it in the plate heating example. The color PMB format uses triplets of '0' to '255' numbers as R, G, B colors - we will use this format for Mandelbrot set pictures.

The reason we would like to use a PBM picture for the colored graph, is that we reorder the nodes by color classes, so nodes from the same color-class will be next to each other. In this case the coloring may be checked with eye, as in the adjacency matrix - the picture - we will see boxes without '1'-s around the main diagonal, this will mean independent sets as no edge will run inside that set. We may even count the boxes and check the number of colors the coloring used! We cannot stress enough, how important it is to check the results for high performance computation, as usually the output is also so big, that checking is a problem by itself.

For the above described reorder we use the permutation_color function, which constructs the permutation order of the nodes according to color classes. The perm vector of ints is used for this purpose, and the testperm function checks whether the permutation is a correct permutation, e.g. each number appears once and only once. The permto function is building the adjacency matrix according to these permutations.

Last, the initialize function sets the initial values of the data structures. These data structures we use are the adjacency matrix (adj[][]); the color classes where for each color class for each node we work out that the node is in that color class (colors[][]); the free colors where for each node for each color class we work out that the node can be put in that color class (free_color[][]); the saturation of a node where we collect the number of neighbours for each node (sat[]); the number of yet free color classes for each node (free[]); the fact if the node has been already colored (OK[]); and the number of color classes we used (colors). The global variable N indicates the number of nodes.

8.2.1. 8.2.0.1 The program

The beginning of the program is the declaration of the global variables and the functions. The main prints out the usage, or reads in the adjacency matrix from the .clq file if given as a parameter, then it initializes the data structures.

```
1: #include <iostream>
  2: #include <fstream>
  3: #include <vector>
  4: #include <algorithm>
  5: using namespace std;
  6:
  7: int N;
  8: string comment="";
  9:
 10: bool **adj; // adjacency matrix
 11: bool **colors; // color classes
 12: bool **free color; // free color classes of nodes
 13: int *sat; // saturation of a node
 14: int *free_num; // number of free color classes for nodes
 15: bool *OK; // is the node ready
 16: int num_color=0;
 17:
 18: void read_clq(string);
 19: void write_pbm(string);
 20: void initialize();
 21: void permutation_color();
 22: void permto();
 23: vector<int> perm; // the node list of permutation
 24: void testperm();
 25:
 26: int main(int argc, char **argv){
 27:   if(argc<2){
 28:     cerr<<"Usage: "<<argv[0]<<" file.clq"<<endl;
 29:     exit (1);
 30:   }
 31:   int i,j,c,min_free, min_sat, min_id;
 32:   string file_name(argv[1]);
 33:   read_clq(file_name);
 34:
 35:   initialize();
```

The coloring itself is a loop for assign color to each node, so it is executed N times. In the loop we first find the node of smallest freedom, and from those the maximal saturated one. This node, which has the index of min_id will be colored, so we note this fact in the OK[] array.

```
37:   //we color N nodes each after other:
 38:   for(int v=0;v<N;++v){
 39:     min_free=N;
 40:     min_sat=N;
 41:     //find node of minimum freedom:
 42:     for(i=0;i<N;++i)
 43:       if(!OK[i] && (min_free>free_num[i] ||
 44:       min_free==free_num[i] && sat[i]<min_sat)){
 45:  min_free=free_num[i];
 46:  min_sat=sat[i];
 47:  min_id=i;
 48:       }
 49:     OK[min_id]=1; //ready to color
```

The next part is the actual coloring and the updating of the data structures where we count the freedom of a node, and save the color classes it can be put into. This falls into two categories. The node we color may be of non freedom, so it cannot be put into any existing color class. In this case, we make a new color class (actually they were made in the initializing part, we just use it), and put the node into it. Then we increase the freedom for all not yet colored nodes, and next decrease this freedom for those which are connected. (It can be made in one step, but we chose to do it in two for a reason to be more clear. It does not effect the running time complexity.)

In the last step we update the number of color classes. (The num_color variable gives us the number of color classes while we number the color classes starting with 0. So when we point to the num_color-th color class it means the next then the last one we yet used. That is why we increase this variable only at the end.)

```
51:     //color it:
 52:     if(min_free==0){
 53:       //We need a new color class.
 54:       colors[num_color][min_id]=1;
 55:       //the new color class posible class for the rest:
 56:       for(i=0;i<N;++i)
 57:  if(!OK[i]){
 58:    free_color[i][num_color]=1;
 59:    ++free_num[i];
 60:  }
 61:       //but not for the connected nodes:
 62:       for(i=0;i<N;++i){
 63:  if(!OK[i] && adj[i][min_id]){
 64:    free_color[i][num_color]=0;
 65:    --free_num[i];
 66:  }
 67:       }
 68:       ++num_color;
```

The other category is when there is a color class into which the node can be put freely. First we find a suitable color class simply by finding the first free one (denoted by the variable c). We put the node into this color class, and again, for each neighbour we decrease the freedom of those nodes. decrease the freedom for those nodes that we have not yet colored, can be put into this color class, and adjacent to the node just been colored.

```
69:     }else{
 70:       //We put node into an old color class.
 71:       //find the class:
 72:       for(c=0;!free_color[min_id][c];++c);
 73:       colors[c][min_id]=1;
 74:       //the connected nodes' freedom decreases:
 75:       for(i=0;i<N;++i){
 76:  if(!OK[i] && free_color[i][c] && adj[i][min_id]){
 77:    free_color[i][c]=0;
 78:    --free_num[i];
 79:  }
 80:       }
 81:     }
 82:   }
```

At the end we print out the number of colors we used, construct the permutation according to color classes and check it, rearrange the adjacency matrix and write out the PBM picture file.

```
83:   cout<<"number of DSATUR colors: "<<num_color<<endl;
 84:
 85:   permutation_color();
 86:   testperm();
 87:   permto();
 88:   write_pbm(file_name);
 89: }
```

The initialize function sets the number of used colors to 0, the free color classes of all nodes to 0, the nodes uncolored (OK[]), and zeroes out the color classes and the possible color classes of the nodes. It also counts the neighbours of all the nodes to use the saturation number later.

```
91: void initialize(){
 92:   int sum;
 93:   num_color=0;
 94:   for(int i=0;i<N;++i){
 95:     sum=0;
 96:     for(int j=0;j<N;++j){
 97:       sum += adj[i][j];
 98:     }
```

```
 99:     sat[i]=sum;
100:     free_num[i]=0;
101:     OK[i]=0;
102:   }
103:   for(int i=0;i<N/6;++i)
104:     for(int j=0;j<N;++j){
105:       colors[i][j]=0;
106:       free color[j][i]=0;
107:     }
108: }
```

The read_clq function apart from reading in the adjacency matrix from the given file also dynamically allocates memory for all arrays. We cannot do this in other place, as the clq file gives us the value of N, and we have to allocate memory at least for the adjacency matrix after it but before reading in the values. For color classes we hope that there will be maximum of $N/4$ of them.

The clq file numerates the edges from number 1, while we index the arrays from 0, hence the shifts x-1 and y-1 in the end.

```
110: void read_clq(string file_name){
111:   int i,j,n,m,x,y;
112:   string type,line;
113:   ifstream fin(file_name.c_str());
114:   if(!fin.is_open()){
115:     cout<<"ERROR opening file"<<endl;
116:     exit(1);
117:   }
118:
119:   //eliminate comment lines
120:   while(fin.peek()=='c'){
121:     getline(fin,line);
122:     comment=comment+"#"+line+'\n';
123:   }
124:
125:   fin>>type; // type of image/matrix (P1)
126:   fin>>type; // edge
127:   fin>>N; // vertexes
128:   fin>>m; // edges
129:
130:   //allocate apropriate space according to N:
131:   adj=new bool*[N];
132:   colors=new bool*[N/6];
133:   free_color=new bool*[N];
134:   for(i=0;i<N;i++){
135:     adj[i]=new bool[N];
136:     free color[i]=new bool[N/6];
137:   }
138:   for(i=0;i<N/4;i++)
139:     colors[i]=new bool[N];
140:   sat = new int[N];
141:   free_num = new int[N];
142:   OK = new bool[N];
143:
144:   for(i=0;i<m;i++){
145:     fin>>type; // e
146:     if(type=="e"){
147:       fin>>x;fin>>y;
148:       //cout<<"x: "<<x<<"; y: "<<y<<endl;
149:       adj[x-1][y-1]=1;
150:       adj[y-1][x-1]=1;
151:     }
152:   }
153:   fin.close();
154: }
```

The function write_pbm simply writes out the adjacency matrix' '0'-s and '1'-s to the pbm file.

```
156: void write_pbm(string file_name){
```

```
157:   int i,j;
158:   file_name=file_name+".pbm";
159:   ofstream fout(file_name.c_str());
160:   if(!fout.is_open()){
161:     cout<<"ERROR opening file"<<endl;
162:     exit(1);
163:   }
164:   fout<<"P1"<<endl;
165:   fout<<comment;
166:   fout<<N<<" "<<N<<endl;
167:
168:   for(i=0;i<N;i++){
169:     for(j=0;j<N;j++){
170:       fout<<adj[i][j]<<" ";
171:     }
172:     fout<<endl;
173:   }
174:   fout.close();
175: }
```

The permutation_colors function sums the number of nodes in all color classes, and then finds the biggest number, pushes it into a vector and zeroes it out till all color classes are done.

```
177: void permutation_color(){
178:   int i,j,k, max, maxcol, sum;
179:   int *colors_sum= new int[N];
180:
181:   for(i=0;i<num_color;++i){
182:     sum=0;
183:     for(j=0;j<N;++j)
184:       sum += colors[i][j];
185:     colors_sum[i]=sum;
186:   }
187:   for(i=0;i<num_color;++i){
188:     max=0;
189:     for(j=0;j<num_color;++j)
190:       if(colors_sum[j]>max){
191:  max=colors_sum[j];
192:  maxcol=j;
193:       }
194:     for(j=0;j<N;++j)
195:       if(colors[maxcol][j])
196:  perm.push_back(j);
197:     colors_sum[maxcol]=0;
198:   }
199: }
```

The permto function constructs a new adjacency matrix from a given permutation (the vector perm).

```
201: void permto(){
202:   int i,j;
203:   bool **tmp=new bool*[N];
204:   for(i=0;i<N;i++)
205:     tmp[i]=new bool[N];
206:   for(i=0;i<N;i++)
207:     for(j=0;j<N;j++)
208:       tmp[i][j]=adj[perm[i]][perm[j]];
209:   for(i=0;i<N;i++)
210:     for(j=0;j<N;j++)
211:       adj[i][j]=tmp[i][j];
212: }
```

And the last function, test_perm, checks whether a permutation is correct, which means that each number occurs in it once and only once.

```
214: void testperm(){
215:   int sum;
216:   for(int i=0;i<N;++i){
```

```
217:     sum=0;
218:     for(int j=0;j<N;++j)
219:       if(perm[j]==i) ++sum;
220:     if(sum!=1){
221:       cerr<<"Wrong permutation!!"<<endl;
222:       exit (1);
223:     }
224:   }
225: }
```

8.3. 8.3 Parallelization

The problem of parallelization is quite similar to the previous chapter's Dijkstra's algorithm. There is no clear parallelization, but again we can do a task parallelization distributing the nodes between the processes. The coloring of the nodes will be done by every process for each node, but the administration of the free colors and the freedom of colors will be done by each process only to a partition of the nodes that are assigned to them.

The first step, finding the node of minimal freedom, first made locally from those nodes that assigned, and for which the process have right information. Then by reducing local minimums we get a global minimum. This node is to be colored, this step is to be made by all processes. Then only the assigned nodes' information is updated.

We will use block scheduling for which we have two reasons. First, the distribution of nodes is quite indifferent for a graph as the nodes can be enumerated in any order, so we are free to chose any distribution. As in the previous example we used loop splitting, here we will use block scheduling. Dynamic scheduling cannot be really used in this example. Second, there are two aims of parallelizing such an algorithm: reduction of running time and to be able to deal with bigger problems as a distributed algorithm can be constructed to use only a fraction of memory, as it may store only part of the information. Graph algorithms can be used for really huge graph instances, where memory limit can be a question. In this case we only need to send a part of a graph to a process, where block scheduling is quite useful, as sending by blocks is much easier.

Actually our program is not built this way to simplify it and differentiate it from the sequential program as little as possible. But for huge graphs one perhaps need to rewrite the program according to the previous notes.

8.4. 8.4 Program

Most of the functions and data are the same with two exceptions. The adjacency matrix is constructed in a bit tricky way. For being sent in one block the matrix needs to be in continuous memory space, so it is allocated in one big array and pointers of the adj array will point to different parts. Thus we have a one dimensional array to which we still can refer as a two dimensional array. The second difference is a new function, the memory_init, which we need for the following reason: The master process which reads in the clq file, must allocate memory at points different from the other processes, which can allocate memory only after they notified about the value of N. The other functions remain the same, so we will not show them again here.

The program starts the same way, in the main function the MPI framework is started, and the master process ($id = 0$) reads in the clq file.

```
1: #include <iostream>
  2: #include <fstream>
  3: #include <vector>
  4: #include <algorithm>
  5: #include <mpi.h>
  6: using namespace std;
  7:
  8: int N;
  9: string comment="";
 10:
 11: char **adj; // adjacency matrix
 12: char *adj_matrix; // actual store
 13: bool **colors; // color classes
 14: bool **free_color; // free color classes of nodes
```

```
 15: int *sat; // saturation of a node
 16: int *free_num; // number of free color classes for nodes
 17: bool *OK; // is the node ready
 18: int num_color=0;
 19:
 20: void read_clq(string);
 21: void write_pbm(string);
 22: void initialize();
 23: void memory_init();
 24: void permutation_color();
 25: void permto();
 26: vector<int> perm; // the node list of permutation
 27: void testperm();
 28:
 29: int main(int argc, char **argv){
 30:   int i,j,c,min_free, min_sat, min_id;
 31:   int id, nproc, id_from;
 32:   int startval, endval, sv, ev;
 33:   if(argc<2){
 34:     cerr<<"Usage: "<<argv[0]<<" file.clq"<<endl;
 35:     exit (1);
 36:   }
 37:   string file_name(argv[1]);
 38:
 39:   MPI_Status status;
 40:   // Initialize MPI:
 41:   MPI_Init(&argc, &argv);
 42:   // Get my rank:
 43:   MPI_Comm_rank(MPI_COMM_WORLD, &id);
 44:   // Get the total number of processors:
 45:   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 46:
 47:   if(id == 0){
 48:     read_clq(file_name);
 49:   }
```

After the master process reads in the clq file he will know the value of N, so it can be broadcast to all other processes. Only after this point, the memory allocation can be made by the slaves, so for all slave processes (id!=0) we call the memory_init function. As we are going to construct a block scheduling, we need to calculate the startval and the endval, which two variables will define the block of nodes for each process. The whole adjacency matrix is broadcast and the initialization function is called. Note that the broadcast of the adjacency matrix is done through the actual one dimensional array. We also need to mention here that instead of bool values we used char-s because of the openmpi framework we use. The present openmpi program supports only MPI 2.2 version and no bool data type is available yet for MPI communication.

For more complex program we only need to send the matrix for a block or rather band between the startval and endval nodes. As the program is rather complicated we decided to skip this part and broadcast the whole matrix for the sake of simplicity.

```
50:   MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
 51:   if(id!=0)
 52:     memory_init();
 53:   startval = N*id/nproc;
 54:   endval =   N*(id+1)/nproc;
 55:
 56:   MPI_Bcast(adj_matrix, N*N, MPI_CHAR, 0, MPI_COMM_WORLD);
 57:
 58:   initialize();
```

The actual coloring starts. Similarly to the sequential program in loop we color all the N nodes. First we find the node with minimal freedom and among similar ones the one with biggest saturation.

```
60:   //we color N nodes each after other:
 61:   for(int v=0;v<N;++v){
 62:     min_free=N;
 63:     min_sat=N;
 64:     min_id=-1;
 65:     //find node of minimum freedom:
```

```
 66:     for(i=startval;i<endval;++i)
 67:       if(!OK[i] && (min_free>free_num[i] ||
 68:       min_free==free_num[i] && sat[i]<min_sat)){
 69:  min_free=free_num[i];
 70:  min_sat=sat[i];
 71:  min_id=i;
 72:       }
```

As for the previous problem of shortest paths we make a minimum location reduction of the local minimums. We construct a structure of the node number (min_id) and of its freedom value (min_free). Then we perform an Allreduce for minimum location, so all the processes will be given the id and the $freedom$ of the global minimum $freedom$. Actually we cheat a little, as for equal freedom values we should look for the maximum saturation by the original algorithm. We do it for the local minimums, but not so for the global one. So it is performed partially. It could have been done the way we did for the first version of shortest paths with sending to the master alongside the local id -s and local $freedom$ the saturation values as well, and let the master choose 'manually' from them according exactly to the algorithm. That would complicate the program slightly, and we aim for simplicity. Apart from this we already mentioned that the choosing of maximal saturation is debated to have real impact for the result of the algorithm. So choosing this minimum location seemed a good choice.

```
74:     struct{
 75:       int free;
 76:       int id;
 77:     } p, tmp_p;
 78:     p.id=min_id;
 79:     p.free=min_free;
 80:     //not optimal,
 81:     //we would've need to look for minimum of free and sat together
 82:     MPI_Allreduce(&p, &tmp_p, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
 83:
 84:     min_id=tmp_p.id;
 85:     min_free=tmp_p.free;
```

The update of the information we store about nodes (freedom, and which color class they can be put into) is quite similar to the sequential program. First if the node we color cannot be put into any color class (its freedom is zero), then we open a new color class, and put the node into it. Note that the fact of the freedom being zero is known by all processes, so all the processes will put the node into a new color class. Again we increase the freedom number of all nodes and then decrease them if they are neighbours. The difference from the sequential program is that the loops run from $startval$ to $endval$ as we do block scheduling, and update only information of those nodes that assigned to this process.

```
87:     OK[min id]=1; //ready to color
 88:     //color it:
 89:     if(min_free==0){
 90:       //We need a new color class.
 91:       colors[num_color][min_id]=1;
 92:       //the new color class posible class for the rest:
 93:       for(i=startval;i<endval;++i)
 94:  if(!OK[i]){
 95:    free_color[i][num_color]=1;
 96:    ++free_num[i];
 97:  }
 98:       //but not for the connected nodes:
 99:       for(i=startval;i<endval;++i){
100:  if(!OK[i] && adj[i][min_id]){
101:    free_color[i][num_color]=0;
102:    --free_num[i];
103:  }
104:       }
105:       ++num_color;
106:     }else{
```

The other case is when the node to be colored can be placed into an existing color class. This case is a little more complex, as only the process to which this node is assigned can know the exact color class the node will be placed into. So first we must find the process which 'has' the node - we run a simple loop and use the same

equation as we counted the $startval$ and $endval$. The resulting process number will be stored in the variable id_from and it is known by every process because of the parallel computation by the loop. Then only the process in charge of this node (id==id_from) calculates the color class, which value (c)is broadcast to all processes. The following program code is the same as the sequential one except from the block scheduling of the loops that run from $startval$ to $endval$.

```
106:     }else{
107:       //We put node into an old color class.
108:       //find the class:
109:       int id_from;
110:       for(id_from=0;id_from<nproc;++id_from)
111:  if(N*id_from/nproc<=min_id && min_id<N*(id_from+1)/nproc) break;
112:       if(id==id_from)
113:  for(c=0;!free_color[min_id][c];++c);
114:       MPI_Bcast(&c, 1, MPI_INT, id_from, MPI_COMM_WORLD);
115:       colors[c][min_id]=1;
116:       //the connected nodes' freedom decreases:
117:       for(i=startval;i<endval;++i){
118:  if(!OK[i] && free_color[i][c] && adj[i][min_id]){
119:    free_color[i][c]=0;
120:    --free_num[i];
121:  }
122:       }
123:     }
124:   }
```

In the end of the main we print out the number of color classes (we do it with all processes for testing purposes to see whether they did it all right), and the master process does the permutation by color classes and calls the write_pbm function to write out the PMB picture. The Finalize is called at the end to close the MPI framework.

```
125:   cout<<"number of DSATUR colors: "<<num_color<<endl;
126:   if(id==0){
127:     permutation_color();
128:     testperm();
129:     permto();
130:     write_pbm(file_name);
131:   }
132:   // Terminate MPI:
133:   MPI_Finalize();
134:
135: }
```

The the new memory_init function we wrote, will allocate the memory for data structures. It also makes the trick of allocating one dimensional array for the storage of adjacency matrix (adj_matrix) and making the pointers of the adj point to the proper parts of it to make it accessible through the adj matrix as a two dimensional array.

```
152: void memory_init(){
153:   int i,j;
154:   adj=new char*[N];
155:   adj_matrix=new char[N*N];
156:   colors=new bool*[N];
157:   free_color=new bool*[N];
158:   for(i=0;i<N;i++){
159:     adj[i]=adj_matrix+(i*N);
160:     colors[i]=new bool[N];
161:     free_color[i]=new bool[N];
162:   }
163:   sat = new int[N];
164:   free_num = new int[N];
165:   OK = new bool[N];
166:   for(i=0;i<N;++i)
167:     for(j=0;j<N;++j)
168:       adj[i][j]=0;
169: }
```

The read_clq differs only in that we take out the memory allocation from it to be placed in the memory_init, so we call that function instead of the memory allocations.

8.4.1. 8.4.1 The complete parallel program

```
1: #include <iostream>
  2: #include <fstream>
  3: #include <vector>
  4: #include <algorithm>
  5: #include <mpi.h>
  6: using namespace std;
  7:
  8: int N;
  9: string comment="";
 10:
 11: char **adj; // adjacency matrix
 12: char *adj_matrix; // actual store
 13: bool **colors; // color classes
 14: bool **free_color; // free color classes of nodes
 15: int *sat; // saturation of a node
 16: int *free num; // number of free color classes for nodes
 17: bool *OK; // is the node ready
 18: int num_color=0;
 19:
 20: void read_clq(string);
 21: void write_pbm(string);
 22: void initialize();
 23: void memory_init();
 24: void permutation_color();
 25: void permto();
 26: vector<int> perm; // the node list of permutation
 27: void testperm();
 28:
 29: int main(int argc, char **argv){
 30:   int i,j,c,min_free, min_sat, min_id;
 31:   int id, nproc, id_from;
 32:   int startval, endval, sv, ev;
 33:   if(argc<2){
 34:     cerr<<"Usage: "<<argv[0]<<" file.clq"<<endl;
 35:     exit (1);
 36:   }
 37:   string file_name(argv[1]);
 38:
 39:   MPI_Status status;
 40:   // Initialize MPI:
 41:   MPI_Init(&argc, &argv);
 42:   // Get my rank:
 43:   MPI_Comm_rank(MPI_COMM_WORLD, &id);
 44:   // Get the total number of processors:
 45:   MPI Comm size(MPI COMM WORLD, &nproc);
 46:
 47:   if(id == 0){
 48:     read_clq(file_name);
 49:   }
 50:   MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
 51:   if(id!=0)
 52:     memory init();
 53:   startval = N*id/nproc;
 54:   endval =   N*(id+1)/nproc;
 55:
 56:   MPI_Bcast(adj_matrix, N*N, MPI_CHAR, 0, MPI_COMM_WORLD);
 57:
 58:   initialize();
 59:
 60:   //we color N nodes each after other:
 61:   for(int v=0;v<N;++v){
 62:     min_free=N;
 63:     min_sat=N;
 64:     min_id=-1;
```

```
 65:     //find node of minimum freedom:
 66:     for(i=startval;i<endval;++i)
 67:       if(!OK[i] && (min_free>free_num[i] ||
 68:       min_free==free_num[i] && sat[i]<min_sat)){
 69:  min_free=free_num[i];
 70:  min_sat=sat[i];
 71:  min_id=i;
 72:       }
 73:
 74:     struct{
 75:       int free;
 76:       int id;
 77:     } p, tmp_p;
 78:     p.id=min_id;
 79:     p.free=min_free;
 80:     //not optimal,
 81:     //we would've need to look for minimum of free and sat together
 82:     MPI_Allreduce(&p, &tmp_p, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);
 83:
 84:     min_id=tmp_p.id;
 85:     min free=tmp p.free;
 86:
 87:     OK[min_id]=1; //ready to color
 88:     //color it:
 89:     if(min_free==0){
 90:       //We need a new color class.
 91:       colors[num_color][min_id]=1;
 92:       //the new color class posible class for the rest:
 93:       for(i=startval;i<endval;++i)
 94:  if(!OK[i]){
 95:    free_color[i][num_color]=1;
 96:    ++free_num[i];
 97:  }
 98:       //but not for the connected nodes:
 99:       for(i=startval;i<endval;++i){
100:  if(!OK[i] && adj[i][min_id]){
101:    free color[i][num color]=0;
102:    --free_num[i];
103:  }
104:       }
105:       ++num_color;
106:     }else{
107:       //We put node into an old color class.
108:       //find the class:
109:       int id_from;
110:       for(id_from=0;id_from<nproc;++id_from)
111:  if(N*id_from/nproc<=min_id && min_id<N*(id_from+1)/nproc) break;
112:       if(id==id_from)
113:  for(c=0;!free_color[min_id][c];++c);
114:       MPI Bcast(&c, 1, MPI INT, id from, MPI COMM WORLD);
115:       colors[c][min_id]=1;
116:       //the connected nodes' freedom decreases:
117:       for(i=startval;i<endval;++i){
118:  if(!OK[i] && free_color[i][c] && adj[i][min_id]){
119:    free_color[i][c]=0;
120:    --free_num[i];
121:  }
122:       }
123:     }
124:   }
125:   cout<<"number of DSATUR colors: "<<num_color<<endl;
126:   if(id==0){
127:     permutation_color();
128:     testperm();
129:     permto();
130:     write_pbm(file_name);
131:   }
132:   // Terminate MPI:
133:   MPI_Finalize();
134:
135: }
136:
```

```
137: void initialize(){
138:   int sum;
139:   for(int i=0;i<N;++i){
140:     sum=0;
141:     for(int j=0;j<N;++j){
142:       colors[i][j]=0;
143:       free_color[i][j]=0;
144:       sum += adj[i][j];
145:     }
146:     sat[i]=sum;
147:     free_num[i]=0;
148:     OK[i]=0;
149:   }
150: }
151:
152: void memory_init(){
153:   int i,j;
154:   adj=new char*[N];
155:   adj_matrix=new char[N*N];
156:   colors=new bool*[N];
157:   free color=new bool*[N];
158:   for(i=0;i<N;i++){
159:     adj[i]=adj_matrix+(i*N);
160:     colors[i]=new bool[N];
161:     free_color[i]=new bool[N];
162:   }
163:   sat = new int[N];
164:   free_num = new int[N];
165:   OK = new bool[N];
166:   for(i=0;i<N;++i)
167:     for(j=0;j<N;++j)
168:       adj[i][j]=0;
169: }
170:
171: void read_clq(string file_name){
172:   int i,j,n,m,x,y;
173:   string type,line;
174:   ifstream fin(file_name.c_str());
175:   if(!fin.is_open()){
176:     cout<<"ERROR opening file"<<endl;
177:     exit(1);
178:   }
179:
180:   //eliminate comment lines
181:   while(fin.peek()=='c'){
182:     getline(fin,line);
183:     comment=comment+"#"+line+'\n';
184:   }
185:
186:   fin>>type; // type of image/matrix (P1)
187:   fin>>type; // edge
188:   fin>>N; // vertexes
189:   fin>>m; // edges
190:
191:   //allocate apropriate space according to N:
192:   memory_init();
193:
194:   for(i=0;i<m;i++){
195:     fin>>type; // e
196:     if(type=="e"){
197:       fin>>x;fin>>y;
198:       //cout<<"x: "<<x<<"; y: "<<y<<endl;
199:       adj[x-1][y-1]=1;
200:       adj[y-1][x-1]=1;
201:     }
202:   }
203:   fin.close();
204: }
205:
206: void write_pbm(string file_name){
207:   int i,j;
208:   file_name=file_name+".pbm";
```

```
209:   ofstream fout(file_name.c_str());
210:   if(!fout.is_open()){
211:     cout<<"ERROR opening file"<<endl;
212:     exit(1);
213:   }
214:   fout<<"P1"<<endl;
215:   fout<<comment;
216:   fout<<N<<" "<<N<<endl;
217:
218:   for(i=0;i<N;i++){
219:     for(j=0;j<N;j++){
220:       if(adj[i][j])
221:  fout<<"1 ";
222:       else
223:  fout<<"0 ";
224:     }
225:     fout<<endl;
226:   }
227:   fout.close();
228: }
229:
230: void permutation_color(){
231:   int i,j,k, max, maxcol, sum;
232:   int *colors_sum= new int[num_color];
233:
234:   for(i=0;i<num_color;++i){
235:     sum=0;
236:     for(j=0;j<N;++j)
237:       sum += colors[i][j];
238:     colors_sum[i]=sum;
239:   }
240:   for(i=0;i<num_color;++i){
241:     max=0;
242:     for(j=0;j<num_color;++j)
243:       if(colors_sum[j]>max){
244:  max=colors_sum[j];
245:  maxcol=j;
246:       }
247:     for(j=0;j<N;++j)
248:       if(colors[maxcol][j])
249:  perm.push_back(j);
250:     colors_sum[maxcol]=0;
251:   }
252: }
253:
254: void permto(){
255:   int i,j;
256:   bool **tmp=new bool*[N];
257:   for(i=0;i<N;i++)
258:     tmp[i]=new bool[N];
259:   for(i=0;i<N;i++)
260:     for(j=0;j<N;j++)
261:       tmp[i][j]=adj[perm[i]][perm[j]];
262:   for(i=0;i<N;i++)
263:     for(j=0;j<N;j++)
264:       adj[i][j]=tmp[i][j];
265: }
266:
267: void testperm(){
268:   int sum;
269:   for(int i=0;i<N;++i){
270:     sum=0;
271:     for(int j=0;j<N;++j)
272:       if(perm[j]==i) ++sum;
273:     if(sum!=1){
274:       cerr<<"Wrong permutation!!"<<endl;
275:       exit (1);
276:     }
277:   }
278: }
```

We would like to demonstrate our ptogram with two examples. We conwerted the original clq file to a picture file of the adjacency matrix and we show the reordered by color classes picture of the same graphs. The two graphs are from the above mentioned DIMACS challenge graph set, and represent the problem of Keller's conjecture.

The DSATUR algorithm colored the keller4 graph with 25 colors:

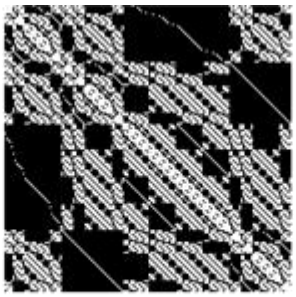

Figure 8: Original keller4 graph

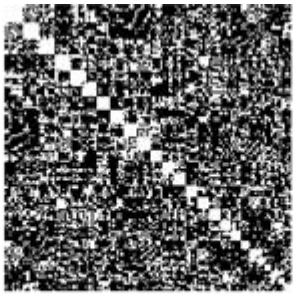

Figure 9: Reordered keller4 graph

The DSATUR algorithm colored the keller5 graph with 50 colors:

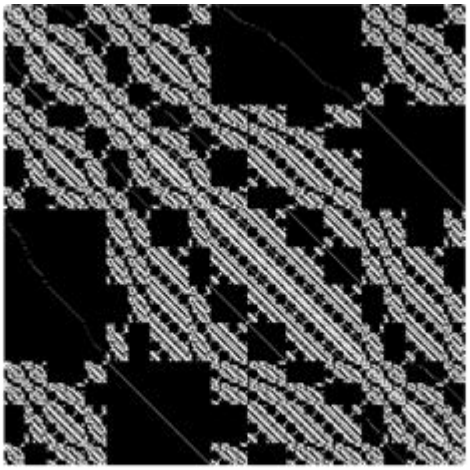

Figure 10: Original keller5 graph

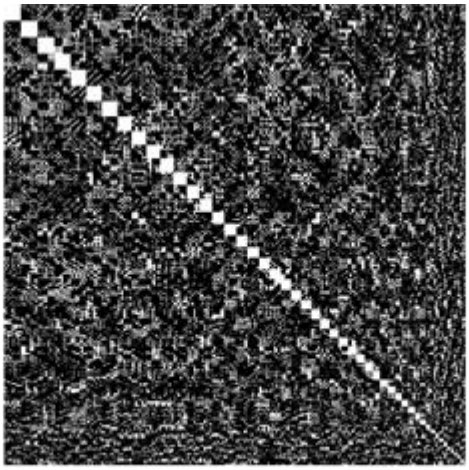

Figure 11: Reordered keller5 graph

8.5. 8.5 Usage

In this section we would like to point out the advantages of the presented solution. As we already mentioned with the help of parallel programs we can overcome two barriers: memory and time limits.

Such parallel programs that use distributed memory can be used to color much larger graphs. The parallel program will be able to use literally hundred gigabytes of memory even if one uses a classroom of PCs. One such example would be edge coloring as proposed in the article by Sandor Szabo.[Szab2012a] Edge coloring of this type is equvivalent of node coloring of a derived graph, where each edge is represented by a node. The parallel program is able to color edges of graphs where there are 200 000-2 000 000 (two million) nodes in the derived graph. This method of DSATUR coloring of edges resulted better optimum than node coloring of the same graph. Computations of this size can only be performed with parallel computation in reasonable time limits, as the coloring of biggest graphs take hours even with 100 processors.

The other aim is obviously faster computation. Some type of node colorings require much more computation for decision if a node can be put into a color class or not. An example of this coloring is the s -clique free coloring presented in the article of Sandor Szabo and Bogdan Zavalnij.[Szab2012b] Again, with s beeing bigger and bigger the computational demand of this problem quickly becomes infeasible a stand alone computer. So only parallel programs can solve the problem.

9. 9 Solving system of linear equations

In this chapter we will discuss the possibilities of solving system of linear equations with the aim of parallel programming. Systems of linear equations are used in several scientific and engineering problems, and numerical algorithms are used most frequently to solve these problems. They are used in differential equation solvers and for optimization problems as well. One of the possible solution for this problem is the Gaussian elimination, or as sometimes referred, the Gauss-Jordan elimination. We will use this algorithm in this chapter.

The modern era high performance computing quite usually concentrate on this problem. Actually, the nowadays de facto standard for supercomputing is the LINPACK Benchmark, whose aim is "to solve a dense system of linear equations". This very benchmark is used by the most famous supercomputer site, the TOP500 list (http://www.top500.org/.

In our book we do not wish to compete with the modern serial and parallel implementations of this problem but, nevertheless, would like to show, as a good example, the possibilities of the parallelization of the Gaussian elimination.

9.1. 9.1 The problem

The problem is to solve a set of simultaneous algebraic equations. We will present a basic algorithm and describe a simple parallelization. The set of simultaneous equations is:

$$\begin{aligned} a(1,1)x_1 + a(1,2)x_2 + a(1,3)x_3 + \cdots + a(1,n)x_n - b_1 = 0 \\ a(2,1)x_1 + a(2,2)x_2 + a(2,3)x_3 + \cdots + a(2,n)x_n - b_2 = 0 \\ \vdots \\ a(n,1)x_1 + a(n,2)x_2 + a(n,3)x_3 + \cdots + a(n,n)x_n - b_n = 0 \end{aligned} \tag{0.1}$$

We know initially the values of $a(i,j)$ and b_i, and the problem is to solve the equations for $x_1, x_2, \ldots, x_n$.

These equations are usually written in the matrix form

$$\mathbf{Ax} - \mathbf{b}$$

where $\mathbf{A}$ is the $n \times n$ matrix with elements $a(i,j)$, $\mathbf{x}$ is the vector of unknowns with elements $x_1, x_2, \ldots, x_n$ and $\mathbf{b}$ is the vector of known constants $b_1, b_2, \ldots, b_n$. We will first describe the general algorithm for solving the set of equations and provide an illustrative example. Then we will present a serial program for the algorithm and a parallel version.

To simplify the problem it is most efficient to incorporate the $\mathbf{b}$ vector into the array $\mathbf{A}$, so that $\mathbf{A}$ has n rows and $n+1$ columns. Let

$$\begin{aligned} a(1,n+1) &= -b_1 \\ a(2,n+1) &= -b_2 \\ &\vdots \\ a(n,n+1) &= -b_n \end{aligned}$$

Equation (9.1) become

$$\begin{aligned} a(1,1)x_1 + a(1,2)x_2 + a(1,3)x_3 + \cdots + a(1,n)x_n + a(1,n+1) = 0 \\ a(2,1)x_1 + a(2,2)x_2 + a(2,3)x_3 + \cdots + a(2,n)x_n + a(2,n+1) = 0 \\ \vdots \\ a(n,1)x_1 + a(n,2)x_2 + a(n,3)x_3 + \cdots + a(n,n)x_n + a(n,n+1) = 0 \end{aligned}$$

We will discuss the reason for writing the equations in this manner after the algorithm is described.

9.2. 9.2 Elimination

There are two steps in the algorithm: first is the elimination step, and the second is the back-substitution step. To illustrate the elimination step, consider the set of three simultaneous equations:

$$\begin{aligned} 2x_1 - 4x_2 + 24x_3 - 12 = 0 \\ 6x_1 + 18x_2 + 12x_3 + 24 = 0 \\ 3x_1 + 7x_2 - 2x_3 - 4 = 0 \end{aligned}$$

In matrix form:

$$\left[\begin{array}{ccc|c} 2 & -4 & 24 & -12 \\ 6 & 18 & 12 & 24 \\ 3 & 7 & -2 & 4 \end{array}\right]$$

First, we eliminate x_1 from two of the equations. We choose the second equation to remain in it, because the coefficient of x_1 has the biggest absolute magnitude in it. For the purpose of elimination the coefficients of that equation from which x_1 is not eliminated should be divided by the coefficient of x_1 in that equation.

The equations are usually reordered, so the equation from which x_1 is not eliminated appear at the top. The reordered equations are:

In matrix form:

$$\left[\begin{array}{rrr|r} 6 & 18 & 12 & 24 \\ 2 & -4 & 24 & -12 \\ 3 & 7 & -2 & 4 \end{array}\right]$$

Now eliminate x_1 from all but equation (9.2). Equation (9.2) called the pivot equation and the coefficient of x_1 in it called the pivot element. We must multiply the first equation by $-2/6$, that is the negative of the ratio of coefficients of x_1 in that equation, and add this equation to the equation (9.2). So after multiplying these equations become:

$$\begin{aligned} -2x_1 - 6x_2 - 4x_3 - 8 = 0 \\ 2x_1 - 4x_2 + 24x_3 - 12 = 0 \end{aligned}$$

Adding these equations the result be:

$$-10x_2 + 20x_3 - 20 = 0$$

Equations (9.2)-(9.2) are now:

$$\begin{aligned} 6x_1 + 18x_2 + 12x_3 + 24 = 0 \quad (0.1a) \\ -10x_2 + 20x_3 - 20 = 0 \quad (0.1b) \\ 3x_1 + 7x_2 - 2x_3 - 4 = 0 \quad (0.1c) \end{aligned}$$

In matrix form:

$$\left[\begin{array}{rrr|r} 6 & 18 & 12 & 24 \\ & -10 & 20 & -20 \\ 3 & 7 & -2 & 4 \end{array}\right]$$

We repeat the procedure, this time multiplying (9.2) by $-3/6$ and adding the result to (9.2). The resulting set of equations:

$$\begin{aligned} 6x_1 + 18x_2 + 12x_3 + 24 = 0 \quad (0.1a) \\ -10x_2 + 20x_3 - 20 = 0 \quad (0.1b) \\ -2x_2 - 8x_3 - 16 = 0 \quad (0.1c) \end{aligned}$$

In matrix form:

$$\left[\begin{array}{rrr|r} 6 & 18 & 12 & 24 \\ & -10 & 20 & -20 \\ & -2 & -8 & -16 \end{array}\right]$$

Next we would like to eliminate x_2 from one of the equations (9.2) or (9.2). As equation (9.2) has the larger absolute coefficient of we will eliminate from the equation (9.2), and equation (9.2) will be the pivot equation. We multiply this equation by and add the resulting equation to (9.2). This gives

$$6x_1 + 18x_2 + 12x_3 + 24 = 0 \tag{0.1a}$$
$$-10x_2 + 20x_3 - 20 = 0 \tag{0.1b}$$
$$-12x_3 - 12 = 0 \tag{0.1c}$$

In matrix form:

$$\left[\begin{array}{rrr|r} 6 & 18 & 12 & 24 \\ & -10 & 20 & -20 \\ & & -12 & -12 \end{array}\right]$$

This completes the elimination phase. The last equation (9.2) has only one unknown.

