Scientific Computing Mandatory Assignment 2

Magnus Erik Hvass Pedersen
University of Aarhus, Student #971055
Danish Technical University, Student #053375
November 2005

1 Introduction

The purpose of this document is to verify attendance of the author to the *Scientific Computing* course, at the Danish Technical University (DTU). The reader is assumed to be familiar with the project description, as well as the programming language C++, and the distributed computing interfaces OpenMP and MPI.

After a brief outline of the 2-dimensional Poisson problem, numerical methods for approximating its solution are given. An implementation of these methods is described for the C++ programming language, including the distributed computation through the use of OpenMP and MPI. The resulting software is tested for correctness, and various performance benchmarks are produced. This is believed to be in correspondance with the project description, and the project is hence considered wholly solved.

2 The Poisson Problem

According to the problem description, the 2-dimensional Poisson problem consists of finding the function u over some domain $\Omega \subset \mathbb{R}^2$:

$$u : \Omega \to \mathbb{R}$$

such that the function u satisfies the following partial differential equation:

$$\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2} = f(x, y) \tag{1}$$

for all $(x, y) \in \Omega$. The function f is the socalled *source-term*, and is a function given in advance, for example modelling some real-world scenario. The values that u takes at the boundaries of Ω, are also given in advance, and are called the boundary conditions.

2.1 Heat-Radiator Example

Here, we shall consider a simplified model of heating a living-room, by placing a heat-radiator close to one of the walls. To make things easier for ourselves, we shall restrict the domain to $[-1, 1]^2$, that is:

$$\Omega = \{(x, y) \ : \ |x| \leq 1 \wedge |y| \leq 1\}$$

The source-term f representing the heat-emitting radiator, is then defined by:

$$f(x,y) = \begin{cases} 200 & , |x| \leq 1/3 \wedge -2/3 \leq y \leq -1/3 \\ 0 & , \text{else} \end{cases}$$

And the boundary conditions for this problem, that is, the values u takes at the boundaries of the domain Ω, are as follows:

$$\begin{array}{rcl} u(x,1) & = & 20 \\ u(x,-1) & = & 0 \end{array} \quad \text{when } |x| \leq 1$$

and

$$u(1,y) = u(-1,y) = 20, \ |y| \leq 1$$

Note that the exact meaning of this, in terms of the relationship between the mathematical model and the actual physical heating of a room, is not really important for this project, as we are merely interested in the computational aspects.

2.2 Gauss-Seidel Iteration

One way of solving Eq.(1) numerically, is to make a discrete and square grid A, consisting of $N \times N$ points, and then iteratively apply the socalled *Gauss-Seidel* formula:

$$a_{i,j} \leftarrow \frac{1}{4}\left(a_{i,j-1} + a_{i,j+1} + a_{i-1,j} + a_{i+1,j} + \Delta^2 \cdot f(x,y)\right) \tag{2}$$

where $A = (a_{i,j})$ is the $N \times N$ grid (or matrix) of approximations to the function u,[1] $\Delta = 1/N$ is the grid-spacing, and $f(x,y)$ is the value of the source-term at position (x,y), which for the time being, we shall say corresponds to grid-position (i,j).[2]

2.2.1 Initialization & Convergence

It is beyond the scope of this document to derive the Gauss-Seidel formula, and also to show that the grid-values of A will converge to the function that we seek; that is $a_{i,j} \to u(x,y)$ after enough iterations of Eq.(2).

Assuming convergence is inevitable, the speed with which the values of A converge to the sought function u, is naturally also of great importance. However, if we have no prior knowledge about the function u, then one choice of initial values, is of course as good as any other.

In this paper however, our testing and benchmarking is not concerned with the quality of the results (as long as they appear correct), but our interest is

[1] Note the notation $a_{i,j}$ instead of $u_{i,j}$ as suggested in the problem description, which is a bit confusing, as u would then represent both a function as well as a value approximating it.

[2] The exact mapping between a position (x,y) in Ω and the discrete grid-indices (i,j), naturally depends on how the grid is indexed, e.g. from 0 to $N-1$, or from 1 to N, etc. For the present implementation, the actual mapping is given in section 3.1 below.

merely in how long it takes to compute, say, a thousand iterations of Eq.(2) for the entire grid. We will therefore not try and guess the initial values for A, but merely set $a_{i,j} = 0$ for all points in the grid.

2.2.2 Algorithm

Before depicting the algorithm for computing approximations to the 2D Poisson problem, let us first describe how we intend to store the grid and its boundary, and access the individual points or elements.

To make computation easier, we will store the values for the boundaries in the grid also, which merely requires a slight accomodation in the indexing of its elements, but which easens the application of the Gauss-Seidel formula on grid-points neighbouring the boundaries. That is, we create a grid of size $(N+2)\times(N+2)$, where $i \in \{0, \cdots, N+1\}$ is the vertical index, thus designating a row in the grid or matrix, and likewise $j \in \{0, \cdots, N+1\}$ is the horizontal index, designating a column in the grid. The algorithm for applying the Gauss-Seidel formula iteratively, is therefore:

- Initialize all inner grid-points by setting $a_{i,j} = 0$ for each $i, j \in \{1, \cdots, N\}$.[3]
- Initialize the boundaries with their appropriate values.
- Repeat the following a certain number of times:
  - Update each of the inner grid-points by using Eq.(2), for example by having an outer-loop iterating over i (going from 1 up to N), and an inner-loop iterating over j (also going from 1 up to N).

As noted above, this project is not concerned with convergence or quality of the approximation, but merely in performance tuning. One suggestion for measuring the convergence however, would be to sum the absolute differences for each pair of the $a_{i,j}$'s, that is, before and after one step of applying the Gauss-Seidel formula. This measure can presumably be used as an indication of convergence, provided of course the progression of convergence is fairly smooth (or at least that it does not alternate between improving and worsening the approximations).

2.3 Red-Black Iteration

In the implementation below, we are going to parallelize the algorithm from section 2.2.2 in two ways, first on a *Shared Memory Processor* (SMP), and then on a network of seperate computers (distributed memory). To parallelize an algorithm on an SMP, we will use the *OpenMP* framework, which merely supplies a set of compiler directives, that will generate the code necessary for splitting and combining the work onto several processors sharing the same memory. This

[3] Actually, the implementation sets $a_{i,j} = 0$ for $i, j \in \{0, \cdots, N+1\}$, because it must support distributed computation also.

splitting and combining, should of course be done in a way, so as to maximize the degree of parallelization, and minimize the degree of serialization, which means each of the processors should work on a part of the problem, that is as independent as possible from what the other processors are currently working on.

2.3.1 Dependencies of Gauss-Seidel

Now, if we were to analyze the internal depencies of the algorithm in section 2.2.2, so as to deduce whether or not there is a required ordering of the computations, then we would find that the computation of each element $a_{i,j}$, depends on the value of every other element in the solution! Which can be seen by applying Eq.(2) recursively.

Naturally, it is not possible to use values that have not yet been computed, regardless of how we order the computation, and indeed, this is not even required by the Gauss-Seidel algorithm, as any ordering of the computations should do.

2.3.2 Red-Black Colouring

We can however, make it easier for any dependency analysis to see that the grid-points are not strongly dependent on each other. To do this, start by noticing that if we colour the points or elements of the grid, in the fashion of a red and black chess-board, then if $a_{i,j}$ is coloured red, its update only depends on directly neighbouring values (see Eq.(2)), that because of the colouring scheme, are of course white.

2.3.3 Red-Black Algorithm

So we may create a variant of the algorithm in section 2.2.2, for which the iteration consists of first updating the values of, say, all red grid-points, and then all white grid-points. This means there are no dependencies between values computed in each of these parts. The loop of the algorithm from section 2.2.2 therefore becomes something like:[4]

- Repeat the following a certain number of times:
  - Update each of the *red* inner grid-points by using Eq.(2), and by having an outer-loop iterating over i (going from 1 up to N), and having an inner-loop iterating over every other j; and now going from $1 + ((i + 1) \mod 2)$ up to N (possibly $N - 1$, depending on the value of N).
  - Update each of the *black* inner grid-points by using Eq.(2), and again by having an outer-loop iterating over i (going from 1 up to N), and

[4] In order for the OpenMP compiler to understand the loop-conditions, it is necessary that they be of a rather simple form. So even though there are many ways of creating this red-black separation, not all of them will work with OpenMP.

having an inner-loop iterating over every other j; now going from $1 + (i \mod 2)$ up to N (possibly $N - 1$).

