TMA4280: Introduction to Supercomputing

Numerical solution of the Poisson problem using finite differences, the conjugate gradient method, and parallel processing

Spring 2011

©Einar M. Rønquist
Department of Mathematical Sciences
NTNU, N-7491 Trondheim, Norway

1 Introduction

We consider here the Poisson problem

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega, && (1)\\ u &= 0 \quad \text{on } \partial\Omega, && (2) \end{aligned}$$

where Ω is the two-dimensional square domain depicted in Figure 1. We will solve this Poisson problem numerically based on a central difference approximation of the Laplace operator. This will result in the five-point finite difference stencil indicated in Figure 1. The discretization is based on a uniform structured grid with $n + 1$ points in each spatial direction. Due to the prescribed boundary conditions, the total number of unknowns is thus $N = (n - 1)^2$.

The discretization of the Poisson problem results in a linear system of algebraic equations,

$$\underline{\underline{A}}\,\underline{u} = \underline{f}. \qquad (3)$$

We will now discuss the solution of this system using the conjugate gradient method, both in a single-processor and multi-processor context. In contrast to the direct solution methods we have considered earlier, the conjugate gradient method is an *iterative* method.

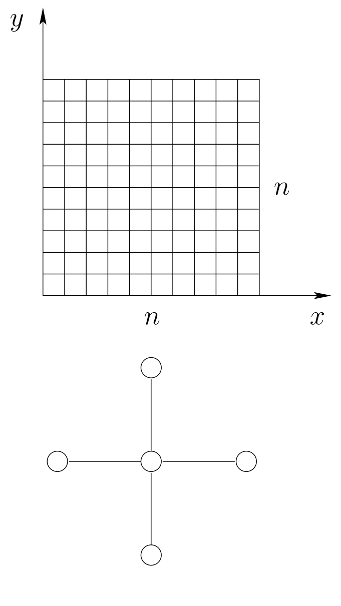


Figure 1: Computational domain and discretization using finite differences.

2 Discretization

The grid points are denoted as (x_i, y_j), $0 \leq i, j \leq n$. We will use a uniform grid with grid spacing $h = L/n$ in both the x- and y-direction; here, $\Omega = (0, L) \times (0, L)$.

The finite difference approximation of $u(x_i, y_j)$ is denoted as $u_{i,j}$. Using a five-point central difference formula for the approximation of the Laplacian, the system of algebraic equations can be expressed as

$$-\frac{(u_{i+1,j} - 2u_{i,j} + u_{i-1,j})}{h^2} - \frac{(u_{i,j+1} - 2u_{i,j} + u_{i,j-1})}{h^2} = f_{i,j} \quad 1 \leq i, j \leq n-1. \tag{4}$$

Numerical analysis of this discretization scheme shows that we can expect the discretization error to scale as $\mathcal{O}(h^2)$, i.e.,

$$|u(x_i, y_j) - u_{i,j}| \sim \mathcal{O}(h^2), \quad 1 \leq i, j \leq n-1. \tag{5}$$

3 System of algebraic equations

It is possible to explicitly construct the global system of algebraic equation as

$$\underline{A}\,\underline{u} = \underline{f}. \tag{6}$$

Here, $\underline{A}$ represents the discrete Laplace operator, $\underline{u}$ is a vector representing the unknowns, and $\underline{f}$ is a known right hand side vector. Note that this representation necessitates the use of a *global* numbering scheme, e.g., a natural ordering of the unknowns; see Section 5 for more details. With such a numbering scheme, $\underline{A}$ can be classified as a band matrix with five non-zero matrix elements in each row (except for boundary effects).

We recall some key properties of the matrix $\underline{A}$:

- $\dim(\underline{A}) = N = (n-1)^2 \sim \mathcal{O}(n^2)$
- $\underline{A}$ is symmetric ($\underline{A} = \underline{A}^T$)
- $\underline{A}$ is positive definite ($\underline{v}^T \underline{A}\,\underline{v} > 0$ for all $\underline{v} \in \mathbb{R}^N$, $\underline{v} \neq \underline{0}$)
- $\underline{A}$ is non-singular, i.e., invertible ($\det(\underline{A}) \neq 0$)

4 The conjugate gradient method

We will here solve the system (6) using an iterative method. In particular, we will use the conjugate gradient method since $\underline{A}$ is symmetric and positive definite (abbreviated as SPD); these properties are prerequisites for using this method.