For general case, the algorithm for the elimination step is

Algorithm 1 Elimination step

1: **for** $irow \leftarrow 1, n-1$ **do**
2: $icol \leftarrow irow$
3: For column $icol$ find the maximum $|a(jrow, icol)|$ for all $jrow$ between $irow$ and n inclusive.
4: If $irow \neq jrow$ interchange the rows $irow$ and $jrow$. Row $irow$ is now the pivot row.
5: **for** $jrow \leftarrow irow + 1, n$ **do**
6: Multiply each $a(irow, kcol)$ by $-a(jrow, irow)/a(irow, irow)$ for all $kcol \geq irow$.
7: Add row $irow$ to row $jrow$. The result becomes the new row $jrow$.

At the end of the elimination step, the set of equations can be written in the form

$$\mathbf{Wx} = \mathbf{0}$$

which is in the form

$$\begin{aligned} w(1,1)x_1 + w(1,2)x_2 + w(1,3)x_3 + \cdots + +w(1,n-1)x_{n-1} + w(1,n)x_n + w(1,n+1) &= 0 \\ w(1,2)x_2 + w(1,3)x_3 + \cdots + +w(1,n-1)x_{n-1} + w(1,n)x_n + w(1,n+1) &= 0 \\ w(1,3)x_3 + \cdots + +w(1,n-1)x_{n-1} + w(1,n)x_n + w(1,n+1) &= 0 \\ \vdots & \\ w(1,n-1)x_{n-1} + w(1,n)x_n + w(1,n+1) &= 0 \\ w(1,n)x_n + w(1,n+1) &= 0 \end{aligned} \tag{0.1}$$

where for row i all $w(i,k) = 0$ if $k < i$. Thus for equation i (row i) all variables $x_j, j < i$ have been eliminated. It is this elimination step which accounts for the name of the algorithm: Gaussian elimination.

9.3. 9.3 Back substitution

In the second phase we solve the equations 9.2. As the last, n th row has only one unknown, namely x_n , it can be directly computed. Afterwards, the calculated value of x_n can be substituted into the previous equations in the $(n-1)$ st- 1 th rows. Then, since we substituted the value of x_n into the equation of the $(n-1)$ th row, there is only one unknown left, the x_{n-1} . So now, we can directly calculate the value of $(n-1)$ from the equation of the $(n-1)$ th row. And we do so with all the equations from the bottom to the top. This phase is called the back substitution because we are calculating a value of an unknown and substituting it back to the remaining equations above.

The back substitution can be demonstrated by solving the equations 9.2-9.2 in our example. First, we calculate x_3 from 9.2, which gives

$$x_3 = -1,$$

while the other two equations remain

$$6x_1 + 18x_2 + 12x_3 + 24 = 0 \tag{0.1a}$$
$$-10x_2 + 20x_3 - 20 = 0 \tag{0.1b}$$

In the back substitution step we substitute the calculated value of x_3 into these equations, and get

$$6x_1 + 18x_2 + 12 = 0 \tag{0.1a}$$
$$-10x_2 - 40 = 0 \tag{0.1b}$$

The equation 9.3 has only one unknown left, the x_2, which gives

$$x_2 = -4$$

Proceeding with back substitution of this value to equation 9.3,

$$6x_1 - 60 = 0 \tag{0.1a}$$

resulting

$$x_1 = 10$$

Thus we solved the system of equations for x_1, x_2 and x_3.

The general algorithm for back substitution is

Algorithm 1 Back substitution step

1: **for** $icol \leftarrow n, 1$ down by -1 **do**
2: $\quad x_{icol} \leftarrow -w(icol, n+1)/w(icol, icol)$
3: $\quad$ **for** $jrow \leftarrow 1, icol - 1$ **do**
4: $\quad\quad w(jrow, n+1) \leftarrow w(jrow, n+1) + a(jrow, icol)x(icol)$

0.1 The program

An example sequential program would be like presented.

In the header part we declare the **Ab** matrix and the solution vector x. The constant N specifies the size of the problem.

```
1: #include <iostream>
  2: #include <algorithm>
  3: #include <math.h>
  4: #include <iomanip>
  5: using namespace std;
  6:
  7: const int N=8;
  8: double Ab[N][N+1];
  9: //the A matrix and the b column
 10: //the N-th column is the -b values
 11:
 12: double oriAb[N][N+1]; //original Matrix Ab for testing the solution
 13: double x[N]={0}; //the solution vector
 14:
 15: void PrintMatrix();
 16: void SetMatrix();
 17: void TestSolution();
 18:
 19: int main(){
```

The int main() function, which does the calculation itself. The first part sets up the problem, the last part substitutes back the solution to the original problem. The three steps are the pivoting, the elimination - these are done in a for loop; and the back substitution. The actual $\mathbf{Ab}$ matrix with the x elements are displayed at each stage.

```
19: int main(){
 20:   int irow,jrow, j, jmax;
 21:   double t, amul, item, item_max;
 22:   time_t ts, te;
 23:
 24:   SetMatrix();
 25:   //the original matrix:
 26:   PrintMatrix();
 27:
 28:   ts=clock();
 29:   for(irow=0;irow<N-1;++irow){
 30:
 31:     //pivoting step
 32:     jmax=irow;
 33:     item_max=fabs(Ab[irow][irow]);
 34:     for(jrow=irow+1;jrow<N;++jrow)
 35:       if(fabs(Ab[jrow][irow])>item_max){
 36:  jmax=jrow;
 37:  item_max=fabs(Ab[jrow][irow]);
 38:       }
 39:     //interchange the rows, if necessary
 40:     if(jmax!=irow)
 41:       for(j=0;j<N+1;++j){
 42:  item=Ab[irow][j];
 43:  Ab[irow][j]=Ab[jmax][j];
 44:  Ab[jmax][j]=item;
 45:       }
 46:
 47:     //elimination step
 48:     t=-1.0/Ab[irow][irow];
 49:     for(jrow=irow+1;jrow<N;++jrow){
 50:       amul=Ab[jrow][irow] * t;
 51:       //elimination of the row
 52:       for(j=irow;j<N+1;++j)
 53:  Ab[jrow][j] += amul * Ab[irow][j];
 54:     }
 55:   }
 56:
 57:   //the upper triangle matrix:
 58:   PrintMatrix();
 59:
 60:   //back substitution step
 61:   for(irow=N-1;irow>=0;--irow){
 62:     x[irow]= - Ab[irow][N]/Ab[irow][irow];
```

```
 63:     for(jrow=0;jrow<irow;++jrow){
 64:       Ab[jrow][N] += x[irow] * Ab[jrow][irow];
 65:       Ab[jrow][irow]=0;
 66:     }
 67:   }
 68:
 69:   te=clock();
 70:   cout<<"time elapsed: "<<difftime(te,ts)/CLOCKS PER SEC<<endl;
 71:
 72:   //the solution matrix:
 73:   PrintMatrix();
 74:   TestSolution();
 75: }
 76:
 77: void SetMatrix(){
```

The function of setting up fills in the matrix with random elements. In this case positive values from 0 to 9 is chosen. For a realistic case one would better use something like: (double)rand()/rand() which would give us floating point numbers with uneven distribution more likely to be in the range 0 to 1. As we will see small absolute values could cause problem to the algorithm if no pivoting is used.

```
77: void SetMatrix(){
 78:   int i,j;
 79:   srand(time(NULL));
 80:   for(i=0;i<N;++i)
 81:     for(j=0;j<N+1;++j)
 82:       oriAb[i][j]=Ab[i][j]=rand()%10;
 83: }
 84:
 85: void PrintMatrix(){
```

The printing function prints the matrix and the values of x. If the matrix is too big we would not like it to be printed.

```
85: void PrintMatrix(){
 86:   //Beware, the precision is set to one digit!
 87:   //Remember this when checkning visually.
 88:   int i,j;
 89:   if(N>20){cout<<"Too big to display!"<<endl;return;}
 90:   cout.precision(1);
 91:   for(i=0;i<N+1;++i)cout<<"------";
 92:   cout<<fixed<<"-----"<<endl;
 93:   for(i=0;i<N;++i){
 94:     cout<<"| ";
 95:     for(j=0;j<N;++j)
 96:       cout<<setw(5)<<Ab[i][j]<<" ";
 97:     cout<<"| "<<setw(5)<<Ab[i][j];
 98:     cout<<" |  x["<<setw(2)<<i<<"] = "<<setw(5)<<x[i]<<endl;
 99:   }
100:   for(i=0;i<N+1;++i)cout<<"------";
101:   cout<<"-----"<<endl;
102: }
103:
104: void TestSolution(){
```

The testing function tests the solution by substiting back the values of x to the original problem. We display the difference of the sum of the row and the value of b in 20 digit precision, to see the round off errors as well. The function alerts us if the difference is too big, which would mean a possible error in the algorithm.

```
104: void TestSolution(){
105:   int i,j;
106:   double diff, sum;
107:   cout.precision(20);
108:   for(i=0;i<N;++i){
109:     sum=0;
110:     for(j=0;j<N;++j)
111:       sum += x[j] * oriAb[i][j];
112:     diff=sum+oriAb[i][N];
```

```
113:     if(diff>0.0001 || diff<-0.0001)
114:       cout<<"ERROR! "<<sum<<" ~ "<<oriAb[i][N]<<", diff:"<<diff<<endl;
115:     if(N<50){
116:       cout<<setw(4)<<sum<<" ~ "<<setw(4)<<oriAb[i][N];
117:       cout<<", diff:"<<setw(4)<<fixed<<diff<<endl;
118:     }
119:   }
120: }
```

A probable output of the program for $N = 8$ would look like this:

```
The original matrix:
-------------------------------------------------------------
|   5.0   2.0   7.0   3.0   4.0   2.0   1.0   7.0 |   6.0 |  x[ 0] =   0.0
|   6.0   2.0   9.0   1.0   3.0   4.0   4.0   8.0 |   8.0 |  x[ 1] =   0.0
|   8.0   2.0   9.0   8.0   8.0   4.0   1.0   9.0 |   4.0 |  x[ 2] =   0.0
|   3.0   0.0   7.0   2.0   5.0   1.0   2.0   1.0 |   7.0 |  x[ 3] =   0.0
|   4.0   2.0   6.0   2.0   1.0   8.0   1.0   2.0 |   2.0 |  x[ 4] =   0.0
|   8.0   8.0   2.0   8.0   9.0   6.0   9.0   7.0 |   4.0 |  x[ 5] =   0.0
|   4.0   8.0   5.0   8.0   4.0   6.0   7.0   6.0 |   3.0 |  x[ 6] =   0.0
|   1.0   8.0   4.0   8.0   4.0   7.0   5.0   7.0 |   8.0 |  x[ 7] =   0.0
-------------------------------------------------------------
The upper triangular matrix:
-------------------------------------------------------------
|   8.0   2.0   9.0   8.0   8.0   4.0   1.0   9.0 |   4.0 |  x[ 0] =   0.0
|   0.0   7.8   2.9   7.0   3.0   6.5   4.9   5.9 |   7.5 |  x[ 1] =   0.0
|   0.0   0.0  -9.2  -5.4  -1.3  -3.0   4.2  -6.5 |  -5.8 |  x[ 2] =   0.0
|   0.0   0.0   0.0  -6.7  -3.5  -0.1   3.9  -0.6 |   3.2 |  x[ 3] =   0.0
|   0.0   0.0   0.0   0.0   3.1  -1.1   2.4  -4.3 |   2.5 |  x[ 4] =   0.0
|   0.0   0.0   0.0   0.0   0.0   4.2  -0.4  -6.1 |  -2.0 |  x[ 5] =   0.0
|   0.0   0.0   0.0   0.0  -0.0   0.0  -1.9  -1.2 |  -0.4 |  x[ 6] =   0.0
|   0.0   0.0   0.0   0.0   0.0   0.0   0.0  -8.5 |  -4.7 |  x[ 7] =   0.0
-------------------------------------------------------------
time elapsed: 0.0
The solution matrix:
-------------------------------------------------------------
|   8.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0 | -12.2 |  x[ 0] =   1.5
|   0.0   7.8   0.0   0.0   0.0   0.0   0.0   0.0 |   6.4 |  x[ 1] =  -0.8
|   0.0   0.0  -9.2   0.0   0.0   0.0   0.0   0.0 |  -6.7 |  x[ 2] =  -0.7
|   0.0   0.0   0.0  -6.7   0.0   0.0   0.0   0.0 |  10.4 |  x[ 3] =   1.6
|   0.0   0.0   0.0   0.0   3.1   0.0   0.0   0.0 |   5.6 |  x[ 4] =  -1.8
|   0.0   0.0   0.0   0.0   0.0   4.2   0.0   0.0 |   1.3 |  x[ 5] =  -0.3
|   0.0   0.0   0.0   0.0  -0.0   0.0  -1.9   0.0 |   0.3 |  x[ 6] =   0.1
|   0.0   0.0   0.0   0.0   0.0   0.0   0.0  -8.5 |  -4.7 |  x[ 7] =  -0.6
-------------------------------------------------------------
-6.0000000000000000000 ~ 6.0000000000000000000, diff:0.0000000000000000000
-8.0000000000000000000 ~ 8.0000000000000000000, diff:0.0000000000000000000
-4.0000000000000000000 ~ 4.0000000000000000000, diff:0.0000000000000000000
-7.0000000000000000000 ~ 7.0000000000000000000, diff:0.0000000000000000000
-2.0000000000000000000 ~ 2.0000000000000000000, diff:0.0000000000000000000
-3.9999999999999822364 ~ 4.0000000000000000000, diff:0.0000000000000177636
-2.9999999999999955591 ~ 3.0000000000000000000, diff:0.0000000000000044409
-8.0000000000000000000 ~ 8.0000000000000000000, diff:0.0000000000000000000
```

The problem is especially problematic and calculation intense. It can be seen that the three nested for-loops will cause the running time be the magnitude of $O(n^3)$. In fact a decent modern computer will solve a dense equation system of size $10\,000 \times 10\,000$, that means 10 000 equations and variables in about 10-30 minutes depending on the computer. This demonstrates the need for parallelization.

9.4. 9.4 Parallelization

For the purpose of parallelization we need to distribute the work between more processes. The two steps of the algorithm - the elimination with pivoting and the back substitution - are clearly not equivalent in time complexity. The back substitution has $O(n^2)$, while the elimination has $O(n^3)$ complexity. This means that

there is little use in writing a parallel program for the back substitution as the other part dominates the running time.

The distribution of the work may be done by assigning each process to some of the rows, as elimination will be made between the pivot row and all the other rows. So the processes will make the elimination on some of the rows, which are assigned to them. This leads us to four problems.

First, when we need to interchange the 'next' row with the pivot row - the pivoting step. In some cases the interchange will be done between rows assigned to the same process, in other cases between rows assigned to different processes. In both cases we will need to find the maximum pivot element and the process to which it is assigned. Also, in the latter case we will need some interprocess communication for interchanging where we can use the just calculated process ids.

Second, we need to distribute the pivot row. This can be done easyly with an MPI_Bcast.

Third, we need to distribute the rows for even work. This problem is a little tricky. If we use block scheduling, then from the p processes the process to which the first, say $1/p$ part of the rows assigned will finish its work very soon, in $1/p$ time, and will have no work in the future. This scheduling is clearly unefficient. We cannot either use dynamic scheduling, as the rows from which the pivot row should be substracted must be present at the process memory, that means scheduling the rows in prior. The third option is the loop splitting. Let us remind of the main loop of the elimination:

```
48:     t=-1.0/Ab[irow][irow];
 49:     for(jrow=irow+1;jrow<N;++jrow){
 50:       amul=Ab[jrow][irow] * t;
 51:       //elimination of the row
 52:       for(j=irow;j<N+1;++j)
 53:  Ab[jrow][j] += amul * Ab[irow][j];
 54:     }
```

With usual loop splitting we would get the following:

```
t=-1.0/Ab[irow][irow];
for(jrow=irow+1+id;jrow<N;jrow += nproc){
  amul=Ab[jrow][irow] * t;
  //elimination of the row
  for(j=irow;j<N+1;++j)
    Ab[jrow][j] += amul * Ab[irow][j];
}
```

But actually this will lead us to a problem. Depending on the value of irow there will be different rows assigned to one process. We can show it in a small table, where $nproc = 3$ and the number of rows is 10. We indicate the row numbers assigned to different processes:

Table 1: Assigned rows with usual loop splitting

irow=	id=0	id=1	id=2
0	1,4,7	2,5,8	3,6,9
1	2,5,8	3,6,9	4,7
2	3,6,9	4,7	5,8
3	4,7	5,8	6,9
4	5,8	6,9	7
5	6,9	7	8
6	7	8	9
7	8	9	
8	9		

Although the distribution is even, the rows assigned to one process differ from one loop to another. Which means interchanging the values of the rows at the beginning of each loop, that would cause intolerable time

delay. (The same problem arises with the implementation of Gaussian elimination on shared memory systems, where this loop splitting scheme would lead to inefficient cache usage, and one needs to deal with it the same way as described below.)

In order to make the loop assign the same rows to the same process in different runs we reorder the inner loop from:

```
for(jrow=irow+1;jrow<N;++jrow)
```

to:

```
for(jrow=N-1;jrow>=irow+1;--jrow)
```

This is the same loop, just in decreasing order. This one can be loop split in usual way:

```
98:     for(jrow=N-1-id;jrow>=irow+1;jrow-=nproc){
 99:        amul=Ab[jrow][irow] * t;
100:        //elimination of the row
101:        for(j=irow;j<N+1;++j)
102:  Ab[jrow][j] += amul * Ab[irow][j];
103:     }
104:   }
```

The main difference is the assigned rows. Let us show the previous table with this scheduling. The gain is obvious, as we assign the same rows to the same processes at each loop:

Table 2: Assigned rows with modified loop splitting

irow=	id=0	id=1	id=2
0	9,6,3	8,5,2	7,4,1
1	9,6,3	8,5,2	7,4
2	9,6,3	8,5	7,4
3	9,6	8,5	7,5
4	9,6	8,5	7
5	9,6	8	7
6	9	8	7
7	9	8	
8	9		

The fourth and last problem is to collect the 'ready' rows by the master process that will perform the back substitution. Note, that during the elimination step in the beginning of each loop the pivot row was broadcasted, and this pivot row at that time was already 'ready'. So if the master process do not throw out the broadcasted pivot rows, but instead saves them, at the end it will have the upper triangular matrix ready. Obviously this means more intensive memory usage, as if the processes - including the master - save only the rows assigned to them. In that case at the end the master would need to collect this data. We, in our example, followed the simpler way, and let all the processes save the whole matrix, and update only those rows, that assigned to that very process. Also, this version is closer to the original program, as we will store the broadcasted pivot row in its own place, and can modify the program as little as possible. It does not only help to understand the parallel program better, but also helps us to construct a correct program. Do not forget that constructing a correct program is more important than gaining a few percent of running time even in High Performance Computing.

9.5. 9.5 Parallel program

After the previous considerations we present the program itself. As the called functions are absolutely the same as in the serial program, we will not show them again, but only the main function. In the beginning we have the declaration of variables, the usual starting of MPI framework, and for the master process (id==0) we start the measurement of time, construct the Ab matrix, print it, and broadcast it to all other processes.

```
20: int main(int argc, char **argv){
 21:   int id, nproc, id_from, id_to;
 22:   MPI_Status status;
 23:   // Initialize MPI:
 24:   MPI_Init(&argc, &argv);
 25:   // Get my rank:
 26:   MPI_Comm_rank(MPI_COMM_WORLD, &id);
 27:   // Get the total number of processors:
 28:   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 29:
 30:   int irow,jrow, i,j, jmax;
 31:   double t, amul, item, item_max;
 32:   time t ts, te;
 33:
 34:   if(id==0){
 35:   SetMatrix();
 36:   cout<<"The original matrix:"<<endl;
 37:   PrintMatrix();
 38:
 39:   ts=MPI_Wtime();
 40:   }
 41:   MPI_Bcast(Ab, N*(N+1), MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Next, in the main loop for irow we first must do the pivoting. For this purpose we need to find out which process has the pivot row, and store this in variable id_from. Before this process broadcasts the pivot row, we need to find and interchange the right pivot row with the next row. For this purpose the, id_from process broadcasts the absolute value of the pivot element.

```
43:   for(irow=0;irow<N-1;++irow){
 44:
 45:     //the thread who has the next row to eliminate
 46:     //he will broadcast this row to all others
 47:     id_from=(N-1-irow)%nproc;
 48:
 49:     //pivoting step
 50:     jmax=irow;
 51:     item_max=fabs(Ab[irow][irow]);
 52:     MPI_Bcast(&item_max, 1, MPI_DOUBLE, id_from, MPI_COMM_WORLD);
```

Then the processes find the local maximum, and with MAXLOC reduction, already described in previous chapters, we find the global maximum absolute value pivot element, and the number of its row, and the ID of the process which have this row that will be stored in variable id_to.

```
54:     for(jrow=N-1-id;jrow>=irow+1;jrow-=nproc)
 55:       if(fabs(Ab[jrow][irow])>item_max){
 56:  jmax=jrow;
 57:  item_max=fabs(Ab[jrow][irow]);
 58:       }
 59:     //interchange the rows, if necessary
 60:     struct{
 61:       double item;
 62:       int row;
 63:     } p, tmp_p;
 64:     p.item=item_max; p.row=jmax;
 65:     MPI_Allreduce(&p, &tmp_p, 1, MPI_DOUBLE_INT, MPI_MAXLOC, MPI_COMM_WORLD);
 66:     jmax=tmp p.row;
 67:     //the rank of the other slave with whom the pivot must be exchanged:
 68:     id_to=(N-1-jmax)%nproc;
```

If the two rows must be interchanged at the same process, we do the same as in the sequential program:

```
70:     if(id_from==id_to){ //replacement at the same slave: no communication
 71:       if(id==id_from && jmax!=irow)
 72:   for(j=0;j<N+1;++j){
 73:     item=Ab[irow][j];
 74:     Ab[irow][j]=Ab[jmax][j];
 75:     Ab[jmax][j]=item;
 76:   }
```

If the two rows in question reside at different processes, then we need to interchange them by message passing. The process id_to copies the pivot row to row irow, then sends it to the other process. The process id_from saves the irow row to the place of the found pivot row, receives the irow-th row from the other process, and sends back the saved row. Thus the two rows interchanged at both processes.

```
77:     }else{ //communication is needed for pivot exchange
 78:       if(id==id_to){
 79:   for(j=irow;j<N+1;++j)
 80:     Ab[irow][j]=Ab[jmax][j];
 81:   MPI_Send(&Ab[irow], N+1, MPI_DOUBLE, id_from, 77, MPI_COMM_WORLD);
 82:   MPI_Recv(&Ab[jmax], N+1, MPI_DOUBLE, id_from, 88,
 83:     MPI_COMM_WORLD, &status);
 84:   }
 85:       if(id==id_from){
 86:   for(j=irow;j<N+1;++j)
 87:     Ab[jmax][j]=Ab[irow][j];
 88:   MPI_Recv(&Ab[irow], N+1, MPI_DOUBLE, id_to, 77,
 89:     MPI_COMM_WORLD, &status);
 90:   MPI_Send(&Ab[jmax], N+1, MPI_DOUBLE, id_to, 88, MPI_COMM_WORLD);
 91:       }
 92:     }
```

After the necessary interchange the pivot row is broadcasted to all processes, and the processes perform the elimination on rows assigned to them. This ends the elimination loop.

```
94:     MPI_Bcast(&Ab[irow], N+1, MPI_DOUBLE, id_from, MPI_COMM_WORLD);
 95:
 96:     //elimination step
 97:     t=-1.0/Ab[irow][irow];
 98:     for(jrow=N-1-id;jrow>=irow+1;jrow-=nproc){
 99:       amul=Ab[jrow][irow] * t;
100:       //elimination of the row
101:       for(j=irow;j<N+1;++j)
102:   Ab[jrow][j] += amul * Ab[irow][j];
103:     }
104:   }
```

At the end the master process prints the upper triangular matrix and performes the back substitution the same way as in the sequential program. Finaly, the measured execution time is printed with the substituted matrix. We also check the solution here. The MPI_Finalize() call ends the MPI framework.

```
106:   if(id==0){
107:     cout<<"The upper triangular matrix:"<<endl;
108:     PrintMatrix();
109:
110:     //back substitution step
111:     for(irow=N-1;irow>=0;--irow){
112:       x[irow]= - Ab[irow][N]/Ab[irow][irow];
113:       for(jrow=0;jrow<irow;++jrow){
114:   Ab[jrow][N] += x[irow] * Ab[jrow][irow];
115:   Ab[jrow][irow]=0;
116:       }
117:     }
118:     te=MPI_Wtime();
119:     cout<<"time elapsed: "<<te-ts<<endl;
120:
121:     cout<<"The solution matrix:"<<endl;
122:     PrintMatrix();
123:     TestSolution();
```

```
124:   }
125:
126:   // Terminate MPI:
127:   MPI_Finalize();
128: }
```

The whole main function altogether is:

```
1: #include <iostream>
  2: #include <algorithm>
  3: #include <math.h>
  4: #include <iomanip>
  5: #include <mpi.h>
  6: using namespace std;
  7:
  8: const int N=8000;
  9: double Ab[N][N+1];
 10: //the A matrix and the b column
 11: //the N-th column is the -b values
 12:
 13: double oriAb[N][N+1]; //original Matrix Ab for testing the solution
 14: double x[N]={0}; //the solution vector
 15:
 16: void PrintMatrix();
 17: void SetMatrix();
 18: void TestSolution();
 19:
 20: int main(int argc, char **argv){
 21:   int id, nproc, id_from, id_to;
 22:   MPI_Status status;
 23:   // Initialize MPI:
 24:   MPI_Init(&argc, &argv);
 25:   // Get my rank:
 26:   MPI Comm rank(MPI COMM WORLD, &id);
 27:   // Get the total number of processors:
 28:   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 29:
 30:   int irow,jrow, i,j, jmax;
 31:   double t, amul, item, item_max;
 32:   time_t ts, te;
 33:
 34:   if(id==0){
 35:   SetMatrix();
 36:   cout<<"The original matrix:"<<endl;
 37:   PrintMatrix();
 38:
 39:   ts=MPI Wtime();
 40:   }
 41:   MPI_Bcast(Ab, N*(N+1), MPI_DOUBLE, 0, MPI_COMM_WORLD);
 42:
 43:   for(irow=0;irow<N-1;++irow){
 44:
 45:     //the thread who has the next row to eliminate
 46:     //he will broadcast this row to all others
 47:     id_from=(N-1-irow)%nproc;
 48:
 49:     //pivoting step
 50:     jmax=irow;
 51:     item_max=fabs(Ab[irow][irow]);
 52:     MPI_Bcast(&item_max, 1, MPI_DOUBLE, id_from, MPI_COMM_WORLD);
 53:
 54:     for(jrow=N-1-id;jrow>=irow+1;jrow-=nproc)
 55:       if(fabs(Ab[jrow][irow])>item_max){
 56:  jmax=jrow;
 57:  item_max=fabs(Ab[jrow][irow]);
 58:       }
 59:     //interchange the rows, if necessary
 60:     struct{
 61:       double item;
 62:       int row;
 63:     } p, tmp_p;
```

```
 64:     p.item=item_max; p.row=jmax;
 65:     MPI_Allreduce(&p, &tmp_p, 1, MPI_DOUBLE_INT, MPI_MAXLOC, MPI_COMM_WORLD);
 66:     jmax=tmp_p.row;
 67:     //the rank of the other slave with whom the pivot must be exchanged:
 68:     id_to=(N-1-jmax)%nproc;
 69:
 70:     if(id_from==id_to){ //replacement at the same slave: no communication
 71:       if(id==id from && jmax!=irow)
 72:   for(j=0;j<N+1;++j){
 73:     item=Ab[irow][j];
 74:     Ab[irow][j]=Ab[jmax][j];
 75:     Ab[jmax][j]=item;
 76:   }
 77:     }else{ //communication is needed for pivot exchange
 78:       if(id==id_to){
 79:   for(j=irow;j<N+1;++j)
 80:     Ab[irow][j]=Ab[jmax][j];
 81:   MPI_Send(&Ab[irow], N+1, MPI_DOUBLE, id_from, 77, MPI_COMM_WORLD);
 82:   MPI_Recv(&Ab[jmax], N+1, MPI_DOUBLE, id_from, 88,
 83:     MPI_COMM_WORLD, &status);
 84:   }
 85:       if(id==id_from){
 86:   for(j=irow;j<N+1;++j)
 87:     Ab[jmax][j]=Ab[irow][j];
 88:   MPI_Recv(&Ab[irow], N+1, MPI_DOUBLE, id_to, 77,
 89:     MPI_COMM_WORLD, &status);
 90:   MPI_Send(&Ab[jmax], N+1, MPI_DOUBLE, id_to, 88, MPI_COMM_WORLD);
 91:       }
 92:     }
 93:
 94:     MPI_Bcast(&Ab[irow], N+1, MPI_DOUBLE, id_from, MPI_COMM_WORLD);
 95:
 96:     //elimination step
 97:     t=-1.0/Ab[irow][irow];
 98:     for(jrow=N-1-id;jrow>=irow+1;jrow-=nproc){
 99:       amul=Ab[jrow][irow] * t;
100:       //elimination of the row
101:       for(j=irow;j<N+1;++j)
102:   Ab[jrow][j] += amul * Ab[irow][j];
103:     }
104:    }
105:
106:    if(id==0){
107:     cout<<"The upper triangular matrix:"<<endl;
108:     PrintMatrix();
109:
110:     //back substitution step
111:     for(irow=N-1;irow>=0;--irow){
112:       x[irow]= - Ab[irow][N]/Ab[irow][irow];
113:       for(jrow=0;jrow<irow;++jrow){
114:   Ab[jrow][N] += x[irow] * Ab[jrow][irow];
115:   Ab[jrow][irow]=0;
116:       }
117:     }
118:     te=MPI_Wtime();
119:     cout<<"time elapsed: "<<te-ts<<endl;
120:
121:     cout<<"The solution matrix:"<<endl;
122:     PrintMatrix();
123:     TestSolution();
124:    }
125:
126:    // Terminate MPI:
127:    MPI_Finalize();
128: }
129:
130: void SetMatrix(){
131:   int i,j;
132:   srand(time(NULL));
133:   for(i=0;i<N;++i)
134:     for(j=0;j<N+1;++j)
135:       oriAb[i][j]=Ab[i][j]=rand()%10;
```

```
136: }
137:
138: void PrintMatrix(){
139:   //Beware, the precision is set to one digit!
140:   //Remember this when checkning visually.
141:   int i,j;
142:   if(N>20){cout<<"Too big to display!"<<endl;return;}
143:   cout.precision(1);
144:   for(i=0;i<N+1;++i)cout<<"------";
145:   cout<<fixed<<"-----"<<endl;
146:   for(i=0;i<N;++i){
147:     cout<<"| ";
148:     for(j=0;j<N;++j)
149:       cout<<setw(5)<<Ab[i][j]<<" ";
150:     cout<<"| "<<setw(5)<<Ab[i][j];
151:     cout<<" |  x["<<setw(2)<<i<<"] = "<<setw(5)<<x[i]<<endl;
152:   }
153:   for(i=0;i<N+1;++i)cout<<"------";
154:   cout<<"-----"<<endl;
155: }
156:
157: void TestSolution(){
158:   int i,j;
159:   double diff, sum;
160:   cout.precision(20);
161:   for(i=0;i<N;++i){
162:     sum=0;
163:     for(j=0;j<N;++j)
164:       sum += x[j] * oriAb[i][j];
165:     diff=sum+oriAb[i][N];
166:     if(diff>0.0001 || diff<-0.0001)
167:       cout<<"ERROR! "<<sum<<" ~ "<<oriAb[i][N]<<", diff:"<<diff<<endl;
168:     if(N<50){
169:       cout<<setw(4)<<sum<<" ~ "<<setw(4)<<oriAb[i][N];
170:       cout<<", diff:"<<setw(4)<<fixed<<diff<<endl;
171:     }
172:   }
173: }
```

9.6. 9.6 Running times

We measured the running times for 10 000 variables and equations on the two supercomputers.

Table 3: Gaussian elimination with pivoting

nproc	A		B	
1	665s		1510s	
12	184s	3.6x	339s	4.5x
24	103s	6.5x	221s	6.8x
48	61s	10.9x	118s	12.8x
96	49s	13.6x	136s	11.1x
192	31s	21.5x	161s	9.4x
384	29s	22.9x	192s	7.9x

As one can see there is a moderate speed-up, which has its limits. For larger problems the speed-up would be greater, as elimination would take up more time than communication. Though we make too much communication altogether. One may think that the interchanging step at the beginning of each loop is taking too long time, so we measured the same program without the pivot interchanging step as well.

Table 4: Gaussian elimination without pivoting

nproc	A		B	
1	614s		1498s	
12	186s	3.3x	271s	5.5x
24	96s	6.4x	173s	8.7x
48	52s	11.8x	94s	15.9x
96	34s	18.1x	73s	20.5x
192	29s	21.2x	66s	22.7x
384	34s	18.1x	49s	30.6x

Clearly the speed-ups are a little better, but still not too bright. Clearly, broadcasting the entire row at the beginning of each loop takes up too much time.

It is possible to reconstruct the the program, that way much less message exchanges are required. In this case we need to assign the processes not rows but columns, as elimination is always done between elements of the same column. There still must be some communication, as for the interchange of the pivot row one process must tell the others which rows must be interchanged, but after this all work can be done locally. Obviously we also need to deal with the fourth problem of collecting the data to the master process. Such modification of the program can be a good assignment for learning purposes.

9.7. 9.7 The role of pivoting

When we chose the pivot element, we always chose from the x_i coefficients the one with the biggest absolute value. This equation we called the pivot equation and we called this coefficient the pivot element. Then we interchanged the equations, so that the pivot equation should be the next one in the procedure.

There are two reasons for choosing such pivot element. First reason is that we must multiply the pivot (i th) equation before adding it to the j th equation with the constant $-a(i,j)/a(i,i)$. This constant can be calculated only if $a(i,i)$ is not equal to zero. There are two possibilities. If there is at least one co-efficient of x_i which is not zero, than choosing the pivot being the biggest absolute value will choose a co-efficient different from zero, and save us from division by zero. If there is no such co-efficient, than the system of equations cannot be solved explicitly, because there are less equations defining x_i than there should be. (Because of this phenomenon and aim of simplicity we didn't check this possibility in our program. Obviously the proper program should deal with such case.)

The second reason is that in the Gaussian elimination one would desire to improve the numerical stability. The following system is dramatically affected by round-off error when Gaussian elimination and backwards substitution are performed.

$$\begin{aligned} 0.003x_1 + 59.14x_2 + 59.17 &= 0 \\ 5.291x_1 - \ 6.13x_2 + 46.78 &= 0 \end{aligned}$$

In matrix form:

$$\left[\begin{array}{cc|c} 0.003 & 59.14 & 59.17 \\ 5.291 & -6.13 & 46.78 \end{array}\right]$$

This system has the exact solution of $x_1 = 10.00$ and $x_2 = 1.000$, but when the elimination algorithm and backwards substitution are performed using four-digit arithmetic, the small value of $a(1,1)$ causes small round-off errors to be propagated. The algorithm without pivoting yields the approximation of $x_1 = 9873.3$ and $x_2 = 4$. In this case it is desirable that we interchange the two rows so that $a(2,1)$ is in the pivot position, as it has the bigger absolute value.

$$5.291x_1 - \ 6.13x_2 + 46.78 = 0$$
$$0.003x_1 + 59.14x_2 + 59.17 = 0$$

In matrix form:

$$\left[\begin{array}{cc|c} 5.291 & -6.13 & 46.78 \\ 0.003 & 59.14 & 59.17 \end{array}\right]$$

Considering this system, the elimination algorithm and backwards substitution using four-digit arithmetic yield the correct values $x_1 = 10.00$ and $x_2 = 1.000$.

10. 10 Heating a plate

A further good example for parallelization is the heated plate model. Let's consider a plate of a certain $X \times Y$ size. We would like to model the thermal energy propagation in a time when we heat one edge of the plate. For our example, ideal assumptions can be made, so the density of the plate is uniform, the heat propagation parameters of the material are also uniform, the insulation at the edges is perfect, etc.. An analytical solution would be to solve the two dimensional heat equation, where the new temperature is calculated with the help of the Laplacian operator and the current temperature. According to our ideal assumptions and the basic model, we can give a numerical solution to this problem as well. This solution is an iterative calculation, where actual heat values are used for calculating new heat values. The main idea is very simple. We can model the heat propagation by going through all points and averaging the heat value of surrounding points. This average will be the new heat value for the actual point.

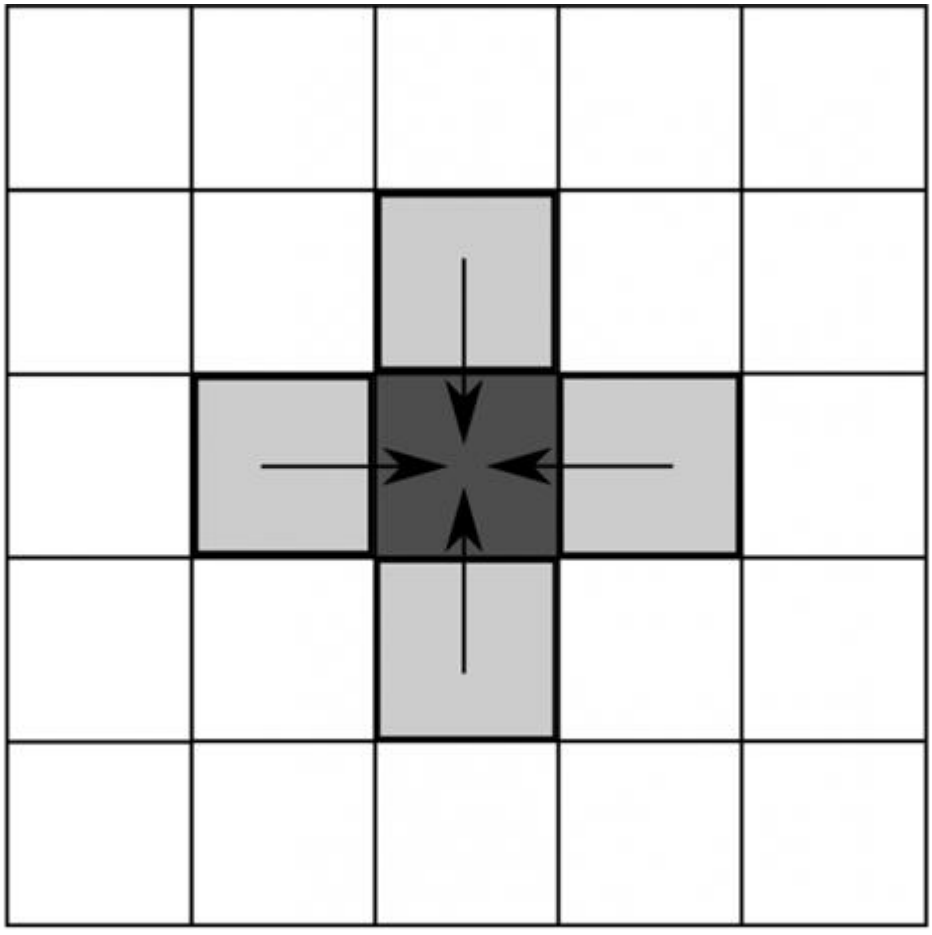

Figure 12: Averaging pattern of the heat propagation model

If we talk about an image, this method would give some sort of blur effect. Since one edge (or some pixels at the edge) is heated, its value will be constant, thus, this input energy will propagate inside the plate. If we run the iteration long enough, the value of the heated edge would reach all points of the plate. Under real conditions we should consider the leaking of the heat to the environment, but in our model we ignore this. We don't need to get too many surrounding points to get good results. Actually, it is enough to use the upper, lower, right and left neighbours of all points (at the edges, we leave out the corresponding non-existing neighbours). We have to run the iteration until we reach the appropriate condition. This could be a given small measure of change, thus we stop when change decreases below a given level. The condition could be also a given heat value at a given position. For our example, we will use a simple fixed length iteration so we won't loose focus on our parallelization problem.

10.1. 10.1 Sequential calculation method

For the averaging calculation we have to go through all points of the array. At every point, we have to use its neighbours. For that we have to define two arrays with the dimensions of X and Y. One array stores the actual point-wise heat values of the plate, and the other will store the new, calculated values. This is necessary, because in every time-step, we need more of the original values to be able to calculate several of the new ones, so we are not allowed to overwrite actual values during calculation. After we are done with all points, we can copy the new values to the actual array and begin a new iteration. Special cases are the edges, where only three neighbours are present, and the corners, where only two neighbours have to be considered.

10.2. 10.2 Parallel method

The high number of points we work with suggests that parallelization would speed up the calculation. Since every point needs values from its small neighbourhood, it seems to be appropriate to share the work between parallelly working nodes.

The parallel version seems to be straightforward. We have to cut our plate into slices. Every slice will be processed by a dedicated working node. However, there are some issues we need to consider before we start.

10.3. 10.3 Mapping - sharing the work

The first question could be: how should we split the data between working nodes? We should give continuous blocks, since we have to work with horizontal and vertical neighbours. It is also evident, that we should give only whole lines to the workers, otherwise data exchange would become pretty complicated. But how many lines should one node work on? If the number of lines is divisible by working nodes, we just have to hand out equal numbers of lines to each node. If it is not, then we have further options. One option could be, that we do not use all of our free nodes, just as many as we need to divide the number of lines without a residual. It can be a big waste, if we have some nodes, e.g. 7 and we have to drop out 2 or 3. But why should we drop any nodes? We could hand out the residual lines for nodes, to make the extra work. The speed will depend on the slowest node, so we have to try for an equal distribution of workload. In general, mapping should be carefully planned depending on the characteristics of the work. Mapping can affect speed, so at parallelization, it is a key question of efficiency. Static mapping techniques do the mapping before the main algorithm runs. Dynamic mapping techniques do this during the execution of the main algorithm. For this example, we choose a static mapping technique, where static sets of data are distributed between working nodes.

10.4. 10.4 Communicating border data

For every point, we need its neighbours to calculate the new heat value. Since we distributed whole lines for the working nodes, the right and left neighbours are at hand for any working node. However, the upper neighbours of our first line or the bottom neighbours of our last line are at different nodes, so we have to collect them after every iteration step to get the new heat values required for the next iteration step. If one working node has the first n lines of all, it has to collect the $n+1$ line for the next calculation from it's bottom neighbour node. Middle position working nodes need to do the same with the line before their first lines too. These lines gathered

from neighbour nodes are often called "ghost lines", because they are only virtually present and have to be updated through inter-nodal communication.

10.5. 10.5 Example code, linear solution

This section shows a simple plate heating simulation code implemented in C++ according the above guidelines and ideas.

The first lines are libraries which are necessary for functions for I/O operations, mathematical calculations and for being able to use the vector type. This latter adds the convenience of dynamic length over normal arrays.

After this section we define some constant values. X, Y are the dimensions of our plate in points, thus in our model we use an array to represent the plate. The heatp value represents the temperature level of our heated points. The iter_n constant shows how many iterations will be made and the speed constant tunes the speed of the heat propagation, thus representing a multiplying parameter at the averaging.

The actual and new heat values are stored in the old_array and new_array arrays.