Notice the modulo-arithmetics, which are used to interchange between starting the row-iteration at the first and the second elements of the given row, thus mimicking the Red-Black colouring. Also note that for the top-row, we start at column-index $j = 1$, which probably does not affect cache locality any (see next section), but still appears to be the orderly point of starting the iteration.

2.4 Block Decomposition

Recall the concept of *cache locality* and how it may significantly impact computational performance (see e.g. [1]). To briefly summarize: When we are working on very large data-sets, we should try and ensure that memory access follows two patterns, so as to maximize memory cache use, and hence improve computational performance:

1. Data should be accessed in the order in which it is stored in memory; which means data should be accessed linearly. This is known as *spatial locality*.

2. Adding to this; whenever data must be re-used, we should try and do it as soon as possible, without performing computations on a lot of other data first. This is known as *temporal locality*.

When the grid-size N is *very* large, both the basic Gauss-Seidel algorithm from section 2.2.2, and its Red-Black variant from section 2.3, have poor temporal cache locality. The reason is, that they first update the values of one row, and then move on to the next row, and as the Gauss-Seidel formula in Eq.(2) requires values from the rows immediately above and below,[5] these may no longer be in cache, as they are stored approximately N elements away from the current grid-point.[6]

The basic idea for alleviating this problem, is to add an extra layer of outer loops, that break the grid into smaller sub-grids, for which the rows can be kept in cache. For the algorithm in section 2.2.2, we just need to break the grid into several column-wise blocks, of some appropriate (and somewhat machine-dependent) width. For the Red-Black variant, we would like both the red and black iterations to have good cache locality, so we need to break the grid into smaller and possibly non-quadratic rectangles, so the black iteration can re-use cache-data left over by the red iteration; also for the first row of that block. The implementations of both of these block-decompositions are detailed in section 3 below.

[5] As well as the values to the left and right of the grid-point to be updated; but these should be in cache.

[6] Not including the grid-boundary, and not taking into account if the storage for the individual rows is allocated consecutively or not; which would only make cache locality worse.

2.5 Necessity of Block Decomposition

Modern machines have maybe 1 or 2 MB of socalled L2 cache, and the implementation below, stores the value of a grid-point as a floating-point datatype, that requires 8 bytes per instance. So for the basic Gauss-Seidel algorithm, we would need approximately $2 \cdot N \cdot 8$ bytes of cache, to ensure both $a_{i-1,j}$ and $a_{i+1,j}$ can be kept in cache after their first use, and re-used in the updating of $a_{i,j}$. This means a machine with 1 MB of cache, can accomodate a 2D-Poisson problem with $N \simeq 65536$, and still have good cache locality. Of course, storing a grid of this size, would take up about 32 GB of memory, and would require vast amounts of computational time to perform just a single Gauss-Seidel iteration on. In short, we need not worry about cache locality for the basic algorithm from section 2.2.2.

2.5.1 Red-Black Iteration

At a first glance, we would like to store the entire grid in cache for the Red-Black variant, because each iteration will go through the grid twice; once for the Red grid-points, and once for the Black. This will also improve performance when performing several Red-Black iterations in turn; and naturally, as we also execute several iterations when using the basic algorithm from section 2.2.2, it is beneficial to keep the entire grid in cache for that algorithm as well.

This means the following considerations also apply there, and in fact, when viewed over several iterations, the two algorithms are more or less the same, in terms of cache locality.

2.5.2 Entire Grid in Cache Memory

If we wish to keep the entire grid in cache memory, we would need approximately $8 \cdot (N+1)^2$ bytes; so for a 1 MB cache, we would therefore be able to keep grids of size $N \simeq 127$ in cache.[7]

When the grid-size N is above 127, but below approximately 65536 (for 1 MB of cache memory), the cache is sufficiently large to hold the grid-points needed by the Gauss-Seidel formula, and it is first when N becomes higher than 65536, that we will experience socalled *thrashing* of the cache; in which the data we need, has always just been discarded from the cache.

2.5.3 Nested Parallelism

So is block-decomposition even relevant? No, not really. But it gives us an opportunity to study something interesting about distributed computation using OpenMP, as we will make use of socalled *nested* parallelism, and thereby discuss the pro's and con's of splitting the computation in different ways, which may be interesting and much more relevant in other contexts.

[7] Other data is also held in cache, so the actual grid-size our implementation can keep in cache, is likely less.

3 Implementation

Implementation was carried out in two phases, first at home and using *MS Visual C++ .NET 2003*, and then using the SUN machines from DTU, which have multiple CPUs, both for use with OpenMP, but also for simulating distribution with MPI.

3.1 Grid & Coordinates

Before anything else, let us start by describing how the grid is stored, and how its points are indexed.

The grid is stored as an array-of-arrays, where the grid-points are double-precision floating point numbers. That is, the grid is of datatype `double**`. Points in the grid are enumerated and indexed from 0 through $N + 1$, where those two extremes are the boundaries. In the source-code below, the index i designates a row, and index j designates a column. These may be mapped to x and y coordinates (both in the $[-1, 1]$ range) as follows:

$$x = 2 \cdot \frac{i}{N+1} - 1$$

and similarly for y:

$$y = 2 \cdot \frac{j}{N+1} - 1$$

Note that i maps to x, and j to y.

Functions are provided for getting and setting grid-values, and are named `GetU()` and `SetU()`, taking as parameters the indices i and j.[8]

3.2 Object-Oriented Design

The object-oriented design is very simple, in that there are only two classes; one class for numerically solving the Poisson problem (this class is named `LPoisson`), and one class for distributing computation using MPI (named `LPoissonMPI`). The source-code still exhibits a high degree of modularity though, but at a function-level rather than the class-level, as illustrated in the following sections.

3.2.1 Poisson Instance

Instantiating the Poisson problem and performing iterations on it, is then a rather simple matter, as one merely creates an object of the desired class (either `LPoisson` or `LPoissonMPI`), by passing appropriate arguments to its constructor, and then calls the desired iteration-function a number of times.

[8] Even though the grid-values are denoted a above, they are (somewhat confusingly) called `u` in the source-code.

3.2.2 Fixed Source-Term Function

This implementation is fixed into solving the Heat-Radiator example from section 2.1. The function f is therefore hardcoded as follows:

```
double f (double x, double y) const
{
    assert(std::fabs(x) < 1);
    assert(std::fabs(y) < 1);

    return (std::fabs(x) <= 1.0/3.0 &&
            -2.0/3.0 <= y && y <= -1.0/3.0) ? (200) : (0);
}
```

Sure, this could be made as a virtual function, and hence specialized in a subclass, but as the function gets evaluated once per grid-point, the overhead would be significant. Another solution is therefore needed, if the `LPoisson`-class must be used in a more general setting that requires solving Poisson problems with different source-terms.

3.2.3 Source-Term Storage

The simplest solution to avoiding fixation of the source-term, is perhaps to store the values of the source-term in a grid also. Then the overhead of declaring `f` to be a virtual function, would only be negligible, as the function would only be called during initialization of the `LPoisson`-object.

Naturally, this would require additional storage for the grid holding the source-term values, which effectively doubles our memory requirements. It also means that the source-term can not be made to change over the course of time, which may be needed by some particular instances of the 2D-Poisson problem.

3.3 Reverse Inheritance

A small trick can be employed to enable the specialization of the f-function, without the costly overhead of making it a virtual function, and without any additional storage-requirements. The trick may be called *reverse* inheritance, in that it makes use of inheritance, but does it the other way around than usual.

3.3.1 Regular Inheritance & Specialization

Usually, we would allow for specialization of the `LPoisson`-class by making the `f()`-function virtual like this:[9]

```
class LPoisson
{
public:
```

[9]Note that three dots `...` are used to indicate trivial matters. The reader is expected to have a fair knowledge of object-oriented programming in C++, to fully appreciate section 3.3.

```
    LPoisson (...) : ... { ... }

    virtual double f (double x, double y) const = 0;
}
```

Where the `=0` at the end, indicates the function is *pure* virtual, and therefore *must* be specialized in a sub-class of `LPoisson`. Using virtual functions (regardless of purity), is generally known as specialization through inheritance, and has the problems with computational overhead discussed above.

3.3.2 Template Super-Class

Instead of making `f()` a virtual function, we may remove `f()` from the `LPoisson`-class altogether, and make the `LPoisson`-class itself inherit from some template-argument `T`, which is then assumed to be a super-class containing the `f()`-function:

```
template <class T>
class LPoisson : public T
{
public:
    LPoisson (...) : T(), ... { ... }

    // Same implementation as usual, only "f" is omitted.
}
```

The `f()`-function may then be implemented in a separate class, for example called `LHeat` for the Heat-Radiator example of this project:

```
class LHeat
{
public:
    inline double f (double x, double y) const { ... }
}
```

Note that the `f()`-function can now be declared `inline` as it is no longer a virtual function.