The conjugate gradient method represents an example of a modern iterative solution method, despite the fact that the method was first proposed by Hestenes and Stiefel about half a century ago. One reason for this is the fact that the method initially was considered to be a direct method; see comments at the end of this section. Another reason is the much more recent development of sophisticated and powerful preconditioning techniques, which are used to speed up the convergence rate of the conjugate gradient iteration; however, we will not be able to go into the details of such techniques in this course.

In this section, we will first briefly discuss the mathematical framework behind the conjugate gradient method. However, our main emphasis here will be on implementation issues, computational complexity, as well as parallel processing.

We consider a system of the type (6) where $\underline{A}$ is symmetric and positive definite. Furthermore, let $\underline{u} \in \mathbb{R}^N$ and $\underline{A} \in \mathbb{R}^{N\times N}$.

Consider the quadratic functional $J : \mathbb{R}^N \to \mathbb{R}$ defined as

$$J(\underline{w}) = \frac{1}{2}\underline{w}^T \underline{A}\,\underline{w} - \underline{w}^T \underline{f}, \qquad \underline{w} \in \mathbb{R}^N. \tag{7}$$

We claim that solving the system $\underline{A}\,\underline{u} = \underline{f}$ is equivalent to minimizing $J(\underline{w})$ over all $\underline{w} \in \mathbb{R}^N$; see Figure 2. That is,

$$\boxed{\underline{A}\,\underline{u} = \underline{f} \qquad \Longleftrightarrow \qquad \underline{u} = \arg\min_{\underline{w}\in\mathbb{R}^N} J(\underline{w})} \tag{8}$$

To see this, consider $\underline{w} = \underline{u} + \underline{v}$ for any $\underline{v} \in \mathbb{R}^N$. Then

$$\begin{aligned} J(\underline{u}+\underline{v}) &= \frac{1}{2}(\underline{u}+\underline{v})^T \underline{A}(\underline{u}+\underline{v}) - (\underline{u}+\underline{v})^T \underline{f} && (9)\\ &= \underbrace{(\frac{1}{2}\underline{u}^T \underline{A}\,\underline{u} - \underline{u}^T \underline{f})}_{J(\underline{u})} + \underbrace{\underline{v}^T(\underline{A}\,\underline{u} - \underline{f})}_{=0} + \underbrace{\frac{1}{2}\underline{v}^T \underline{A}\,\underline{v}}_{>0 \quad \forall \underline{v}\neq 0} && (10)\end{aligned}$$

Thus,

$$J(\underline{u}+\underline{v}) = J(\underline{u}) + \frac{1}{2}\underline{v}^T \underline{A}\,\underline{v} \quad > J(\underline{u}) \qquad \forall \underline{v} \neq 0. \tag{11}$$

Since $\underline{A}$ is symmetric and positive definite, we can diagonalize $\underline{\mathrm{A}}$ as

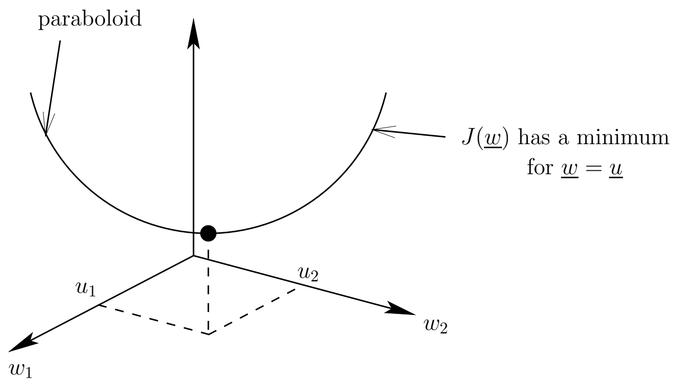
$$\underline{A}\,\underline{Q} = \underline{Q}\,\underline{\Lambda} \tag{12}$$