```
1: #include <iostream>
  2: #include <fstream>
  3: #include <sstream>
  4: #include <math.h>
  5: #include <vector>
  6:
  7: using namespace std;
  8: const int X=60;
  9: const int Y=60;
 10: const double heatp=100;
 11: const int iter_n=300;
 12: const double speed=1;
 13:
 14: double old_array[X][Y] = {0};
 15: double new_array[X][Y] = {0};
 16:
 17:
```

The iteration() function calculates the the new values of the new_array() array from the old_array() values for every point. The calculation runs in an embedded iteration between 0-X and 0-Y.

After calculating the new values, they are copied back to the old_array() To keep the heating value at a constant rate, we change the heat value of three of the upper left points to the constant value heatp.

```
18: void iteration()
 19: {
 20:  int i=0;
 21:  int j=0;
 22:  int a,w,d,x;
 23:  for(i=0;i<X;i++)
 24:  {
 25:     for(j=0;j<Y;j++)
 26:     {
 27:         // If at an edge
 28:      if( i>0 && i<X-1 && j>0 && j<Y-1)
 29:      {
 30:          new_array[i][j] = (old_array[i-1][j] + old_array[i+1][j] + old_array[i][j-
1] + old_array[i][j+1] ) / 4 * speed;
 31:      }
 32:      // if at upper edge
 33:      if( j==0 && i>0 && i<X-1 )
 34:      {
 35:         new_array[i][j] = ( old_array[i-1][j] + old_array[i+1][j] +
old_array[i][j+1] ) / 3 * speed;
 36:      }
 37:      //if at right edge
 38:      if( i==0 && j>0 && j<Y-1 )
```

```
 39:      {
 40:         new_array[i][j] = ( old_array[i][j-1] + old_array[i][j+1] +
old_array[i+1][j] ) / 3 * speed;
 41:      }
 42:      // if at bottom edge
 43:      if( j==Y-1 && i>0 && i<X-1 )
 44:      {
 45:         new array[i][j] = ( old array[i-1][j] + old array[i+1][j] + old array[i][j-
1] ) / 3 * speed;
 46:      }
 47:      // if at left edge
 48:      if( i==X-1 && j>0 && j<Y-1 )
 49:      {
 50:         new_array[i][j] = ( old_array[i][j-1] + old_array[i][j+1] + old_array[i-
1][j] ) / 3 * speed;
 51:      }
 52:      // we have to handle the four squares
 53:      if( i==0 && j==0)
 54:      {
 55:         new_array[i][j]=(old_array[i+1][j]+old_array[i][j+1] ) / 2 * speed;
 56:      }
 57:      if( i==0 && j==Y-1)
 58:      {
 59:        new_array[i][j]=(old_array[i+1][j]+old_array[i][j-1] ) / 2 * speed;
 60:      }
 61:      if( i==X-1 && j==0)
 62:      {
 63:        new_array[i][j]=(old_array[i-1][j]+old_array[i][j+1] ) / 2 * speed;
 64:      }
 65:      if( i==X-1 && j==Y-1)
 66:      {
 67:        new_array[i][j]=(old_array[i-1][j]+old_array[i][j-1] ) / 2 * speed;
 68:      }
 69:     }
 70:  }
 71:
 72:  for(i=0;i<X;i++)
 73:  {
 74:     for(j=0;j<Y;j++)
 75:     {
 76:         old_array[i][j]=new_array[i][j];
 77:     }
 78:  }
 79: old_array[0][0] = heatp; // heated points
 80: old_array[0][1] = heatp;
 81: old_array[1][0] = heatp;
 82:
 83: }
 84:
 85:
```

The main() function consists of file output and is running the iteration() function. First, we open a file handle (myfile) for writing the output to a PPM file. At the start, we add some preliminary heat value to the plate points and a heated value to the left corner.

```
86: int main()
 87: {
 88:     ofstream myfile;  // we will write to this file
 89:
 90:  // seting the initial heat value of the whole plate
 91:  for(int i=0;i<X;i++)
 92:  {
 93:     for(int j=0;j<Y;j++)
 94:     {
 95:         old_array[i][j]=5;  // some starting heat value for the plate
 96:     }
 97:  }
 98:
 99:  // set hot spots in a corner
100:  old_array[0][0] = heatp; // heated points
```

```
101:  old_array[0][1] = heatp;
102:  old_array[1][0] = heatp;
103:
```

The iteration will run from 0 to $iter\_n$. At every step, we run the iteration() function to calculate the whole surface of the plate. After every iteration, we write the actual heat state out into a file to be able to investigate the heat propagation. The file-name will be "plate_XXX" where XXX are numbers from 0 to $iter\_n$.

```
104:  for(int x=0;x<iter_n;x++)  // we run until a fixed "iter_n" iteration number
105:  {
106:     iteration();
107:     string filename1,filename2;
108:     filename1 = "plate_";
109:     filename2 =".ppm";
110:     std::stringstream ss;
111:     if(x<10) {int b=0; ss << b;};
112:     if(x<100) {int b=0; ss << b;};
113:     ss << x;
114:
115:     filename1.append(ss.str());
116:     filename1.append(filename2);
117:        char *fileName = (char*)filename1.c_str();
118:
119:        myfile.open(fileName);
120:        myfile << "P3\r\n";
121:        myfile << X;
122:        myfile << " ";
123:        myfile << Y;
124:        myfile << "\r\n";
125:        myfile << "255\r\n";
126:
127:
```

Since the PPM format needs the RGB pixel data to be written as triads of decimal numbers, we have to go through the new_array array (at this state, the two arrays are the same) to get the actual heat value. This heat value has to be written to all the R,G,B components, resulting in a grey-scale PPM image at the end.

```
128:       for(int i=0;i<Y;i++)
129:      {
130:        for(int j=0;j<X;j++)
131:        {
132:   myfile << min( (int)round(new_array[i][j]*255/(heatp*speed)*6), 255);
133:   myfile << " ";
134:   myfile << min( (int)round(new_array[i][j]*255/heatp*speed)*6, 255);
135:   myfile << " ";
136:   myfile << min( (int)round(new_array[i][j]*255/heatp*speed)*6, 255);
137:   myfile << " ";
138:        }
139:       myfile << "\r\n";
140:     }
141:
142:     myfile.close();
143:   }
144:   return 0;
145: }
```

The result will be a series of PPM files with the transient states of the heat propagation.

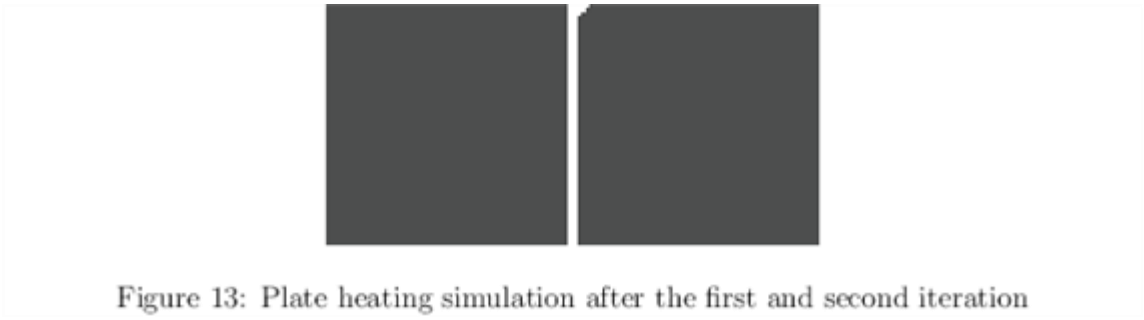

Figure 13: Plate heating simulation after the first and second iteration

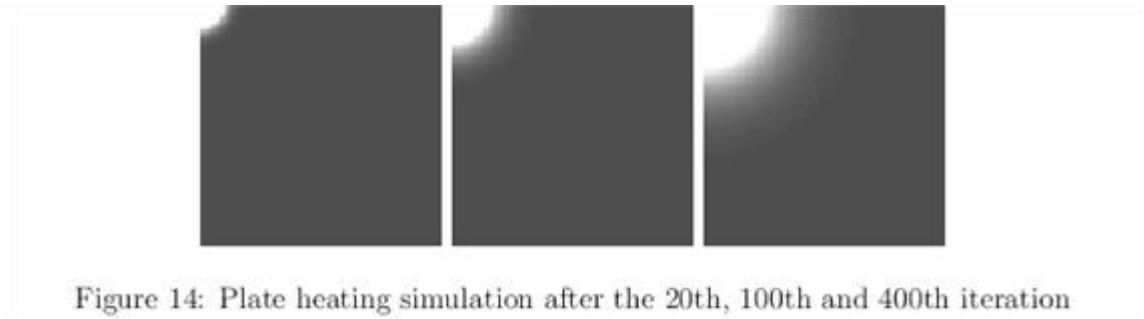

Figure 14: Plate heating simulation after the 20th, 100th and 400th iteration

10.6. 10.6 Example code, parallel solution

In this section we will show a parallelized version of the plate heating model. As discussed above, the parallelization is done by sharing the array, representing the plate, between working nodes. For this, we have to count our working nodes and distribute the data among them.

The first include lines are similar to the serial code, we have only one extra line, the mpi.h library, which we need for the mpi functions. The constants are also the same, though we had to define some extra variables. The FROM and TO variables will define the unique starting and ending lines to be worked with for every working node. The world_rank and world_size variables will contain the serial number and the number of working nodes.

```
1: #include <mpi.h>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7:
  8: using namespace std;
  9: const int X=800; //683;
 10: const int Y=800; //384;
 11: const double heatp=400;
 12: const int iter_n=1600;
 13: int FROM=0;
 14: int TO=0;
 15:
 16: double old_array[X][Y] = {0};
 17: double new_array[X][Y] = {0};
 18:
 19: int world_rank;
 20: int world_size;
 21:
```

The iteration() function is very similar to the serial one, except for that this time, we need some extra parameters. The i_from and i_to parameters will adjust which lines to process. This is necessary, because different working nodes will work on different set of lines.

```
22: void iteration(int i_from, int i_to)
```

```
 23: {
 24:  int i=0;
 25:  int j=0;
 26:
 27:  for(i=i_from;i<i_to;i++)
 28:  {
 29:     for(j=0;j<Y;j++)
 30:     {
 31:         // If its not at an edge
 32:      if( i>0 && i<X-1 && j>0 && j<Y-1)
 33:      {
 34:          new_array[i][j] = (old_array[i-1][j] + old_array[i+1][j] + old_array[i][j-
1] + old_array[i][j+1] ) / 4;
 35:      }
 36:      // At upper edge
 37:      if( j==0 && i>0 && i<X-1 )
 38:      {
 39:         new_array[i][j] = ( old_array[i-1][j] + old_array[i+1][j] +
old_array[i][j+1] ) / 3;
 40:      }
 41:      // At right edge
 42:      if( i==0 && j>0 && j<Y-1 )
 43:      {
 44:         new_array[i][j] = ( old_array[i][j-1] + old_array[i][j+1] +
old_array[i+1][j] ) / 3;
 45:      }
 46:      // At bottom edge
 47:      if( j==Y-1 && i>0 && i<X-1 )
 48:      {
 49:         new_array[i][j] = ( old_array[i-1][j] + old_array[i+1][j] + old_array[i][j-
1] ) / 3;
 50:      }
 51:      // At left edge
 52:      if( i==X-1 && j>0 && j<Y-1 )
 53:      {
 54:         new_array[i][j] = ( old_array[i][j-1] + old_array[i][j+1] + old_array[i-
1][j] ) / 3;
 55:      }
 56:      // Four corners
 57:      if( i==0 && j==0)
 58:      {
 59:         new_array[i][j]=(old_array[i+1][j]+old_array[i][j+1] ) / 2;
 60:      }
 61:      if( i==0 && j==Y-1)
 62:      {
 63:        new_array[i][j]=(old_array[i+1][j]+old_array[i][j-1] ) / 2;
 64:      }
 65:      if( i==X-1 && j==0)
 66:      {
 67:        new array[i][j]=(old array[i-1][j]+old array[i][j+1] ) / 2;
 68:      }
 69:      if( i==X-1 && j==Y-1)
 70:      {
 71:        new_array[i][j]=(old_array[i-1][j]+old_array[i][j-1] ) / 2;
 72:      }
 73:     }
 74:  }
```

The parameters also appear when copying the new data on the old one.

```
72:      }
 73:     }
 74:  }
 75: // Lets copy the new data to the old one
 76:  for(i=i_from;i<i_to;i++)
 77:  {
 78:     for(j=0;j<Y;j++)
 79:     {
 80:         old_array[i][j]=new_array[i][j];
 81:     }
 82:  }
```

```
 83:
 84: // we let the corners hot spots at a constant value
 85: old_array[0][0] = heatp; // heated point
 86: old_array[0][1] = heatp;
 87: old_array[1][0] = heatp;
 88:
 89: }
 90:
```

The main() function starts the same way as the previous with defining a file handle to write the result to a PPM file at the end. After this, we need to initialize the MPI environment with the MPI_Init() line. With the MPI_Comm_size() function, we get the number of nodes accessible for work. This number will be accessible in the second parameter world_size. The MPI_Comm_rank() function will give the rank the number of the actual process. From this point, dedicated actions can run on different nodes according to the rank of the worker.

In the next steps, we have to share the tasks between nodes. According to the above, we just equally divide the lines to work on amongst the workers. If there should be some residuals left, we give it to the last worker. This method is somewhat unbalanced, but it is a very simple and fast way to share the work. With an increasing number of lines, the difference between a normal worker and the last one will decrease. The variable p_row will contain the number of lines every worker has to work with, except the last worker. The variable p_row_last will contain the number of lines to be processed by the last worker node.

```
177:
178:  // we have to hand out the work for different workers
179:  // we do a simple sharing, we hand out equal work for everyone, and the last has
to do some residual extra work (that could be in worse case near the double of a normal
workers work!
180:
181:  double p_row = (double)X / world_size;   // we divide the number of rows (X) by
the number of workers
182:  double p_row_last = 0; // who will be the last? If no others, then the first will
be the last as well..
183:  if(floor(p_row) != p_row) // there is a residual from line 179
184:  {
185:   p_row = floor(p_row);  // whole number of lines a worker will get
186:   p_row_last = X - ((world_size-1)*p_row); // and the number of lines for the last
worker (normal number + residual lines)
187:  }
188:  else  // there is no residual, everyone get equal work
189:  {
190:     p_row = floor(p_row);
191:     p_row_last = p_row;
192:  }
193:  // if we are not the last ones
194:  if(world_rank < world_size-1)
195:  {
196:    cout << "Normal workers work: " << p_row << " lines.. \r\n";
197:  }
198:  else
199:  {
200:    cout << "Last workers work: " << p_row_last << " lines.. \r\n";
201:  }
202:
203:  // lets calculate what FROM, TO values do we have to work with
```

Some feedback is given by the nodes by a simple cout on the console. Now, we know the number of lines a worker has to work with. We have to calculate the starting (FROM) and ending (TO) line numbers to work with. If we are last, we might have to do some extra work.

```
202:
203:  // lets calculate what FROM, TO values do we have to work with
204:  if(world_rank < world_size-1) // if I'm not the last one
205:  {
206:   FROM = world_rank*p_row;
207:   TO = FROM+p_row;
208:  }
209:  else  // I'm the last one
```

```
210:  {
211:   FROM = world_rank*p_row;
212:   TO = FROM + p_row_last;
213:  }
214:
215:  // every worker reports their scope
216:  cout << "FROM: " << FROM << " -- TO: " << TO << " \r\n";
217:
218:  // we start the outer iteration - every worker runs this iteration on their own
set of lines
```

In the outer iteration, at every step, we calculate every point value for the heated plate and start over. Every worker does this whole iteration, but only on their own set of lines. We can use the FROM and TO values to indicate the set. If we are not the only worker, we have to interchange our borders, or ghost lines. We use the interchange_borders() function for this purpose.

```
218:  // we start the outer iteration - every worker runs this iteration on their own
set of lines
219:  for(int x=0;x<iter_n;x++)
220:  {
221:     //  the inner loop is a good object for parallelization
222:     iteration(FROM, TO); // the FROM , TO values are already calculated for every
worker based on their rank
223:     if(world_size>1) {interchange_borders(FROM,TO);} // we have to interchange
border values between workers, except we are alone
224:  }
225:
226:  // at this point, every process worked on their own data-slice and we are ready
227:  // the data-slices are at different nodes, so we have to collect the data to the
first one, to write out into a file
228:  //
229:
```

At this point, we skip back to the interchange_borders() function we left out before. This function is also an addition to the serial code. Since the nodes need neighbour line data to calculate their line data, the borders have to be communicated between the nodes. This inter-nodal communication will go through the MPI environment with the help of MPI_Send() and MPI_Recv(). Every node reaches this point at about the same time. According to the nodes position, some nodes have to get and send ghost lines from and to both neighbours, some need to communicate only in one direction. We also have to know whether we are the only node in the MPI world. Since every MPI_Send needs to communicate with an MPI_Recv and vice-versa, we have to carefully construct the communication. We will create communication pairs. At first, every even node will pass data up to their odd neighbours. At the same time, every odd node has to receive this data. After this, odd ones pass data upwards to even ones and they receive this.

```
90:
 91: void interchange_borders(int FROM, int TO)
 92: {
 93:    // passing borders UP
 94:
 95:    // even ones pass data up.
 96:    if( !(world_rank % 2)  && world_rank < world_size-1 )
 97:    {
 98:       MPI_Send(&old_array[TO-1][0], Y, MPI_DOUBLE, world_rank + 1, 222 ,
MPI_COMM_WORLD); // 222 - unique tag
 99:       //cout << world_rank << " sending to " << world_rank + 1 << " \r\n";
100:    }
101:    // odd ones get data from beneath (except if lowest)
102:    if( (world_rank % 2)  && world_size>1 )
103:    {
104:       MPI Recv(&old array[FROM-1][0], Y, MPI DOUBLE, world rank - 1, 222 ,
MPI_COMM_WORLD , MPI_STATUS_IGNORE); // 222 - unique tag
105:       //cout << world_rank << " receiving from " << world_rank - 1 << " \r\n";
106:    }
107:    // odd ones pass data up (except if on top)
108:    if( (world_rank % 2)  && world_rank < (world_size-1) )
109:    {
110:       MPI_Send(&old_array[TO-1][0], Y, MPI_DOUBLE, world_rank + 1, 333 ,
```

```
MPI_COMM_WORLD); // 333 - unique tag
111:        //cout << world_rank << " sending to " << world_rank + 1 << " \r\n";
112:     }
113:     // even ones get data from beneath (except if lowest)
114:     if( !(world_rank % 2)  && world_rank > 0)
115:     {
116:        MPI_Recv(&old_array[FROM-1][0], Y, MPI_DOUBLE, world_rank - 1, 333 ,
MPI COMM WORLD , MPI STATUS IGNORE); // 333 - unique tag
117:        //cout << world_rank << " receiving from " << world_rank - 1 << " \r\n";
118:     }
119:
```

After all necessary data are passed upwards, we have to pass data downwards in a similar way: even nodes pass the ghost lines down, odd ones get them and after that, odd ones pass data up and even ones get it.

```

120:     // Passing data DOWN
121:     // even ones pass data down (except lowest)
122:     if( !(world_rank % 2)  && world_rank > 0 )
123:     {
124:        MPI_Send(&old_array[FROM][0], Y, MPI_DOUBLE, world_rank - 1, 222 ,
MPI_COMM_WORLD); // 222 - unique tag
125:        //cout << world_rank << " sending to " << world_rank - 1 << "tol: " << FROM
<< "amimegy: " << old_array[FROM][0] <<" \r\n";
126:     }
127:     // odd ones get data from above (except if at top)
128:     if( (world_rank % 2)  && world_rank < (world_size - 1))
129:     {
130:        MPI_Recv(&old_array[TO][0], Y, MPI_DOUBLE, world_rank + 1, 222 ,
MPI_COMM_WORLD , MPI_STATUS_IGNORE); // 222 - unique tag
131:        //cout << world_rank << " receiving from " << world_rank + 1 << "ig: " <<
TO+1 << "amijon: " << old array[TO+1][0] <<" \r\n";
132:     }
133: // odd ones pass data down
134:     if( (world_rank % 2) )  // if we are odd, we are at least the second node
(counting goes from zero)
135:     {
136:        MPI_Send(&old_array[FROM][0], Y, MPI_DOUBLE, world_rank - 1, 333 ,
MPI_COMM_WORLD); // 333 - unique tag
137:        //cout << world_rank << " sending to " << world_rank - 1 << "tol:" << FROM <<
" \r\n";
138:     }
139:     // even ones get data from above
140:     if( !(world_rank % 2)  && world_rank < (world_size -1) ) // even ones get from
above
141:     {
142:        MPI_Recv(&old_array[TO][0], Y, MPI_DOUBLE, world_rank + 1, 333 ,
MPI_COMM_WORLD , MPI_STATUS_IGNORE); // 333 - unique tag
143:        //cout << world_rank << " receiving from " << world_rank + 1 << "ig: " <<
TO+1 << " \r\n";
144:     }
145:
146:
147: }
148:
149: int main()
150: {
151:     ofstream myfile;  // this will be the output file handle
152:
153:  // Initialize the MPI environment
154:   MPI_Init(NULL, NULL);
155:
156:   // Get the number of processes
157:   // int world_size;
158:   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
159:
160:   // Get the rank of the process
161:   // int world_rank;
162:   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
163:
164:   // setting a start value for the plate
165:   for(int i=0;i<X;i++)
```

```
166:  {
167:     for(int j=0;j<Y;j++)
168:     {
169:         old_array[i][j]=5;
170:     }
171:  }
172:
173:  // setting heated point at the plate
174:  old_array[0][0] = heatp; // heated point
175:  old_array[0][1] = heatp;
176:  old_array[1][0] = heatp;
177:
```

After all the calculations are done, every working node has a portion of the final result. To be able to write it to a file at the first node, it has to collect all data from the other nodes. Since every node knows only its own starting and ending line numbers (FROM, TO), the first node has to calculate these data for every receiver. These values will be stored in rFROM and rTO. First, we have to check, whether there are more than one working nodes in our MPI world. If there are more workers, all of them have to send their data to the first one. Since every node runs the same code, we have to arrange that every node sends their data once and the first node receives it several times. This is done by a simple for cycle, where every node sends its data according to the cycle variable p. In every cycle step, the first node receives these data.

```
232:
233:  // the first node has to collect data from others
234:
235:  if(world_size>1)  // are we alone? If not, then we do some collecting
236:  {
237:
238:      // for every worker id (world_rank) we do some send or receive
239:   for(int p=1;p<world_size;p++)
240:   {
241:
242:    if(world_rank == p) // if I'm a sender
243:    {
244:        cout << "I'm (" << world_rank << ") sending lines from:  " << FROM << " to:
" << TO << " \r\n";
245:     for(int r=FROM;r<TO;r++)  // I send my own lines
246:     {
247:      MPI_Send(&old_array[r][0], Y, MPI_DOUBLE, 0 , p*1000+r , MPI_COMM_WORLD); //
999 - unique tag    // itt az unique tag ne legyen a for " R "je? megelozik egymast?
248:     }
249:    }
250:    else if ( world_rank == 0) // I'm not "p" and not the others, I'm worker one (0)
251:    {
252:     // what FROM, TO values should I read from? From every worker (p value) once.
receive_from is rFROM    and     receive_to is rTO
253:     if(p<world_size-1) // not the last (the last has some extra work)
254:     {
255:      rFROM = p_row*p;
256:      rTO = p row*(p+1);
257:     }
258:     else // the last ones work
259:     {
260:      rFROM = p_row*p;
261:      rTO = X;
262:     }
263:     // cout << "receiving (" << world_rank << ") lines from: " << p << " interval:
" << rFROM << " - " << rTO << " \r\n";
264:     for(int r=rFROM;r<rTO;r++)
265:     {
266:      MPI_Recv(&new_array[r][0], Y, MPI_DOUBLE, p , p*1000+r , MPI_COMM_WORLD,
MPI_STATUS_IGNORE); // 999 - unique tag
267:     }
268:    }
269:   }
270:
271:  }
```

After receiving all the data fragments, the whole array is at the first node. At this point the file output is exactly the same as in the serial code, only we have to write one file with the final result.

```
271:  }
272:
273:
274:  if(world_rank==0)
275:  {
276:
277:     // we will write a file at the very end only
278:     string filename1;
279:     filename1 = "plate_par.ppm";
280:     char *fileName = (char*)filename1.c_str();
281:
282:    cout << " irom a fajlt: " << fileName << "\r\n";
283:
284:   myfile.open(fileName);
285:   myfile << "P3\r\n";
286:   myfile << X;
287:   myfile << " ";
288:   myfile << Y;
289:   myfile << "\r\n";
290:   myfile << "255\r\n";
291:
292:
293:   for(int i=0;i<Y;i++)
294:   {
295:    for(int j=0;j<X;j++)
296:    {
297:
298:   myfile << min( (int)round(new array[i][j]*255/heatp*6), 255);
299:   myfile << " ";
300:      myfile << min( (int)round(new_array[i][j]*255/heatp*6), 255);
301:   myfile << " ";
302:      myfile << min( (int)round(new_array[i][j]*255/heatp*6), 255);
303:      myfile << " ";
304:    }
305:    myfile << "\r\n";
306:   }
307:
308:   myfile.close();
309: }
310:
311:   MPI Finalize();
312:   return 0;
313: }
```

10.7. 10.7 A more easy data exchange

If we look at the interchange_borders() function, we can see that the implementation of this usual job seems to be a bit complicated. With the help of the MPI_Sendrecv() function, we can make this code more compact, thus cleaner. The MPI_Sendrecv() function is able to send data to a given node and receive data from a given node at the same time (more precisely, it will handle the send and receive operations in the background at the MPI layer). As such, we have to do the data exchange in three blocks. From bottom to top, one block is the interchange of upper border of the first and the bottom border of the second worker. The second block is for the inner workers, who have to exchange their two borders in both directions. The third block is the communication between the last and the next to last workers.

```
115: void interchange borders(int TOL, int IG)
116: {
117:    // we get in here, if there are at least 2 processors (function is called after
an if outside)
118:    // processor with rank 0 has to communicate only with process 1
119:    if (world_rank == 0)
120:    {
121:         // we send the upper line of process 0 to process 1 and receive the bottom
```

```
line of process 1. We use an unique tag, ex. "222".
122:   MPI_Sendrecv(&regi[IG-1][0], Y, MPI_DOUBLE, world_rank + 1, 222 ,
123:           &regi[IG][0], Y, MPI_DOUBLE, world_rank + 1, 222 ,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
124:    }
125:    // if our rank is between RANK_0 and RANK_LAST, we are a middle point and have
to communicate with two neighbours. We have to send our border lines and receive the
ghost lines of the neighbours
126:    if (world_rank > 0 && world_rank < world_size-1)
127:    {
128:        // upper border line is going up to next process and we receive a ghost line
from beneath.
129:   MPI_Sendrecv(&regi[IG-1][0], Y, MPI_DOUBLE, world_rank + 1, 222 ,
130:           &regi[TOL-1][0], Y, MPI_DOUBLE, world_rank - 1, 222 ,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
131:   // lower border is going beneath and we receive a ghost line from above.
132:   MPI_Sendrecv(&regi[TOL][0], Y, MPI_DOUBLE, world_rank - 1, 222 ,
133:           &regi[IG][0], Y, MPI_DOUBLE, world_rank + 1, 222 ,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
134:    }
135:    // if we are the last process, we only communicate beneath
136:    if (world_rank == world_size-1)
137:    {
138:   // we send lower bottom line beneath and get a ghost line from beneath
139:   MPI_Sendrecv(&regi[TOL][0], Y, MPI_DOUBLE, world_rank - 1, 222 ,
140:           &regi[TOL-1][0], Y, MPI_DOUBLE, world_rank - 1, 222 ,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
141:    }
142:
143: }
```

11. 11 Fast Fourier Transformation (FFT)

11.1. 11.1 Fourier Transformation

The Fourier transformation ((1)) is a mathematical tool, with which we can switch from a function of time to a function of frequency. The method is named after Joseph Fourier, who showed that any periodic function can be constructed as a sum of simple sine and cosine waves (Fourier series). The Fourier transformation extends this idea by allowing the period of the function to approach infinity.

The definition [Weisst] of the Fourier transformation is:

$$f(\nu) = F_t[f(t)](\nu) = \int_{-\infty}^{+\infty} f(t)e^{-2\pi i\nu t}dt \qquad (11.1)$$

How can we take advantage of this? According to this, we can consider functions as mixtures of more simple functions. If we know the components of this mixture, we can deal with them separately or even disable arbitrary sets of them, changing the form and behaviour of the original function. We can analyse the original function better by decomposing it to several simple sinusoidal functions.

In practice, we can filter off special frequencies of the signal (noise, or too high, or too low frequencies), we can visualize a given frequency spectrum (e.g. spectrum analyser of sound software). A further, widespread application of the Fourier transformation is image processing. Several image compression techniques rely on this concept.

Since most of the engineering and computing applications work with discrete values, we need to look at the Discrete Fourier Transformation (DFT).

11.2. 11.2 Discrete Fourier Transformation (DFT)

Digital computers can work with discrete values, so in this case, the Fourier Transformation has to be simplified to work with discrete values, thus we use the Discrete Fourier Transformation (DFT). We get these discrete values from the analogue signal, where we sample the analogue signal at a given frequency and represent the signal with them. If we choose the right sampling frequency, we can fully reconstruct the original analogue signal from the discrete values. The work of Shannon and Nyquist showed that the sampling frequency should be greater than the twice of the maximum frequency of the sampled signal. This also suggests that we should choose the minimal sufficient sampling frequency to be able to work with as small data as possible to increase speed. Further on, we assume that we made the appropriate sampling and look at the data series as such. The discrete version of the Fourier transformation is very similar to the original form.

Original form:

$$F(j\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt \tag{11.2}$$

For a set of given discrete values, the form of the integral would be the summation of these discrete values:

$$F(j\omega) = T\sum_{k=0}^{N-1} f(kT)e^{-j\omega kT} \tag{11.3}$$

where we have N samples of the signals with T sampling times. For a finite number of data points, the DFT assumes that this dataset is periodic. As such, the fundamental frequency and its harmonics have to be evaluated. According to this,

$$\omega = (0, \frac{2\pi}{NT}, \frac{2\pi}{NT} \times 2, .., \frac{2\pi}{NT} \times (N-1)) \tag{11.4}$$

formulating in general:

$$F[n] = \sum_{k=0}^{N-1} f[k]e^{-j\frac{2\pi}{N}nk}(n = 0 : N-1) \tag{11.5}$$

$F[n]$ is the discrete Fourier Transform of sequence $f[k]$.

Although, the values of the input series $f[k]$ are mostly $real$, note, that the $F[n]$ values are complex. The inverse transform is:

$$f[k] = \frac{1}{N}\sum_{n=0}^{N-1} F[n]e^{+j\frac{2\pi}{N}nk} \tag{11.6}$$

11.3. 11.3 Fast Fourier Transformation (FFT)

The DFT calculation involves addition and multiplication operations. Multiplication is the slowest operation of them, thus, speed of the calculation depends on the number of multiplications. If we look at the DFT equation, we can see, that for every

$$n \in \{0, 1, .., N-1\} \tag{11.7}$$

discrete value, we have to sum N number of multiplications. This implies a time dependence of N^2 ($O(N^2)$). Since transformations are mostly required real-time, and real world applications work with several ten or hundred thousands or even more input data points, computational speed is a major question.

There are several methods developed to exploit the properties and estimate the DFT. The ideas are based on simplifying redundant calculations and on dividing the sums into one-element-components. It is said, that Gauss was aware of such a method and he was able to reduce the time factor of a DFT from $O(N^2)$ to $O(N \log N)$.

These methods are called the Fast Fourier Transformation (FFT)[Hilb2013b].

The method able to make the DFT in $O(N \log N)$ was rediscovered and published by the creators of The Cooley-Tukey Algorithm[Cool1965] which was selected in 1999 into the Top 10 Algorithms of 20th Century.

The Danielson-Lanczos Lemma states, that a DFT of N points can be decomposed into two DFTs of $N/2$ points. One for every first and one for every second, thus, for even and odd members of the original N inputs.

Let

$$e^{-j\frac{2\pi}{N}nk} = W_N^{nk} \tag{11.8}$$

be.

The Danielson-Lanczos Lemma states:

$$F[n] = \sum_{k=0}^{\frac{N}{2}-1} f[2k] W_N^{2kn} + \sum_{k=0}^{\frac{N}{2}-1} f[2k+1] W_N^{(\mathbf{2k+1})n} = \sum_{k=0}^{\frac{N}{2}-1} f[2k] W_N^{2kn} + \sum_{k=0}^{\frac{N}{2}-1} f[2k+1] W_N^{\mathbf{2kn}} W_N^{\mathbf{n}} \tag{11.9}$$

In a more compact form:

$$F[n] = E[n] + W_N^n O[n] \tag{11.10}$$

where $E[n]$ denotes the even input data and $O[n]$ denotes the odd input data and W_N denotes the Twiddle Factor.

Since the even and odd parts have half-half of the original input, the summation of them goes from 0 to $\frac{N}{2}$.

This idea can be used again on the $E[n]$ and $O[n]$ sets, getting a sum of four components:

$$F[n] = E[n] + W_N^n O[n] = EE[n] + W_{\frac{N}{2}}^n EO[n] + W_N^n OE[n] + W_N^n W_{\frac{N}{2}}^n OO[n] \tag{11.11}$$

This decomposition can go on until we can't split any further (expansion of the Danielson-Lanczos lemma), thus we get one element summations. A one element summation is itself, so we need only to deal with the appropriate W_N^n values. One prerequisite for that is, that we always have to be able to halve the input, thus, the number of the inputs should be a power of two (other strategies are also known). If this is the case, then we have to do $m = \log_2 N$ steps for all N elements. Hence the speed-up from $O(N^2)$ to $O(Nlog_2N)$. This latter splitting method is also referred to as the Radix-2 method.

A widespread technique for input number different from power of two, is to pad zeros to the end of the data series, filling them to the number of the next power of two [Hilb2013a] [Lyons2004]. This won't affect the result technically, just make the frequency resolution more dense. Although, this won't give any new data, besides the possibility to use the Radix-2 FFT method, it has the advantage of representing the corresponding frequencies more scenic. In our example later on, we will use this kind of padding technique as well.

According to the above, for $N = 2$ the summation looks like

$$F(n) = x(0) + W_2^n x(1) \tag{11.12}$$

for $N = 4$ it is

$$F(n) = x(0) + W_2^n x(2) + W_4^n x(1) + W_4^n W_2^n x(3) \tag{11.13}$$

and for $N = 8$ it looks

$$F(n) = x(0) + W_2^n x(4) + W_4^n x(2) + W_4^n W_2^n x(6) + W_8^n x(1) + \\ + W_8^n W_2^n x(5) + W_8^n W_4^n x(3) + W_8^n W_4^n W_2^n x(7) \quad (11.14)$$

If we closely look at the equations , we can see some strange order of the $x(n)$ components. This order is the outcome of the repeated odd - even separation of the input values. For 8 values, we have to split (divide into two) three times to get the appropriate one element sums. Let's suppose we have the values $0, 1, 2, 3, 4, 5, 6, 7$ as inputs. We choose these values, so the values also represent their original positions in the series. If we split them to odd-even parts (DL lemma), we get two four-member-parts: $0, 4, 2, 6$ and $1, 5, 3, 7$. If we split them further, we get: $0, 4$, $2, 6$, $1, 5$, $3, 7$. For the last split, we get the order we showed in (14). This order can be achieved by a little trick, by sorting the positions binary-reverse. Hence the name reverse binary order.

By reverting the binary order of the index values of the original inputs (5), we get exactly the above order.

Table 5: Reverse binary values

value	binary	reverse binary (rb)	rb value
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Implementing a reverse binary function can come handy at the algorithmic solution of the FFT. According to the Danielson-Lanczos lemma, our FFT algorithm has to recombine pairs of odd and even inputs, where the even part is multiplied by a W_N^n value. After this recombination, we have a new series, with the half of the original length ($N/2$). We have to start again on it and recombine the odd and even parts with a multiplication of W_{2N}^n at the even part. We have to do this until we get one value. This will be the nth outcome of the FTT. With a reverse binary function, we can directly choose the appropriate pairs from the input to mix at the first step and use this reverse binary order later on for calculating further n values.

Let's summarize the above. We want to calculate the Discrete Fourier Transform (DFT) in a fast way (FFT). We have sampled input values from the time domain and we have to use them for the calculation. If we extend the Danielson-Lanczos lemma, we can separate the input values always into two parts (odd, even) recursively. That could be thought of like a tree. The input values are from left to right the leaves of the tree, in a special order (reverse binary). We only have to go from left to right and recombine two adjacent leaves. It is not enough to add them, we have to multiply the second ones (even) with a special W_N^n Twiddle Factor. The value of this factor depends on the actual level (of the tree) we are at. For N input points, we will have $\log_2 N$ levels. After adding the two parts, we get a new series of data, with which we have to do the same as before, using a new W_N^n factor. Since we had $\log_2 N$ input data (Radix-2), we will get only one value at the end. This value will be the n th value of our output, thus the n th value of our frequency domain data. The above procedure has to be done for every n of the N time-domain sampled inputs, thus giving an n part output.

11.3.1. 11.3.1 FFT serial example code

This subsection will show a serial FFT implementation. The code uses the above methods. Since the FFT works with complex numbers, where we have a real and an imaginary part, we have chosen to represent the real and imaginary parts in two different arrays.

The first part of the code has the usual include lines and some variable definitions.