3.3.3 Instantiating Poisson Specialization

All we have to do now, is to instantiate the `LPoisson`-class with `LHeat` as its template argument:

```
LPoisson<LHeat> myPoisson(...);
```

Or we can make a type-definition first:

```
typedef LPoisson<LHeat> TPoisson_Heat;
TPoisson_Heat myPoisson(...);
```

Or we can make a whole new class:

```
class LPoisson_Heat : public LPoisson<LHeat>
{
public:
    LPoisson_Heat (...) : LPoisson<LHeat>(...) { }
}
```

3.3.4 Reverse Inheritance in General

A problem with reverse inheritance in general, is that `f()` could have needed some of the variables or functions from `LPoisson`.[10] However, there are many ways of solving this, depending on the particulars of the given situation. An easy solution is to pass references to the required variables as parameters to the `f()`-function. This becomes a bit more difficult when `f()` calls functions from `LPoisson`, especially if we want those functions to be compiled `inline` – but, it can be done, although it is beyond the scope of this document to go into a full *how-to* description.

One example of this problem, that could indeed arise in the `LPoisson`-class, is when initializing the values in the grid; and in particular the boundaries. Although the performance-issue is not really pressing here, we should still somehow specialize the function that determines the values of the boundaries, and we may therefore use this as illustration of the problem. Now, if we choose to have the specialized function actually initialize the boundaries by itself, instead of just returning the value given some indices, then the function would need access to the grid, which is of course stored in `LPoisson`, the *sub*-class of `LHeat`, and not the *super*-class. But passing a reference to the storage of the grid, would solve this problem.

3.3.5 Performance & Implementation Issues

There are primarily two reasons why the implementation has not been made to support specialization through reverse inheritance – apart from the fact that it is not required by the project description. The reasons are:

1. Not all C++ compilers support templates equally well, and particularly when it comes to automatically optimizing the code for template-classes and -functions. This would void the purpose of reverse inheritance, which was to provide high-performing specialization of a class.

2. At present, the source-code for template-classes must be written entirely in the header-file, so the implementation of a class can not be written separately in a source-file (filename-extension `.cpp`), which makes the source-code more difficult to maintain.

[10] We are just using those class- and function-names to conceptually illustrate this particular problem with reverse inheritance.

3.4 Gauss-Seidel Algorithm

Now let us turn to the Gauss-Seidel algorithms. First is the basic algorithm from section 2.2.2, where the function that one should call for performing a single Gauss-Seidel iteration of the entire grid, is the following:

```
void Iterate ()
{
    DoIterate(1, kSizeN);
}
```

As can be seen, the function merely calls another function, `DoIterate()`, with two parameters, of which the latter is the number of columns in the grid, `kSizeN`. Remember that an `LPoisson`-object may only hold a smaller part of the entire grid, as it must support distributed computing through MPI also. The function `DoIterate()` takes as parameters the column-indices `J1` and `J2`, between which the Gauss-Seidel iteration will be performed:[11]

```
void DoIterate (const int J1, const int J2)
{
    #pragma omp parallel for
    for (int i=1; i<=kSizeM; i++)
    {
        double y = MapIndexI2Y(i);

        DoIterateRow(i, y, J1, J2);
    }
}
```

So `DoIterate()` iterates over each row, and after mapping the row-index `i` to the coordinate `y`, it calls `DoIterateRow()` with the current row-index, the mapped coordinate `y`, and the same column-indices `J1` and `J2` again. Finally, the function `DoIterateRow()` iterates over the grid-points in a row, and the function also takes a fifth parameter, which is the *stride* between increments of the column-index `j`. The stride defaults to 1, and makes it possible to re-use the function for the Red-Black variant of the Gauss-Seidel algorithm. The function `DoIterateRow()` is hence implemented as follows:

```
void DoIterateRow (const int i, const double y,
                   const int J1, const int J2,
                   const int stride=1)
{
    assert(J1>=1 && J1<=J2 && J2 <=kSizeN);
    assert(stride>=1);

    for (int j=J1; j<=J2; j+=stride)
    {
```

[11] The OpenMP pragmas will be discussed in section 3.7.

```
            double x = MapIndexJ2X(j);
            double u = CalcU(x, y, i, j);

            SetU(i, j, u);
        }
    }
```

Finally, the function `CalcU()` implements the Gauss-Seidel update-formula from Eq.(2), and should have no surprises:

```
double CalcU (double x, double y, int i, int j) const
{
    return 0.25 * ( GetU(i, j-1) +
                    GetU(i, j+1) +
                    GetU(i-1, j) +
                    GetU(i+1, j) +
                    kDelta2 * f(x, y) );
}
```

The reason for calling `GetU()` instead of just addressing the grid-points directly as in `mU[i][j]`, is that `GetU()` allows us to add assertions to the indices, to ensure they are in the correct ranges. It also allows us to more easily exchange the actual storage of the grid, e.g. to a single-dimensional array. In short, it is just good grooming, to have this kind of source-code modularity.

3.4.1 Block Decomposition

For the block-decomposed version of the basic Gauss-Seidel algorithm, we may use `DoIterate()` from above. All we then need to do, is to add outer-loops that split the grid into smaller blocks, and then call `DoIterate()` on those:

```
void Iterate (int blockWidth)
{
    assert(blockWidth>=1);
    blockWidth = std::min(blockWidth, kSizeN);

    #pragma omp parallel for
    for (int J1=1; J1<=kSizeN; J1+=blockWidth)
    {
        int J2 = std::min<int>(J1+blockWidth-1, kSizeN);

        DoIterate(J1, J2);
    }
}
```

Note that the function accepts block-widths greater than the width of the grid, and merely truncates the block-width to fit. Also note the mathematical style of block-decomposition, in which the index `J1` is incremented with the block-width

in the for-loop construct, and the index `J2` is then ensured not to go beyond the grid inside the loop. This block-decomposition could be done in numerous ways, but as the loop must support OpenMP, there are some restrictions, and the above is efficient, easy to understand, and supports OpenMP.

3.5 Red-Black Algorithm

Implementing the Red-Black variant of the Gauss-Seidel algorithm, follows the same style as its basic counterpart. The function one should call for performing a Red-Black iteration of the entire grid, and in a non-block-decomposed manner, is the following:

```
void IterateRedBlack ()
{
    DoIterateRedBlack(1, kSizeM, 1, kSizeN);
}
```

As above, this merely calls a function which takes block-indices as parameters. So to process the entire gir, the parameter for the starting row (and column) is 1, and the end-row is `kSizeM`, and the end-column `kSizeN`.

This other function is then implemented as follows, where it should be noted that the function `DoIterateRow()` from the basic algorithm above is being re-used, only here with the stride-parameter set to 2:

```
void DoIterateRedBlack (const int I1, const int I2,
                        const int J1, const int J2)
{
    assert(I1>=1 && I1<=I2 && I2 <=kSizeM);
    assert(J1>=1 && J1<=J2 && J2 <=kSizeN);

    #pragma omp parallel
    {
        #pragma omp for
        for (int i=I1; i<=I2; i++)
        {
            double y = MapIndexI2Y(i);

            DoIterateRow(i, y, J1+(i+1)%2, J2, 2);
        }

        #pragma omp for
        for (int i=I1; i<=I2; i++)
        {
            double y = MapIndexI2Y(i);

            DoIterateRow(i, y, J1+i%2, J2, 2);
        }
```

```
    } /* end omp parallel */
}
```

3.5.1 Block Decomposition

To block-decompose the Red-Black algorithm, all we need to do, is to make two outer-loops to compute the row- and column-indices for the blocks, and then call the `DoIterateRedBlack()`-function from above with these indices as parameters:

```
void IterateRedBlack (int blockWidth, int blockHeight)
{
    assert(blockWidth>=1);
    assert(blockHeight>=1);

    blockWidth = std::min(blockWidth, kSizeN);
    blockHeight = std::min(blockHeight, kSizeM);

    #pragma omp parallel for
    for (int J1=1; J1<=kSizeN; J1+=blockWidth)
    {
        int J2 = std::min<int>(J1+blockWidth-1, kSizeN);

        for (int I1=1; I1<=kSizeM; I1+=blockHeight)
        {
            int I2 = std::min<int>(I1+blockHeight-1, kSizeM);

            DoIterateRedBlack(I1, I2, J1, J2);
        }
    }
}
```

3.6 Output

Outputting the grid-points is done by the following function, which takes as argument, an output-stream. This stream can be either a file, or e.g. the standard-output (`std::cout`), which was useful during basic testing during implementation:

```
void OutputGnuPlot (std::ostream& out) const
{
    for (int i=0; i<=kSizeM+1; i++)
    {
        for (int j=0; j<=kSizeN+1; j++)
        {
            double x = MapIndexJ2X(j);
            double y = MapIndexI2Y(i);
```

```
            double u = GetU(i, j);

            out << x << " " << y << " " << u << "\n";
        }
    }
}
```

3.7 OpenMP

We will not go into a detailed discussion of OpenMP variable scoping, thread teams, and what not, but merely focus on the overall aspects in relation to iterations on grids in general, and 2-D Poisson grids in particular. The reader may find it helpful to draw grids and block-decompositions as we go along.