```
1: #include <iostream>
  2: #include <fstream>
  3: #include <sstream>
  4: #include <math.h>
  5: #include <vector>
  6: #include <time.h>
  7: #include <cstdlib>
  8: #include <bitset> // we require it for special binary representation
  9:
 10: using namespace std;
 11: // input data
 12: double * data_in_r;
 13: double * data_in_i;
 14:
 15: //output data
 16: double * data_out_r;
 17: double * data_out_i;
 18:
 19: //working buffer
 20: double * buffer r;
 21: double * buffer_i;
 22:
 23: //reverse binary input data
 24: double * rb_buffer_r;
 25: double * rb_buffer_i;
 26:
 27: int p_length=32380;
 28: int p2_length=0;
 29: int bitlength;
 30: int what_power=0;
 31: unsigned int mask=0x00000001;
 32:
 33: unsigned int revbit(unsigned int val)
```

The eighth row has an include file named bitset. With the help of this include, we will be able to output binary numbers easily. This functionality is commented out, but left in for possible monitoring purposes. The data_in_r[] and data_in_i[] arrays will contain the real and imaginary parts of the input data. Similarly the data_out_r[] and data_out_i[] arrays will hold the spectral output data. The algorithm will need a temporary buffer (buffer_r[], buffer_i[]), where the temporary results are stored for further calculations. During the calculation, we will need a special order of the input data (reverse binary). Since the reverse binary calculation is somewhat time consuming, it is a good idea to store it in an array (rb_buffer_r[], rb_buffer_i[]) and only copy it to the buffer when we need it.

The p_length variable contains the number of input data points. For real life usage, this should be read and calculated from the actual data from arbitrary input (a file, or some streamed real-time data). For our test purpose, we will generate some input data. Should the size of the data not be a power of two, we have to pad it with zeros. The padded length (next power of two) will be stored in p2_length. The bitlength variable will be used for monitoring purposes, to see the original bit length of the inputs. The variable what_power will contain the exponent n of 2^n where we have 2^n inputs (stored in p2_length). Thus, it will be used by the next_2pow() function and will store the minimal bit length on which the inputs number (p2_length) can be represented (e.g. for 8 inputs it will be 3). The mask variable will be used by the revbit() function. This function will produce the reverse binary value of it's input.

The next lines in the code include some short function definitions.

```
32:
 33: unsigned int revbit(unsigned int val)
 34: {
 35:  unsigned int rval=0;
 36:  unsigned int fval=0;
 37:  for(int i=0;i<what_power;i++)
 38:  {
 39:     rval = (val >> i) & mask;
 40:     fval = fval | (rval << (what_power-i-1));
 41:  }
 42:     return fval;
```

```
 43: }
 44:
```

The revbit() function returns the reverse binary value of the given input val. There are several tricky solutions for this to do, but one of the most compact and straightforward method is to get the binary values from the right bit by bit, and put them to the other side. This reverses the order of the bits. It's important to note, that this reverse binary method works with dynamic lengths, thus, we need to know the minimum number of bits required, to represent the index of the inputs. If we have 8 input data, this number is 3, if we have 16 input data, this number is 4, if we have 1024 input data, this number is 10. The rval gets the first right, second right, and so on.. -bits of the input, we shift them one position further to the left and do a logical OR with them and fval. After what_power steps, fval has the final reverse binary value and we return it.

```
45: // this function generates the input values
 46: // in this case, this will be some sawtooth wave
 47: void gen_data(unsigned int size)
 48: {
 49:  if (size>p2_length) {size=p2_length;} // to avoid writes to wrong addresses
 50:  for(unsigned int i=0;i<size;i++)
 51:     {
 52:         data_in_r[i] = (double)i;
 53:   data in i[i] = 0;
 54:     }
 55:
 56:  for(unsigned int i=size;i<p2_length;i++) // padding with zeros, if necessary
 57:     {
 58:         data_in_r[i] = 0;
 59:   data_in_i[i] = 0;
 60:     }
 61:
 62: }
 63:
```

The gen_data() function generates a sawtooth form input signal by putting increasing values from 0 to $size$, where size is the input parameter of the function. After a short check to avoid wrong writes, we put the increasing data into data_in_r[] and data_in_i[]. Although we usually work with real numbers as input, it could be possible to work with complex numbers. For our example, we simply zero out the imaginary part of the input. If the size parameter is less than the upper next power of two, we have to pad the remaining input positions with zeros.

```
63:
 64: // this function gives the next power of two
 65: int next_2pow(int input)
 66: {
 67:  what_power = ceil(log((double)input)/log(2.0));
 68:     return (int)pow(2.0, what_power);
 69: }
 70:
```

The next_2pow() function will return the upper next power of two, which is above the parameter. This function is necessary for the zero padding function, which ensures that the number of the inputs is a power of two. First, we get the two based logarithm of the input ($\log_a b = \frac{\log_x b}{\log_x a}$) and round it to the next whole number (ceil()). This value is stored in the global variable what_power and the function returns the upper next power of two.

```
70:
 71: // W Twiddle factor - real part
 72: //
 73: double WLn_r(unsigned int L, unsigned int n)  // W is complex, we handle it with a
real (_r) and with an imaginary (_i) part
 74: {
 75:  return( (double)cos( 2*M_PI*n /L ));
 76: }
 77:
 78: // W Twiddle factor - imaginary part
 79: double WLn_i(unsigned int L, unsigned int n)
 80: {
```

```
 81:  return( (double)sin( 2*M_PI*n /L ));
 82: }
 83:
```

The next two functions, WLn_r() and WLn_i() are the real and imaginary part of the Twiddle Factor W_N^n defined previously. Since the Twiddle Factor is a complex number

$$W_N^n = e^{\frac{-j2\pi n}{N}} \tag{11.15}$$

we can use the Euler formula

$$e^{ix} = \cos(x) + i\sin(x) \tag{11.16}$$

and calculate the real (cos()) and imaginary (sin()) parts separately.

$$WLn\_r(L,n) = \cos(\frac{2\pi n}{L}) \text{ and } WLn\_i(L,n) = \sin(\frac{2\pi n}{L})$$

The main() function starts with calculating the p2_length value which will be an optionally zero padded input series with a length of power of two, closest to the original input length. After some monitoring outputs to the console, we allocate memory for our input data (data_in_r[], data_in_i[]), output data (data_out_r[], data_out_r[]), temporary buffer (buffer_r[], buffer_r[]) and for the reverse binary sorted data (rb_buffer_r[], rb_buffer_r[]).

```
83:
 84: int main()
 85: {
 86:   p2_length=next_2pow(p_length); // next power 2 length
 87:
 88:   bitlength = log(p2_length) / log(2); // bit length
 89:
 90:   // information about calculated values
 91:   cout << "p_length " << p_length << endl;;
 92:   cout << "p2_length " << p2_length << endl;;
 93:   cout << "bitlength " << bitlength << endl;;
 94:   cout << "what_power " << what_power << endl;;
 95:
 96:   // allocate memory for input data REAL
 97:   data_in_r = new double[p2_length];
 98:
 99:   // allocate memory for input data IMAGINARY
100:   data_in_i = new double[p2_length];
101:
102:   // allocate memory for output data (real)
103:   data_out_r = new double[p2_length];
104:
105:   // allocate memory for output data (im)
106:   data out i = new double[p2 length];
107:
108: // allocate memory for buffer data (working buffer)
109:   buffer_r = new double[p2_length];
110:
111:   buffer_i = new double[p2_length];
112:
113: // allocate memory for reverse binary sorted data (working buffer)
114:   rb_buffer_r = new double[p2_length];
115:
116:   rb_buffer_i = new double[p2_length];
117:
118:   // generate input data as a saw form
119:   gen data(p length);  // we generate data of p length length, if it's not a power
of two, it will be padded with zeros in the function
120:   // put data in index-bit reverse order to data_out
121:
122:   /* only a short debug to look at the reverse
```

The next few lines are commented out, but left in the example.

```
122:   /* only a short debug to look at the reverse
123:   for(int i=0;i<p2_length;i++)
124:   {
125:     std::bitset<32> x(i);
126:  unsigned int rev = revbit(i);
127:  std::bitset<32> y(rev);
128:  // cout << "szam: " << i << " (" << x << ") reverse: " << rev << " (" << y << ")
\r\n";
129:   }
130:   */
131:
```

This code could be uncommented and used to monitor the binary representation of the values and the reverse binary ordered values. It is not needed for the calculation, though.

```
132:   // we will start the loops here.
133:   // we work with complex numbers, so we have a real ( r) and an imaginary ( i)
part as well
134:
135:   unsigned int puf_l = p2_length;  // working variable - starting length of input
data - it will be changed during the algorithm (halved and halved..)
136:   unsigned int divider = 1;  // divider variable, it will be doubled as we step
forward with the series of Even Odd recombination levels
137:   unsigned int ind = 0 ;  // temp variable for copying data
138:
139:   // at first, we generate a reverse binary ordered working buffer from the input.
It's worth to do it once and just copy it into the working buffer whenever it's
necessary
140:  // copy input data into buffer in reverse binary order
141:  // input data is complex, hence the real and imaginary part
142:   for(unsigned int i=0;i<p2_length;i++)
143:   {
144:   ind = revbit(i);
145:   rb_buffer_r[i]=data_in_r[ind];
146:   rb_buffer_i[i]=data_in_i[ind];
147:   }
148:
149:   for(unsigned int n=0;n<p2_length;n++)   // F(n)
```

Since we have to deal with p2_length number of data input, we will have to make an outer loop of this size. As mentioned before, we will recursively compose new array elements from pairs. To book the size of the intermediate array, we will use the puf_l variable. The divider variable will be of use to calculate the actual W_N^n Twiddle Factor (WLn_r and WLn_i). The ind variable will contain the reverse binary index value. Before the outer loop, we reorder the input data according to the reverse binary scheme. Since we will need this reverse binary order for the calculation at every input data, it is a good idea to do it only once and just copy it again and again. This reordered input value list will be stored in the rb_buffer_r[] and rb_buffer_i[] arrays. Prepared the outer loop, we can start with that.

```
148:
149:   for(unsigned int n=0;n<p2_length;n++)   // F(n)
150:   {
151:   // copy reverse binary pre-ordered buffer into working buffer, we have to start
from this state at every "n"-step
152:   for(unsigned int i=0;i<p2_length;i++)
153:   {
154:     buffer_r[i]=rb_buffer_r[i];
155:     buffer_i[i]=rb_buffer_i[i];
156:   }
157:   divider=1;    // we will use this variable to indicate "N"
158:   puf_l = p2_length;    // length starts always from the input length and will
change inside the next loop
159:   for(unsigned int b=0;b<what_power;b++)  // b is running from 0 to logN levels
160:   {
161:    divider = divider*2;  // as we step forward, we have to double the value of N
162:    for(unsigned int k=0;k<puf_l;k++)
```

```
163:    {
164:         buffer_r[k] = buffer_r[2*k] + buffer_r[2*k+1]*WLn_r(divider ,n)  -
buffer_i[2*k+1]*WLn_i(divider, n);
165:         // a+bi    x      c+di    = (ac - bd) + i(ad + bc)
166:         buffer_i[k] = buffer_i[2*k]  + buffer_r[2*k+1]*WLn_i(divider ,n) +
buffer_i[2*k+1]*WLn_r(divider, n);
167:         // m+ni     +   (a+bi x c+di)   even + odd * w   -->   r:   m+(ac - bd)
i:  n+(ad + bc)
168:    }
169:    puf_l = puf_l / 2;
170:   }
171:   data_out_r[n]=buffer_r[0];
172:   data_out_i[n]=buffer_i[0];
173:
174:   // we can look at the input and output data
175:   cout << "-- Input " << n << ": " << data_in_r[n] << "," << data_in_i[n] << " --
output: " << data_out_r[n] << "," << data_out_i[n] << endl;;
176:  }
177:
```

For N inputs, we will calculate N outputs during the FFT, so we have to make an outer loop that runs from 0 to $p2\_length$. At every step, we start from the specially reordered input values, so we have to copy the previously generated reverse binary ordered input data from rb_buffer_r[] and rb_buffer_i[] to our temporary buffer buffer_r[] and buffer_i[]. The next loop goes from 0 to $what\_power$, thus we define here, how many pairing steps do we have to make. If we consider, that the inputs are power of two, we can easily make out that we have to make $\log_2^N$ steps. And that is exactly the value in what_power. At every step, we have to pair adjacent values in the reverse binary ordered list. Besides adding these pairs, we have to deal with the special complex Twiddle Factor WLn_r and WLn_i. The even part of the pair (the second) has to be multiplied by the Twiddle Factor before adding it to the odd part. The Twiddle Factor has two changing parameters. One of them is connected to the pairing level or step level. This parameter has been referenced as N previously. We reverence it in the Twiddle Factor functions as L not to mix it with n which has it's own meaning. The value of N indicates the number of input parameters. At the top level, it is the number of $p2\_length$ and it is always divided into two at each sub-levels. At the lowest level, at the leaves of our tree, the value of N is 2. Going from bottom to top, this value has to double itself at every step. The variable divider acts like this and is the first parameter for the Twiddle Factor. The second parameter is n which indicates the actual index of the input values. This comes from the outer loop. The third loop which runs from 0 to puf_l handles the pairing procedure. At every level, we have to choose the first (odd) and second (even) value from the buffer. After one pairing, the length of the new dataset will be only the half, and again the half, until it is only one element. The puf_l variable changes accordingly. At this point we have to remember that the algorithm works with complex numbers. Although, our typical input data has only real part, the calculations must be done with complex numbers. Adding two complex numbers is quite easy, we have to add the real parts together and the imaginary parts together. Multiplying is more tricky: $(a+ib) \times (c+id) = (ac-bd) + i(ad+bc)$. In our loop, the first element of the pairs is indexed as $2k$ and the second element is indexed as $2k+1$. If we denote the first element of the pair with $m+in$, the second element with $a+ib$ and the Twiddle Factor with $c+di$, then we have to calculate this: $m+in+(a+ib)*(c+id)$. This will result in $(m+(ac-bd))+i(n+(ad+bc))$. Substituting the arrays, we calculate the real and imaginary parts in two lines:

```
buffer_r[k] = buffer_r[2*k] + buffer_r[2*k+1]*WLn_r(divider ,n)
              - buffer_i[2*k+1]*WLn_i(divider, n);
```

and

```
buffer_i[k] = buffer_i[2*k] + buffer_r[2*k+1]*WLn_i(divider ,n)
              + buffer_i[2*k+1]*WLn_r(divider, n);
```

After going through every pair of every level, we gain one complex value for the actual n input. We have to put this data into the data_out_r[] and data_out_i[] result arrays. For monitoring reasons, we can get some feedback to the console, comparing the input and output values for different n-s.

```
176:   }
177:
178:    // we loose data after this lines, any output should be done here
179:    delete[] data_out_i; // free up memory
180:    delete[] data_out_r;
181:    delete[] data_in_i;
182:    delete[] data in r;
183:    delete[] buffer_r;
184:    delete[] buffer_i;
185:    delete[] rb_buffer_r;
186:    delete[] rb_buffer_i;
187:
188:    return 0;
189: }
```

After calculating the FFT for the n inputs, we have our frequency data. This can now be written out to a file or directly drawn in some graphical applications.

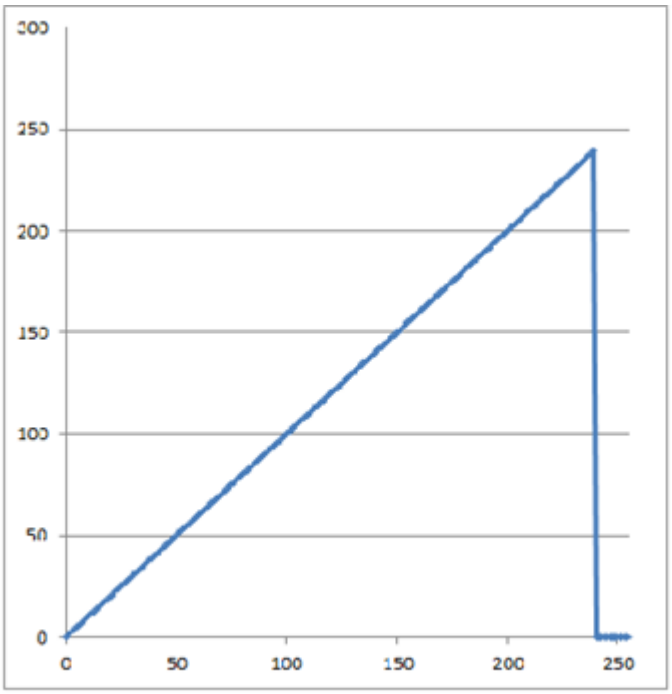

Figure 15: Input data: Jigsaw form from 0 to 240, padded to 256.

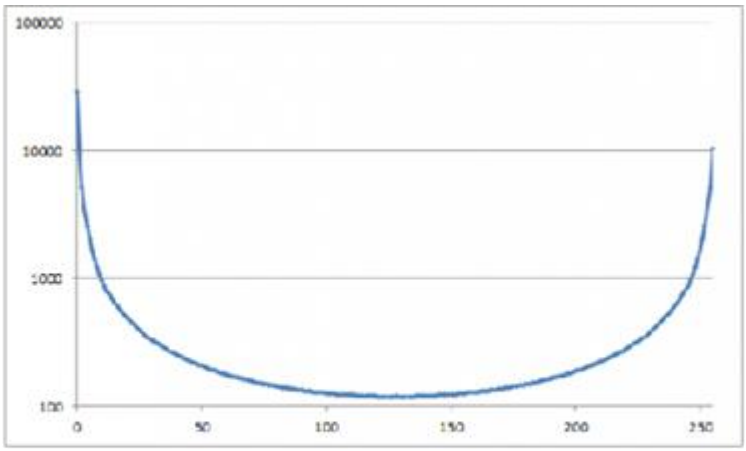

Figure 16: Ouput data: power spectrum. If sampling occurred at 256 Hz, our frequency step is 1 Hz, thus spectrum goes from 0 to 256 Hz. Values are represented on a $\log_{10}$ scale.

As we can see, the FFT speeds up the DFT process and gives us a handy tool. Although, calculating huge data-blocks is still a lengthy process. The implementation has some potential for further optimization, but the running time can't be decreased. If we look at the algorithm, we can notice, that parallelization could help to gain further speed-up.

11.3.2. 11.3.2 FFT example code parallelization

The FFT algorithm previously dealt with, has the potential for parallelization. According to the serial algorithm, the huge amount of input data is processed one by one running an outer loop. Although every calculation involves the whole input series, our input is known at the very beginning. This yields to a possible parallelization, where working nodes could calculate with arbitrary amount of input data, parallel.

The first include and definition block differs only in some extra lines from the serial version.

```
1: #include <mpi.h>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7: #include <time.h>
  8: #include <cstdlib>
  9: #include <bitset> // we require it for special binary representation
 10:
 11: using namespace std;
 12: // input data
 13: double * data_in_r;
 14: double * data_in_i;
 15:
 16: //output data
 17: double * data_out_r;
 18: double * data_out_i;
 19:
 20: //working buffer
 21: double * buffer_r;
 22: double * buffer i;
 23:
 24: //reverse binary input data
 25: double * rb_buffer_r;
 26: double * rb_buffer_i;
 27:
 28: int p_length=240;
 29: int p2_length=0;
 30: int bitlength;
 31: int what_power=0;
 32: unsigned int mask=0x00000001;
 33:
 34: // MPI variables
 35: int world_rank;  // the rank of this process
 36: int world_size; // size of world, thus number of processors in this world
 37: int FROM;
 38: int TO;
 39: int rFROM;
 40: int rTO;
 41:
```

First, we have to include again the mpi.h include file to be able to use the MPI environment. For the communication, we need some variables. As before, the world_rank and world_size variables will contain the number of the actual working node and the number of all nodes. For data partition, we define some variables, with which we can follow up the data range for different nodes (FROM, TO, rFROM, rTO).

Since the parallel solution will use the same functions as the serial one, we define them the same way.

```
41:
 42: unsigned int revbit(unsigned int val)
```

```
 43: {
 44:  unsigned int rval=0;
 45:  unsigned int fval=0;
 46:  for(int i=0;i<what_power;i++)
 47:  {
 48:     rval = (val >> i) & mask;
 49:     fval = fval | (rval << (what_power-i-1));
 50:  }
 51:     return fval;
 52: }
 53:
 54: // this function generates the input values
 55: // in this case, this will be some sawtooth wave
 56: void gen_data(int size)
 57: {
 58:  for(int i=0;i<size;i++)
 59:     {
 60:         data_in_r[i] = (double)i;
 61:   data_in_i[i] = 0;
 62:     }
 63:
 64:  for(int i=size;i<p2_length;i++) // padding with zeros, if necessary
 65:     {
 66:         data_in_r[i] = 0;
 67:   data_in_i[i] = 0;
 68:     }
 69:
 70: }
 71:
 72: // this function gives the next power of two
 73: int next_2pow(int input)
 74: {
 75:  what_power = ceil(log((double)input)/log(2.0));
 76:     return (int)pow(2.0, what_power);
 77: }
 78:
 79: // this function gives the last power of two
 80: int last_2pow(int input)
 81: {
 82:     return (int)pow(2.0, floor(log((double)input)/log(2.0)));
 83: }
 84:
 85: // W Twiddle factor - real part
 86: //
 87: double WLn_r(unsigned int L, unsigned int n)  // W is complex, we handle it with a
real (_r) and with an imaginary (_i)
 88: {
 89:  return ( (double)cos( 2*M_PI*n /L ));
 90: }
 91:
 92: // W Twiddle factor - imaginary part
 93: double WLn_i(unsigned int L, unsigned int n)
 94: {
 95:  return ( (double)sin( 2*M_PI*n /L ));
 96: }
 97:
```

These functions are not affected by parallelizing the code, except one function which we will use for determining the number of workers. This function is the last_2pow() function and it calculates the lower neighbouring power of two of the input. We will get back to this later on.

The next change is in the first rows of the main() function.

```
96: }
 97:
 98: int main()
 99: {
100:  double s_time; // MPI timing
101:  // Initialize the MPI environment - we can use the MPI functions and world until
MPI_Finalize
```

```
102:     MPI_Init(NULL, NULL);
103:
104:  s_time=MPI_Wtime();
105:  // number of processes
106:     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
107:     // Get the rank of the process
108:     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
109:
110:  // we set the required input size to power of two
111:  p2_length=next_2pow(p_length); // next power 2 length
112:  bitlength = log(p2_length) / log(2); // bit length  - for monitoring purposes
```

We define a double variable named s_time and use it for MPI timing with the MPI_Wtime() function. This later gives a time in seconds elapsed since an arbitrary time in the past. For us, differences between such values will be of use. After initializing the MPI environment (MPI_Init()), we enumerate the workers in our default MPI communicator with MPI_Comm_size() and put the number into world_size. The next step is to identify ourselves and get our rank with the MPI_Comm_rank() function and store it to world_rank. The next two lines are familiar from the serial version, we set the input number to power of two and set some monitoring parameters.

```
113:
114:  // we have to split the work among workers
115:  // we wan't power of 2 workers
116:  int ws_temp = world_size;
117:  // should we have less data than workers..
118:  if ( p2_length < world_size )
119:  {
120:   ws_temp = p2_length;
121:   // this will be always power of two, since our p2_length is such
122:  }
123:  else
124:  {
125:   ws_temp = last_2pow(world_size);  // we use power of 2 number of workers
126:  }
127:
128:     // unnecessary workers check
129:  if(world_rank > ws_temp-1)
130:  {
131:   // we are not needed, so we quit
132:   cout << world_rank << ": ---- Bye! ---- " << endl;
133:   MPI_Finalize();
134:   exit(1);
135:  }
```

At this point, we have to prepare the data distribution. In this example, we will hand out equal portions of data to every working node. If the amount of data is less than the number of workers, we select data number of workers active, thus we set the variable ws_temp to p2_length. If we have more data than workers, which is mostly the case, we limit the number of workers to the maximal power of two possible. For example, if we have 5 workers and 32 input data, we use only 4 workers. Why do we want to use power of two number of workers? Because we can't do anything else. If we have power of two number of input data and we want to divide it to equal parts, the number of parts will be power of two and the number of data in one part will be a power of two as well. Since we don't need the unnecessary workers, we just let them say "Bye", finalize the MPI environment and exit the program. Although this move works in our example, we have to pay attention to one issue regarding this. If some of the initial members exit, we will loose all MPI functionality regarding collective communication. A workaround for that will be discussed at the end of this subsection.

The next block is similar to the serial version, except some data range calculations at the beginning.

```
135:  }
136:
137:   // we divide the number of input data with the number of workers
138:  unsigned int dataslice = p2_length / ws_temp;
139:  FROM = dataslice*world_rank;
140:  TO = FROM + dataslice;
141:
142:   // information about calculated values
```

```
143:   if (world_rank == 0)
144:   {
145:     cout << "p_length " << p_length << endl;
146:     cout << "p2_length " << p2_length << endl;
147:     cout << "bitlength " << bitlength << endl;
148:     cout << "what_power " << what_power << endl;
149:     cout << "No. of workers: " << ws_temp << endl;
150:   }
151:
152:  // allocate memory for input data REAL
153:   data_in_r = new double[p2_length];
154:
155:   // allocate memory for input data IMAGINARY
156:   data_in_i = new double[p2_length];
157:
158:   // allocate memory for output data (real)
159:   data_out_r = new double[p2_length];
160:
161:   // allocate memory for output data (im)
162:   data_out_i = new double[p2_length];
163:
164: // allocate memory for buffer data (working buffer)
165:   buffer_r = new double[p2_length];
166:
167:   buffer_i = new double[p2_length];
168:
169: // allocate memory for reverse binary sorted data (working buffer)
170:   rb_buffer_r = new double[p2_length];
171:
172:   rb_buffer_i = new double[p2_length];
173:
174:   // generate input data as a saw form
175:   gen_data(p_length);  // we generate data of p_length length, if it's not a power
of two, it will be padded with zeros in the function
176:   // put data in index-bit reverse order to data_out
177:
178:
179:   /* only a short debug to look at the reverse
180:   for(int i=0;i<p2_length;i++)
181:   {
182:     std::bitset<32> x(i);
183:  unsigned int rev = revbit(i);
184:  std::bitset<32> y(rev);
185:  // cout << "szam: " << i << " (" << x << ") reverse: " << rev << " (" << y << ")
\r\n";
186:   }
187:   */
188:
189:   // we will start the loops here.
190:   // we work with complex numbers, so we have a real ( r) and an imaginary ( i)
part as well
191:
192:   unsigned int puf_l = p2_length;  // working variable - starting length of input
data - it will be changed during the algorithm (halved and halved..)
193:   unsigned int divider = 0;  // divider variable, it will be doubled as we step
forward with the series of Even Odd recombination levels
194:   unsigned int ind=0 ;  // temp variable for copying data
195:
196:   // at first, we generate a reverse binary ordered working buffer from the input.
It's worth to do it once and just copy it into the working buffer whenever it's
necessary
197:  // copy input data into buffer in reverse binary order
198:  // input data is complex, hence the real and imaginary part
199:   for(unsigned int i=0;i<p2_length;i++)
200:   {
201:   ind = revbit(i);
202:   rb_buffer_r[i]=data_in_r[ind];
203:   rb_buffer_i[i]=data_in_i[ind];
204:   }
```

First, we set the amount of data for one worker in dataslice and every worker determines its range of work (FROM, TO). In the next few lines, we define some variables, allocate memory for our lists, generate the input data and the reverse binary order data series just as in the serial version.

```
204:   }
205:
206:   // the FROM and TO variables are set for every worker different
207:   for(unsigned int n=FROM;n<TO;n++)   // F(n)
208:   {
209:   // copy reverse binary pre-ordered buffer into working buffer, we have to start
from this state at every "n"-step
210:   for(unsigned int i=0;i<p2_length;i++)
211:   {
212:     buffer_r[i]=rb_buffer_r[i];
213:     buffer_i[i]=rb_buffer_i[i];
214:   }
215:   divider=1;    // we will use this variable to indicate "N"
216:   puf_l = p2_length;    // length starts always from the input length and will
change inside the next loop
217:   for(unsigned int b=0;b<what_power;b++)  // b is running from 0 to logN levels
218:   {
219:       divider = divider*2;  // as we step forward, we have to double the value of N
220:    for(unsigned int k=0;k<puf_l;k++)
221:    {
222:          buffer_r[k] = buffer_r[2*k] + buffer_r[2*k+1]*WLn_r(divider ,n)  -
buffer_i[2*k+1]*WLn_i(divider, n);
223:       // a+bi    x      c+di   = (ac - bd) + i(ad + bc)
224:          buffer_i[k] = buffer_i[2*k]  + buffer_r[2*k+1]*WLn_i(divider ,n) +
buffer_i[2*k+1]*WLn_r(divider, n);
225:       // m+ni     +   (a+bi x c+di)   even + odd * w   -->   r:   m+(ac - bd)
i:  n+(ad + bc)
226:       }
227:    puf_l = puf_l / 2;
228:   }
229:   data_out_r[n]=buffer_r[0];
230:   data_out_i[n]=buffer_i[0];
231:
232:   // we look at the input and output data
233:   // cout << "-- Input " << n << ": " << data_in_r[n] << "," << data_in_i[n] << " -
- output: " << data_out_r[n] << "," << data_out_i[n] << endl;
234:  }
```

The outer loop, which processes the input data is running on a definite interval for every worker. The inside of this loop doesn't change compared to the serial code. Since every worker has the whole dataset at the beginning and the algorithm doesn't rely on previous results, we can run the algorithm on every worker parallel. After doing the FFT on the given input range, every worker has its own result chunk. This has to be collected to get the whole result at one process.

```
234:  }
235:
236:    // we have to collect all data
237:    if(ws_temp>1)  // we collect only, if there are more than one processes
238:  {
239:   for(int p=1;p<ws_temp;p++) // we have to collect ws_temp-1 slices
240:   {
241:    if(world_rank == p) // if it's my turn, as sender, I send.
242:    {
243:        // cout << "I'm sending (me: " << world_rank << ") data from " << FROM << "
to " << TO << ". dataslice: " << dataslice << endl;
244:     MPI_Send(&data_out_r[FROM], dataslice, MPI_DOUBLE, 0 , 900 , MPI_COMM_WORLD);
245:     MPI_Send(&data_out_i[FROM], dataslice, MPI_DOUBLE, 0 , 900 , MPI_COMM_WORLD);
246:
247:    }
248:    else if ( world_rank == 0) // if we are process 0, we collect the sended data
249:    {
250:        // what shall rFROM and rTO be? What lines do we have to collect. We have to
calculate it from the remote process' rank (p)
251:      rFROM = dataslice*p;
252:      rTO = dataslice*(p+1);
```

```
253:
254:     // cout << "I receive (me: " << world_rank << ") the lines from: " << p << ".
Interval: " << rFROM << " - " << rTO << endl;
255:     MPI_Recv(&data_out_r[rFROM], dataslice, MPI_DOUBLE, p , 900 , MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
256:     MPI_Recv(&data_out_i[rFROM], dataslice, MPI_DOUBLE, p , 900 , MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
257:    }
258:   }
259:  }
260:  if(world_rank == 0)
261:  {
262:   cout << world_rank << " collecting done! - MPI_time: " << MPI_Wtime()-s_time <<
endl;
263:  }
```

First, we check whether more than one workers are running. If yes, we have to collect data from ws_temp-1 co-workers. It is always a good idea to collect data at the first worker with rank zero. This rank is always present. If there are more workers, then every worker with a greater rank than zero has to send (MPI_Send()) the real (data_out_r[]) and imaginary (data_out_i[]) part of its data chunk to process 0. Parallel to this, the zero rank worker has to receive (MPI_Recv()) these data chunks (dataslice) into the right place of the real and imaginary arrays. Although the right place has to be calculated before. Since we know the rank of the other workers we are waiting data from, we can easily calculate the starting (rFROM) and ending (rTO) points of the data chunks we are getting from the remote processes.

```
265:  // some Monitoring output to look what data did we calculate and collect
266:  if(world_rank == 0)
267:  {
268:    for(int n=0;n<p2_length;n++)
269:    cout << "-- Input " << n << ": " << data_in_r[n] << "," << data_in_i[n] << " --
output: " << data_out_r[n] << "," << data_out_i[n] << endl;
270:  }
271:
272:   // we loose data after this lines, any output should be done here
273:   delete[] data_out_i; // free up memory
274:   delete[] data_out_r;
275:   delete[] data_in_i;
276:   delete[] data_in_r;
277:   delete[] buffer_r;
278:   delete[] buffer_i;
279:   delete[] rb_buffer_r;
280:   delete[] rb_buffer_i;
281:
282:  // MPI end - we loose contact with the other nodes at this point
283:  MPI_Finalize();
284:
285:   return 0;
286: }
```

At the end, we output our original inputs and the calculated FFT in a list. This can be saved and processed on. At the end, we have to free up our buffers in memory (delete[] some buffer). At this point, our data is erased from memory. We close the MPI environment for our active workers as well and from this point, no communication can be done between workers. The different processes can still work on and do arbitrary duties, but the MPI environment can't be initialized again.

11.3.2.1. 11.3.2.1 Finalize workaround

As we mentioned, after starting with m workers and needing only n where $m > n$, we can let the unneeded workers to quit. In our example, we just did an MPI_Finalize() and exit() on them, which did the trick in this case. What we have to be aware of, is that with closing some of the workers dynamically, they will be acting as dead at collective communications. This can lead to wrong behaviour, even to the freezing of the program (e.g. MPI_Barrier() would cause such a hang). Fortunately there is an easy workaround for this matter. Before letting the unwanted workers stop, we can define a new communicator with the wanted workers list and use this communicator after the quitting of the unwanted processes. Thus, in our example, we switch from the

MPI_COMM_WORLD communicator to an arbitrary MPI_COMM_ANYTHING one. Some basic steps for that will be presented as follows.