3.7.1 Gauss-Seidel Algorithm

Executing the function `Iterate()` for the basic Gauss-Seidel algorithm, results in splitting into several OpenMP threads in the `for`-loop of the function `DoIterate()` from section 3.4.

Generally speaking, we should not assume anything about the way OpenMP chooses to split the iterations of a `for`-loop amongst its threads, however, a fair guess would be that each processor iterates approximately from index $k \cdot M/P$ to index $(k+1) \cdot M/P$, where k is the id of the thread, ranging from zero to $P-1$, and M is the width of the sub-grid stored in that particular instance of the `LPoisson`-class. Under these assumptions, and because of the way this particular `for`-loop is made, each OpenMP thread will process a *vertical* block of the grid.

For grid-sizes N, large enough to cause thrashing of the memory cache (see section 2.4), one should either directly split the parallelism horizontally, or use the block-decomposed version of the algorithm. In the latter case, one should ensure that parallelism is as expected; namely horizontally. The way to do this while retaining the OpenMP directives in the `DoIterate()`-function, that is used by both the non-block-decomposed and block-decomposed algorithms, is described next.

3.7.2 Block-Decomposition & Nested Parallelism

If we choose to run the block-decomposed version of the Gauss-Seidel algorithm instead, then the OpenMP parallelism is first done *horizontally*, as can be seen from the `for`-loop in section 3.4.1, which means each OpenMP thread should execute in line with those block-decompositions.

Depending on the OpenMP facilities, either an additional layer of nested parallelism is created in the call to the `DoIterate()`-function, or the OpenMP directives in that function are simply ignored. (See [2, Section 2.4.1] for details on nested OpenMP directives.)

3.7.3 Red-Black Algorithm

OpenMP parallelism of the Red-Black algorithm, is done similarly, by having the normal algorithm split the threads vertically over the grid, and the block-decomposed algorithm split horizontally.

3.8 MPI

The first implementation of the `LPoisson`-class only supported quadratic grids. An MPI implementation was made that utilized this version of the `LPoisson`-class, but which was limited by the fact that all the nodes' sub-grids had to be quadratic also. Then considerations on whether or not to use the MPI-library's topology-support, finally lead to the conclusion, that by far the easiest way to distribute computation of the 2D-Poisson problem, would be to split the grid either row- or column-wise.

Initially, one might think that it is advantageous in regards to cache locality, to split the grid into blocks with approximately N rows and N/P columns (with P being the number of MPI processing nodes). But from the discussion in section 2.4, we know this makes no difference on cache locality, unless the entire sub-grid can be kept in cache; in which case, we might as well have split the grid into blocks of N/P rows and N columns.

However, the MPI-function that we will use to communicate between nodes, requires buffers to be consecutive strings of bytes. So if we were to split the grid into blocks of N rows and N/P columns, we would need to do this copying back and forth to the communications buffer, as elements constituting rows of the grid, are not stored consecutively in memory. If we split the grid the other way around, we may use the actual storage of the grid as communication-buffers.

3.8.1 MPI Class Constructor

The class implementing all of this, is called `LPoissonMPI`. As construction arguments, the class takes the number of MPI processing nodes P, the total grid-size N, and the rank of the current processing node (ranging from 0 to $P-1$). This class does *not* inherit from `LPoisson`, for the simple reason, that the book-keeping of finding out the size and offset of the sub-grid, would have to be clumsily written if done as a sub-class. Instead we have the following constructor for `LPoissonMPI`, which creates an `LPoisson`-object towards the end:

```
LPoissonMPI::LPoissonMPI (int N, int P, int rank) :
kP(P),
kRank(rank),
kUpNeighbour((rank == P-1) ? (MPI_PROC_NULL) : (rank+1)),
kDownNeighbour((rank == 0) ? (MPI_PROC_NULL) : (rank-1))
{
    assert(rank>=0 && rank<P);
    assert(P>=0 && P<=N);
```

```
    // Calculate the height of the sub-grid for each MPI node.
    std::div_t div_result = std::div(N, P);
    int height = div_result.quot;

    // Ensure the computational burden is evenly distributed.
    if (div_result.rem != 0)
    {
        height++;
    }

    // Grid is split into sub-grids of width N and height
    // approximately N/P. That is, the first P-1 nodes have
    // height std::ceil(N/P), and the last node has a sub-grid
    // of height N-(P-1)*std::ceil(N/P).
    int offsetM = rank*height;
    int sizeM = std::min(N-offsetM, height);
    int offsetN = 0;
    int sizeN = N;

    // Create a Poisson sub-grid of this size and offset.
    mPoisson = new LPoisson(N, sizeM, offsetM, sizeN, offsetN);
}
```

As you can see, care is put into ensuring that the size of the sub-grid, is such that the last node, does not have (in the worst case) a sub-grid with $P - 1$ more rows than the other nodes. Since the nodes synchronize through their communication once per iteration, this would skew the time-usage, as all the other nodes would have to wait for that last node.

Note the use of `MPI_PROC_NULL` as possible ranks of the processing node's neighbours. This special rank-number indicates to the communication functions of MPI, that communication should not actually take place, but the functions should merely report that the transfer was successful, however with zero data transferred. This makes it much easier to use MPI for the topology we have chosen, as we do not need any code to handle these special cases, arising only at the first and last processing nodes. (See the function `Communicate()` in the next section.)

Also note that even though `LPoissonMPI` does not inherit from `LPoisson`, the use of reverse inheritance is still possible, simply by making `LPoissonMPI` a template-class also, and then pass its template-argument on to `LPoisson`.

3.8.2 MPI Iteration & Communication

Performing a single iteration of the sub-grid of an instance of the `LPoissonMPI`, is simply done by the following function, which first uses the function from the `LPoisson`-class, and then calls another function in `LPoissonMPI` to perform the communication through MPI:

```
void Iterate ()
{
    assert(mPoisson);

    mPoisson->Iterate();
    Communicate();
}
```

The communication function is made as follows, which is straight-forward for this kind of distributed problem topology, where each node starts by sending data upwards, and receiving from below. Then, the node sends downwards and receives from above. The `MPI_Sendrecv()`-function handles any synchronization between the processing nodes, thus providing a transparent way of avoiding deadlocks etc. The `Communicate()`-function is as follows:

```
void Communicate ()
{
    assert(mPoisson);

    // Conveniece variables and constants from Poisson-object.
    double** u = mPoisson->mU;
    const int N = mPoisson->kN;
    const int offset = mPoisson->kOffsetM;
    const int height = mPoisson->kSizeM;

    // Status-information for MPI.
    MPI_Status status;

    // Send up, receive from below.
    MPI_Sendrecv( u[height]+1, N, MPI_DOUBLE, kUpNeighbour, 0,
                  u[0]+1, N, MPI_DOUBLE, kDownNeighbour, 0,
                  MPI_COMM_WORLD, &status);

    // Send down, receive from above.
    MPI_Sendrecv( u[1]+1, N, MPI_DOUBLE, kDownNeighbour, 0,
                  u[height+1]+1, N, MPI_DOUBLE, kUpNeighbour, 0,
                  MPI_COMM_WORLD, &status);
}
```

There is not much to note here, except perhaps that all pointers to buffers have the number 1 added to them – this is simply to avoid communicating the grid-points of the vertical boundaries, which do not change.

3.9 MPI Communication World

Two things should be observed in regards to MPI and its socalled Communication's World. First note that we simply use `MPI_COMM_WORLD` directly. Sec-

ondly, that `MPI_Init()` and `MPI_Finalize()` are called somewhere outside the `LPoissonMPI`-class.