```
1: // MPI_Finalize() megoldas
  2:
  3:  int world_size;
  4:  int world_rank;
  5:
  6:  // Az MPI kornyezet inditasa - az MPI_Finalize fuggvenyig hasznalhatjuk az MPI
fuggvenyeket.
  7:     MPI_Init(NULL, NULL);
  8:
  9:  s_time=MPI_Wtime();
 10:  // processzusok szama
 11:     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
 12:     // processzusunk rangja
 13:     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 14:
 15:  // hanyan vannak a csoportban
 16:  MPI_Group wgroup;
 17:  MPI_Comm_group(MPI_COMM_WORLD, &wgroup);
 18:
 19:  // vagjuk le a szuksegtelen munkasokat
 20:  MPI_Group newgroup;
 21:  MPI_Group_range_excl(wgroup, 1, { last_active_worker, world_size-1, 1},
&new_group);
 22:
 23:  // we create a new communicator
 24:  MPI_Comm MPI_COMM_ANYTHING;
 25:  MPI_Comm_create(MPI_COMM_WORLD, newgroup, &MPI_COMM_ANYTHING);
 26:
 27:  if(MPI_COMM_ANYTHING == MPI_COMM_NULL)
 28:  {
 29:     // we are unwanted workers
 30:   MPI_Finalize();
 31:   exit(0);
 32:  }
 33:
```

The first rows are already familiar, we initialize the MPI environment, enumerate the active workers and get our rank. We do get the group of processes in our MPI world and store it into an MPI_Group structure. An MPI group is an ordered list of processes. Because it's ordered, processes have a *rank* in the group. A *communicator* consists of a *group* and some rules regarding communication. Our default communicator is MPI_COMM_WORLD, it is our communication world, hence the name. After we have our group of processes, we have to chop unwanted workers and copy the wanted ones into a new group. This is done by the MPI_Group_range_excl() function. Its first input is the original group (wgroup). The second input is the number of ranges (triplets) coming in the third parameter. The third parameter is a set of triplets (ranges), with the meaning of 'start of range' (last_active_worker), 'end of range' (world_size-1) and 'step size in range' (1). The output of the function is the new, tailored group (new_group). With this new group, we can create a new communicator with the help of the MPI_Comm_create() function. The function returns MPI_COMM_NULL if such a process is calling it that is not the part of the new group. We can use this fact for selecting who has to leave the MPI world.

Another, rather rough method could be to check the number of workers at the beginning, and if it doesn't fit our idea, simply quit the program with a short message about the required process number.

12. 12 Mandelbrot

The Mandelbrot set is a set of points, where a special property holds. To this property colors can be assigned, thus representing the set in colorful and spectacular images.

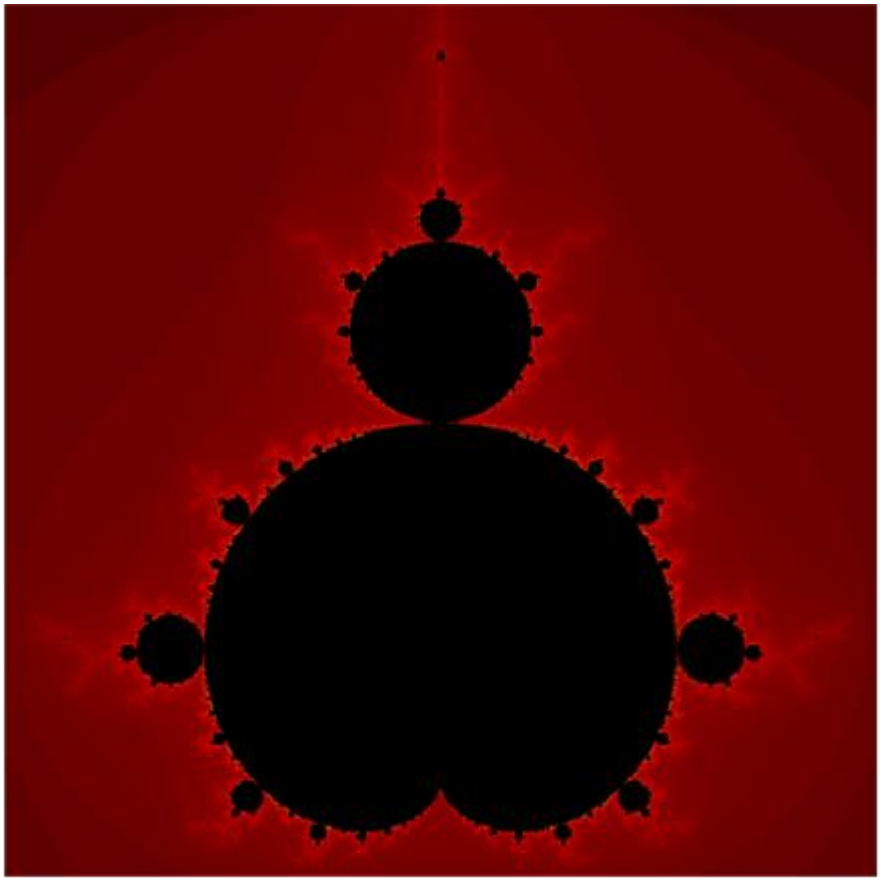

Figure 17: Mandelbrot subset generated by our serial program

This special set is named after the Polish-born, French and American mathematician, Benoit Mandelbrot, who made extensive studies on the field of fractal geometry. Fractals are mathematical sets, where an extra fractal dimension is present. This dimension is a statistical index of complexity, thus, a ratio of detail change with the change of the measuring scale.

The Mandelbrot set is a never-ending fractal shape, which can be zoomed in an arbitrary scale and its special forms are always repeating themselves in it. This is also referred as self-similarity.

The mathematical definition is as follows:

$$M = \{c \in \mathbb{C} | \lim_{n \to \infty} \mathbb{Z}_n \neq \infty\} \tag{12.1}$$

where:

$$Z_0 = cZ_{n+1} = Z_n^2 + c \tag{12.2}$$

According to this, the Mandelbrot set is the set of c complex numbers, where the recursive function Z_n for the number c is not infinite, when n approaches infinity. Infinity is now the attractor of this function, as used in context of fractals.

Regarding the picture above (17), the Mandelbrot set is the black region in the middle. The colored points are outside the set and have different colors set by their n iteration values.

To be able to calculate such sets, we have to make some practical, yet inevitable moves. Our first problem is to handle the question of infinity. We can't test in finite time whether a function approaches infinity or not. It is not practical to test whether it approaches a very-very big number either. Fortunately, it can be proven that if the absolute value of Z gets bigger than 2, thus, its distance from $0 + i0$ is more than 2, it will run to infinity. That's a big help to check this in our program code.

The maximum iteration number has to be more than a few tens, to be able to check whether a Z is going over 2 or not. However, this value should be tuned to the given resolution we work with. At a given resolution, it doesn't make sense to raise this iteration count, since we will loose data because of the resolution. If we want to make a very big resolution (practically zoomable) image, we have to raise the maximum iteration count as well. This will, of course, take more time to compute.

If we have our data and coloring scheme, we have to choose some good looking palette for this operation.

In our example, we simply take the n values and assign the following R, G, B values to them:

Table 6: RGB color settings for n values

n value	R value	G value	B value
0-255	n	0	0
256-511	255	n	0
512-767	255	255	n

This color scheme is going from black to red, then from red to yellow and from yellow to white. This has a fancy impression of flames, depicted in a zoomed in image generated by our example program (18).

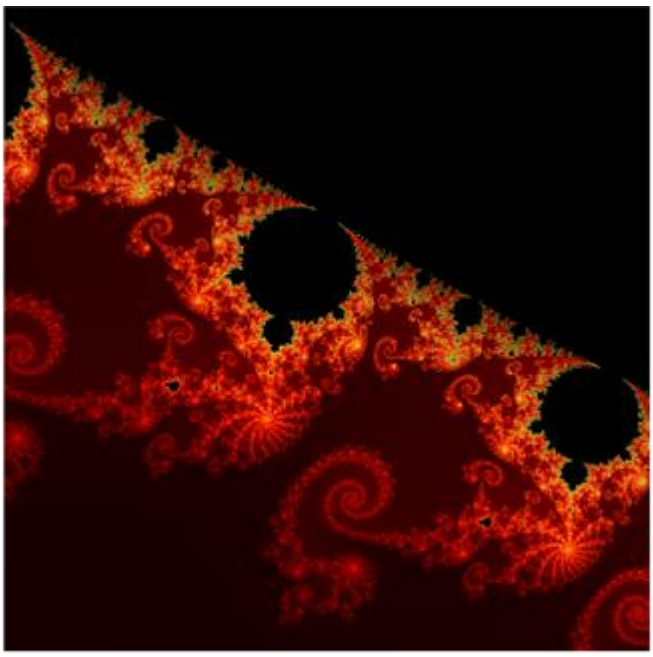

Figure 18: Mandelbrot subset generated by serial program

12.1. 12.1 Mandelbrot set serial example

In this subsection, the serial code for generating Mandelbrot sets will be discussed.

As usual, the first lines of the code start with include lines and constant definitions.

```
1: #include <cstdlib>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7: #include <time.h>
  8:
```

```
  9: using namespace std;
 10: const int imgX=3000; //  horizontal image resolution
 11: const int imgY=3000; //  vertical image resolution
 12: const int iter_n=3000; // max. iteration number
 13: const double yMin= -0.135;   // Mandelbrot scene y - range
 14: const double yMax= -0.115;
 15: const double xMin= -0.79;  // Mandelbrot scene x -range
 16: const double xMax= -0.77;
 17:
 18: int img_array[imgX][imgY] = {0};  // our MAndelbrot set values array
```

The actual resolution of our output image is set in imgX and imgY to 3000. The iter_n value declares the maximum iteration count at different points in our investigated complex plane. The actual complex plane is set with two ranges between yMin and yMax, and xMin and xMax. This later plane will be walked through point by point with the previous image resolution. The img_array[imgX] will store the calculated n values.

```
19:
 20: // convergation function - base of the Mandelbrot set value generation
 21: // it will get two parameters (x and y coordinates) and will give an iteration
count in return
 22: int converges(double cx,double cy)
 23: {
 24:  int n=0;
 25:  double zx=0;
 26:  double new_zx=0;
 27:  double zy=0;
 28:  // we iterate until max. iteration count iter_n, or until z^2 (complex!) runs over
4 - this means, our series will run to infinity, so it's not part of the set
 29:  while( (n<iter_n) && (zx*zx + zy*zy)<4 )
 30:  {
 31:      // z * z => new_zx = (zx*zx - zy*zy)  new_zy = (zx*zy + zx*zy)   // we work
with complex numbers
 32:   // z*z + c = zx^2 - zy^2 +cx   +  i(zx*zy*2 + cy)
 33:   new_zx = zx*zx - zy*zy + cx;
 34:   zy = 2*zx*zy + cy;
 35:   zx = new_zx;
 36:   n++;
 37:  }
 38:  return n;
 39: }
 40:
```

The converges() function has two parameters, cx and cy which are the x, y -coordinates on our complex plane and returns the iteration number, n. Inside the function we define the missing necessary variables for the $z_{n+1} = z_n^2 + c$ calculation. The loop in the converges() function runs from zero until we reach the maximum iteration number (iter_n) or we exceed the 2 unit boundary of $0 + i0$. Since $\sqrt{x^2 + y^2}$ has to be calculated, we can simplify it to $x^2 + y^2$ and check whether it is larger than 4.

```
40:
 41:
 42: int main(int argc, char **argv)
 43: {
 44:     if(argc<2)
 45:  {
 46:     cout << "Usage:" <<endl;
 47:     cout<<argv[0]<<" out_file.ppm"<<endl;
 48:     exit(1);
 49:  }
 50:  double resX=0;  // Resolution of our iteration steps
 51:  double resY=0; // this will be calculated by (yMax-yMin) / imgY later..
 52:  double cx=xMin; // calculation will start at this point and we will change this
dynamically
 53:  double cy=yMin;
 54:  time t mytime1,mytime2; // we will show some timing data
 55:
 56:     ofstream myfile;  // we will write to this file
 57:
```

```
 58:  string filename1(argv[1]);
 59:  // filename1 = "mandel.ppm";
 60:  char *fileName1 = (char*)filename1.c_str();
 61:
 62:     //prepare the step resolution
 63:  resX = (xMax-xMin) / imgX;
 64:  resY = (yMax-yMin) / imgY;
 65:
```

The main() function starts with processing the necessary arguments. If the argument count (argc) is less than 2, we print what it should have been. We wait for an output name. We defined the Mandelbrot set plane in yMin, yMax, xMin, xMax and our graphical resolution in imgX and imgY . From these, we can calculate our step size (resX and resY) by which we move on our set plane. The starting point will be xMin and yMin and we handle the actual position in cx and cy. We also define some time_t variables for timing and prepare our output file with the name given in our argument.

```
65:
 66:  time(&mytime1);  // we get a time value at this point
 67:
 68:  // we do the calculation for every point of our complex plane, thus on our 2D
image with appropriate steps
 69:  for(int i=0;i<imgX;i++)
 70:  {
 71:   cy=yMin;  // at every new step in X direction, we start at the first Y value
 72:   for(int j=0;j<imgY;j++)
 73:   {
 74:      img_array[i][j]=converges(cx,cy);   // we get the number of convergation steps
or the maximum value (iter_n)
 75:     // cout << img_array[i][j] << endl;
 76:      cy=cy+resY;
 77:   }
 78:   cx=cx+resX;
 79:  }
 80:  // we get another time at this point, so we can calculate the elapsed time for the
calculation
 81:     time(&mytime2);
 82:
 83:  cout << "Time elapsed during calculation: " << mytime2-mytime1 << " secs." <<
endl;;
```

After getting a current time into mytime1 we start with our main embedded loops. We calculate the appropriate value for every point of our image (going from 0 to imgX and from 0 to imgY). Every new line starts from yMin so we have to assign this value to cy. In the inner loop, we calculate the convergence value of the given cx, cy point and put it into the img_array[] array. At the end, we get the new current time into mytime2 and output the difference of them, that is the required running time for the embedded loops.

```
82:
 83:  cout << "Time elapsed during calculation: " << mytime2-mytime1 << " secs." <<
endl;;
 84:
 85:  // file IO
 86:  myfile.open(fileName1);
 87:  myfile << "P3\r\n";
 88:  myfile << imgX;
 89:  myfile << " ";
 90:  myfile << imgY;
 91:  myfile << "\r\n";
 92:  myfile << "255\r\n";
 93:
 94:     // we have to colour our dataset. Actually, the members of the Mandelbrot set
are used to be the same colour (black?) and have from point of visualisations view no
interest.
 95:  // the outer points are represented by assigning colours to their iteration steps
and this generates vivid forms and colors
 96:  for(int i=0;i<imgX;i++)
 97:  {
 98:   for(int j=0;j<imgY;j++)
 99:   {
```

```
100:
101:    if( (img_array[i][j] < 256) )  // we go from black to red in this range
102:    {
103:     myfile << img_array[i][j] << " 0 0"; // (int)(84*pow(img_array[i][j],0.2)) << "
0 0"; //myfile << img_array[i][j] << " 0 0";
104:    }
105:    else if( img_array[i][j] < 512)  // we go from red to yellow in this range
106:    {
107:     myfile << "255 " << img_array[i][j]-256 << " 0";
108:    }
109:    else if( img_array[i][j] < 768)  // we go from yellow to white in this range
110:    {
111:     myfile << "255 255 " << img_array[i][j]-512;
112:    }
113:    /*  // we could refine our palette for more resolution, more iteration-step
images
114:    else if( img_array[i][j] < 1024)
115:    {
116:     myfile << 1024-img_array[i][j] << " 255 255";
117:    }
118:    else if( img array[i][j] < 1280)
119:    {
120:     myfile << "0 "  << 1280-img_array[i][j] << " 255";
121:    }
122:    else if( img_array[i][j] < 1536)
123:    {
124:     myfile << "0 0 "  << 1536-img_array[i][j];
125:    }
126:    */
127:    else  // everything else is black
128:    {
129:        myfile << "0 0 0 ";
130:    }
131:
132:    myfile << " ";
133:
134:   }
135:         myfile << "\r\n";
136:  }
137:
138:   myfile.close();  // we close our file
139:
140:   time(&mytime2);  // and give another elapsed time info (IO included)
141:
142:   cout << "Time elapsed total: " << mytime2-mytime1 << " secs \r\n";
143:   return 0;
```

Our output will be an .PPM file, again. After setting the filename and the image parameters, we have to output our data into our image file. Since the image file consists of R,G,B triplets, we have to convert our Mandelbrot set data into this range. To be more precise, our dataset in the img_array[] array contains the Mandelbrot set and all other points not belonging to the set. If the convergence values of the set don't reach the maximum iteration number, the given coordinates of the dataset don't belong to the Mandelbrot set. We will color it to different colors. The values below 256 will set the R (red) channel from zero to maximum with a zero G (green) and B (blue) channel. This means a color gradient from black to red. Similarly, the next 256 values ($255 < n < 512$) are colored from red to $yellow$ and the next 256 values are going from $yellow$ to $white$. Raising the resolution and defining further colors (commented out) generate additional gradients to the palette.

At the end, we close our .PPM file and output some running time for that.

By adjusting the yMin, yMax, xMin and xMax values, we can "zoom in" on the plane.

12.2. 12.2 Parallel versions of the Mandelbrot set generator

Since one of the spectacular property of the Mandelbrot set is that its borderline is definable to infinity, an arbitrary zoom can be applied on the generated image of the set. However, this needs to generate images real-time or generate very high resolution images, or both. Since every point of an image needs to run an

approximation function, the whole procedure needs a lot of computing power. This is a case of parallelization again. In the next few examples, we show some work sharing strategies, their advantages and drawbacks.

12.2.1. 12.2.1 Master-Slave, an initial solution

In our first example, we show a master-slave communication model of the working nodes. The basic concept is to set a master node, practically the node with the lowest rank, and let all other nodes be slaves. Slaves ask the master for work, which in turn gives the slaves one point on the complex plane to make the approaching calculation and collects data from them. The idea is simple. Every process can work at its own speed, according to the complexity of its task. If the task is done, data is sent back to the master and new data is obtained. The master does only the data sharing and collecting task, and at the end, it shuts down all the slaves. Slaves work in an infinite loop and will stop only when the master shuts them down.

The first lines of the code are similar to the serial version.

```
1: #include <cstdlib>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7: #include <time.h>
  8: #include <mpi.h>
  9:
 10: using namespace std;
 11: const int imgX=3000; //  horizontal image resolution
 12: const int imgY=3000; //  vertical image resolution
 13: const int iter_n=3000; // max. iteration number
 14: const double yMin= -0.135;   // Mandelbrot scene y - range
 15: const double yMax= -0.115;
 16: const double xMin= -0.79;  // Mandelbrot scene x -range
 17: const double xMax= -0.77;
 18: int img_array[imgX][imgY] = {0};  // our MAndelbrot set values array
 19:
 20: // convergation function - base of the Mandelbrot set value generation
 21: // it will get two parameters (x and y coordinates) and will give an iteration
count in return
 22: int converges(double cx,double cy)
 23: {
 24:  int n=0;
 25:  double zx=0;
 26:  double new_zx=0;
 27:  double zy=0;
 28:  // we iterate until max. iteration count iter_n, or until z^2 (complex!) runs over
4 - this means, our series will run to infinity, so it's not part of the set
 29:  while( (n<iter_n) && (zx*zx + zy*zy)<4 )
 30:  {
 31:      // z * z => new_zx = (zx*zx - zy*zy)  new_zy = (zx*zy + zx*zy)   // we work
with complex numbers
 32:   // z*z + c = zx^2 - zy^2 +cx   +  i(zx*zy*2 + cy)
 33:   new_zx = zx*zx - zy*zy + cx;
 34:   zy = 2*zx*zy + cy;
 35:   zx = new_zx;
 36:   n++;
 37:  }
 38:  return n;
 39: }
 40:
 41:
 42: int main(int argc, char **argv)
```

The only new line is the mpi.h inlcude file. The main() function starts with some variable definitions already familiar. We define here a MPI_Status variable named status to be able to get the status (sender, message tag, error status) of an MPI message later. The answer[4] array will hold the result structure of a message (id of sender, iteration count0 or -1 for first request, X coordinate, Y coordinate). The coordinates[2] structure will

hold the coordinates of our image processed and the structure question[2] will hold the real and imaginary part of the processed point in our complex plane.

```
42: int main(int argc, char **argv)
 43: {
 44:   //variables for MPI communication:
 45:   int id, nproc;
 46:   MPI_Status status;
 47:   int answer[4];
 48:   int coordinates[2];
 49:   double question[2];
 50:
 51:   double resX=0;  // Resolution of our iteration steps
 52:   double resY=0; // this will be calculated by (yMax-yMin) / imgY later..
 53:   double cx=xMin; // calculation will start at this point and we will change this
dynamically
 54:   double cy=yMin;
 55:   double s_time, e_time;
 56:
 57:   // Initialize MPI:
 58:   MPI_Init(&argc, &argv);
 59:   // Get my rank:
 60:   MPI_Comm_rank(MPI_COMM_WORLD, &id);
 61:   // Get the total number of processors:
 62:   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 63:
 64:   MPI_Barrier(MPI_COMM_WORLD) ;//for precize timing
 65:
 66:   if(id == 0){ //Master
```

After the variable definitions we initialize the MPI environment. We get our rank into id and the size of the MPI World we see into nproc. With the MPI_Barrier() function, we synchronize all workers to this point, everyone has to reach this point to go on in any process. This is necessary now for the precise measurement of elapsed time during distributed work.

The Master-Slave method starts at this point.

```
66:   if(id == 0){ //Master
 67:
 68:     if(argc<2)
 69:       {
 70:  cout << "Usage:" <<endl;
 71:  cout<<argv[0]<<" out_file.ppm"<<endl;
 72:  MPI Abort(MPI COMM WORLD, 1);
 73:  exit(1);
 74:       }
 75:
 76:     ofstream myfile;  // we will write to this file
 77:     string filename1(argv[1]);
 78:     // filename1 = "mandel.ppm";
 79:     char *fileName1 = (char*)filename1.c_str();
 80:
 81:     //prepare the step resolution
 82:     resX = (xMax-xMin) / imgX;
 83:     resY = (yMax-yMin) / imgY;
 84:
 85:     s_time=MPI_Wtime(); // we get a time value at this point
 86:
 87:     // we do the calculation for every point of our complex plane,
 88:     // thus on our 2D image with appropriate steps
 89:
 90:     for(int i=0;i<imgX;i++)
 91:       {
 92:  cy=yMin;  // at every new step in X direction, we start at the first Y value
 93:  for(int j=0;j<imgY;j++)
 94:    {
 95:       MPI Recv(answer, 4, MPI INT, MPI ANY SOURCE, 1, MPI COMM WORLD,
 96:        &status);
 97:      // answer[0] -- id from whom
```

```
 98:      // answer[1] -- '-1' or the number of iterations
 99:      // answer[2] -- X coordinate
100:      // answer[3] -- Y coordinate
101:
102:       if(answer[1]>=0){ //not the first answer
103:         img_array[answer[2]][answer[3]]=answer[1];   // we get the number
104:         //of convergation steps or the maximum value (iter_n)
105:       }
106:      coordinates[0]=i;   //new coordinates to calculate
107:      coordinates[1]=j;
108:       MPI_Send(coordinates, 2, MPI_INT, answer[0], 2, MPI_COMM_WORLD);
109:       question[0]=cx;
110:       question[1]=cy;
111:       MPI_Send(question, 2, MPI_DOUBLE, answer[0], 3, MPI_COMM_WORLD);
112:       cy=cy+resY;
113:     }
114:   cx=cx+resX;
115:       }
116:     //the remaining answers:
```

If the rank our process (id) is 0 then we are the master process. If our argument count is ok, we open a file for output and set the step resolution and get an actual time into s_time. At this point, we start the embedded loop where we go through the image points. For every image point, we wait for a message from any of the slaves. At first, this is only a request for some work, but later, this is both an answer and a request for work. If it is an answer as well, we put the received data to the appropriate coordinate. The id (answer(1)), the iteration number (answer[1]) and the coordinates of the calculated image point (answer[2], answer[3]) are part of the answer. The slave indicates that this is the first answer, thus, it is only a request for work by setting the iteration number to -1. After receiving an answer, the master sends out a new point for processing. This consists of an MPI_Send() of the new image coordinates ("where") and the questioned cx , cy values. We hand out all the points one-by-one until no coordinates remain. After the last sends, we have to collect the last answer as well, and send out a termination request to the slaves. This is done by sending a -1 coordinate.

The next few lines do tasks similarly to the serial version. We output the elapsed time of the above procedure, and output the collected whole dataset into an image file. We also show the elapsed time for the I/O operation.

If the rank of the process is greater than zero, the actual process is a slave. Slaves request work, do their jobs and send results back to the master. Every slave starts with a special message, with that it indicates a request for work only (answer[0] is the rank id , answer[1] is -1), then starts an infinite loop of work. In this loop, we receive the new coordinates to process. If we receive a termination request, thus coordinates[0] < 0, the "infinite" loop will be breaked and after finalizing the MPI environment, the program will stop. Until termination, the slaves receive the image coordinates (coordinates), the set coordinates (question) and according to them, generate an iteration count with the converges() function. The 4-element-answer is constructed and will be sent back to the master.

The above example is called an initial example. Although the idea to share small portions to slaves on request is good, the calculation of one point is too small a problem compared to the communication overhead.

The program is tested on two different architectures. As mentioned in the introduction of Part II, A is an SMP machine and B is a cluster machine. The table shows the job times on both machines with different number of cores.

Table 7: Master-Slave communications, with Master asking for one point

nproc	A-v1		B-v1	
1	56.0s		89s	
12	9.9s	5.7x	24s	3.7x
24	16.6s	3.4x	25s	3.6x
48	22.6s	2.5x	31s	2.9x
96	25.2s	2.2x	51s	1.7x
192	29.4s	1.9x	38s	2.3x

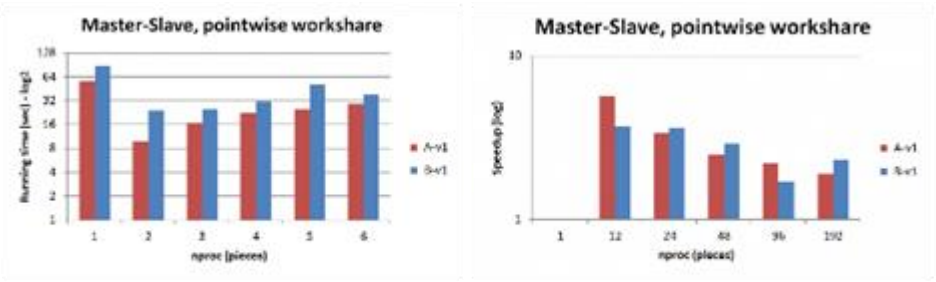

Figure 19: Running times and speedup

As can be seen in the table, the speed-up trend with raising number of cores is much more a slow-down on both architectures. However, even with the communication overhead, there is a speed-up compared to a one core run.

A better idea would be to give a bigger portion of work for one node, as done in the next example.

12.2.2. 12.2.2 Master-Slave - a better solution

The second example code works very similarly to the first one, except for that we hand out bigger parts, whole lines to the slaves for processing. The first lines of the program are very similar to the ones in the first version. We introduce a new array, img_line[imgY] with which we will communicate the lines between master and slaves.

```
1: #include <cstdlib>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7: #include <time.h>
  8: #include <mpi.h>
  9:
 10: using namespace std;
 11: const int imgX=3000; //  horizontal image resolution
 12: const int imgY=3000; //  vertical image resolution
 13: const int iter_n=3000; // max. iteration number
 14: const double yMin= -0.135;   // Mandelbrot scene y - range
 15: const double yMax= -0.115;
 16: const double xMin= -0.79;  // Mandelbrot scene x -range
 17: const double xMax= -0.77;
 18: int img_array[imgX][imgY] = {0};  // our MAndelbrot set values array
 19: int img_line[imgY] = {0};
 20:
 21: // convergation function - base of the Mandelbrot set value generation
 22: // it will get two parameters (x and y coordinates) and will give an iteration
count in return
 23: int converges(double cx,double cy)
 24: {
 25:  int n=0;
 26:  double zx=0;
 27:  double new_zx=0;
 28:  double zy=0;
 29:  // we iterate until max. iteration count iter_n, or until z^2 (complex!) runs over
4 - this means, our series will run to infinity, so it's not part of the set
 30:  while( (n<iter_n) && (zx*zx + zy*zy)<4 )
 31:  {
 32:      // z * z => new_zx = (zx*zx - zy*zy)  new_zy = (zx*zy + zx*zy)   // we work
with complex numbers
 33:   // z*z + c = zx^2 - zy^2 +cx   +  i(zx*zy*2 + cy)
 34:   new_zx = zx*zx - zy*zy + cx;
 35:   zy = 2*zx*zy + cy;
 36:   zx = new_zx;
 37:   n++;
 38:  }
```

```
 39:  return n;
 40: }
 41:
 42:
 43: int main(int argc, char **argv)
```

The main() function starts also similarly to the first example, we just omit the coordinates[] variable, since we work with whole lines.

```
43: int main(int argc, char **argv)
 44: {
 45:   //variables for MPI communication:
 46:   int id, nproc;
 47:   MPI_Status status;
 48:   int answer[2];
 49:   double question;
 50:
 51:   double resX=0;  // Resolution of our iteration steps
 52:   double resY=0; // this will be calculated by (yMax-yMin) / imgY later..
 53:   double cx=xMin; // calculation will start at this point and we will change this
dynamically
 54:   double cy=yMin;
 55:   double s_time, e_time; // we will show some timing data
 56:
 57:   // Initialize MPI:
 58:   MPI_Init(&argc, &argv);
 59:   // Get my rank:
 60:   MPI_Comm_rank(MPI_COMM_WORLD, &id);
 61:   // Get the total number of processors:
 62:   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 63:
 64:   MPI_Barrier(MPI_COMM_WORLD); //for precize timing
 65:
 66:
 67:   if(id == 0){ //Master
 68:
 69:     if(argc<2)
 70:       {
 71:  cout << "Usage:" <<endl;
 72:  cout<<argv[0]<<" out_file.ppm"<<endl;
 73:  MPI_Abort(MPI_COMM_WORLD, 1);
 74:  exit(1);
 75:       }
 76:
 77:     ofstream myfile;  // we will write to this file
 78:     string filename1(argv[1]);
 79:     // filename1 = "mandel.ppm";
 80:     char *fileName1 = (char*)filename1.c_str();
 81:
 82:     //prepare the step resolution
 83:     resX = (xMax-xMin) / imgX;
 84:     resY = (yMax-yMin) / imgY;
 85:
 86:     s_time=MPI_Wtime(); // we get a time value at this point
 87:
 88:     // we do the calculation for every point of our complex plane, thus on our 2D
image with appropriate steps
 89:     for(int i=0;i<imgX;i++)
 90:       {
 91:  MPI_Recv(answer, 2, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,
 92:    &status);
 93:  // answer[0] -- from whom
 94:  // answer[1] -- '-1' or the X coordinate
 95:
 96:  if(answer[1]>=0){ //not the first answer
 97:    MPI_Recv(&img_array[answer[1]][0], imgY, MPI_INT, answer[0],
 98:      2, MPI_COMM_WORLD, &status);
 99:  }
100:  MPI_Send(&i, 1, MPI_INT, answer[0], 3, MPI_COMM_WORLD);
101:  MPI_Send(&cx, 1, MPI_DOUBLE, answer[0], 4, MPI_COMM_WORLD);
102:
```

```
103:  cx=cx+resX;
104:       }
105:
```

As a master we will receive a whole line in img_array[answer[1]][0] where answer[1] is the X coordinate, the number of the processed line.

```
105:
106:     //the remaining answers:
107:     int term=-1;
108:     for(int i=1;i<nproc;++i){
109:       MPI_Recv(answer, 2, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);
110:       MPI_Recv(&img_array[answer[1]][0], imgY, MPI_INT, answer[0],
111:      2, MPI_COMM_WORLD, &status);
112:
113:       //sending the termination signal
114:       MPI_Send(&term, 1, MPI_INT, answer[0], 3, MPI_COMM_WORLD);
115:     }
116:
117:     // we get another time at this point, so we can calculate the elapsed time for
the calculation
118:    e_time=MPI_Wtime();
119:
120:     cout << "Time elapsed during calculation: " << e_time-s_time << " secs." <<
endl;;
121:
122:     // file IO
123:     myfile.open(fileName1);
124:     myfile << "P3\r\n";
125:     myfile << imgX;
126:     myfile << " ";
127:     myfile << imgY;
128:     myfile << "\r\n";
129:     myfile << "255\r\n";
130:
131:     // we have to colour our dataset. Actually, the members of the Mandelbrot set
are used to be the same colour (black?) and have from point of visualisations view no
interest.
132:     // the outer points are represented by assigning colours to their iteration
steps and this generates vivid forms and colors
133:     for(int i=0;i<imgX;i++)
134:       {
135:  for(int j=0;j<imgY;j++)
136:   {
137:
138:     if( (img_array[i][j] < 256) )  // we go from black to red in this range
139:       {
140:         myfile << img_array[i][j] << " 0 0"; // (int)(84*pow(img_array[i][j],0.2))
<< " 0 0"; //myfile << img_array[i][j] << " 0 0";
141:       }
142:     else if( img_array[i][j] < 512)  // we go from red to yellow in this range
143:       {
144:         myfile << "255 " << img_array[i][j]-256 << " 0";
145:       }
146:     else if( img_array[i][j] < 768)  // we go from yellow to white in this range
147:       {
148:         myfile << "255 255 " << img_array[i][j]-512;
149:       }
150:     /*  // we could refine our palette for more resolution, more iteration-step
images
151:         else if( img_array[i][j] < 1024)
152:         {
153:         myfile << 1024-img_array[i][j] << " 255 255";
154:         }
155:         else if( img_array[i][j] < 1280)
156:         {
157:         myfile << "0 "  << 1280-img_array[i][j] << " 255";
158:         }
159:         else if( img_array[i][j] < 1536)
160:         {
161:         myfile << "0 0 "  << 1536-img_array[i][j];
```

```
162:          }
163:     */
164:     else  // everything else is black
165:       {
166:         myfile << "0 0 0 ";
167:       }
168:
169:     myfile << " ";
170:
171:   }
172:         myfile << "\r\n";
173:       }
174:
175:   myfile.close();  // we close our file
176:
177:   e_time=MPI_Wtime(); // and give another elapsed time info (IO included)
178:
179:   cout << "Time elapsed total: " << e_time-s_time << " secs \r\n";
180:   }
```

After the iteration, we have to collect the remaining, last lines and write the whole data into a .PPM file again.

If we are slaves, we have to send out a first message in which we indicate that we are ready to work. After this first message, we will get into a loop, where we get a row number and send the whole row back.

```
180:   }
181:   else{ //Slave
182:     //prepare the step resolution
183:     resX = (xMax-xMin) / imgX;
184:     resY = (yMax-yMin) / imgY;
185:
186:     int i;
187:     answer[0]=id;
188:     answer[1]=-1;
189:     MPI_Send(answer, 2, MPI_INT, 0, 1, MPI_COMM_WORLD);
190:
191:     while(1){
192:       MPI_Recv(&i, 1, MPI_INT, 0, 3, MPI_COMM_WORLD, &status);
193:       if(i<0) break; //got termination command!
194:       answer[1]=i;
195:
196:       MPI_Recv(&question, 1, MPI_DOUBLE, 0, 4, MPI_COMM_WORLD, &status);
197:
198:       cy=yMin;  // at every new step in X direction, we start at the first Y value
199:
200:       for(int j=0;j<imgY;j++)
201:  {
202:    img_line[j]=converges(question,cy);
203:    cy=cy+resY;
204:  }
205:       MPI_Send(answer, 2, MPI_INT, 0, 1, MPI_COMM_WORLD);
206:       MPI_Send(img_line, imgY, MPI_INT, 0, 2, MPI_COMM_WORLD);
207:     }
208:
209:   }
210:
211:   // Terminate MPI:
212:   MPI_Finalize();
213:
214:   return 0;
215: }
```

The received cx value in question indicates the line number, and we go through this line and calculate the iteration count for every cx, cy position. The series of these positions will get into the img_line[] array.

This amount of work seems to be big enough, the communication overhead is not a big issue any more, as can be seen in table 8. The speed-up is much better on both architectures. This method brings almost a linear scale up until 192 processors.

Table 8: Master-Slave communications, with Master asking for one line

nproc	A-v2		B-v2	
1	56.0s		89s	
12	5.3s	10.6x	7.4s	12.0x
24	2.5s	22.4x	3.7s	26.8x
48	1.3s	43.1x	2.2s	40.5x
96	0.7s	80.0x	2.0s	44.5x
192	0.4s	140.0x	2.1s	42.4x
384	0.3s	186.7x	2.1s	42.4x
512	0.2s	280.0x		

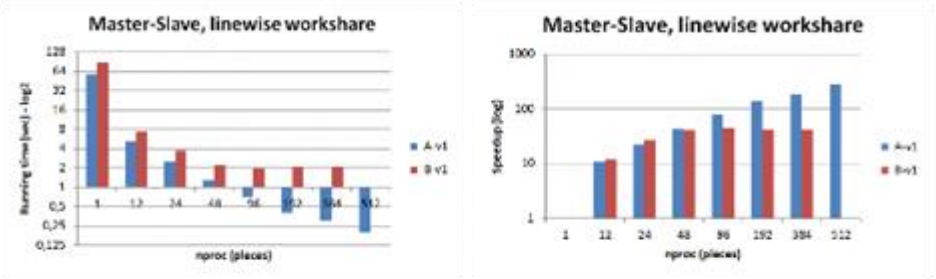

Figure 20: Running times and speedup

12.2.3. 12.2.3 Loop splitting

In the third example, we change our work distribution strategy. The so called loop splitting technique distributes various iterations of a loop to different workers. Since we do not hand out blocks, but different fragments from different positions in our set, some balancing is done. In this example, the master only distributes and collects work and results. The code is similar except the work distribution step.

```
1: #include <cstdlib>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7: #include <time.h>
  8: #include <mpi.h>
  9:
 10: using namespace std;
 11: const int imgX=3000; //  horizontal image resolution
 12: const int imgY=3000; //  vertical image resolution
 13: const int iter n=3000; // max. iteration number
 14: const double yMin= -0.135;   // Mandelbrot scene y - range
 15: const double yMax= -0.115;
 16: const double xMin= -0.79;  // Mandelbrot scene x -range
 17: const double xMax= -0.77;
 18: int img_array[imgX][imgY] = {0};  // our MAndelbrot set values array
 19:
 20: // convergation function - base of the Mandelbrot set value generation
 21: // it will get two parameters (x and y coordinates) and will give an iteration
count in return
 22: int converges(double cx,double cy)
 23: {
 24:  int n=0;
 25:  double zx=0;
 26:  double new_zx=0;
 27:  double zy=0;
```

```
 28:  // we iterate until max. iteration count iter_n, or until z^2 (complex!) runs over
4 - this means, our series will run to infinity, so it's not part of the set
 29:  while( (n<iter_n) && (zx*zx + zy*zy)<4 )
 30:  {
 31:      // z * z => new_zx = (zx*zx - zy*zy)  new_zy = (zx*zy + zx*zy)   // we work
with complex numbers
 32:   // z*z + c = zx^2 - zy^2 +cx   +  i(zx*zy*2 + cy)
 33:   new_zx = zx*zx - zy*zy + cx;
 34:   zy = 2*zx*zy + cy;
 35:   zx = new_zx;
 36:   n++;
 37:  }
 38:  return n;
 39: }
 40:
 41:
 42: int main(int argc, char **argv)
 43: {
 44:   //variables for MPI communication:
 45:   int id, nproc;
 46:   MPI_Status status;
 47:   int id_from;
 48:
 49:   double resX=0;  // Resolution of our iteration steps
 50:   double resY=0; // this will be calculated by (yMax-yMin) / imgY later..
 51:   double cx=xMin; // calculation will start at this point and we will change this
dynamically
 52:   double cy=yMin;
 53:   double s_time, e_time; // we will show some timing data
 54:
 55:   // Initialize MPI:
 56:   MPI_Init(&argc, &argv);
 57:   // Get my rank:
 58:   MPI_Comm_rank(MPI_COMM_WORLD, &id);
 59:   // Get the total number of processors:
 60:   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 61:
 62:   MPI_Barrier(MPI_COMM_WORLD); //for precize timing
 63:
 64:
 65:   if(id == 0){ //Master
 66:
 67:     if(argc<2)
 68:       {
 69:  cout << "Usage:" <<endl;
 70:  cout<<argv[0]<<" out_file.ppm"<<endl;
 71:  MPI_Abort(MPI_COMM_WORLD, 1);
 72:  exit(1);
 73:       }
 74:
 75:     ofstream myfile;  // we will write to this file
 76:     string filename1(argv[1]);
 77:     // filename1 = "mandel.ppm";
 78:     char *fileName1 = (char*)filename1.c_str();
 79:
 80:     s_time=MPI_Wtime(); // we get a time value at this point
 81:
 82:     //recieve
 83:     for(int j=1;j<nproc;++j){
 84:       MPI_Recv(&id_from, 1, MPI_INT, MPI_ANY_SOURCE, 1,
 85:         MPI_COMM_WORLD, &status);
 86:       //it is important to recive from ANY_SOURCE,
 87:       //as process execution may differ greatly
 88:
 89:       for(int i=id_from-1;i<imgX;i += (nproc-1))
 90:  MPI_Recv(&img_array[i][0], imgY, MPI_INT, id_from,
 91:    2, MPI_COMM_WORLD, &status);
 92:     }
 93:
 94:     // we get another time at this point, so we can calculate the elapsed time for
the calculation
```

If we are masters, we receive answers in a loop. The first slave to answer, will give us its rank, id_from. Working with this value, we can get every value starting from . Doing this for every worker (the order of incoming id-s vary) we get the fragments of the result.

The output is similar to the above versions.