In a more general setting, one would probably let the communication's world be provided to the `LPoissonMPI`-class as a parameter, so as to allow for different optimizations of the underlying hardware network (e.g. a ring-type network topology seems ideal for problems distributed in this way, or at least one should ensure that nodes with neighbouring ranks, also are *neighbours* in terms of communication speed and latency).

Moreover, one should consider where best to call `MPI_Init()` and `MPI_Finalize()`, as it is not quite graceful, to have the user of the `LPoissonMPI`-class call them manually; for example as done here, in the entry-point (the `main()`-function) of the program.

3.9.1 MPI Output

An important aspect of distributed computing, is of course to gather and output the results from all the processing nodes. There are (at least) two ways of doing this with MPI, one is by outputting to one or more files; which does not actually require any MPI calls, not even for synchronization. The other way is to use the MPI library to transfer data from all the processing nodes, to a single node that supports output – e.g. the node with rank zero. The former method is fairly easy, while the latter is more involved. In this implementation, it was chosen to let each node output to a file.

There are essentially two ways of outputting the results from the individual processing nodes to files. First, each node could output to a distinct file, e.g. by adding the node's rank to the desired filename. When there are many processing nodes, this means there would be a lot of files. Alternatively, we may append to a single file, thus assuming each node has access to that file, and that the file-system can interleave the different nodes' writings, so the file does not become garbled. This appending is the way it is done in this implementation, and both requirements appear to be satisfied when testing the MPI-distributed Poisson implementation on a single machine; as done in section 4.4.

This however, does not appear to work on the system with actual distribution (tested on the `hald` machine on DTU, as described in section 5), and no visual results are therefore shown for computations done on that system. To make this work in general, one could for example implement file-locking and a signal-handler for when the lock was freed. This was considered beyond the scope of this project.

3.9.2 Compiling & Running on UNIX

To compile and run on a UNIX system supporting the OpenMP and MPI extensions for its C++ compiler, one must do the following for compiling the OpenMP Poisson-code:[12]

[12] It is important the get the order of the source-files correct.

```
CC -o poisson -fast -xtarget=ultra3 -xarch=v8plusb \
   -xopenmp LPoisson.cpp poisson.cpp
```

And one may run the resulting program by typing the following at the command-prompt:

```
OMP_NUM_THREADS=2 ./poisson -n 30 -k 100
```

Which instructs the OpenMP-framework to use two threads.

Compiling the MPI-version (also including OpenMP), is done by typing the following:

```
mpCC -o poisson_MPI -fast -xtarget=ultra3 -xarch=v8plusb \
     -xopenmp -mt -lmpi LPoisson.cpp LPoissonMPI.cpp \
      poisson_MPI.cpp
```

without the line-breakings, as indicated by the characters at the end of the first two lines. Running the program can be done by typing the following:

```
OMP_NUM_THREADS=2 mprun -np 4 ./poisson_MPI -n 30 -k 100
```

Which is believed to set the environment variable `OMP_NUM_THREADS` to 2 on each of the four MPI nodes, so as to execute 4 MPI nodes, each running 2 OpenMP threads.

4 Testing

The implementation was tested for correctness during development, until no further errors were uncovered. Assertions in the source-code often played a crucial role in this testing, as assertions not only advise the programmer on the valid parameter ranges and execution states, but efficiently enforce these when the program is built and executed in debug-mode.

On page 26 and onwards, are displayed a number of figures, demonstrating convergence of the numerical solutions to the Poisson problem, as well as indicating correctness of the various algorithms.

4.1 Basic Gauss-Seidel Algorithm

First is figure 1, showing the initial solution to this particular 2D-Poisson problem, for a grid-size of $N = 30$ (that is, the grid has 32×32 points including boundaries). Figures 2, 3, 4, and 5, show the state of the numerical solution after 10, 100, 1000, and 10000 iterations, respectively. As can be seen, the solution approaches what is believed to be the actual solution, already after 100 iterations. Therefore, the experiments for the remainder of this project, are conducted with only 100 iterations.

4.2 Red-Black Algorithm

Executing 100 iterations of the Red-Black algorithm from section 2.3, also with grid-size $N = 30$, results in figure 6. Clearly, this resembles figure 3 for the basic Gauss-Seidel algorithm, and is therefore considered correct.

4.3 Block-Decomposed Algorithms

Testing for correctness of the block-decomposed variants of the basic as well as the Red-Black algorithms, is again done with grid-size $N = 30$ and 100 iterations for each algorithm. The results are shown in figure 7 for the basic algorithm with block-width 7, and figure 8 for the Red-Black algorithm with block-width 13 and block-height 5. Different block-sizes were tested during development, but the ones shown here, merely illustrate block-decompositions that are both divisible in N (as is the case for the Red-Black algorithm's block-height of 5), and that are not (as is the case with the block-widths 13 and 7). Again, the results resemble those in figure 3 for the basic Gauss-Seidel algorithm, and are therefore considered to be correct.

4.4 MPI, Single Machine Simulation

As for correctness of the distributed computation of the Poisson problem using MPI, the implementation was tested on a single *Microsoft Windows XP* machine, made to run 4 separate threads, simulating a network of 4 computational nodes.[13] The results of using the basic and Red-Black algorithms on these distributed nodes (and neither being block-decomposed), are shown in figures 9 and 10, respectively. Again, the results resemble those in figure 3 for the basic Gauss-Seidel algorithm, and the MPI implementation is therefore also considered to be logically correct, in terms of communicating the right data between the nodes – however, in a real-world scenario, testing of the MPI implementation, should also be conducted on a real physical network of computers, which is omitted here though, as previously discussed in section 3.9.1.

4.5 OpenMP, Multi-Processor Machine

Testing of the OpenMP implementation, was done on the machine described in section 5. Figures 11 through 13 display the results after 100 iterations of the basic Gauss-Seidel algorithm, on a grid with size $N = 30$, and having 1, 2, and 4 OpenMP threads, respectively. As can be seen, the results are as expected, and the OpenMP implementation for this algorithm, is therefore considered to be correct and in working order.

The last test with OpenMP, is for the Red-Black variant of the Gauss-Seidel algorithm, and the result is shown in figure 14. Again, the result is as expected, and the implementation is hence deemed correct.

[13] Note that the grid-size $N = 30$ is not divisible by the number of processing nodes $P = 4$. During development, tests were also conducted with e.g. $P = 2$ and $P = 3$, which both divide $N = 30$.

We will not conduct detailed experiments with the block-decomposed algorithms, and therefore omit their test-results here.

5 Experimental Results

To uncover the relationship between time-usage and number of processors in the distributed computing of the 2D-Poisson problem, a number of experiments are conducted. All experiments are conducted on the machine name `hald`, located at DTU. The details of the machine are unimportant, all we need to know is the following: `hald` has 48 processors, and it has enough physical memory to avoid swapping to disk, with the grid-sizes used here (the largest grid consumes approximately 32 MB). Moreover, the machine is used by many students simultaneously, which should be taken into account when assessing the results.

The experiments focus on performance of the basic Gauss-Seidel algorithm, its Red-Black variant, and distributing the computation with OpenMP and MPI. That is, we will not conduct extensive experiments on the performance of the block-decomposed algorithms. All experiments perform 100 iterations of the entire grid, and we merely vary the grid-size and number of processors; again, we are not concerned with the quality of the results, as long as they appear to be correct, and hence do not alter the number of iterations performed.

5.1 OpenMP

Table 1 shows the time-usage in seconds, from running the basic algorithm from section 2.2.2 (and as implemented in section 3.4). As can be seen, the performance starts to degenerate when the number of OpenMP threads reaches a certain point, depending on the grid-size N. Generally however, the optimal performance seems to be with somewhere around 16 OpenMP threads.

5.1.1 Parallel Scaling

In regards to scaling of the basic Gauss-Seidel algorithm (up until approximately 16 OpenMP threads), we see that the performance is not fully linear in the number of processors involved: Sometimes the time-usage is less than expected, but mostly it is worse than expected. Take for example the case with $N = 2000$ and one and two OpenMP threads. Here we would expect a socalled *embarrasingly* parallel algorithm[14] to spend only 42.45 seconds with two threads, as opposed to the actual 45.91. However, for four threads, the time-usage is roughly the expected 21.25 seconds.

[14] One that has no serialization of the executed code, and which the Gauss-Seidel algorithm comes close to being, with the only exception being its most outer-loop that calls `Iterate()` successively.