```
94:     // we get another time at this point, so we can calculate the elapsed time for
the calculation
 95:    e_time=MPI_Wtime();
 96:
 97:     cout << "Time elapsed during calculation: " << e_time-s_time << " secs." <<
endl;;
 98:
 99:     // file IO
100:     myfile.open(fileName1);
101:     myfile << "P3\r\n";
102:     myfile << imgX;
103:     myfile << " ";
104:     myfile << imgY;
105:     myfile << "\r\n";
106:     myfile << "255\r\n";
107:
108:     // we have to colour our dataset. Actually, the members of the Mandelbrot set
are used to be the same colour (black?) and have from point of visualisations view no
interest.
109:     // the outer points are represented by assigning colours to their iteration
steps and this generates vivid forms and colors
110:     for(int i=0;i<imgX;i++)
111:       {
112:  for(int j=0;j<imgY;j++)
113:   {
114:
115:     if( (img_array[i][j] < 256) )  // we go from black to red in this range
116:       {
117:         myfile << img_array[i][j] << " 0 0"; // (int)(84*pow(img_array[i][j],0.2))
<< " 0 0"; //myfile << img_array[i][j] << " 0 0";
118:       }
119:     else if( img array[i][j] < 512)  // we go from red to yellow in this range
120:       {
121:         myfile << "255 " << img_array[i][j]-256 << " 0";
122:       }
123:     else if( img_array[i][j] < 768)  // we go from yellow to white in this range
124:       {
125:         myfile << "255 255 " << img_array[i][j]-512;
126:       }
127:     /*  // we could refine our palette for more resolution, more iteration-step
images
128:         else if( img_array[i][j] < 1024)
129:         {
130:         myfile << 1024-img_array[i][j] << " 255 255";
131:         }
132:         else if( img_array[i][j] < 1280)
133:         {
134:         myfile << "0 "  << 1280-img array[i][j] << " 255";
135:         }
136:         else if( img_array[i][j] < 1536)
137:         {
138:         myfile << "0 0 "  << 1536-img_array[i][j];
139:         }
140:     */
141:     else  // everything else is black
142:       {
143:         myfile << "0 0 0 ";
144:       }
145:
146:     myfile << " ";
147:
148:   }
149:         myfile << "\r\n";
150:       }
```

```
151:
152:   myfile.close();  // we close our file
153:
154:   e_time=MPI_Wtime(); // and give another elapsed time info (IO included)
155:
156:   cout << "Time elapsed total: " << e_time-s_time << " secs \r\n";
157:   }
```

If we are a slave, we have to serve these arbitrary data fragments the same way as the master.

```
157:   }
158:   else{ //Slave
159:     //because the slaves numbered 1..nproc:
160:     //id -> id-1
161:     //nproc -> nproc-1
162:
163:     //prepare the step resolution
164:     resX = (xMax-xMin) / imgX;
165:     resY = (yMax-yMin) / imgY;
166:
167:     //because of the Master-Slave execution process 0 does no work!
168:
169:     cx=cx+resX*(id-1); //jump to the right cx
170:     //beware, round-off error!
171:     //this value is slightly different from
172:     //adding resX startval times to cx
173:     //this will cause unspottable difference to the picture
174:
175:     for(int i=id-1;i<imgX;i += (nproc-1))
176:       {
177:  cy=yMin;  // at every new step in X direction,
178:  //we start at the first Y value
179:  for(int j=0;j<imgY;j++)
180:    {
181:      img_array[i][j]=converges(cx,cy);
182:      cy=cy+resY;
183:    }
184:
185:  cx=cx+resX*(nproc-1); //the steps are by nproc!
186:  //beware, round-off error!
187:
188:       }
189:
190:    MPI_Send(&id, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
191:     for(int i=id-1;i<imgX;i += (nproc-1))
192:       {
193:  MPI Send(&img array[i][0], imgY, MPI INT, 0, 2, MPI COMM WORLD);
194:       }
195:
196:   }
197:
198:   // Terminate MPI:
199:   MPI_Finalize();
200:
201:   return 0;
202: }
```

As a slave, we have to calculate the starting line (cx) where we have to start the calculation. This will be the cx=cx+resX*(id-1) value. This value will be slightly different from the cx value used in our previous examples. It is a round-off error, where adding resX id-1 times to cx is id-1 rounding at the addition, while adding resX*(id-1) is only one addition and a multiplication. This will slightly modify our image. To avoid this, we could change back to a loop of additions, to get the same rounding as in our previous examples.

After setting up the right cx and calculating the right lines, we have to send our id and the data to the master.

As shown in the following table, speed up is still very good, running times show that this method is also viable for this problem. However, the speed up is not as good as before.

Table 9: Loop splitting, master is not working

nproc	A-v3a		B-v3a	
1	56.0s		89s	
12	5.8s	9.7x	7.7s	11.6x
24	2.8s	20.0x	4.4s	20.2x
48	1.5s	37.3x	2.3s	38.7x
96	1.7s	32.9x	1.7s	52.3x
192	1.1s	50.9x	2.1s	42.4x
384	1.3s	43.1x	3.4s	26.2x
512	1.4s	40.0x		

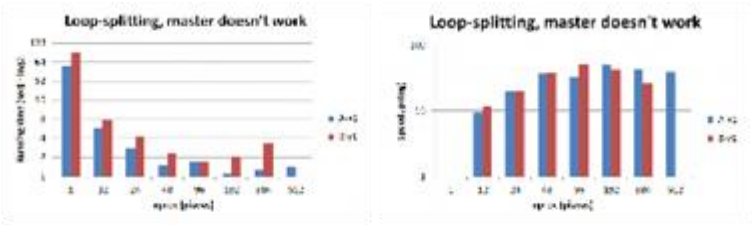

Figure 21: Running times and speedup

12.2.4. 12.2.4 Loop splitting variant

The fourth example is a variant of the loop splitting technique. What makes it different from the previous one is that the master will also perform some calculations. First, this sounds better, since more worker should do the job faster. In fact, it is true until some number of processes. After a while, the administrative overhead of the master process will make him the slowest process among the others and all the slaves have to wait for the master, that collects data only after it has done its job. The master has to receive results from all slaves in a row. In the previous example, the master process did the file name administration parallel to the slaves who began the work. Now, the master as worker has to do this extra.

The code is similar to the previous ones except where we check whether we are master or slave.

```
1: #include <cstdlib>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7: #include <time.h>
  8: #include <mpi.h>
  9:
 10: using namespace std;
 11: const int imgX=3000; //  horizontal image resolution
 12: const int imgY=3000; //  vertical image resolution
 13: const int iter_n=3000; // max. iteration number
 14: const double yMin= -0.135;   // Mandelbrot scene y - range
 15: const double yMax= -0.115;
 16: const double xMin= -0.79;  // Mandelbrot scene x -range
 17: const double xMax= -0.77;
 18: int img_array[imgX][imgY] = {0};  // our MAndelbrot set values array
 19:
 20: // convergation function - base of the Mandelbrot set value generation
 21: // it will get two parameters (x and y coordinates) and will give an iteration
count in return
 22: int converges(double cx,double cy)
 23: {
```

```
 24:  int n=0;
 25:  double zx=0;
 26:  double new_zx=0;
 27:  double zy=0;
 28:  // we iterate until max. iteration count iter_n, or until z^2 (complex!) runs over
4 - this means, our series will run to infinity, so it's not part of the set
 29:  while( (n<iter_n) && (zx*zx + zy*zy)<4 )
 30:  {
 31:      // z * z => new_zx = (zx*zx - zy*zy)  new_zy = (zx*zy + zx*zy)   // we work
with complex numbers
 32:   // z*z + c = zx^2 - zy^2 +cx   +  i(zx*zy*2 + cy)
 33:   new_zx = zx*zx - zy*zy + cx;
 34:   zy = 2*zx*zy + cy;
 35:   zx = new_zx;
 36:   n++;
 37:  }
 38:  return n;
 39: }
 40:
 41:
 42: int main(int argc, char **argv)
 43: {
 44:   //variables for MPI communication:
 45:   int id, nproc;
 46:   MPI_Status status;
 47:   int id_from;
 48:
 49:   double resX=0;  // Resolution of our iteration steps
 50:   double resY=0; // this will be calculated by (yMax-yMin) / imgY later..
 51:   double cx=xMin; // calculation will start at this point and we will change this
dynamically
 52:   double cy=yMin;
 53:   double s_time, e_time; // we will show some timing data
 54:   ofstream myfile;  // we will write to this file
 55:   char *fileName1;
 56:
 57:   // Initialize MPI:
 58:   MPI_Init(&argc, &argv);
 59:   // Get my rank:
 60:   MPI_Comm_rank(MPI_COMM_WORLD, &id);
 61:   // Get the total number of processors:
 62:   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 63:
 64:   MPI_Barrier(MPI_COMM_WORLD); //for precize timing
 65:
 66:
 67:   if(id == 0){ //Master
 68:
 69:     if(argc<2)
 70:       {
 71:  cout << "Usage:" <<endl;
 72:  cout<<argv[0]<<" out_file.ppm"<<endl;
 73:  MPI_Abort(MPI_COMM_WORLD, 1);
 74:  exit(1);
 75:       }
 76:
 77:     string filename1(argv[1]);
 78:     // filename1 = "mandel.ppm";
 79:     fileName1 = (char*)filename1.c_str();
 80:
 81:     //prepare the step resolution
 82:     s_time=MPI_Wtime(); // we get a time value at this point
 83:   }
 84:
```

If we are the master, we have to do some extra work by checking the arguments, preparing the output file and getting a time value. After this, all processes do the same: they calculate their part of work with the help of their rank (id) and calculate the starting cx value. Knowing the starting cx value, every process works on its cx+nproc*resX lines. After doing all the calculation and building the individual img_array[i][j] blocks, we have to send/receive them.

```
84:
 85:   //prepare the step resolution
 86:   resX = (xMax-xMin) / imgX;
 87:   resY = (yMax-yMin) / imgY;
 88:
 89:   cx=cx+resX*id; //jump to the right cx
 90:   //beware, round-off error!
 91:   //this value is slightly different from
 92:   //adding resX startval times to cx
 93:   //this will cause unspottable difference to the picture
 94:
 95:   for(int i=id;i<imgX;i += nproc)
 96:     {
 97:       cy=yMin;  // at every new step in X direction,
 98:       //we start at the first Y value
 99:       for(int j=0;j<imgY;j++)
100:  {
101:    img_array[i][j]=converges(cx,cy);
102:    cy=cy+resY;
103:  }
104:
105:       cx=cx+resX*nproc; //the steps are by nproc!
106:       //beware, round-off error!
107:    }
108:
109:   if(id != 0){ //Slaves sending
110:     MPI_Send(&id, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
111:     for(int i=id;i<imgX;i += nproc)
112:       {
113:  MPI_Send(&img_array[i][0], imgY, MPI_INT, 0, 2, MPI_COMM_WORLD);
114:       }
115:
116:   }else{//Master recieving
117:     for(int j=1;j<nproc;++j){
118:       MPI_Recv(&id_from, 1, MPI_INT, MPI_ANY_SOURCE, 1,
119:         MPI COMM WORLD, &status);
120:       //it is important to recive from ANY_SOURCE,
121:       //as process execution may differ greatly
122:
123:       for(int i=id_from;i<imgX;i += nproc)
124:  MPI_Recv(&img_array[i][0], imgY, MPI_INT, id_from,
125:    2, MPI_COMM_WORLD, &status);
126:     }
127:
128:     // we get another time at this point, so we can calculate the elapsed time for
the calculation
129:    e_time=MPI_Wtime();
130:
131:     cout << "Time elapsed during calculation: " << e time-s time << " secs." <<
endl;;
132:
133:     // file IO
134:     myfile.open(fileName1);
135:     myfile << "P3\r\n";
136:     myfile << imgX;
137:     myfile << " ";
138:     myfile << imgY;
139:     myfile << "\r\n";
140:     myfile << "255\r\n";
141:
142:     // we have to colour our dataset. Actually, the members of the Mandelbrot set
are used to be the same colour (black?) and have from point of visualisations view no
interest.
143:     // the outer points are represented by assigning colours to their iteration
steps and this generates vivid forms and colors
144:     for(int i=0;i<imgX;i++)
145:       {
146:  for(int j=0;j<imgY;j++)
147:   {
148:
149:     if( (img_array[i][j] < 256) )  // we go from black to red in this range
```

```
150:       {
151:         myfile << img_array[i][j] << " 0 0"; // (int)(84*pow(img_array[i][j],0.2))
<< " 0 0"; //myfile << img_array[i][j] << " 0 0";
152:       }
153:     else if( img_array[i][j] < 512)  // we go from red to yellow in this range
154:       {
155:         myfile << "255 " << img_array[i][j]-256 << " 0";
156:       }
157:     else if( img_array[i][j] < 768)  // we go from yellow to white in this range
158:       {
159:         myfile << "255 255 " << img_array[i][j]-512;
160:       }
161:     /*  // we could refine our palette for more resolution, more iteration-step
images
162:         else if( img_array[i][j] < 1024)
163:         {
164:         myfile << 1024-img_array[i][j] << " 255 255";
165:         }
166:         else if( img_array[i][j] < 1280)
167:         {
168:         myfile << "0 "  << 1280-img array[i][j] << " 255";
169:         }
170:         else if( img_array[i][j] < 1536)
171:         {
172:         myfile << "0 0 "  << 1536-img_array[i][j];
173:         }
174:     */
175:     else  // everything else is black
176:       {
177:         myfile << "0 0 0 ";
178:       }
179:
180:     myfile << " ";
181:
182:   }
183:         myfile << "\r\n";
184:       }
185:
186:   myfile.close();  // we close our file
187:
188:   e_time=MPI_Wtime(); // and give another elapsed time info (IO included)
189:
190:   cout << "Time elapsed total: " << e_time-s_time << " secs \r\n";
191:   }
192:
193:   // Terminate MPI:
194:   MPI_Finalize();
195:
196:   return 0;
197: }
```

If we are slaves (rank is not zero), we have to do several MPI_Send()s. The first will send our id and the followings will contain our part of data lines. If we are masters, we have to receive these lines by doing the iteration parametrised by the first received rank value. As masters, we have to do some file output as well. Table 10 shows the speed-up on both machines.

Table 10: Loop splitting, master is working as well

nproc	A-v3b		B-v3b	
1	56.0s		89s	
12	5.8s	9.7x	7.0s	12.7x
24	2.9s	19.3x	4.1s	21.7x
48	1.5s	37.3x	2.3s	38.7x
96	1.8s	31.1x	1.7s	52.3x
192	1.2s	46.7x	1.8s	49.4x
384	0.7s	80.0x	3.4s	26.2x
512	1.5s	37.3x		

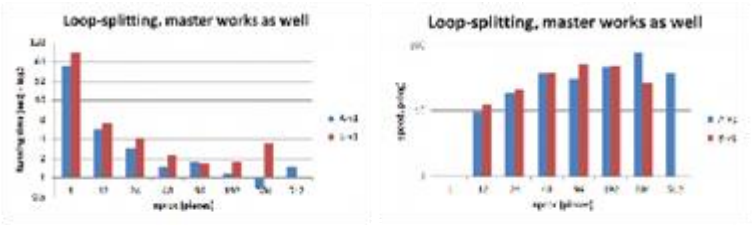

Figure 22: Running times and speedup

12.2.5. 12.2.5 Block scheduling

The Block scheduling is our next, well known strategy. In this method, we just split the dataset into consecutive blocks and hand out them for the working nodes. In our first block scheduling example, the master won't work again, it will only collect data from other workers, slaves. The code is unchanged until splitting the data.

```
1: #include <cstdlib>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7: #include <time.h>
  8: #include <mpi.h>
  9:
 10: using namespace std;
 11: const int imgX=3000; //  horizontal image resolution
 12: const int imgY=3000; //  vertical image resolution
 13: const int iter_n=3000; // max. iteration number
 14: const double yMin= -0.135;   // Mandelbrot scene y - range
 15: const double yMax= -0.115;
 16: const double xMin= -0.79;  // Mandelbrot scene x -range
 17: const double xMax= -0.77;
 18: int img_array[imgX][imgY] = {0};  // our MAndelbrot set values array
 19:
 20: // convergation function - base of the Mandelbrot set value generation
 21: // it will get two parameters (x and y coordinates) and will give an iteration
count in return
 22: int converges(double cx,double cy)
 23: {
 24:  int n=0;
 25:  double zx=0;
 26:  double new_zx=0;
 27:  double zy=0;
 28:  // we iterate until max. iteration count iter_n, or until z^2 (complex!) runs over
4 - this means, our series will run to infinity, so it's not part of the set
 29:  while( (n<iter_n) && (zx*zx + zy*zy)<4 )
```

```
 30:  {
 31:      // z * z => new_zx = (zx*zx - zy*zy)  new_zy = (zx*zy + zx*zy)    // we work
with complex numbers
 32:    // z*z + c = zx^2 - zy^2 +cx    +  i(zx*zy*2 + cy)
 33:    new_zx = zx*zx - zy*zy + cx;
 34:    zy = 2*zx*zy + cy;
 35:    zx = new_zx;
 36:    n++;
 37:   }
 38:   return n;
 39: }
 40:
 41:
 42: int main(int argc, char **argv)
 43: {
 44:    //variables for MPI communication:
 45:    int id, nproc;
 46:    MPI_Status status;
 47:    int id_from;
 48:    int startval;
 49:    int endval;
 50:
 51:    double resX=0;  // Resolution of our iteration steps
 52:    double resY=0; // this will be calculated by (yMax-yMin) / imgY later..
 53:    double cx=xMin; // calculation will start at this point and we will change this
dynamically
 54:    double cy=yMin;
 55:    double s_time, e_time; // we will show some timing data
 56:    ofstream myfile;  // we will write to this file
 57:    char *fileName1;
 58:
 59:    // Initialize MPI:
 60:    MPI_Init(&argc, &argv);
 61:    // Get my rank:
 62:    MPI_Comm_rank(MPI_COMM_WORLD, &id);
 63:    // Get the total number of processors:
 64:    MPI Comm size(MPI COMM WORLD, &nproc);
 65:
 66:    MPI_Barrier(MPI_COMM_WORLD); //for precize timing
 67:
 68:
 69:    if(id == 0){ //Master
 70:
 71:      if(argc<2)
 72:        {
 73:   cout << "Usage:" <<endl;
 74:   cout<<argv[0]<<" out_file.ppm"<<endl;
 75:   MPI_Abort(MPI_COMM_WORLD, 1);
 76:   exit(1);
 77:        }
 78:
 79:      string filename1(argv[1]);
 80:      // filename1 = "mandel.ppm";
 81:      fileName1 = (char*)filename1.c_str();
 82:
 83:      s_time=MPI_Wtime(); // we get a time value at this point
 84:
 85:      //recieving
 86:      for(int j=1;j<nproc;++j){
```

If we are master we start with collecting data as usual. The difference here is the block definition of the data to be received. In the first message, we got the id_from rank data and using this, we can calulate the starting position and size of the data block (endval - startval). We will receive a datablock of $(endval - startval)$ lines $\times imgY$.

```
86:      for(int j=1;j<nproc;++j){
 87:         MPI_Recv(&id_from, 1, MPI_INT, MPI_ANY_SOURCE, 1,
 88:           MPI_COMM_WORLD, &status);
 89:         //it is important to recive from ANY_SOURCE,
 90:         //as process execution may differ greatly
```

```
 91:
 92:       startval = imgX*(id_from-1)/(nproc-1);
 93:       endval =   imgX*id_from/(nproc-1);
 94:
 95:       //for(int i=startval;i<endval; ++i)
 96:  MPI_Recv(&img_array[startval][0], (endval-startval)*imgY, MPI_INT, id_from,
 97:    2, MPI_COMM_WORLD, &status);
 98:     }
 99:
100:     // we get another time at this point, so we can calculate the elapsed time for
the calculation
101:    e_time=MPI_Wtime();
102:
103:     cout << "Time elapsed during calculation: " << e_time-s_time << " secs." <<
endl;;
104:
105:     // file IO
106:     myfile.open(fileName1);
107:     myfile << "P3\r\n";
108:     myfile << imgX;
109:     myfile << " ";
110:     myfile << imgY;
111:     myfile << "\r\n";
112:     myfile << "255\r\n";
113:
114:     // we have to colour our dataset. Actually, the members of the Mandelbrot set
are used to be the same colour (black?) and have from point of visualisations view no
interest.
115:     // the outer points are represented by assigning colours to their iteration
steps and this generates vivid forms and colors
116:     for(int i=0;i<imgX;i++)
117:       {
118:  for(int j=0;j<imgY;j++)
119:   {
120:
121:     if( (img_array[i][j] < 256) )  // we go from black to red in this range
122:       {
123:         myfile << img_array[i][j] << " 0 0"; // (int)(84*pow(img_array[i][j],0.2))
<< " 0 0"; //myfile << img_array[i][j] << " 0 0";
124:       }
125:     else if( img_array[i][j] < 512)  // we go from red to yellow in this range
126:       {
127:         myfile << "255 " << img_array[i][j]-256 << " 0";
128:       }
129:     else if( img_array[i][j] < 768)  // we go from yellow to white in this range
130:       {
131:         myfile << "255 255 " << img_array[i][j]-512;
132:       }
133:     /*  // we could refine our palette for more resolution, more iteration-step
images
134:         else if( img_array[i][j] < 1024)
135:         {
136:         myfile << 1024-img_array[i][j] << " 255 255";
137:         }
138:         else if( img_array[i][j] < 1280)
139:         {
140:         myfile << "0 "  << 1280-img_array[i][j] << " 255";
141:         }
142:         else if( img_array[i][j] < 1536)
143:         {
144:         myfile << "0 0 "  << 1536-img_array[i][j];
145:         }
146:     */
147:     else  // everything else is black
148:       {
149:         myfile << "0 0 0 ";
150:       }
151:
152:     myfile << " ";
153:
154:   }
155:         myfile << "\r\n";
```

```
156:       }
157:
158:   myfile.close();  // we close our file
159:
160:   e_time=MPI_Wtime(); // and give another elapsed time info (IO included)
161:
162:   cout << "Time elapsed total: " << e_time-s_time << " secs \r\n";
163:
164:   }else{//Slave
165:
166:     //because the slaves numbered 1..nproc:
167:     //id -> id-1
168:     //nproc -> nproc-1
169:
170:     //prepare the step resolution
171:     resX = (xMax-xMin) / imgX;
172:     resY = (yMax-yMin) / imgY;
173:
174:     startval = imgX*(id-1)/(nproc-1);
```

The code is again the same as before, except the slaves data block calculation. This goes the same way as by the master side. We have to calculate the startval and endval values and to send the datalines accordingly.

```
174:     startval = imgX*(id-1)/(nproc-1);
175:     endval =   imgX*id/(nproc-1);
176:
177:     cx=cx+resX*startval; //jump to the right cx
178:     //beware, round-off error!
179:     //this value is slightly different from
180:     //adding resX startval times to cx
181:     //this will cause unspottable difference to the picture
182:
183:     for(int i=startval;i<endval;++i)
184:       {
185:  cy=yMin;  // at every new step in X direction,
186:  //we start at the first Y value
187:  for(int j=0;j<imgY;j++)
188:    {
189:      img_array[i][j]=converges(cx,cy);
190:      cy=cy+resY;
191:    }
192:
193:  cx=cx+resX;
194:       }
195:
196:     //sending
197:     MPI_Send(&id, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
198:     MPI_Send(&img_array[startval][0], (endval-startval)*imgY, MPI_INT, 0, 2,
MPI_COMM_WORLD);
199:   }
200:
201:   // Terminate MPI:
202:   MPI_Finalize();
203:
204:   return 0;
205: }
```

This kind of data split is simple, the consecutive blocks can be sent at once. However, the workload could be more imbalanced, since we take contiguous parts of the dataset. Calculating the Mandelbrot set, contiguous parts can be fully inside the borderline, thus every point has to iterate till the maximum, and some parts could be fully out of the set. The running times scale quite well on both machines, though (11).

Table 11: Block scheduling, master doesn't work

nproc	A-v4a		B-v4a	
1	56.0s		89s	
12	13.2s	4.2x	17.3s	5.1x
24	6.7s	8.4x	8.3s	10.7x
48	3.3s	17.0x	4.1s	21.7x
96	1.7s	32.9x	2.2s	40.5x
192	1.1s	50.9x	5.2s	17.1x
384	1.2s	46.7x	12.1s	7.4x
512	0.9s	62.2x		

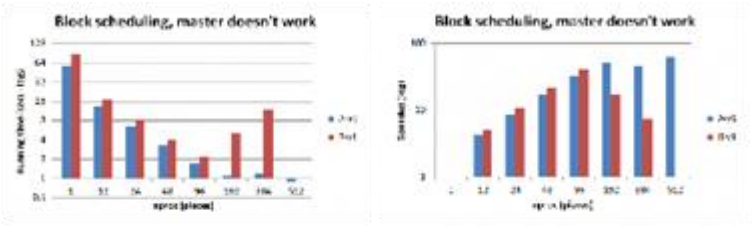

Figure 23: Running times and speedup

12.2.6. 12.2.6 Block scheduling variant

Our last parallel Mandelbrot set generator example is a variant of the block scheduling. In this variant, the master is also working. This case is similar to the loop splitting variant (12.2.4). The consecutive data blocks are handed out, and the master gets one as well. This is on one hand good, more workers are working. On the other hand, if the master is slower than some of the slaves, they all have to wait for the master and after one can send, the others have to wait for each other, causing a jam, which can be shown at the test runs as well.

```
1: #include <cstdlib>
  2: #include <iostream>
  3: #include <fstream>
  4: #include <sstream>
  5: #include <math.h>
  6: #include <vector>
  7: #include <time.h>
  8: #include <mpi.h>
  9:
 10: using namespace std;
 11: const int imgX=3000; //  horizontal image resolution
 12: const int imgY=3000; //  vertical image resolution
 13: const int iter_n=3000; // max. iteration number
 14: const double yMin= -0.135;   // Mandelbrot scene y - range
 15: const double yMax= -0.115;
 16: const double xMin= -0.79;  // Mandelbrot scene x -range
 17: const double xMax= -0.77;
 18: int img_array[imgX][imgY] = {0};  // our MAndelbrot set values array
 19:
 20: // convergation function - base of the Mandelbrot set value generation
 21: // it will get two parameters (x and y coordinates) and will give an iteration
count in return
 22: int converges(double cx,double cy)
 23: {
 24:  int n=0;
 25:  double zx=0;
 26:  double new_zx=0;
 27:  double zy=0;
 28:  // we iterate until max. iteration count iter_n, or until z^2 (complex!) runs over
```

```
4 - this means, our series will run to infinity, so it's not part of the set
 29:  while( (n<iter_n) && (zx*zx + zy*zy)<4 )
 30:  {
 31:      // z * z => new_zx = (zx*zx - zy*zy)  new_zy = (zx*zy + zx*zy)   // we work
with complex numbers
 32:   // z*z + c = zx^2 - zy^2 +cx   +  i(zx*zy*2 + cy)
 33:   new_zx = zx*zx - zy*zy + cx;
 34:   zy = 2*zx*zy + cy;
 35:   zx = new_zx;
 36:   n++;
 37:  }
 38:  return n;
 39: }
 40:
 41:
 42: int main(int argc, char **argv)
 43: {
 44:   //variables for MPI communication:
 45:   int id, nproc;
 46:   MPI_Status status;
 47:   int id from;
 48:   int startval;
 49:   int endval;
 50:
 51:   double resX=0;  // Resolution of our iteration steps
 52:   double resY=0; // this will be calculated by (yMax-yMin) / imgY later..
 53:   double cx=xMin; // calculation will start at this point and we will change this
dynamically
 54:   double cy=yMin;
 55:   double s_time, e_time; // we will show some timing data
 56:   ofstream myfile;  // we will write to this file
 57:   char *fileName1;
 58:
 59:   // Initialize MPI:
 60:   MPI_Init(&argc, &argv);
 61:   // Get my rank:
 62:   MPI Comm rank(MPI COMM WORLD, &id);
 63:   // Get the total number of processors:
 64:   MPI_Comm_size(MPI_COMM_WORLD, &nproc);
 65:
 66:   MPI_Barrier(MPI_COMM_WORLD); //for precize timing
 67:
 68:
 69:   if(id == 0){ //Master
 70:
 71:     if(argc<2)
 72:       {
 73:  cout << "Usage:" <<endl;
 74:  cout<<argv[0]<<" out_file.ppm"<<endl;
 75:  MPI Abort(MPI COMM WORLD, 1);
 76:  exit(1);
 77:       }
 78:
 79:     string filename1(argv[1]);
 80:     // filename1 = "mandel.ppm";
 81:     fileName1 = (char*)filename1.c_str();
 82:
 83:     //prepare the step resolution
 84:     s_time=MPI_Wtime(); // we get a time value at this point
 85:   }
 86:
 87:   //prepare the step resolution
 88:   resX = (xMax-xMin) / imgX;
 89:   resY = (yMax-yMin) / imgY;
 90:
 91:   //because of the Master-Slave execution process 0 does no work!
 92:
 93:   startval = imgX*id/nproc;
 94:   endval =   imgX*(id+1)/nproc;
 95:
```

The code is different at the calculation of startval and endval. The master gets also a part, so we divide work into nproc parts.

```
95:
 96:   cx=cx+resX*startval; //jump to the right cx
 97:   //beware, round-off error!
 98:   //this value is slightly different from
 99:   //adding resX startval times to cx
100:   //this will cause unspottable difference to the picture
101:
102:   for(int i=startval;i<endval;++i)
103:     {
104:       cy=yMin;  // at every new step in X direction,
105:       //we start at the first Y value
106:       for(int j=0;j<imgY;j++)
107:  {
108:    img_array[i][j]=converges(cx,cy);
109:    cy=cy+resY;
110:  }
111:
112:       cx=cx+resX;
113:     }
114:
115:   if(id != 0){ //Slaves sending
116:     MPI_Send(&id, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
117:     //for(int i=startval;i<endval; ++i)
118:     //{
119:     MPI_Send(&img_array[startval][0], (endval-startval)*imgY, MPI_INT, 0, 2,
MPI_COMM_WORLD);
120:     //}
121:
122:   }else{//Master recieving
123:     for(int j=1;j<nproc;++j){
124:       MPI_Recv(&id_from, 1, MPI_INT, MPI_ANY_SOURCE, 1,
125:         MPI_COMM_WORLD, &status);
126:       //it is important to recive from ANY_SOURCE,
127:       //as process execution may differ greatly
128:
129:       startval = imgX*id_from/nproc;
130:       endval =   imgX*(id_from+1)/nproc;
131:
132:       //for(int i=startval;i<endval; ++i)
133:  MPI_Recv(&img_array[startval][0], (endval-startval)*imgY, MPI_INT, id_from,
134:    2, MPI COMM WORLD, &status);
135:     }
136:
137:     // we get another time at this point, so we can calculate the elapsed time for
the calculation
138:    e_time=MPI_Wtime();
139:
140:     cout << "Time elapsed during calculation: " << e_time-s_time << " secs." <<
endl;;
141:
142:     // file IO
143:     myfile.open(fileName1);
144:     myfile << "P3\r\n";
145:     myfile << imgX;
146:     myfile << " ";
147:     myfile << imgY;
148:     myfile << "\r\n";
149:     myfile << "255\r\n";
150:
151:     // we have to colour our dataset. Actually, the members of the Mandelbrot set
are used to be the same colour (black?) and have from point of visualisations view no
interest.
152:     // the outer points are represented by assigning colours to their iteration
steps and this generates vivid forms and colors
153:     for(int i=0;i<imgX;i++)
154:       {
155:  for(int j=0;j<imgY;j++)
156:   {
```

```
157:
158:     if( (img_array[i][j] < 256) )  // we go from black to red in this range
159:       {
160:         myfile << img_array[i][j] << " 0 0"; // (int)(84*pow(img_array[i][j],0.2))
<< " 0 0"; //myfile << img_array[i][j] << " 0 0";
161:       }
162:     else if( img_array[i][j] < 512)  // we go from red to yellow in this range
163:       {
164:         myfile << "255 " << img_array[i][j]-256 << " 0";
165:       }
166:     else if( img_array[i][j] < 768)  // we go from yellow to white in this range
167:       {
168:         myfile << "255 255 " << img_array[i][j]-512;
169:       }
170:     /*  // we could refine our palette for more resolution, more iteration-step
images
171:         else if( img_array[i][j] < 1024)
172:         {
173:         myfile << 1024-img_array[i][j] << " 255 255";
174:         }
175:         else if( img array[i][j] < 1280)
176:         {
177:         myfile << "0 "  << 1280-img_array[i][j] << " 255";
178:         }
179:         else if( img_array[i][j] < 1536)
180:         {
181:         myfile << "0 0 "  << 1536-img_array[i][j];
182:         }
183:     */
184:     else  // everything else is black
185:       {
186:         myfile << "0 0 0 ";
187:       }
188:
189:     myfile << " ";
190:
191:   }
192:         myfile << "\r\n";
193:       }
194:
195:   myfile.close();  // we close our file
196:
197:   e_time=MPI_Wtime(); // and give another elapsed time info (IO included)
198:
199:   cout << "Time elapsed total: " << e_time-s_time << " secs \r\n";
200:   }
201:
202:   // Terminate MPI:
203:   MPI_Finalize();
204:
205:   return 0;
206: }
```

The rest of the code is similar, everyone is calculating, then the master is collecting data. The slaves may wait for that and that is an increasing problem with increasing process numbers. The run times on the A (SMP ccNUMA) and B (Fat-node tree cluster) machines are shown in table 12. The speed-up seems to break down at higher number of working nodes, this is likely due to the jam effect because of the working master node. This has an effect earlier on the B architecture due the communication delay.

Table 12: Block scheduling, master works as well

nproc	A-v4b		B-v4b	
1	56.0s		89s	
12	12.7s	4.4x	15.8s	5.6x
24	6.4s	8.8x	8.3s	10.7x
48	3.5s	16.0x	5.1s	17.5x
96	1.9s	29.5x	3.8s	23.4x
192	1.2s	46.7x	6.0s	14.8x
384	1.4s	40.0x	12.5s	7.1x
512	1.2s	46.7x		

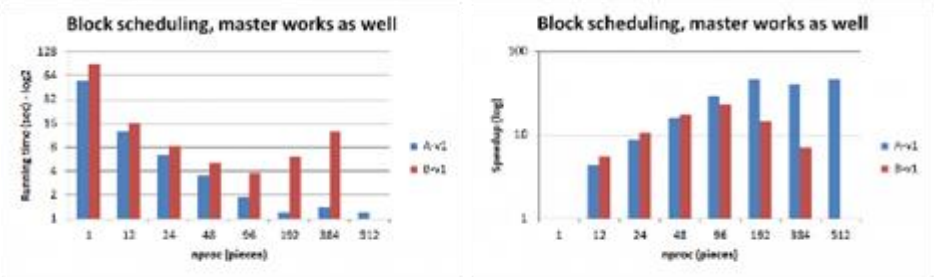

Figure 24: Running times and speedup

12.2.7. 12.2.7 Thoughts on work sharing

The first thing to think about at parallelization is the strategy of work sharing. This has to be designed carefully since hidden delays, unsynchronized work can greatly decrease the gain of some processes coupled power. We must analyse the algorithm we want to use according to its need for communication size and structure. The final set-up should consider the underlying architecture (SMP, cluster) and network properties (throughput, latency) to reach desired speed. Communication has always had an overhead. It is not a good idea to send and receive too small chunks of data. However, by using big data chunks, we take the risk of making the work of nodes unbalanced. Depending on the algorithm and architecture, we have to find the optimal configuration for best run. It is always a good idea to make some benchmarks and test runs with smaller data to speed up the configuration process.

In our Mandelbrot set generation example, we have taken a part of the set for investigation, which is imbalanced in the sense of complexity. As shown in the picture 18, some lines include big portions of the set itself (black), some lines have almost no inner points. This means, that different lines need different computational time, thus every process works with a different speed. A Master-Slave algorithm variant, with whole line sharing is proved to be the best solution for this problem. The comparison of the six different work share variants run on an SMP machine (A) and a Fat Node Cluster (B) are shown in table X and Y.

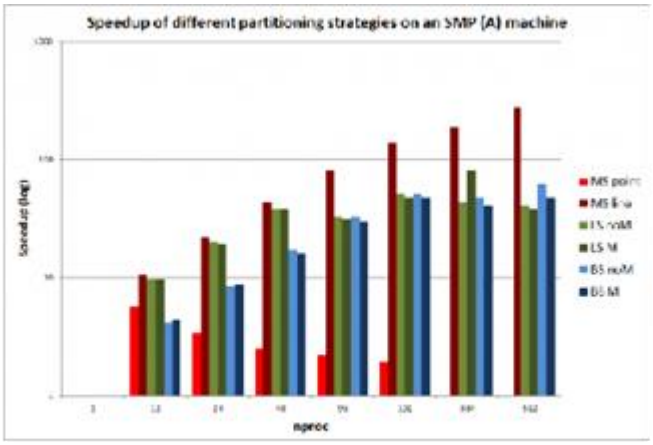

Figure 25: Speedup of the six method on an SMP machine

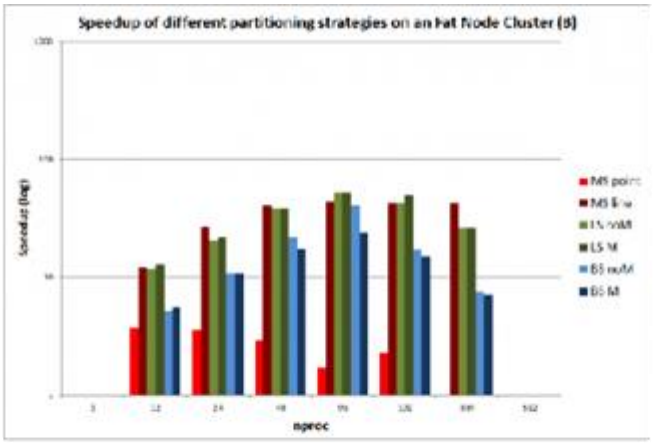

Figure 26: Speedup of the six method on a Fat Node Cluster (48core nodes)

13. 13 Raytracing

13.1. 13.1 Visualization

One large group of computational problems is visualization. Everyone has seen modern computer games and cinema tricks. The main objective of this type of graphics is to render a 3D scene to the 2D monitor as close to reality as possible. Computer graphics has a long and diversified history. There are many different 3D rendering procedures suited for the actual demand. In interactive 3D games speed is a key factor, so objects are commonly simplified and represented by planes and textures; and different tricks are used to mime effects. Where real-time rendering is not necessary, slower but more complex procedures can be used.

The most realistic but the most resource intense rendering method is raytracing. This method doesn't use tricks, it models the whole physical phenomena of the spread of light. Raytracing means tracing the path of rays entering our eyes. Our eye is sensitive to an interval of the optical spectrum (ca. between 380 nm - 780 nm). Electromagnetic waves of this spectrum are propagating straight and are reflected by surfaces. The reflection is different for every wavelength, so light, which is in general a composite of several waves with different wavelengths, will change according to the properties of the surface. This effect influences the perceived color of the light. Some materials absorb light, some materials guide light and the geometric structure of the surface causes reflection of light in different ways. This and further phenomena can be described and modelled mathematically, and as such, very realistic 2D images of a 3D scene can be made.

The basic idea is to follow the way of photons in every direction from light-sources and track their reflections, dispersions or possible loss. This method would need tremendous computing power and most of the calculations would be done unnecessary. To avoid such unwanted work, the raytracing technique reverses the direction and counts only with rays, which have an effect on the observer, thus we start observing the way of rays from our eye and follow up backwards to the light-sources. Using this reverse method, we eliminate all unnecessary calculations, but still, we need a lot of computational power.

13.2. 13.2 Raytracing example

In our example we show the power of parallel computing on a raytraced scene by sharing the work among processors. Since the scene is already defined for every worker and rendered pixels are independent, the parallel work will be quite effective. Calculating the path of one ray is, however, a resource intensive task since several reflections will occur and we have to process the ray for every pixel for every light-source. Two raytraced scene is shown in figure 27. One is with five spheres, visualising reflections, ambient and diffuse lighting and one with a 512 sphere block, as a sample setup for big number of reflections for heavy workload. This later is not so spectacular any more, since a lot of reflections occur on the inner sphere surfaces.

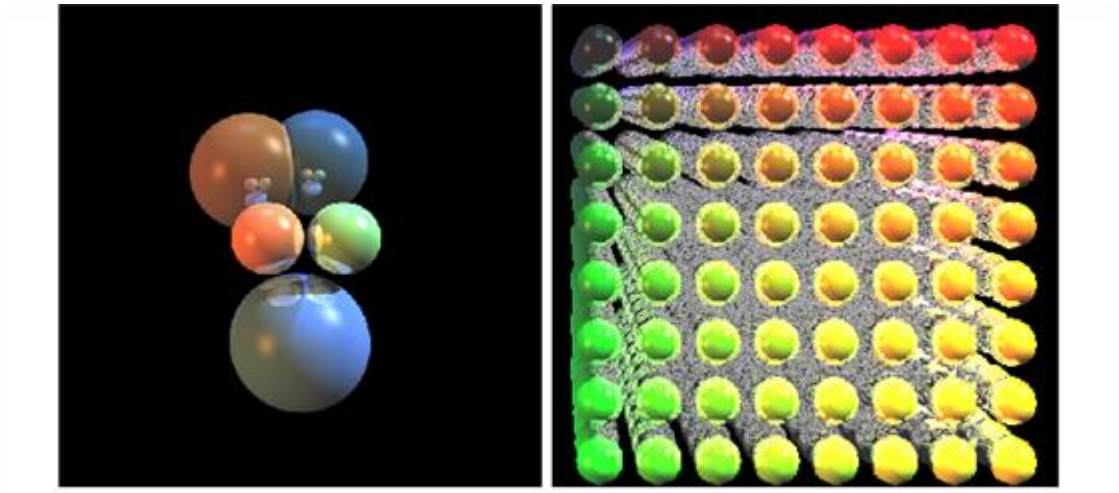

Figure 27: Raytraced scene with 5 and 512 spheres generated by our parallel program

First let us concentrate on the raytracer functionality itself. Secondly we will discuss the parallelization.

13.2.1. 13.2.1 Serial raytracer

The raytracing code is based on the documentation of POV-Ray (Persistence of Vision Ray-Tracer [PovRay]) and on the PHP script of Darren Abbey.

Our example code will implement some basic functionality of a raytracer. It will render five coloured spheres on a black background with three light-sources. There will be ambient lighting, diffuse lighting, specular (mirroring) lighting. These together give the Phong reflection model. The surfaces will provide a mirroring effect.

The 2D image will be calculated pixel-by-pixel. We can look at this 2D image as a viewing plane in our 3D scene, or as a window, through which we see the 3D world. Our rendering will use this abstraction, too. We will define a special point,from where we observe the 3D scene (camera). We will cast a ray from our view point to every coordinate of the viewing plane. This defines a line which will intersect a sphere or point out to nowhere. If the latter occurs, our actual pixel on the image will be of background colour (black in our example). If we intersect a sphere, we have to calculate the lighting parameters of that point and get a colour. These calculations will include

- , where we test whether a light-source illuminates this point or not,

- , which is perpendicular to the surface and will be needed to calculate diffuse lighting and the reflected ray,
- , to get the specular (mirroring) component of lighting and the colour of the reflection.

In the example, we have to initialize some values for the calculation. We define a class Point, which represents a vector in our 3D world. Since we will work with these vectors, it is appropriate to define some methods for this class. The base of this class is a three element array of the type double (p[3]). We can set values to this array by the set() method and get values from it with the get() method. We can get the length of such a vector with the length() method.

The class has also a method to give the normalized vector (norm()), vector operators +, -, *, scalar operators *, / and a friend function point_dot(), with which we can calculate the dot-product of two Point objects.

The new definitions are in the ray6v7.h include file.