5.1.2 Reliability of Time-Results

Still, an important thing to note, is that these figures are not entirely reliable, as the time-usage varies from run to run. For example, it was observed for the measure where there were 19 threads and $N = 100$, that the time-usage varied between 0.94 to 3.07 seconds! This is probably due to the `hald` machine being shared between students, as mentioned. To give a more accurate estimate of time-usage under these conditions, one should have run a series of experiments, say 50, and taken the average time-usage (and possibly the standard deviation as well).

5.1.3 Red-Black Performance

Table 2 shows the results of running the same experiments as in table 1, only with the Red-Black version of the Gauss-Seidel algorithm. Two things should be noted: The performance is always worse than that of the basic Gauss-Seidel algorithm (up to 100% for grid-size $N = 2000$), but otherwise follows the same pattern, where the performance starts to decrease badly, with about 16 OpenMP threads.

5.1.4 When To Use OpenMP

It is not that evident from these results, that one might also need to add conditional expressions to the OpenMP pragmas in the source-code, so as to control when to use OpenMP and when not to. For example, when the grid-size N is very small, say 30, then the overhead of using OpenMP may be too great, causing the overall performance to decrease, as opposed to having just a single thread as normal.

However, it is clear that one should limit the number of OpenMP threads to about 16. The reason is not clear though, and particularly for the larger grids ($N = 1000$ and 2000), one has to wonder why performance starts to degenerate with more than (roughly) 16 threads: There does not appear to be any such limitation in the parallel algorithm – provided OpenMP merely splits the `for`-loops equally amongst the processors. The reason is maybe found in the machine `hald`, and limitations on how many processors a single user may use simultaneously – but, that is just a guess. The same problem was found with the MPI experiments below, so maybe it is not that far fetched.

5.1.5 Block-Decomposed

Using four OpenMP threads, a grid-size of $N = 2000$, and the block-width set to 64 and block-height to 32, the normal Gauss-Seidel algorithm spent 23.87 seconds, and the Red-Black variant spent 30.06 seconds. The result for the basic algorithm was worse when using horizontal block-decomposition with this block-width of 64, however, the result for the Red-Black variant was better than without the use of block-decomposition, and nearly 25% better. We will not investigate this further though, as the Red-Black performance is still not

expected to improve beyond the basic Gauss-Seidel algorithm – which was the entire point of making the Red-Black variant in the first place.

5.2 MPI

On the machine `hald`, distribution with MPI is only simulated, which is done by starting a thread for each desired MPI processing node, and if available, then allocating a separate processor for each of these threads. All MPI nodes execute the basic Gauss-Seidel (non-block-decomposed) algorithm, as it was found to be the fastest in the previous experiments.

Table 3 shows the results of running various numbers of MPI nodes, with each node having a single OpenMP thread. Of course, on `hald` there is no physical difference between OpenMP and MPI threads, as they are all executed on the same machine, but there still remains a difference between the way those threads communicate and synchronize. The results clearly show that performance improves with increasing numbers of MPI nodes, however, as with the OpenMP experiments above, the time-usage does not always decrease linearly as one could perhaps expect, although we are usually quite close to such linear behaviour.

When increasing the number of OpenMP threads running on each MPI node, to four threads instead of just one, we get the results in table 4, which roughly exhibit the same tendency as before. A few things should be noted however. Take for example the results of $N = 2000$ and a single MPI node. This node has 4 OpenMP threads, totalling 4 threads. Yet executing four MPI nodes each with a single OpenMP thread, was found in table 3 to take less time. The same result goes for 2 MPI nodes each with 4 OpenMP threads, as opposed to 8 MPI nodes with 1 OpenMP thread each. Of course, one should not rely blindly on these results, as other users of the `hald` machine may have interfered with the performance measures, yet it is an interesting point, that MPI might be faster than OpenMP, even though MPI also spends time explicitly communicating data between its nodes.

Another and even stronger point shown in table 4, is that performance once again badly degenerates, once the total number of threads reaches about 16. This only adds to the suspicion, that limitations of either the machine or the thread-system are to blame, and not the algorithm, which is believed to be about as parallel as possible, and should therefore scale well.

6 Conclusion

After a brief introduction to the 2-dimensional Poisson problem, the Gauss-Seidel algorithm for solving it numerically was described, along with a socalled Red-Black variant, and block-decomposed versions of both of these.

An object-oriented implementation was presented, which included the distributed computing of these algorithms, using the commonly available OpenMP and MPI interfaces. Section 3.3 was included, documenting an object-oriented

trick, which allowed object-oriented specialization of a class' functions, without losing performance due to the overhead usually involved in this.

Finally, testing for correctness was conducted along with time-usage experiments. The latter showed that both kinds of parallelism worked well, however not quite *embarrasingly* well. In particular, it was found when having more than roughly 16 threads, the performance of the various algorithms started to degrade significantly. The reason for this is still not entirely clear, but as the algorithms themselves appear to be able to scale well beyond this, it was speculated that this deficiency was caused by (possibly) artificial limitations in the given machine, which has 48 processors in total, however shared among many users.

References

[1] Magnus Erik Hvass Pedersen, Mandatory Assignment 1, Scientific Computing Course, Danish Technical University, September 2005

[2] OpenMP Specification version 2.5, May 2005, `http://www.compunity.org/`

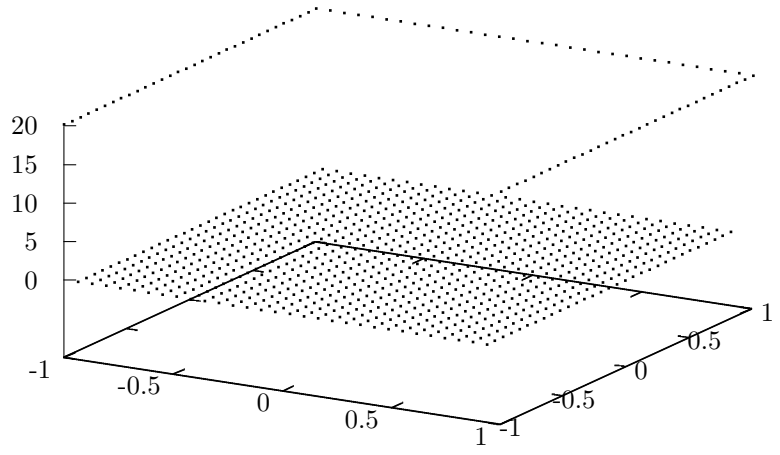

Figure 1: The initial 2D-Poisson Heat-Radiator problem from section 2.1.

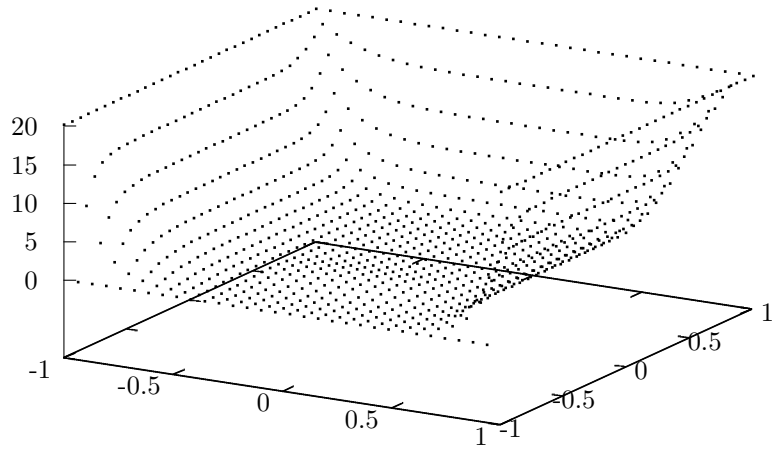

Figure 2: The Poisson solution after 10 iterations of the basic Gauss-Seidel algorithm from section 2.2.2.

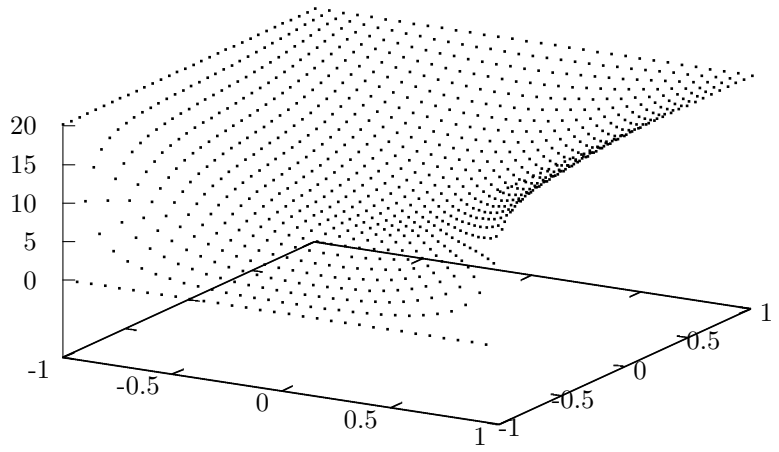

Figure 3: The Poisson solution after 100 iterations of the basic Gauss-Seidel algorithm from section 2.2.2.