```
// include for the Parallel Raytracer
#include <iostream>
#include <math.h>

class Point{   // definition of class Point - a class for 3 dimensional vectors
 private:
  double p[3];

 public:
  Point(){
    p[0]=0;
    p[1]=0;
    p[2]=0;
  }
  Point(double x, double y, double z){
    p[0]=x;
    p[1]=y;
    p[2]=z;
  }
  void set(double x, double y, double z){
    p[0]=x;
    p[1]=y;
    p[2]=z;
  }
  double get(int i){
    return p[i];
  }
  double length() const {
    return( sqrt( p[0]*p[0] + p[1]*p[1] + p[2]*p[2] ) );
  }
  Point norm() const { // normalizing
 double l=length(); // we know the length of the vector
 Point normalized;
 normalized.p[0] = p[0] / l;
 normalized.p[1] = p[1] / l;
 normalized.p[2] = p[2] / l;
 return normalized;
  }
  Point operator+(const Point& a) const{ // output: vector
 Point output;
 output.p[0]=a.p[0]+p[0];
 output.p[1]=a.p[1]+p[1];
 output.p[2]=a.p[2]+p[2];
 return output;
  }
  Point operator-(const Point& a) const{ // output: vector
 Point output;
 output.p[0]=p[0]-a.p[0];
 output.p[1]=p[1]-a.p[1];
 output.p[2]=p[2]-a.p[2];
 return output;
  }
  Point operator*(double scal) const {
 Point output;
```

```
 output.p[0]=p[0]*scal;
 output.p[1]=p[1]*scal;
 output.p[2]=p[2]*scal;
 return output;
  }
  Point operator/(double scal) const {
 Point output;
 output.p[0]=p[0]/scal;
 output.p[1]=p[1]/scal;
 output.p[2]=p[2]/scal;
 return output;
  }
  Point operator*(const Point& a) const {
 Point output;
 output.p[0]=a.p[0]*p[0];
 output.p[1]=a.p[1]*p[1];
 output.p[2]=a.p[2]*p[2];
 return output;
  }
  friend double point_dot(const Point&, const Point&);
};



double point_dot(const Point& a, const Point& b)  // dot-product of two vectors (Points)
{
 return (a.p[0]*b.p[0]+b.p[1]*a.p[1]+a.p[2]*b.p[2]);
}


// diffuse & specular lighting switches
const bool Diffuse_test = true;
const bool Specular_test = true;
const bool Reflection_test = true;


// the next variables need to be defined global
  int max_Reclev=7;    // reflexion limit
  double bg_Color[3] = {0,0,0}; // background color
  int max_Dist = 3000;  // maximum distance (size of rendering world)


  // scene definition
  // Lightsources (direction(x,y,z),color(r,g,b))
  // three lightsources
  double lights[3][6] =
  {
    {-1, 0,-0.5,0.8,0.4,0.1},
 { 1,-1,-0.5,1,  1,  1},
 { 0,-1,   0,0.1,0.2,0.5}
  };

  // Spheres
  // center 3, radius 1, reflection 1, color(rgb) 3, phongsize 1, amount 1
  double spheres[5][10] =
  {
  {-1.05, 0  ,4,1  ,0.5,1   , 0.5,0.25, 40,0.8},
  { 1.05, 0  ,4,1  ,0.5,0.5 , 1  ,0.5 , 40,0.8},
  {0   , 3  ,5,2  ,0.5,0.25, 0.5,1 ,30,0.4},
  {-1   ,-2.3,9,2  ,0.5 ,0.5 , 0.3,0.1 ,30,0.4},
  { 1.3 ,-2.6,9, 1.8,0.5,0.1 , 0.3,0.5,30,0.4}
   };

const int X=4800;
const int Y=4800;

// MPI
// we have to define a temporary array to store values in
double t_frame[X][Y][3] = {0};
```

In the first lines we will use further temporary and global variables in the code. The variable T is used for storing a distance value. It will be used at the ray-sphere intersection calculation and will be discussed there in detail. The obj_Amnt and light_Amnt variables will store the number of objects (spheres) and the number of light-sources in the scene. We define an ambient lighting into the scene stored in ambient(r,g,b) and we give the camera position in our 3D space in camera(x,y,z).

```
1: #include <iostream>
  2: #include <fstream>
  3: #include <math.h>
  4: #include <vector>
  5:
  6: #include "ray6v7.h"
  7:
  8: using namespace std;
  9:
 10: Point Trace(Point &P_vv, Point &D_vv, int recLev);
 11:
 12: double T = -1; //  ray - sphere intersection result - this will be a distance, if -
1, then no intersection occurs..
 13: int obj_Amnt;
 14: int light Amnt;
 15:
 16: // ambient and  camera values
 17: double ddd = 0.2;
 18: Point ambient(ddd,0.2,0.2);  // diffuse lighting in RGB - not 8bit, yet (some grey
- 0.2,0.2,0.2)
 19: Point camera(-4,0,-16);  // camera position - 0,0,-3
 20:
```

There are three functions in our example. The main() function as the standard main function in C++ programs, the calcRaySphereIntersection() function, which calculates the intersection point of a given ray and a sphere, and the Trace() function, which calculates the color of a given x, y point on the viewing plane.

After defining some temporary and global variables, we start our main() function with the counting of our objects and lightsources in the scene. Since we defined them in arrays, we can use the sizeof() function to get the array size in bytes. If we get the element size in bytes as well, we easily can calculate the number of objects in our arrays.

```
21: int main()
 22: {
 23:   Point Pixel;
 24:   ofstream myfile;  // we will write to this file
 25:
 26:   // we have to count the objects in our scene (spheres and lightsources)
 27:   int array_size,element_size;
 28:   array_size = sizeof(spheres);
 29:   element_size = sizeof(spheres[0]);
 30:
 31:   obj_Amnt = array_size/element_size; // we count by the array size in bytes and by
the element size in bytes
 32:   // counting lightsources
 33:   array_size = sizeof(lights);
 34:   element_size = sizeof(lights[0]);
 35:
 36:   light_Amnt = array_size/element_size; // we do the same as by spheres
```

For our further use, we have to normalize the lightsource direction vectors. We will need unit vectors for our light calculations later. Since we need only the direction, which doesn't change during normalization, we are free to do so. After some initialization of our PPM output file, we declare some variables necessary for the forthcoming main loop. The variables coordX and coordY are used to store the ratio revised X, Y cooridnates we look at from our camera position. This X, Y coordinates and a constant Z value (3) will be put into the temp_coord variable.

```
38:   // we have to normalize the light vectors. We will need this to be able to get
positions as multiples of unit vectors - more detail at povray example
```

```
 39:   int i=0;
 40:
 41:   Point normalized;
 42:   while(i<light_Amnt) {
 43:     normalized.set(lights[i][0],lights[i][1],lights[i][2]);
 44: // we put data temporary to "normalized"
 45:
 46:     normalized=normalized.norm(); //  We use the norm() method to get normalized
values
 47:     // we put back the normalized "i" values to the lights[]array
 48:     lights[i][0] = normalized.get(0);
 49:     lights[i][1] = normalized.get(1);
 50:     lights[i][2] = normalized.get(2);
 51:
 52:     i++;
 53:   }
 54:
 55:   // Constructing the image
 56:
 57:   int c,c2;
 58:
 59:   myfile.open("probakep.ppm");
 60:   myfile << "P3\r\n";
 61:   myfile << X;
 62:   myfile << " ";
 63:   myfile << Y;
 64:   myfile << "\r\n";
 65:   myfile << "255\r\n";
 66:
 67:   double coordX,coordY;
 68:   //double Pf;
 69:   Point temp_coord ;
 70:
 71:   c=0;
 72:   c2=0;
```

The main loop in the main() function is a nested loop with variables i and j . These variables run from 0 to X and from 0 to Y . X and Y represents the resolution of our viewing plane, thus the resolution of the output image of the 3D scene. Actually, the inner loop will run between FROM and TO later - this will be discussed further at the parallel part. After we do some aspect ratio correction, we have the inputs of our Trace() function. We need the camera position (camera), the coordinates we look at (through) from the camera position (temp_coord) and a value indicating our recursion level of tracing. This starts from 1 and goes until max_recLev. After tracing we get a colour for the actual pixel of our plane. We convert its R,G,B coordinates to the 0-255 scale and write it to our PPM file. After we did all the raytracing task for all of our rays (going from our viewpoint through all X, Y coordinates), we are done, we only have to close our file and exit.

```
73:   for(int i=0;i<Y;i++) // we iterate through the pixels
 74:   {
 75:    coordY = (double)i / ((double)Y-1)*2 - 1;  // we do some screen ratio correction
 76:
 77:    for(int j=0;j<X;j++)
 78:    {
 79:      coordX = ((double)j / ((double)X-1)*0.5)*4*(double)X/(double)Y; // screen
ratio correction
 80:      temp_coord.set(coordX,coordY,3);
 81:
 82:  Pixel = Trace( camera, temp_coord, 1);   // here we get the pixel color of a pixel
at the actual coordinate. As input, we need the viewing position (camera),
 83:     // the pixel coordinate we look through (temp_coord) and a recursion level, by
which we define how many reflections should we calculate with. The maximum is defined
 84:   // in max-recLev and we increase the initial value (now 1) at every recursion
 85:
 86:  myfile << min( (int)round(Pixel.get(0)*255), 255);  // we convert our analogue
color value to the 0-255 scale, so we can give a true pixel color
 87:  myfile << " ";
 88:  myfile << min( (int)round(Pixel.get(1)*255), 255);
 89:  myfile << " ";
 90:  myfile << min( (int)round(Pixel.get(2)*255), 255);
 91:  myfile << " ";
```

```
 92:  c2=c2+1;
 93:    }
 94:    c2=0;
 95:    myfile << "\r\n";
 96:    c=c+1;
 97:   }
 98:
 99:   myfile.close();
100:
101:   return 0;
102: }
103:
```

We can see that the difficult task is dealt with in the Trace() function. Let's investigate it in detail.

The Trace() function calculates the colour of a given point in our viewing plane. We can imagine our viewing plane as a window through which we look at the 3D scene. What we really see is the group of various colours at various coordinates of this window. According to the reverse technique of ray tracing, we will cast a ray from our viewing point (camera) toward every pixel of our viewing plane. If the ray hits something, we will see some colour, if not, we will see only the background at the particular coordinates of the image. The inputs of the function are the camera position P_vv, the direction D_vv and a value for preventing infinite recursive calls, recLev. In the while function we iterate through every sphere and search for the nearest intersection point. For every iteration the index closest will contain the index of the closest sphere. To calculate the distance from our starting point(camera), we utilize the calcRaySphereIntersection() function. We will discuss this function later. If the actual T is less than minT thus, we've found a closer intersection, we assign T to minT.

```
142: // Trace function, it needs a starting point, a direction and a recursion level. It
will give a Pixel color
143: Point Trace(Point &P_vv, Point &D_vv, int recLev)  // we need a local recLev, since
we call it recursively
144: {
145:   Point Pixel;
146:
147:   // input P, D, reclev - P starting point, D ray direction, reclev - recursion
level
148:   double minT = max_Dist; // this is the maximum distance (defines our worlds size)
149:   int closest = obj_Amnt; // this will be the index of our closest object (sphere),
we start from the number of objects
150:
151:   // we get the closest intersection
152:   int ind = 0;
153:   while(ind<obj_Amnt)
154:     {
155:       // we get a distance
156:       T = calcRaySphereIntersection(P_vv,D_vv,ind);
157:       if ((T>0) && (T<minT)) // if closer than previous, this will be the new
closest
158:  {
159:    minT = T;
160:    closest = ind;
161:    // cout << "BINGO - ";
162:  }
163:       ind++;
164:       // cout << closest;
165:       // cout << " ";
166:     }
167:   ind=0;
168:
```

If we find an intersection point (ie. closest has a valid value, obj_Amnt is invalid), we have to put it in variable IP. At this point, we calculate a normal vector for the sphere from the intersection point, from the center of the sphere and from the radius of the sphere. The vector, pointing from the sphere center to the intersection point, divided by the radius is equal to the normal vector. We use this normal vector for several purposes. First we can calculate a reflected ray by mirroring the incoming ray to the normal vector. Later on we will need the normal vector also for calculating diffuse lighting.

```
169:   if (closest == obj_Amnt)  // no intersection, color will be background color
```

```
170:     {
171:       Pixel.set(bg_Color[0],bg_Color[1],bg_Color[2]);
172:       // global variable with 3 int value for - RGB
173:     }
174:   else
175:     {
176:    // if there is an iintersection, we have to start a ray from that intersection,
and again, and again, until the maximum recursion level reached
177:       // the pixel color will be calculated from reflected light colors (and object
colors that modifies the light colors)
178:       Point IP = P_vv + (D_vv * minT);  // intersection point
179:
180:       // we need the radius of the sphere our ray intersects with - for normal
vector calculation
181:       double R_sp = spheres[closest][3];
182:
183:       Point normal;
184:       Point SP_cent(spheres[closest][0],
185:       spheres[closest][1],
186:       spheres[closest][2]);
187:
188:       // intersec point - sp_center -> normal vect..
189:       SP_cent = IP - SP_cent;
190:
191:       normal = SP_cent / R_sp;
192:
193:       Point V = P_vv - IP;
194:       Point Refl = (normal * (2 * point_dot(normal, V))) - V;  // we can get the
reflected ray by mirroring the incoming ray to the normal vector at the intersection
point
195:       // if reflections are switched on, we will use this reflected ray later
196:
197:
```

At this point, we start dealing with the construction of the compound color of the actual pixel of the viewing plane. First we have an ambient color value to which we do add further components. The ambient color is like a mask of the 3D scene - if we hit something with our first ray it has an ambient lighting value, if not, then it is coloured as the background (black). Every further colour of different effects will be added to this basic ambient value. To take every further effect into account, we have to check whether a light-source is illuminating the intersection point or not. Accordingly, we have to iterate through every light-source (while(ind<light_Amnt)) and check this. For every light-source, we put the direction of the light-source into Light, the colour of the light-source into Light_col and the colour of the sphere that the ray intersects into sphere_col .

```
198:       // Lighting calculation - getting the superposition of the affecting
lightsources lights at our intersection point, thus getting the color of this point
199:       Pixel = ambient;  // as default, we start with a background color
200:       int ind = 0;
201:       int ind2 = 0;
202:       Point Light;
203:       Point Light col;
204:       Point sphere_col;
205:       bool Shadowed=false;  // we will check whether an intersection point is in
shadow , thus whether an object is between this point and the given lightsource
206:       double Factor; // for Lighting calc
207:       Point Temp;
208:       double Temp_d;
209:
210:       while(ind<light_Amnt )   // we check all lightsources of our scene
211:  {
212:    Light.set(lights[ind][0],lights[ind][1],lights[ind][2]);
213:    // direction vector - first 3 value from 6 (second triplet is the color)
214:    Light_col.set(lights[ind][3],lights[ind][4],lights[ind][5]);
215:    // color of light
216:    sphere_col.set(spheres[closest][5],
217:     spheres[closest][6],
218:     spheres[closest][7]);
219:    // sphere color
220:
221:
```

For every light-source, we have to check whether the intersection point is shadowed by an object (between the light-source and the intersection point) or not. If it is shadowed, we do not take that light-source's effect into account. If not, then we calculate a diffuse lighting component (this can be switched off by the diffuse_test constant). This component is calculated from the light-source's colour, the sphere's color and a variable Factor that is the cosine of the normal vector and the vector light-source's incoming ray. This factor affects the intensity of the color.

```
222:    // Shadow test  - is the lightsource shadowed by any object?
223:    Shadowed = false;
224:    ind2 = 0;
225:    while(ind2<obj_Amnt)
226:      {
227:        if( (ind2!=closest) && (calcRaySphereIntersection(IP,Light,ind2) > 0) )
228:          {
229:     Shadowed=true; //
230:     ind2=obj_Amnt;
231:          }
232:        ind2++;
233:      }
234:
235:    if(!Shadowed) // false
236:      {
237:        if(Diffuse_test)  //  if switched on, diffuse lighting will be calculated
238:        {
239:      // if the cosine of two vectors is negative, we don't see it, if it's
positive, we see it
240:      Factor = point_dot(normal, Light);
241:      // cosine angle
242:      // cout << Factor << " --";
243:      if(Factor>0)
244:      {
245:       Pixel = Pixel
246:           +((Light_col * sphere_col)* Factor);
247:      }
248:        }
```

If the variable Specular_test is true, we add a similar component, what mimes the shining, flare like mirroring of the light-source on the surface. This component is the specular reflection component. The ambient, diffuse and specular components together are also referred as the Phong reflection model.

```
249:        if(Specular_test) // if specular (mirror like) reflection is switched on, we
have to calculate it  (phong effect)
250:          {
251:      Factor = point_dot(Refl.norm(), Light);
252:      if(Factor>0)
253:      {
254:       Temp_d = pow(Factor,
255:        spheres[closest][8])
256:       *spheres[closest][9];  //  pow(factor, phong size) * phong_amount
257:       Temp.set(lights[ind][3],
258:           lights[ind][4],
259:           lights[ind][5]);
260:
261:       Pixel = Pixel + (Temp*Temp_d);
262:      }
263:          }
264:      }
265:    ind++;
266:  }
267:       // Pixel
268:
```

After calculating the effects of all the light-sources, we have to calculate the reflections on the surfaces of the spheres. This part can be switched off as well with the Reflection_test constant. If it is switched on, every pixel will get its value calculating the reflection on other spheres. Since we have to calculate the reflections of the reflections, and so on, this method is recursive and uses the Trace() function to calculate the colors from reflected objects and light-sources. This procedure could run forever, so we need a maximum recursion level (

max_RecLev) which we cannot exceed. Every recursive call to the Trace() function increases its recLev parameter, so we will reach max_RecLev at once. If we didn't reach this limit yet, and the sphere had a reflection component (5th parameter of a sphere), we should calculate a color and weight it with the reflection component.

```
269:       if ( Reflection_test )  // if reflections are switched on, we have to
calculate the reflected lights for a given intersection point
270:   {
271:     if ( recLev < max_Reclev && spheres[closest][4] > 0)  // if we didn't exceed the
maximum reflection count (global variable) and the given object has a reflection value
272:       {
273:         Temp_d = spheres[closest][4];
274:         // cout << "Temp_d: " << Temp_d << " ---";
275:         Temp = Trace( IP, Refl, recLev+1 );  // recursive call for handling all
reflections affecting the actual intersection point
276:         Pixel = Pixel + (Temp*Temp_d);  // we add every further reflection data to
Pixel
277:       }
278:
279:   }
280:
281:      }
282:    // cout << "Pixel: " << (int)(Pixel[0]*255) << "-" << (int)(Pixel[1]*255) << "-"
<< (int)(Pixel[2]*255) << "||  ";
283:    return Pixel;
284: }
```

We have to go back to our last function which we didn't observe closely. The calcRaySphereIntersection() function calculates whether a ray intersects with a sphere or not. The function returns -1 if there is no intersection and a factor T with some positive or zero value if there is. This T value is a distance and if we multiply the direction vector of our ray with this factor, we get the vector pointing to the intersection point. The inputs of the function are:

- P_ray - starting point of the ray
- D_ray - direction of the ray
- sp_index - Sphere index in the sphere array

With the help of the sphere index, we can assign the sphere center coordinates to C_sp and the spheres's radius to R_sp. The main idea is that the following equation holds:

$$|P\_ray + TD\_ray - C\_sp| = R\_sp$$

Thus, if an intersection occurs, the intersecting ray turns into a vector, pointing to the intersection point (T*D_ray). This vector can be constructed by summing the vector pointing to the spheres center (C_sp) and the radius vector (actually, we need only the length of this vector: R_sp).

We can substitute P_ray-C_sp with V. This will appear as follows:

$$|TD\_ray + V| = R\_sp$$

If we open the formula, we get:

$$(TD_x + V_x)^2 + (TD_y + V_y)^2 + (TD_z + V_z)^2 - R\_sp^2 = 0$$

From this formula we get T by solving the following 2nd order polynomial:

$$T = (-DV + -\sqrt{(DV)^2 - D^2(V^2 - R^2)})/D^2$$

If the determinant is non-zero, we have one or two solutions. We need the smaller one, thus the closer one. The function returns with this T factor.

```
104: // this function will  check, whether a given ray intersects with a sphere. If yes,
```

```
it will give a "T" distance from the starting point of the ray and we can calculate the
closest intersection coordinate in 3D (and we can use that as a new initial coordinate )
105: double calcRaySphereIntersection(Point& P_ray, Point& D_ray, int sp_index) // The
start point and ray direction are given by reference, the index is dynamic and we will
get a distance returned
106: {
107:  Point C_sp (spheres[sp_index][0],  // We get the center of the sphere -
spheres[sp index][0,1,2]
108:       spheres[sp_index][1],
109:       spheres[sp_index][2]);
110:
111:  // to get the intersection point, we will use a distance "T" (we know the
direction of the ray)
112:  // P - starting point, D - direction, C - center, R - radius, vlength(P+T*D-C)=R,
V=P-C, vlength(v+T*D)=R
113:  // (T*Dx+Vx)2 + (T*Dy+Vy)2 + (T*Dz+Vz)2 - R2 = 0
114:  // T = (-D·V +/- sqrt((D·V)2 - D2(V2-R2))) / D2     // solution of the quadratic
equation (we get two intersection points, we need the closer one)
115:  // http://www.povray.org/documentation/view/3.6.1/130/
116:  // P_ray - C_sp
117:
118:  Point V = P_ray - C_sp;
119:  double R_sp = spheres[sp_index][3]; // the 3 index shows the radius
120:  // D*V
121:  double DV = point_dot(D_ray, V);
122:  // D*D
123:  double D2 = point_dot(D_ray,D_ray);
124:
125:  double SQ = DV*DV - D2*(point_dot(V,V)-R_sp*R_sp);
126:
127:  if (SQ <= 0)  // if negative, no square root exists - no intersection
128:  {
129:    T = -1;
130:  }
131:  else
132:  {
133:    SQ = sqrt(SQ);
134:    double T1 = (-DV+SQ)/D2;
135:    double T2 = (-DV-SQ)/D2;
136:    T = min(T1, T2);  // the first intersection is what we search for (closer one)
137:  }
138:  // if (T>0) {cout << "T: " << T << " ! ";}
139:  return T; // this is the distance from the starting point P in direction D, where
the intersection occur
140: }
```

The raytracer part of the code is discussed. We see that the method, however simplified, is resource intense. We have to make several recursive calculations for several positions of our viewing plane. If we have a complex scene with mirrors and light-sources, with a large resolution, the rendering of only one frame could last for seconds or minutes. How could we improve performance by adding parallel workers to the task?

We can see that the pixels of the viewing plane are independent. Therefore, theoretically, every single pixel could be calculated by independent workers parallelly. We will see that communication overhead prevents us from making such a move, although for a sophisticated calculation, this could be a realistic way as well. Some parallelization is very helpful at this level of complexity, too.

13.2.2. 13.2.2 Parallel raytracer

The above raytracing example needs computational power to run. The serial code runs between 10 and 20 seconds on a dual-core system (Intel Xeon 2,5GHZ). This generates a 4800x4800 resolution frame with 5 spheres, all mirrorings switched on and with three light-sources. To be able to generate more frames per second, we will need to speed up the process. There could be some programming techniques to speed up the run, but these won't help significantly. We need more power and we could achieve it by splitting up the work to several processors. A trial run of the same code with parallelization with 256 cores results the calculation jumping down to 0.3 seconds. Unfortunately the data collecting and I/O part still remain around 6 seconds, since the output file is around 150 MBs. We can easily get over this problem because the work can be split up to independent procedures, thus we don't need to communicate between cores while the calculations run. Every pixel of our

viewing plane is independent of the other one and dependent only on the 3D scene. The 3D scene is defined by geometrical data, is static and every worker has it at the beginning. Our only task is to split the 2D plane between cores. We use the same simple method as in our heated plate example - we divide the lines of our plane by the number of cores and the final core gets some residual lines to deal with.

Since we have to split the points of our plane, we simply have to run the Trace() function from a given point to another given point. In our main() function, the inner loop is going from FROM to TO. Every working core has its own rank and every worker will have a unique interval defined by its own FROM,TO values.

The include file (ray6v7.h) is the same as at the serial implementation with the exception of one definition.

$$double\ t\_frame[X][Y][3] = 0;$$

This temporary array will serve as a container for every worker to put rendered data in. Since nodes are working in a 3D world, where object and lightsource definitions are present and render a part of a 2D image, we need this temporary array. The X,Y array contains triplets of the RGB pixel values.

The parallel program starts with similar lines to the serial one. The first difference is the MPI include line at the beginning (mpi.h). We also have to declare some MPI related variables as before (world_rank, world_size, FROM, TO).

```
1: // Raytrace demo - parallel solution
  2: // this code is based on the raytrace documentation at povray.org and on the php
script of Darren Abbey ((C) 2002)
  3:
  4: #include <mpi.h>
  5: #include <iostream>
  6: #include <fstream>
  7: #include <math.h>
  8: #include <vector>
  9:
 10: #include "ray6v7.h"
 11:
 12: using namespace std;
 13:
 14: // predefinition of the Trace() function. It will
 15: Point Trace(Point &P_vv, Point &D_vv, int recLev);
 16:
 17: double T = -1;  // ray - sphere intersection result - this will be a distance, if -
1, then no intersection occurs..
 18: int obj_Amnt;
 19: int light_Amnt;
 20:
 21: // ambient and  camera values
 22: double ddd = 0.2;
 23: Point ambient(ddd,0.2,0.2); // diffuse lighting in RGB - not 8bit, yet (some grey -
0.2,0.2,0.2)
 24: Point camera(-4,0,-16);  // camera position - 0,0,-3
 25:
 26: // MPI variables
 27:
 28: int world_rank;  // the rank of this process
 29: int world_size; // size of world, thus number of processors in this world
 30: int FROM;
 31: int TO;
 32:
 33:
 34: int main()
```

The Main() function includes some extra lines at the beginning compared to the serial version. After initializing the MPI world, we read some arbitrary time into s_time with the MPI_Wtime() function. We also list the working nodes into world_size and the rank of the actual node into world_rank. After these lines, the code is very similar to the serial one, up to line 98.

```
34: int main()
 35: {
```

```
 36:   Point Pixel;
 37:   ofstream myfile;  // we will write to this file
 38:   double s_time;
 39:   // Initialize the MPI environment - we can use the MPI functions and world until
MPI_Finalize
 40:     MPI_Init(NULL, NULL);
 41:
 42:  s time=MPI Wtime();
 43:   // number of processes
 44:     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
 45:     // Get the rank of the process
 46:     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 47:
 48:
 49:  // we have to count the objects in our scene (spheres and lightsources)
 50:   int array_size,element_size;
 51:   array_size = sizeof(spheres);
 52:   element_size = sizeof(spheres[0]);
 53:
 54:   obj_Amnt = array_size/element_size; // we count by the array size in bytes and by
the element size in bytes
 55:  // counting lightsources
 56:   array_size = sizeof(lights);
 57:   element_size = sizeof(lights[0]);
 58:
 59:   light_Amnt = array_size/element_size; // we do the same as by spheres
 60:   // we have to normalize the light vectors. We will need this to be able to get
positions as multiples of unit vectors - more detail at povray example
 61:   int i=0;
 62:
 63:   Point normalized;
 64:   while(i<light_Amnt) {
 65:     normalized.set(lights[i][0],lights[i][1],lights[i][2]);
 66: // we put data temporary to "normalized"
 67:
 68:     normalized=normalized.norm(); //  We use the norm() method to get normalized
values
 69:     // we put back the normalized "i" values to the lights[]array
 70:     lights[i][0] = normalized.get(0);
 71:     lights[i][1] = normalized.get(1);
 72:     lights[i][2] = normalized.get(2);
 73:
 74:     i++;
 75:   }
 76:
 77:   // MPI START
```

At this point, every worker calculates his portion of work. Every node works with the lines between FROM and TO. As seen before, we leave the extra work for the last worker.

```
77:   // MPI START
 78:   // we have to calculate how many lines does our process to work with, thus we
have to share the work among workers
 79:  // partitioning the dataset is one key moment of parallel computing, it has a
great effect on running time
 80:
 81:  // we choose a simple, albeit not the most effective partitioning method: we share
data among workers equally and the last one has to do some rest as well.
 82:
 83:  int p_row = Y / world_size;  // we divide the number of lines with the number of
workers
 84:  int p_row_last = 0; // we prepare a special variable for the last process
 85:  if((Y % world_size)!= 0) // if the number of lines can't be divided without
residual
 86:  {
 87:      //
 88:   p_row = floor(p_row);  // we take the whole part of our division, that will be
the number of lines for all workers except the last..
 89:   p_row_last = Y - ((world_size-1)*p_row); // .. and the last gets the same number
of lines plus the rest
 90:  }
```

```
 91:  else // we have luck, we can divide the task equally, the last gets the same work
to do
 92:  {
 93:     p_row = floor(p_row);
 94:     p_row_last = p_row;
 95:  }
 96:
 97:  // we have to calculate the first (FROM) and last (TO-1) rows we have to deal with
 98:  if(world_rank < world_size-1) // we are not the last worker
 99:  {
100:   FROM = world_rank*p_row;
101:   TO = FROM+p_row;
102:  }
103:  else  // we are the last worker
104:  {
105:   FROM = world_rank*p_row;
106:   TO = FROM + p_row_last;
107:  }
108:  // MPI END
```

After printing out some information, we go on preparing and starting the double loop with which we step from point to point for rendering the 2D image of the 3D scene. Since this code runs parallelly on every worker, every worker will calculate their own set of lines.

```
108:  // MPI END
109:
110:  // ---------------------------------------------------------------------
111:  // ------------ Feedback, switch off to speed up --------------
112:  if(world_rank < world_size-1)  // Some feedback, who gets what amount of work
113:  {
114:    cout << "I'm worker " << world_rank << ". I will calculate: " << p_row << "
rows. \r\n";
115:  }
116:  else
117:  {
118:    cout << "I'm the last, worker " << world_rank << ". I will work on all residual
rows: " << p_row_last << ". \r\n";
119:  }
120:  // -------------- Feedback ends ---------------------------------
121:  // ---------------------------------------------------------------------
122:
123:
124:
125:   //   // Constructing the image
126:
127:   int c,c2;
128:
129:   double coordX,coordY;
130:   double Pf;
131:   Point temp_coord ;
132:
133:   c=0;
134:   c2=0;
135:   for(int i=0;i<Y;i++) // we iterate through the pixels
136:   {
137:    coordY = (double)i / ((double)Y-1)*2 - 1;  // we do some screen ratio correction
138:
139:
140:    for(int j=FROM;j<TO;j++)
141:    {
142:      coordX = ((double)j / ((double)X-1)*0.5)*4*(double)X/(double)Y; // screen
ratio correction
143:      temp coord.set(coordX,coordY,3);
144:
145:  Pixel = Trace( camera, temp_coord, 1);   // here we get the pixel color of a pixel
at the actual coordinate. As input, we need the viewing position (camera),
146:     // the pixel coordinate we look through (temp_coord) and a recursion level, by
which we define how many reflections should we calculate with. The maximum is defined
147:   // in max-recLev and we increase the initial value (now 1) at every recursion
148:
```

```
149:
150:  t_frame[j][i][0] = Pixel.get(0); // we convert our analogue color value to the 0-
255 scale, so we can give a true pixel color
151:  t_frame[j][i][1] = Pixel.get(1);
152:  t_frame[j][i][2] = Pixel.get(2);
153:
154:  c2=c2+1;
155:    }
156:    c2=0;
157:    c=c+1;
158:   }
```

After every worker has calculated their multiple points of the scene, we have to collect these data fragments. Every node renders to a temporary container t_frame to the accordant lines of this container. As the first node surely exists, it is a good idea to assign the task of collecting results to that one.

After the elapsed time (MPI_Wtime()-s_time) fed back to the console ("work done!"), the first node has to receive data fragments from every other node (p<world_size). If we are a sender node, we use the MPI_Send() function. At this point, we use a trick which will work in this C++ environment now. C++ array is contiguous on this architecture by definition. Actually, it would be worth to make a dynamic array or structure contiguous in an MPI environment to minimalize communication overhead. It is worth in most of the cases even with some extra code at the node. The trick is that we don't send n times the lines, but we send the whole block once: n*line (line 181). If we are the first node, we do the same with MPI_Recv() Before this, we have to do some calculations to put exactly the right lines to the right places received from different nodes. At this point, we could make a mistake putting the received lines to a wrong position. The code would still work but the final result would be wrong. When the data collection is done, we can write out the rendered image to a PPM file. For benchmarking purposes, we give a final running time value, so we can compare the real parallel calculation time interval with the I/O overhead.