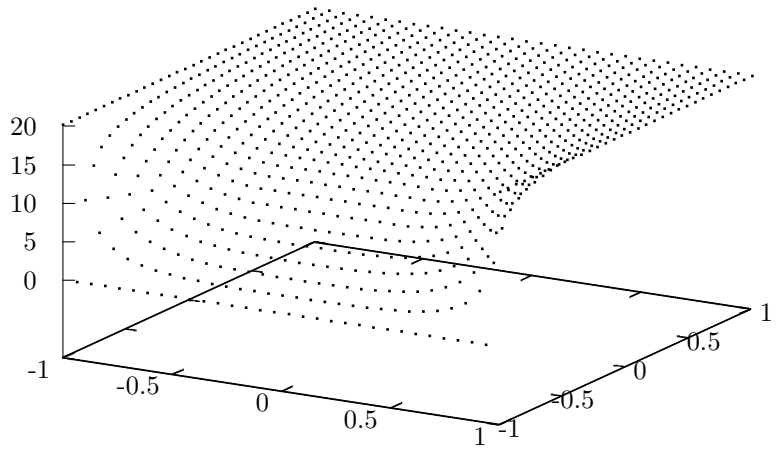

Figure 4: The Poisson solution after 1000 iterations of the basic Gauss-Seidel algorithm from section 2.2.2.

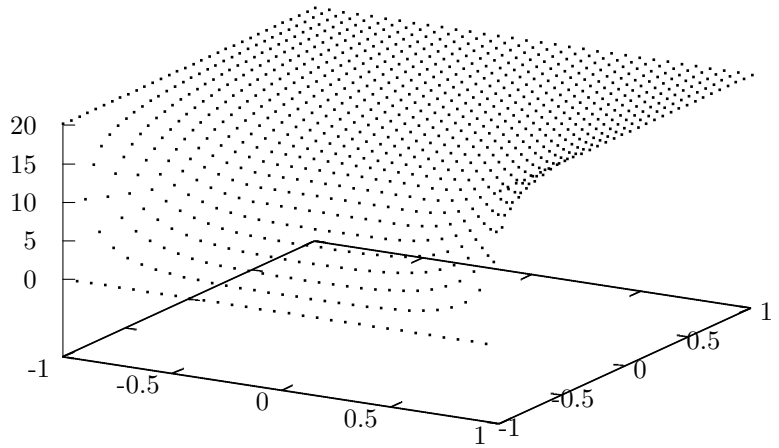

Figure 5: The Poisson solution after 10000 iterations of the basic Gauss-Seidel algorithm from section 2.2.2.

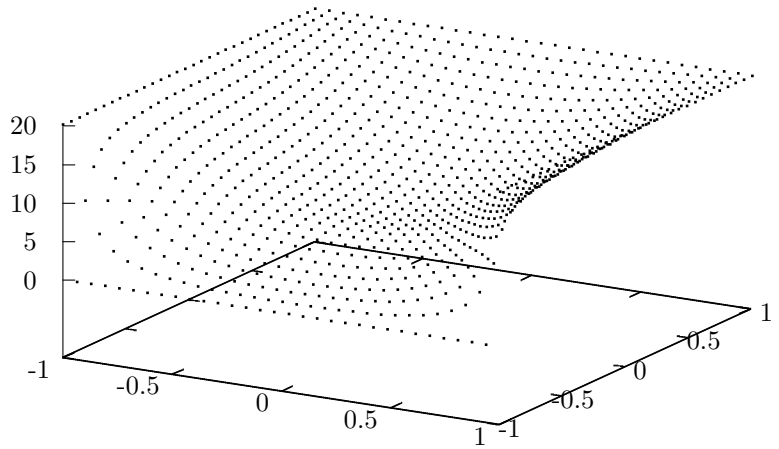

Figure 6: The Poisson solution after 100 iterations of the Red-Black algorithm from section 2.3.

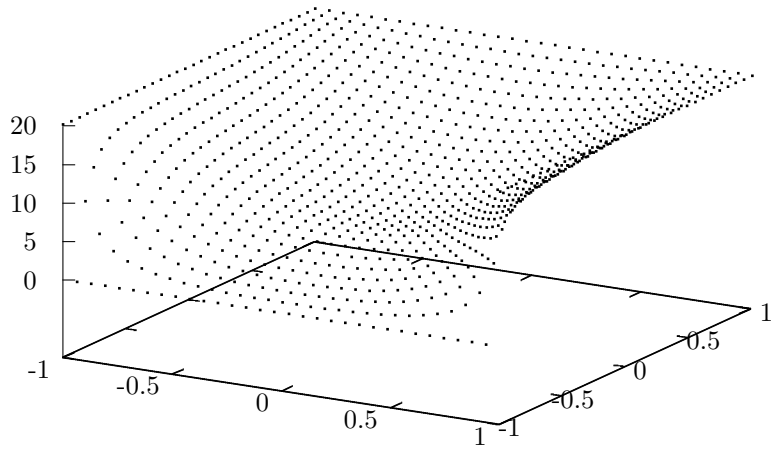

Figure 7: The Poisson solution after 100 iterations of the block-decomposed Gauss-Seidel algorithm, having a block-width of 7.

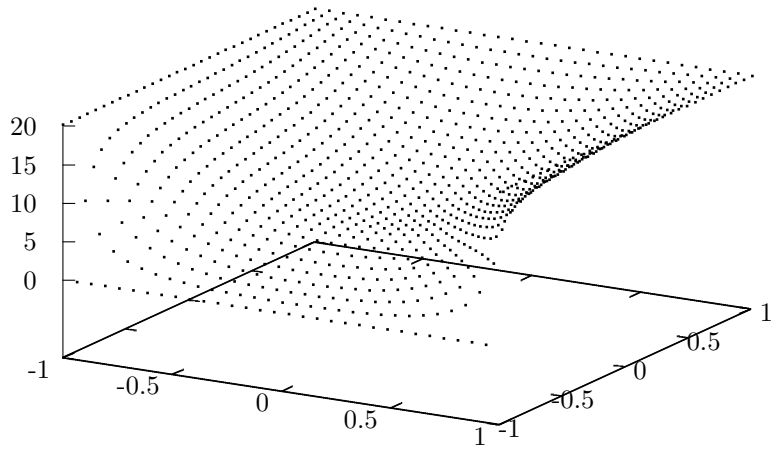

Figure 8: The Poisson solution after 100 iterations of the block-decomposed Red-Black algorithm, having block-width 13 and block-height 5.

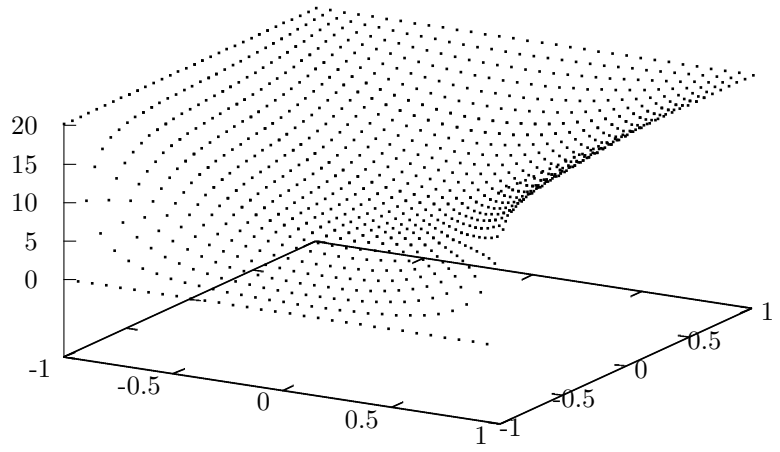

Figure 9: The Poisson solution after 100 iterations of the basic Gauss-Seidel algorithm from section 2.2.2, distributed to 4 nodes using MPI (all nodes running as separate threads on the same machine).

Threads	$N = 100$	$N = 1000$	$N = 2000$
1	0.1226	16.747	84.99
2	0.0765	9.9211	45.91
4	0.0549	5.7641	21.82
8	0.0481	3.3052	13.48
16	0.1971	2.4761	8.093
17	0.2341	2.5463	9.283
20	2.3427	4.2040	9.163
24	7.3426	6.7689	12.43
32	16.790	12.099	16.53
48	33.179	22.639	27.52

Table 1: Time usage in seconds, for performing 100 iterations of the non-block-decomposed OpenMP version of the Gauss-Seidel algorithm, with the given grid-sizes N.