```
159:   // MPI calculation is ready, we have to collect data from other workers
160:   cout << world_rank << " work done! MPI_time: " << MPI_Wtime()-s_time << "\r\n";
161:    // work is done at his point..
162:  // ..but we have the different slices at different processes. We have to collect
them, to have it at one process, practically at process 0
163:
164:  // these variables will serve process 0 to hold the remote process'  FROM TO
values (rFROM rTO)
165:  int rFROM;
166:  int rTO;
167:
168:  if(world_size>1)  // we collect only, if there are more than one processes
169:  {
170:   for(int p=1;p<world_size;p++) // we have to collect world_size-1 slices
171:   {
172:    if(world_rank == p) // if it's my turn, as sender, I send.
173:    {
174:        cout << "I'm sending (me: " << world_rank << ") data from " << FROM << " to
" << TO << ". Y: " << Y << "\r\n";
175:     /*
176:     for(int r=FROM;r<TO;r++)  // I'm sending my lines -- possible simplifications
177:     {
178:      MPI_Send(&t_frame[r][0][0], Y*3, MPI_DOUBLE, 0 , 999 , MPI_COMM_WORLD); // 999
- unique tag
179:     }*/
180:     // Since c++ arrays like t_frame are contiguous in programs memory, we can use
a little trick here:
181:     MPI_Send(&t_frame[FROM][0], Y*3*(TO-FROM), MPI_DOUBLE, 0 , 999 ,
MPI_COMM_WORLD); // we assume, rows are consecutive in memory.
182:    }
183:    else if ( world_rank == 0) // if we are process 0, we collect the sended data
184:    {
185:     // what shall rFROM and rTO be? What lines do we have to collect. We have to
calculate it from the remote process' rank (p)
186:     if(p<world_size-1) // not the last slice is comming. The last one could be
longer than normal (normal + residual!)
187:     {
188:      rFROM = p_row*p;
189:      rTO = p_row*(p+1);
```

```
190:     }
191:     else // last one is coming
192:     {
193:      rFROM = p_row*p;
194:      rTO = X;
195:     }
196:     cout << "I receive (me: " << world_rank << ") the lines from: " << p << ".
Interval: " << rFROM << " - " << rTO << " \r\n";
197:     /*
198:     for(int r=rFROM;r<rTO;r++) // we send all the lines - possible simplification!
199:     {
200:      MPI_Recv(&t_frame[r][0][0], Y*3, MPI_DOUBLE, p , 999 , MPI_COMM_WORLD,
MPI_STATUS_IGNORE); // 999 - unique tag
201:     }*/
202:     MPI_Recv(&t_frame[rFROM][0], Y*3*(rTO-rFROM), MPI_DOUBLE, p , 999 ,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);  // we assume the lines are consecutive in memory.
203:    }
204:   }
205:  }
206:
207:  cout << world rank << " collecting done! - MPI time: " << MPI Wtime()-s time <<
"\r\n";
208:
209:   // MPI_Barrier(MPI_COMM_WORLD);
210:   // MPI
211:   // we are done, we have to collect data
212:    // data is at process 0
213:  // we write it to a file
214:  if(world_rank==0) // if we are process 0, we have to write out data
215:  {
216:
217:     string filename1;
218:     filename1 = "raytrace_par.ppm";
219:     char *fileName = (char*)filename1.c_str();
220:
221:    cout << " I will write the file: " << fileName << ". \r\n";
222:
223:   myfile.open(fileName);
224:   myfile << "P3\r\n";
225:   myfile << X;
226:   myfile << " ";
227:   myfile << Y;
228:   myfile << "\r\n";
229:   myfile << "255\r\n";
230:
231:   for(int i=0;i<Y;i++)
232:   {
233:    for(int j=0;j<X;j++)
234:    {
235:
236:   myfile << min( (int)round(t_frame[j][i][0]*255), 255);  // we convert real values
to the 0-255 RGB plane to generate the PPM file
237:   myfile << " ";
238:      myfile << min( (int)round(t_frame[j][i][1]*255), 255);
239:   myfile << " ";
240:      myfile << min( (int)round(t_frame[j][i][2]*255), 255);
241:      myfile << " ";
242:    }
243:    myfile << "\r\n";
244:   }
245:
246:   myfile.close();
247:   cout << world_rank << " writing done! - MPI_time: " << MPI_Wtime()-s_time <<
"\r\n";
248: }
249:
250:   /*
251:   // File
252:   myfile.open("probakep.ppm");
253:   myfile << "P3\r\n";
254:   myfile << X;
255:   myfile << " ";
```

The Trace() and calcRaySphereIntersection() functions are the same as in the serial code, they are not affected by the parallelization.

14. 14 Project works

In this chapter we will provide some exercises in connection with topics of our work. These exercises are of bigger size, and can be assigned to students for homework project. Some of them can be accomplished in a few weeks time, but some of them are bigger tasks, and can be processed as a BSc thesis work, as a one term task.

14.1. 14.1 Sorting

Sorting is a well known example of basic complex algorithms. The parallel sorting algorithms also help a lot to understand the problem of parallelization.[Akl1985] We would like to propose some project works, that can be accomplished in MPI environments. These tasks require a little more work, for students we propose these as homework for a few weeks term.

The task of sorting is to sort a given array of numbers in increasing (or decreasing) order. Although sorting of other kind (radix sort is a good example)also exists, here we speak about sorting based on comparison. This means that we are allowed to compare two elements of an array and interchange them or sort one of them out to another array or alike. The complexity bound of such sorting is $O(n \log n)$. We propose a simple C++ program which uses the STL sort() algorithm to be the base comparison. The parameters of this function are two pointers, in our case the beginning of the array to be sorted and the next-to-last element pointer of this array. So the null version of the program would be:

```
// basic program for sorting
// with the sort() algorithm

#include<iostream>
#include<algorithm>
using namespace std;
const long long SIZE=1ll<<25;
// 2^25 with use of shifting operators
// 1ll is "1" for long long

double *a = new double[SIZE];

int main(int argc, char ** argv){
  time_t t1, t2;
  srand(time(NULL));

  for(int i=0;i<SIZE;i++)a[i]=(double)rand()/rand();

  for(int i=0;i<40;i++)cout<<a[i]<<", ";

  t1=clock();
  sort(a,a+SIZE);
  t2=clock();

  cout<<endl<<"****************************"<<endl<<endl;

  cout<<"time elapsed: "<<difftime(t2,t1)/CLOCKS_PER_SEC<<endl;


  cout<<endl<<"first 40:"<<endl;
  for(int i=0;i<40;i++)cout<<a[i]<<", ";
  cout<<endl<<"middle 40:"<<endl;
  for(int i=SIZE/2;i<SIZE/2+40;i++)cout<<a[i]<<", ";
  cout<<endl<<"last 40:"<<endl;
  for(int i=SIZE-40;i<SIZE;i++)cout<<a[i]<<", ";
  cout<<endl;
}
```

We would like to make some notes. First, the use of doubles: it makes the comparison and data movement more problematic, which is important to construct a harder problem. Second, the use of (double)rand()/rand() as the generator of random numbers. It will lead to uneven distribution as many values will be around 0, with less and less frequency towards and RAND_MAX. It is well known, that the problem is easier if we know the distribution of the numbers, and an even distribution like (double)rand()/RAND_MAX would lead to such direction.

14.1.1. 14.1.1 Merge sort

Our first proposal is based on merge sort. The original sequential algorithm [Corm2009] is a divide-and-conquer algorithm. We can use it as the base for our parallel version. The master process (where $id = 0$) divides the array and sends one half to process 1. Then process 0 divides its remaining half array and sends the quarter of the array to process 2, while the process 1 also divides its half array and sends a quarter to process 3. And so on each process divides the sub-array and sends to another process.

When no more processes are left to receive a part of the array, the processes sort the sub-array they are left with. We propose to do the sorting just with the STL algorithm sort() as it is a good sequential sorting algorithm.

Next, when local sorting is done, each process routes back the sorted sub-array to the exact processor they received it from. That processor will merge the sub-arrays into a sorted sub-array, and next will route it to the previous processor. This will virtually draw a tree like communication structure, where the master process of rank 0 will collect the whole array at the end .

14.1.1.1. 14.1.1.1 External sorting

A variant of the previous project would be an external sorting, where the numbers to be sorted reside on some storage - hard drive for example -, and are of such magnitude, that they cannot be read at once into one computer memory.

The sorting of this kind could be implemented the following way. Different processes read in a part of the array to be sorted independently and sort it locally. Obviously this part should be able to reside in memory, so either you should use enough computers to split up the array as such, or the processes must do the reading and sorting multiple times on chunks of sub-memory size. The sorted sub-sequences are written back to the storage system.

When the processes are ready, the master process reads in chunks of sorted arrays and merges them while writing out the merged and sorted output to storage again.

14.1.2. 14.1.2 Quick sort

Another example of sorting is the quick sort algorithm.[Corm2009] Again, this is a divide-and-conquer algorithm, so the parallelization could be done the same way as we described previously in merge sort. The difference is that the division of the elements is made prior to sending them to be sorted, so no merging is needed in the end. This means that the sorted sub-arrays can be sent back right to the master process.

14.1.3. 14.1.3 Sample sort

The parallelization of quick sort we proposed before is not the best way for message passing clusters. It consists of too much data movement. So there is a variant of quick sort for clustered architectures and alike, which is called sample sort.

The steps of the algorithm are the following:

1. The master process makes a sampling of the array. It takes out random elements of the array to be sorted in a magnitude just over the number of processes. For 128 processes it can use a hundred times more, so 12800 samples. It sorts the samples, and then takes out every 100-th (in our case). These elements will represent boundaries. The array of boundaries is distributed by broadcast. (With good sampled boundaries we will sort out the original array to nearly equal sized sub-arrays.)

2. The whole array is distributed by broadcast, and every process takes out those elements that fall between the boundaries assigned to that process. If the boundaries are $a_0, a_1, a_2, \ldots, a_{p-1}, a_p$ for p processes, than the k-th process will have the boundaries $a_k \leq a_k < a_{k+1}$, and sort out those elements, where $0 \leq k < p$.

3. The processes sort the elements within their boundaries locally and independently.

4. The sorted sub-sequences are routed back to the root process, which places them after each other and so get the final result.

14.1.3.1. 14.1.3.1 Notes

- The sub-sequences will be of different size, so the slave processes must first send the size of their array, and only next the array itself.
- Because of many send-receive calls be aware of using the right ordering to avoid deadlock, and using different tags for different messages in order to not confuse them.
- Actually the sizes of the sub-arrays are known right after sorting out elements by boundaries, and this information can be sent back to the root right away. With this information from all processes the root can calculate the exact position of each sub-array that it will receive in the end. So receiving the sorted sub-arrays can be done in any order, as long as the processes end with sorting.
- The root can receive the sub-arrays right in place of the original array. In this case the pointer of the receive buffer, instead of the name of the array, say a will be the k-th element of it, a pointer of form a+k. With this we can assign the receive buffer to be in the middle of our array.

14.2. 14.2 K-means clustering

A present problem concerning tasks of data management is the clustering problem. In this problem we have a huge number of data represented by points on a high-dimensional plane, and we would like to cluster them, so grouping them in clusters where points in one cluster are closer to each other than to the points of other clusters. One possible algorithm for this problem is the k-means algorithm, which groups the points into k clusters. The main steps of the algorithm are the following:

1. We choose k arbitrary points in the plain, these will be the centres of our clusters.

2. We calculate the distance of each point to the centers and assign each point to a cluster represented by a center to which the point is closest.

3. After the construction of the clusters, for each cluster we calculate the means of the points fall into this cluster, that is the middle point of the cluster. (We can do it easily by calculating the mean of each dimension parameter for all points.)

4. These mean points will give us the new cluster centres and we shall begin from step 2. again, till when no point moves between the clusters from one loop to the other, so the clustering is relaxed.

The parallelization of such problem can be done like Dijkstra's shortest paths problem parallelization. We assign a part of the points to different processes. The calculation of the distance to the centres can be done independently. Obviously each process then will have only sub-clusters. The means or the center of the clusters can be calculated in the following way. Each process calculates the means of its sub-clusters locally, and then these local centres are grouped together by a master process, the global means should be calculated from them so the new centres are routed back. Do not forget to send, not only the local means, but the number of points it represents, as considering global means we must take into account this number as well as a weight.

For testing purposes one may download and compile the GraphLab package, which have a KMeans++ algorithm implemented. http://docs.graphlab.org/clustering.html The program can generate the input of D dimensional points around given cluster centres, that is several lines of D real numbers. This input can be used for the project. Also the GraphLab software package implements a variation of the KMeans algorithm (the KMeans++), which is good for comparing the results.

A good k-means algorithm (or the improved k-means++) is a practical and useful program. We see the main advantage of its parallelization in the fact that the parallel version is keeping only a fraction of the whole data set in memory. This means that problems which cannot fit into the memory of one computer can be solved by this method. This program can be assigned as part of a BSc thesis work.

14.3. 14.3 Better parallelization for Gaussian elimination

The project aim is to make a better parallelization that we presented in out book. As we have seen that method uses too much communication, so we try to do it with less. The parallelization presented used loop splitting of rows. We propose to make loop splitting of columns, es eliminations always done between elements of the same column. So if we distribute the data by columns, then the elimination step can be done independently on each process.

The communication needed for pivoting only, where the process in charge of the pivoting element finds the maximum absolute value in that row. This information, namely which rows must be changed then may be broadcasted and so every process can make this interchange within those columns they assigned. No other communication is needed apart from the gathering the upper triangular matrix in the end by the master process.

The complexity of this project lays in the unusual data representation. We propose to rotate the whole matrix by 90°, so columns will be rows in our matrix. You should double check the correctness of the program!

14.4. 14.4 FFT further parallelization

In our FFT example, we choose a straightforward parallelization. We send the input data to 2^n processors and let them work on different outputs. Having all the input data at all processors, we can calculate 2^m output chunks at every worker. Again, for this, we have to hand out the whole input to all the processors. Let's consider a different way of parallelization. If we hand out 2^m amount of input data (not all!) for the processors, we could let them work until a some level of steps. At this point we have to pass these data to one processor which goes on with the recombination of these data until the very last number, our nth output. This multi-level work could be done not only at two levels, but also at more, thus, at the first data interchange, several (but not all) processors work on the next steps in the recombination. Could this method improve performance by not sharing all the input data with all the processors? Can we spare on the communication when we have to interchange (much less) data later on?

This new version of the implementation could be a good example to solve the same problem with a different logic. The extra communication during the calculation could be frightening, but we shouldn't forget that we spare the communication of the whole input series at the beginning (in our example, the static input is available for every worker at start, but in many cases, it is the first step to hand data out to every worker).

14.5. 14.5 An example from acoustics

Before modelling the Raytracer example showed the propagation of light in a 3D environment. A similar problem is the propagation of sound in a 3D environment. We simplified the spectrum of the light with three components (R,G,B) and supposed that the output of the light-sources is constant in time. Let's consider a sound source, which gives a sound, composed of different wavelengths, changing in time. This later is not necessary, but would give a more realistic scenario. We could simplify by splitting the whole spectrum into some narrowband components (Ex. 16 parts). We could give a reflecting factor for every object (material) for every narrowband spectrum chunk. We could count the distance of the waves to get the right super-positioned sound at a given time in the scene. With this simplified model, we could calculate acoustic wave propagation in a 3D scene.

This task could build on the Raytracer example with modifications changing colors to sound and counting with propagation delays in the scene.

Of course, this model works as if we could hear sound through a window. One further step could be to create a sphere-like (cube, tetrahedron, dodecahedron or a more complex) 3D object, where we can calculate the incoming sound from every direction, through summing the calculations for every plane of the object.

15. 1 Appendix

15.1. 1 Installation of MPI framework

Because our book is a textbook for those who would like to know the Message Passing Interface programming, we should not give long and detailed installation notes. Those, who would like to set up a system and optimize it, should consult a professional. Still, if one would like to "play" with the presented techniques, or even set-up a medium size environment, we need to help a little.

First, we should note, that the de facto standard of High Performance Computations is the Unix world, often a Linux. It is quite hard to manage windows PCs to give each other distant commands of starting programs. Not to speak about security issues. (Although for really small test reasons a windows computer may be used.) We urge the readers to get familiar with Linux not only because it is fun, but because if they find the subject presented interesting and would like to get more deeply involved in it, they may use a supercomputer in the future. A supercomputer, which will run a Linux or a Unix certainly.

So the reader will need a Linux computer or rather a couple of Linux computers to try out MPI. On a modern Linux one will need a compiler suite, usually the gcc is good enough, which can be installed by the package manager. On some systems one may need to install the g++ package as well if she or he uses the C++ language. The MPI programs, libraries and headers should be installed by the openmpi package. If the reader wants to use mpi over several machines she or he needs to set up the ssh as well, which means the installation of the packeges openssh-client and openssh-server, and possibly (not always) configurating the later by editing the system wide configuration files at /etc/ssh/.

If one has no access to a Linux (although it can be installed even on a virtual machine) then in Windows environment they may use the Cygwyn Unix system. It is a program package installed in Windows and providing a Linux environment. Still we think that a proper Linux installation would be a better idea.

15.1.1. 1.1 Communication over ssh

Our first advice is: Do not use rsh!

The rsh protocol is unsecured, and if you set up your own environment, you should never think of it as an option. (Of course if a special High Performance Computing environment is set up by a professional system administrator and the computers are unaccessible from outside, this possibility may be used with precautions.)

The usage of ssh may cause a little problem at first. The protocol is asking for a password each and every time. To override this behaviour we need to set up our ssh using secure keys.

First, we need to generate the keys (answer yes to all questions and leave the presented parameters unchanged):

```
shell$ ssh-keygen -t rsa
```

Second, we need to copy the key to the appropriate place:

```
shell$ cd $HOME/.ssh
shell$ cp id_rsa.pub authorized_keys
```

Note that the above example is valid in the case of a common file system, like in a computer class room. If the computers have different file systems for home (the usual case at home for example), than the id_rsa.pub of the master system (on which we will run the mpirun program) should be copied to the authorized_keys file of each machines in the home directory of the user. (The user-name of the user must be the same!)

The best way to test the ssh is to ssh over the other machine, and look, whether it is asking a password or not. (The first attempt will ask for a yes/no acknowledgement!) To test the mpirun program, one may start a command like

```
mpirun -hostfile hostfile.txt hostname
```

or

```
mpirun -hostfile hostfile.txt date
```

which will run the hostname Linux command and tell the host-names (we should see different names), and the time and date of the remote machine. The -hostfile switch is detailed later in this chapter.

In details: http://www.open-mpi.org/faq/?category=rsh

15.1.2. 1.2 Running the programs on one machine

The compilation of the programs always should be done with the mpi compiler wrapper. This is a program which calls the actual compiler (gcc, icc or other) and links it with the appropriate libraries. There are different wrappers for each language, in the case of C++ code, as in our book, the name of the wrapper is usually mpic++ or mpicxx. For C code it is usually mpicc, for Fortran code it may be mpif77 or mpif90. The switches are the same as for the compiler itself, for g++ it is usually the -O3 for optimization, and -o for naming the output executable.

On a single computer there is no need to bother with the communication. We can run our program on a single machine with the mpirun command. With the -np switch we can define the number of processes it should start. For example running 4 instances of our program one should type: mpirun -np 4 ./test_program.out.

For program testing purposes the actual number of cores in the computer is not really important. One can easily start 51 processes on a 2 core machine. It is useless in terms of speed, and actually will slow down the running. On the other hand, sometimes exhausting testing is needed, and this feature of the mpirun command is quite useful in such cases.

15.1.3. 1.3 Running the programs on more machines

If the user has more computers, like a classroom of computers, than after setting up the communication as described above, she or he must tell the mpirun command which computers it should use. For this purpose a simple text hostfile should be written, which has a computer name on each line, and alternatively telling the number of CPUs presented in that machine. A possible hostfile may look like this:

```
lab-201b-1 cpu=2
lab-201b-2 cpu=2
lab-201b-3 cpu=2
lab-201b-4 cpu=2
lab-201b-5 cpu=2
lab-201b-6 cpu=2
```

So we will use 6 machines, and each has 2 CPU cores in it. The command will be: mpirun -hostfile hostfile.txt ./test_program.out. The mpirun will find out the number of processes from the hostfile. The usage of the -np and -hostfile switches together is meaningless, so the user should use only one of them.

15.1.4. 1.4 Supercomputer usage

If the user is using a supercomputer, then she or he must consult the documentation of the actual supercomputer, which will describe the actual usage. Usually the programs are compiled with the same wrapper, say mpic++. The executing of the programs is different, as supercomputers may use accounting and batch processing queues. So the user usually should write a simple shell script - example scripts are usually presented in the documentation -, and send the script to the processing queue with a special command. Examples are usually given as well.

In our example (for the second version of our Mandelbrot program on 192 cores) we used a shell script named mpi-2sh. like this:

```
#!/bin/sh
#$ -N Mandel-2-192
mpirun -np $NSLOTS ./mandel2 mandel-2-192.ppm
```

The name of our executable is mandel2, the job will get the name Mandel-2-192. The output file generated was Mandel-2-192.o57979 - as in a supercomputer environment the direct output to the console display is not supported.

Finally we submit the job with the command:

```
qsub -l h_rt=0:5:0 -pe mpi 192 mpi-2.sh
```

telling the system to start a 192 process mpi job with 5 minutes time limit.

15.2. 2 MPI function details

15.2.1. 2.1 Communication

This section will shortly summarize the basic communication functions of the OpenMPI implementation of MPI.

15.2.1.1. 2.1.1 Blocking and non-blocking

There are two basic methods of the communication in MPI, the Blocking and the Non-blocking. The blocking versions will not return until the communication is done, thus, they are blocking the running. This is a safe method, we can be sure that the data will be sent or received after these instructions. In most of the cases, this method is used. However, the non-blocking method has its advantage as well. Although, the non-blocking methods will not guarantee successful sends and receives and we will have to check it with MPI functions, the sending procedure and program running can be overlapped. We can initiate a communication, calculate something and check whether the communication is done or not. With this method, we have to deal with checking ourselves, but because of overlapping, efficiency could be raised.

15.2.1.2. 2.1.2 Send

This function does a blocking send of a message. It will wait until the message is sent (until the send buffer can be used), hence the name blocking.

```
int MPI_Send(
    void *buffer,
        //Address of send buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI Datatype datatype,
        //Data type of elements of send-recieve buffer
```

```
    int dest,
        //Rank of destination
    int tag,
        //Message tag
    MPI_Comm comm
        //Communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Send.3.php

15.2.1.2.1. 2.1.2.1 Buffered MPI_Bsend

This is the buffered send. An user-supplied buffer space is used to buffer the message. Because of that, at sending time, it is hundred percent sure, that the application buffer is copied to the MPI buffer space. The user-specified buffer should be given by the MPI_Buffer_Attach() function 2.1.2.5. We should avoid buffered sends, they bring in latency in almost all MPI implementations.

```
int MPI_Bsend(
    void *buffer,
        //Address of send buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int dest,
        //Rank of destination
    int tag,
        //Message tag
    MPI_Comm comm
        //Communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Bsend.3.php

15.2.1.2.2. 2.1.2.2 MPI_Ssend

This sending method won't complete until a matching receive has been posted. That makes this method a pairwise synchronization event.

```
int MPI_Send(
    void *buffer,
        //Address of send buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int dest,
        //Rank of destination
    int tag,
        //Message tag
    MPI_Comm comm
        //Communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Ssend.3.php

15.2.1.2.3. 2.1.2.3 MPI_Isend

The instant send will prepare and start the sending, and returns immediately. We get back control fast and can go on with work, but we also have to test whether the send is done or not. This later could be done by an MPI_Wait. The MPI_Isend followed by an MPI_Wait immediately equals to an MPI_Send. If we separate them, we can do in our code some work parallel to the sending. At a certain point, we should be sure that the send occurred - so we test it with an MPI_Wait. With this method, we can overlap sending and working, speeding up the whole process.

```
int MPI_ISend(
    void *buffer,
        //Address of send buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int dest,
        //Rank of destination
    int tag,
        //Message tag
    MPI_Comm comm
        //Communicator
    MPI_Request *request
        //Communication request
    )
```

With the communication request handle, we can track the communication status or wait to complete.

http://www.open-mpi.org/doc/current/man3/MPI_Isend.3.php

15.2.1.2.4. 2.1.2.4 MPI_Rsend

The ready mode send. The message will be immediately transmitted under the assumption that a matching receive has already been executed. The code has to be written such, that the receive buffer is ready at the time of sending.

```
int MPI_Rsend(
    void *buffer,
        //Address of send buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int dest,
        //Rank of destination
    int tag,
        //Message tag
    MPI_Comm comm
        //Communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Rsend.3.php

15.2.1.2.5. 2.1.2.5 User supplied buffer

For non-blocking communication, a user defined buffer space is needed. After allocating such a buffer, the sender can attach it to MPI environment for use. This is done by the MPI_Buffer_attach() function and used by the sender in the communication.

```
int MPI_Buffer_attach(
    void *buffer,
        //Address of send buffer
```

```
    int size,
        //Buffer size in bytes
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Buffer_attach.3.php

15.2.1.2.6. 2.1.2.6 Detaching user supplied buffer

With this function, we can detach a previously attached buffer from the MPI environment.

```
int MPI_Buffer_detach(
    void *buffer,
        //Address of send buffer
    int *size,
        //Buffer size in bytes
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Buffer_detach.3.php

15.2.1.2.7. 2.1.2.7 MPI_Wait

Waits for an MPI request to complete. This request was sent by ISend()2.1.2.3. Execution of caller will stop at this call until it returns.

```
int MPI_Wait(
    MPI_Request *request
        //Communication request
    MPI_Status *status
        //Status object
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Wait.3.php

15.2.1.2.8. 2.1.2.8 MPI_Test

Tests for the completion of a specific send or receive. We can do some work and sometimes check whether a communication is ready or not.

```
int MPI_Test(
    MPI_Request *request
        //Communication request handle
    int *flag
        //True if operation completed
    MPI_Status *status
        //Status object
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Test.3.php

15.2.1.2.9. 2.1.2.9 MPI_Cancel

Cancels a previously initialized MPI communication request. E.g. This can be useful if we are sending a longer range of data with MPI_Isend and the master doesn't need it any more (it received it from an other process).

```
int MPI_Wait(
    MPI_Request *request
        //Communication request
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Cancel.3.php

15.2.1.3. 2.1.3 Receive

The MPI_Recv function is the counterpart of MPI_Send. Its function is the blocking receive of a message. It will wait until the message is received, hence the name blocking. The outputs are the address of the receive buffer buffer and the status object status, which is a structure which contains the following fields:

- MPI_SOURCE - id of processor sending the message
- MPI_TAG - the message tag
- MPI_ERROR - error status.

```
int MPI_Recv(
    void *buffer,
        //Address of receive buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int source,
        //Rank of source
    int tag,
        //Message tag
    MPI_Comm comm,
        //Communicator
    MPI_Status *status
        //Status object
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Recv.3.php

15.2.1.3.1. 2.1.3.1 Non-blocking, instant receive

The MPI_Irecv function is the counterpart of MPI_Isend. Its function is the indicating of readiness receiving a message. The returning status object is a structure which contains the following fields:

- MPI_SOURCE - id of processor sending the message
- MPI_TAG - the message tag
- MPI_ERROR - error status.

```
int MPI_Irecv(
    void *buffer,
        //Address of recieve buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int source,
        //Rank of source
    int tag,
        //Message tag
```

```
    MPI_Comm comm,
        //Communicator
    MPI_Status *status
        //Status object
    MPI_Request *request
        //Communicatin request
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Irecv.3.php

15.2.1.4. 2.1.4 SendReceive

The MPI_Sendrecv function sends and receives messages at the same time. This function is useful for executing a shift operation across a chain of processes. In contrary to MPI_Send and MPI_Recv, we don't have to care about correct order of *send* and *receive* pairs. The MPI communication subsystem will handle this issue. The function can communicate with MPI_Send and MPI_Recv functions, though.

```
int MPI_Sendrecv(
    void *sendbuf,
        //Address of send buffer
    int sendcount,
        //Number of elements in send buffer
    MPI_Datatype sendtype,
        //Data type of elements of send buffer
    int dest,
        //Rank of destination
    int sendtag,
        //Message tag
    void *recvbuf,
        //Address of receive buffer
    int recvcount,
        //Number of elements in receive buffer
    MPI_Datatype recvtype,
        //Data type of elements of receive buffer
    int source,
        //Rank of source
    int recvtag,
        //Message tag
    MPI_Comm comm
        //Communicator
    MPI_Status *status
        //Status object
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Sendrecv.3.php

15.2.1.5. 2.1.5 Broadcast

The function MPI_Bcast() sends out count number of datatype from buffer from process root to every other processes in the comm communicator. With this function, we can send the same data to all other processes at once. Since the implementation uses a tree-based network communication, it's more effective than sending the same data in a loop to every other processes.

```
int MPI_Bcast(
    void *buffer,
        //Address of send-recieve buffer
    int count,
        //Number of elements in send-recieve buffer
    MPI_Datatype datatype,
        //Data type of elements of send-recieve buffer
    int root,
        //Rank of root process
```

```
    MPI_Comm comm
        //Communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Bcast.3.php

15.2.2. 2.2 Reduction

The term reduction refers to changing a set of numbers into a smaller set of numbers, often one value, by the help of a function. E.g. summation is a reduction by adding. Reducing across distributed data, could be a big effort from scratch, but luckily MPI offers this functionality as a service.

15.2.2.1. 2.2.1 Reduce

The MPI_Reduce() function gives the basic reduce functionality. From the sendbuf local buffer, it uses count number of datatypes to do the global reduce operation given by op and sends output to the roots recvbuf.

```
int MPI_Reduce(
    void *sendbuf,
        //Address of send buffer
    void *recvbuf,
        //Address of receive buffer (significant only at root)
    int count,
        //Number of elements in send buffer
    MPI_Datatype datatype,
        //Data type of elements of send buffer
    MPI_Op op,
        //Reduce operation
    int root,
        //Rank of root process
    MPI_Comm comm
        //Communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Reduce.3.php

15.2.2.2. 2.2.2 All-reduce

The MPI_Allreduce() function works similarly to the previous one, but every worker gets the results back. From the sendbuf local buffer, it uses count number of datatypes to do the global reduce operation given by op and sends output back to every worker recvbuf.

```
int MPI_Allreduce(
    void *sendbuf,
        //Address of send buffer
    void *recvbuf,
        //Address of receive buffer (significant only at root)
    int count,
        //Number of elements in send buffer
    MPI_Datatype datatype,
        //Data type of elements of send buffer
    MPI_Op op,
        //Reduce operation
    MPI_Comm comm
        //Communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Allreduce.3.php

15.2.2.3. 2.2.3 Scan reduce

The MPI_Scan() performs an inclusive prefix reduction. If we have $0...n$ processes, and the process i is between 0 and n, then this means that the process with rank i gets the result of the reduction of data in processes $0...i$. In other words, the operation defined in Op will be done on processes $0...i$ and the result will be stored in the receive buffer of process i.

```
int MPI_Scan(
    void *sendbuf,
        //Send buffer
    void *recvbuf,
        //Receive buffer
    int count,
        //Number of elements in input buffer
    MPI_Datatype datatype,
        //Data types of elements in input buffer
    MPI_Op op,
        //Operation
    MPI_Comm comm
        //Communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Scan.3.php

15.2.2.4. 2.2.4 Operators

The next operators can be used as op parameter in the reduction functions of MPI.

Table 13: MPI operators of reduction

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

15.2.2.4.1. 2.2.4.1 Minimum and maximum location

There are two special reduction operators which perform on double values. The MPI_MAXLOC and the MPI_MINLOC The first value of the pair is treated as the value which should be compared with the others through the distributed processes. The second value is treated as an index, thus, it will indicate the position of the returned value.

The function MPI_MAXLOC defined as:

$$\begin{bmatrix} u \\ i \end{bmatrix} \star \begin{bmatrix} v \\ j \end{bmatrix} = \begin{bmatrix} w \\ k \end{bmatrix},$$

where $w = max(u, v)$, and

$$k = \begin{cases} i & \text{if } u > v \\ min(i,j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}$$

The function MPI_MINLOC defined similarly:

$$\begin{bmatrix} u \\ i \end{bmatrix} \star \begin{bmatrix} v \\ j \end{bmatrix} = \begin{bmatrix} w \\ k \end{bmatrix},$$

where $w = min(u,v)$, and

$$k = \begin{cases} i & \text{if } u < v \\ min(i,j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}$$

15.2.2.4.2. 2.2.4.2 Special double data types

For the above described MPI_MAXLOC and MPI_MINLOC functions one should use double values. The table below shows the built in double value types of MPI.

Table 14: Double data types

MPI_FLOAT_INT	float and int
MPI_DOUBLE_INT	double and int
MPI_LONG_INT	long and int
MPI_2INT	pair of ints
MPI_SHORT_INT	short and int
MPI_LONG_DOUBLE_INT	long double and int

To use them, one should first construct a structure as in the example in the Dijkstras's Algorithm of Shortest Paths. The send buffer will be p, the receive buffer will be p_tmp for the MPI_Allreduce function.

```
struct{
      int dd;
      int xx;
    } p, tmp_p;
    p.dd=tmp; p.xx=x;

    MPI_Allreduce(&p, &tmp_p, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD);

    x=tmp_p.xx;
    D[x]=tmp_p.dd;
```

15.2.3. 2.3 Dynamically changing the topology

In some cases, like in our FFT example (11.3.2.1) we wish to use special number of processors, or special groups of processors. For this purpose, we can use some MPI *group* and *communicator* functions from which we used the followings.

15.2.3.1. 2.3.1 Getting the group associated with a communicator

This function returns the group associated with the comm communicator.

```
int MPI_Comm_group(
    MPI_Comm comm,
        //Communicator
```

```
    MPI_Group *group
        //Group associated to communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Comm_group.3.php

15.2.3.2. 2.3.2 Creating a new group by excluding

This function creates a new group (newgroup) of ranks by excluding n number of ranges of ranks (triplets: first rank, last rank, stride) from an existing group (group)2.3.1. Stride is the step value we step in the given interval. In other words, we create a new group of processes by kicking out some.

```
int MPI_Group_range_excl(
    MPI_Group group,
        //Group handle
    int n,
        //Number of triplets in array ranges
    int ranges[][3],
        //Array of triplets
    MPI Group *newgroup
        //New group derived from a present one
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Group_range_incl.3.php

15.2.3.3. 2.3.3 Creating a communicator from a group

This function creates a new communicator (newcomm) from the group (group) from the old communicator (comm). The group could have been created by the MPI_Group_range_excl function (2.3.2). If a process is member of the group, it receives the new communicator (newcomm), if not, this value will be MPI_COMM_NULL.

```
int MPI_Comm_create(
    MPI_Comm comm,
        //Communicator
    MPI_Group group,
        //Group, which is a subset of the group of comm
    MPI_Comm *newcomm
        //The new communicator
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Comm_create.3.php

15.2.4. 2.4 Miscellaneous

15.2.4.1. 2.4.1 Initialize

To be able to use MPI functionality, first, we have to initialize the MPI environment with the MPI_Init() function. The argc and argv are the pointer to the number of arguments and the argument vector of main() function.

```
int MPI_Init(
    int *argc,
        //Argc of main()
    char **argv
        //Argv of main().
```

```
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Init.3.php

15.2.4.2. 2.4.2 Rank request

All running processes have a rank in the group of the particular communicator. This rank could be used as a unique identification number of the process in the communicator.

```
int MPI_Comm_rank(
    MPI_Comm comm,
        //Communicator
    int *rank
        //Rank of the calling process in group of comm
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Comm_rank.3.php

15.2.4.3. 2.4.3 Size request

The MPI_Comm_size() function returns the group size of the associated communicator. It is practical to know the size of our group, thus, the number of workers we can count on.

```
int MPI_Comm_size(
    MPI_Comm comm,
        //Communicator
    int *size
        //Number of processes in the group of comm
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Comm_size.3.php

15.2.4.4. 2.4.4 Finalize

The MPI_Finalize() function terminates the MPI execution environment. The running of processes after this is not defined, the best way is to return and exit after this call. However, it is possible to run processes on and carry out non-MPI actions. No MPI functions are allowed to call after finalizing the MPI environment.

```
int MPI_Finalize()
```

http://www.open-mpi.org/doc/current/man3/MPI_Finalize.3.php

15.2.4.5. 2.4.5 Abort

Terminates all processes associated with the communiator comm. The function doesn't return. This is an emergency stop, MPI_Finalize() should be used instead.

```
int MPI_Abort(
    MPI_Comm comm
        //Communicator
    int errorcode
        //Error code to return to invoking environment
```

```
    )
```

http://www.open-mpi.org/doc/current/man3/MPI_Abort.3.php

15.2.4.6. 2.4.6 Barrier

The MPI_Barrier function blocks until all processes in the communicator have called this routine, thus, are at this given point in the execution.

```
int MPI_Barrier(
    MPI_Comm comm
        //Communicator
    )
```

Actually barriers are widely used in shared memory environment, as reading of shared variables can be error prone if some partial computation by an other thread is not ready at that point. In a distributed environment the Send-Receive functions will synchronize the different processes, so in our examples the MPI_Barrier function can or should be used rarely if ever. Sometimes it is used for testing purposes, e.g. before starting to measure running time, we can sync administrative activities and start a clean run from this point. It is worth considering to avoid the usage of this function. There is a good article according to this at *www.niif.hu/en/node/311* in Hungarian with the title "Miért lassulnak le az MPI programok? - Why do MPI programs slow down?".

http://www.open-mpi.org/doc/current/man3/MPI_Barrier.3.php

15.2.4.7. 2.4.7 Wall time

The MPI_WTime gives the time elapsed in seconds, since arbitrary point in the past. This point of start is guaranteed not to be change during running of the process. Thus, we can use the difference of two such values to measure elapsed time for some operations.

```
double MPI_Wtime()
```

Returns elapsed time in seconds.

http://www.open-mpi.org/doc/current/man3/MPI_Wtime.3.php

15.2.5. 2.5 Data types

The standard MPI predefines some basic data types for C like int (MPI_INT), signed long int (MPI_LONG), and all the basic datatypes of C. If we want to send different data types, we have several options. The most simple way is to send different types in different messages. One other option is to MPI_Pack different types into a buffer and send it at once. A further option would be MPI_BYTE where data is sent byte wise. However these two later might make our program not portable. For further options (not scope of this book) refer to derived datatypes in MPI. This advanced method will allow to construct special data structures at runtime. This can be thought as a template for mixed datatypes. The basic MPI datatypes are shown in table 15.[1]

[1]Note, that this is version 3.0 standard, and some of the datatypes may not be present in your MPI implementation yet.

Table 15: MPI basic datatypes

MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char (treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t (defined in ¡stddef.h¿) (treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

15.3. References

- [Akl1985] Akl, S.G. Parallel Sorting Algorithms. Academic Press. 1985.
- [Braw1989] Brawer, S. Introduction to Parallel Programming. Academic Press. 1989.
- [Corm2009] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. Introduction to Algorithms. 3rd ed. MIT Press and McGraw-Hill. 2009.
- [Cool1965] Cooley, J.W. and Tukey, J.W. An algorithm for the machine calculation of complex Fourier series. In: Mathematics of Computation. 19. pp. 297–301, 1965.
- [Dijk1959] Dijkstra, E.W. A note on two problems in connexion with graphs. In: Numerische Mathematik 1. pp. 269-271. 1959.

- [Gram2003] Grama, A., Gupta, A., Karypis, G. and Kumar, V. Introduction to Parallel Computing. 2nd ed. Addison Wesley. 2003.
- [Hilb2013a] Hilbert, S. FFT Zero Padding. BitWeenie, DSP resource page. 2013. http://www.bitweenie.com/listings/fft-zero-padding/
- [Hilb2013b] Hilbert, S. A DFT and FFT TUTORIAL. Alwayslearn, SoundAnalyser Project page. 2013.
- [Ivanyi2013] Iványi P. and Radó J. Előfeldolgozás párhuzamos számításokhoz Tankönyvtar, 2013.
- [Kan1998] Kanevsky, S.A.Y. and Rounbehler, A.Z. MPI/RT an emerging standard for high-performance real-time systems HICSS, pp. 157-166. 1998.
- [Lyons2004] Lyons, R.G. Understanding Digital Signal Processing. Second Edition. Prentice Hall. 2004.
- [Matt2005] Mattson, T.G., Sanders, B.A. and Massingill, B.L. Patterns for Parallel Programming. Addison-Wesly. 2005.
- [PovRay] PovRay - Persistence of Vision Raytracer. http://www.povray.org/documentation
- [Sima1997] Sima, D., Fountain, T. and Kacsuk, P. Advanced Computer Architectures: A Design Space Approach. Addison Wesley. 1997.
- [Szab2012a] Szabó S. A Non-Conventional Coloring of the Edges of a Graph. Open Journal of Discrete Mathematics, 2012, 2, 119-124 doi:10.4236/ojdm.2012.24023 Published Online October 2012. (http://www.SciRP.org/journal/ojdm)
- [Szab2012b] Szabó S. and Zaválnij B. Greedy Algorithms for Triangle Free Coloring. AKCE Int. J. Graphs Comb., 9, No. 2 (2012), pp. 169-186.
- [Weisst] Weisstein, E.W. Discrete Fourier Transform. From MathWorld-A Wolfram Web Resource. http://mathworld.wolfram.com/DiscreteFourierTransform.html