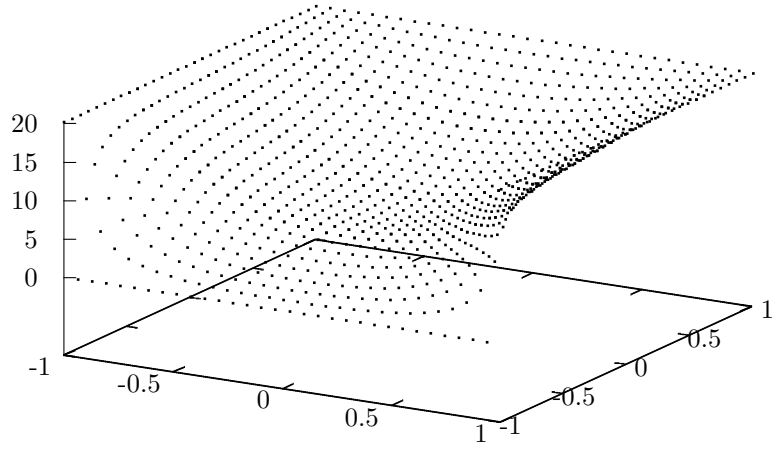

Figure 10: The Poisson solution after 100 iterations of the Red-Black algorithm from section 2.3, distributed to 4 nodes using MPI (all nodes running as separate threads on the same machine).

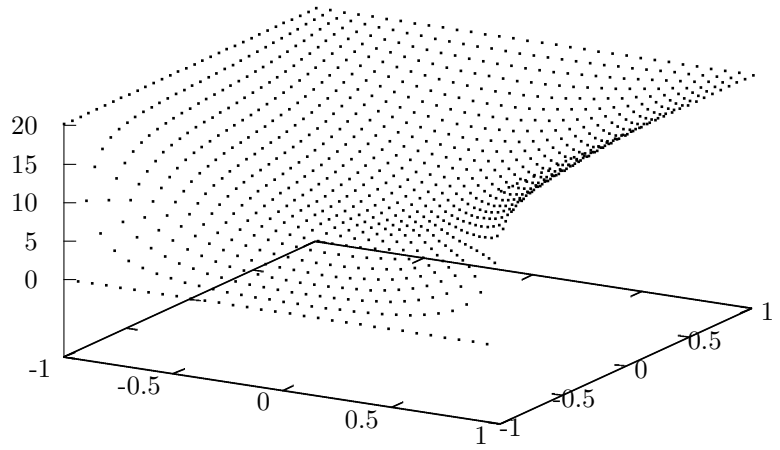

Figure 11: The Poisson solution after 100 iterations of the basic Gauss-Seidel algorithm from section 2.2.2, distributed with OpenMP running a single thread.

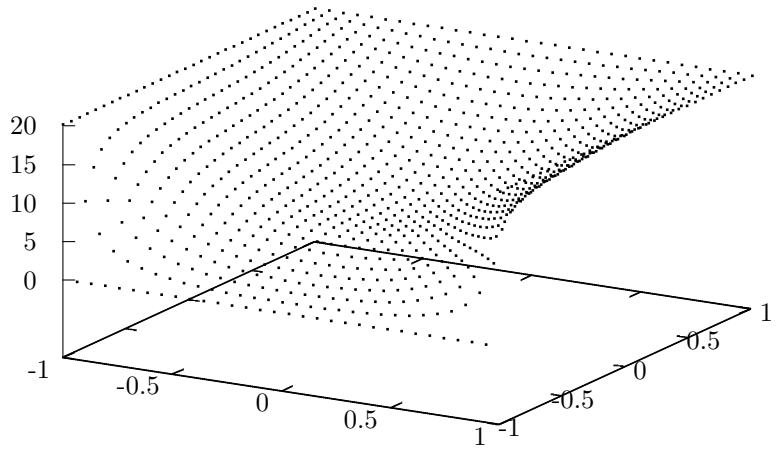

Figure 12: The Poisson solution after 100 iterations of the basic Gauss-Seidel algorithm from section 2.2.2, distributed with OpenMP running two threads.

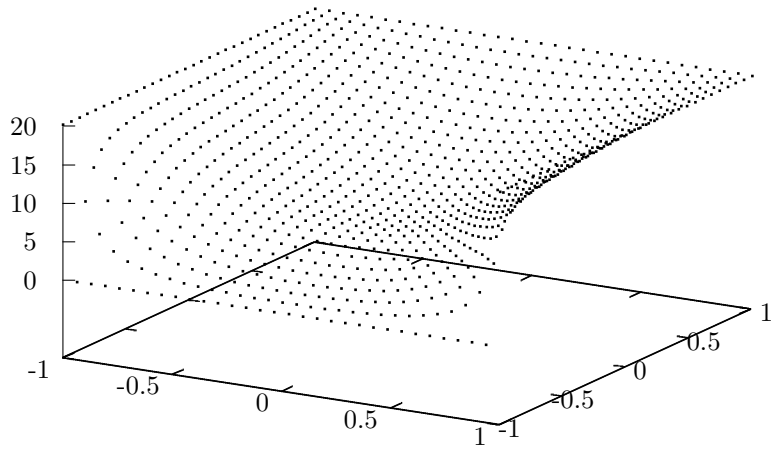

Figure 13: The Poisson solution after 100 iterations of the basic Gauss-Seidel algorithm from section 2.2.2, distributed with OpenMP running four threads.

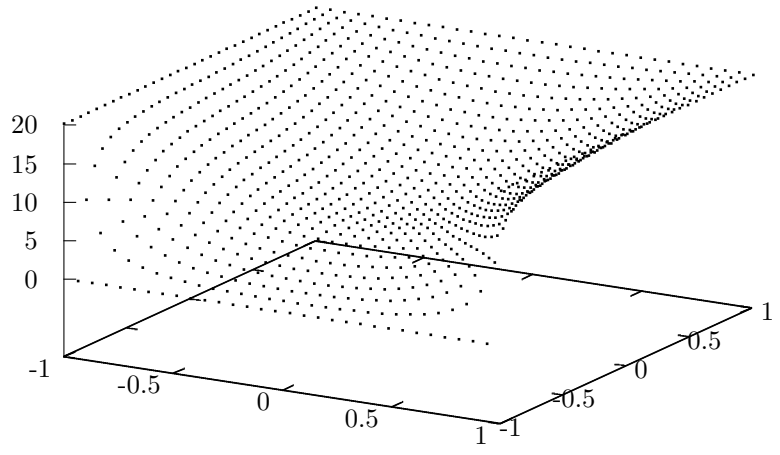

Figure 14: The Poisson solution after 100 iterations of the Red-Black algorithm from section 2.2.2, distributed with OpenMP running four threads.

Threads	$N = 100$	$N = 1000$	$N = 2000$
1	0.1760	20.143	114.6
2	0.1295	17.920	75.95
4	0.1241	10.571	39.27
8	0.1383	5.8372	23.12
16	1.4648	5.4125	16.53
17	1.9080	6.0596	16.36
20	2.6880	7.1797	17.31
24	8.2896	9.6306	19.68
32	17.060	20.489	23.87
48	28.008	32.979	34.96

Table 2: Time usage in seconds, for performing 100 iterations of the non-block-decomposed OpenMP version of the Red-Black algorithm, with the given grid-sizes N.

Nodes	$N = 100$	$N = 1000$	$N = 2000$
1	0.1225	16.304	65.99
2	0.0660	9.0178	42.11
4	0.0377	4.9501	21.58
8	0.0318	2.4811	10.07

Table 3: Time usage in seconds, for performing 100 iterations of the non-block-decomposed OpenMP version of the Gauss-Seidel algorithm (here running only a single OpenMP thread on each MPI node), with the given grid-sizes N, and distributed to the given number of MPI nodes.

Nodes	$N = 100$	$N = 1000$	$N = 2000$
1	0.0512	7.2231	25.51
2	0.0420	3.1491	14.51
4	0.2562	2.6423	9.985
8	8.0832	9.6286	11.51

Table 4: Time usage in seconds, for performing 100 iterations of the non-block-decomposed OpenMP version of the Gauss-Seidel algorithm (here running four OpenMP threads on each MPI node), with the given grid-sizes N, and distributed to the given number of MPI nodes.