or

$$\underline{A}\underline{q}_i = \lambda_i \underline{q}_i \qquad i = 1, \ldots, N, \tag{13}$$

where $\lambda_i > 0$, $1 \le i \le N$ (positive eigenvalues) and $\underline{q}_i^T \underline{q}_j = \delta_{ij}$, $1 \le i, j \le N$ (orthonormal eigenvectors), and where the eigenvector matrix

$$\underline{Q} = [\underline{q}_1, \ldots, \underline{q}_N]. \tag{14}$$



paraboloid
$J(\underline{w})$ has a minimum for $\underline{w} = \underline{u}$
u_1
u_2
w_1
w_2

Figure 2: The quadratic functional $J(\underline{w})$ when $N = 2$.

Hence,

$$\underline{Q}^T \underline{A}\, \underline{Q} = \underline{\Lambda}, \tag{15}$$

where $\underline{\Lambda}$ is a diagonal matrix with the positive eigenvalues along the diagonal.

Now, for any vector $\underline{v} \in \mathbb{R}^N$, we can write

$$\underline{v} = \underline{Q}\underline{z}. \tag{16}$$

This is just a change of coordinate system, in fact, a pure rotation since $\underline{Q}$ is orthonormal; see Figure 3. Combining (11) and (16), we can write

$$\begin{aligned} J(\underline{u} + \underline{v}) &= J(\underline{u}) + \frac{1}{2}\underline{z}^T \underbrace{\underline{Q}^T \underline{A}\, \underline{Q}}_{=\underline{\Lambda}}\, \underline{z}, \\ &= J(\underline{u}) + \frac{1}{2}\underline{z}^T \underline{\Lambda}\, \underline{z}, \\ &= J(\underline{u}) + \frac{1}{2}\sum_{i=1}^{N} \lambda_i z_i^2. \end{aligned}$$

This means that $J(\underline{w}) =$ constant is equivalent to

$$\sum_{i=1}^{N} \lambda_i z_i^2 = \text{constant} \tag{17}$$

or

$$\sum_{i=1}^{N} \left(\frac{z_i}{1/\sqrt{\lambda_i}} \right)^2 = \text{constant}. \tag{18}$$

For example, with $N = 2$ this is the same as saying that

$$\left(\frac{z_1}{1/\sqrt{\lambda_1}}\right)^2 + \left(\frac{z_2}{1/\sqrt{\lambda_2}}\right)^2 = \mathrm{C}, \tag{19}$$

where C is a constant; see Figure 3. In general, for $N > 2$, (18) represents a hyperellipsoid; see Figure 4.

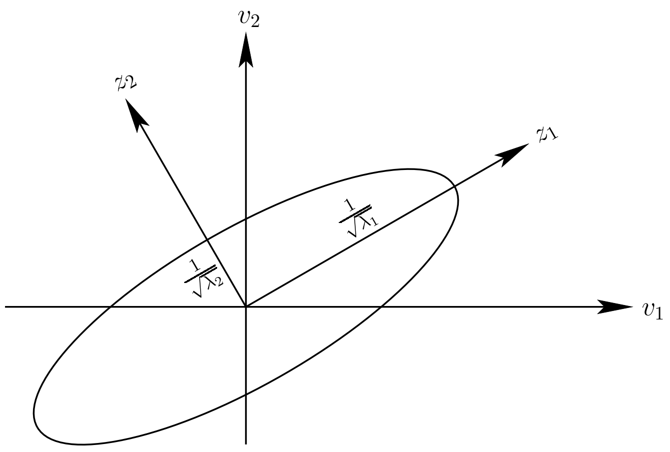


Figure 3: The linear transformation $\underline{v} = \underline{\underline{Q}}\,\underline{z}$ when $N = 2$: a pure rotation. The curve represents an ellipse given by (19) with $C = 1$.

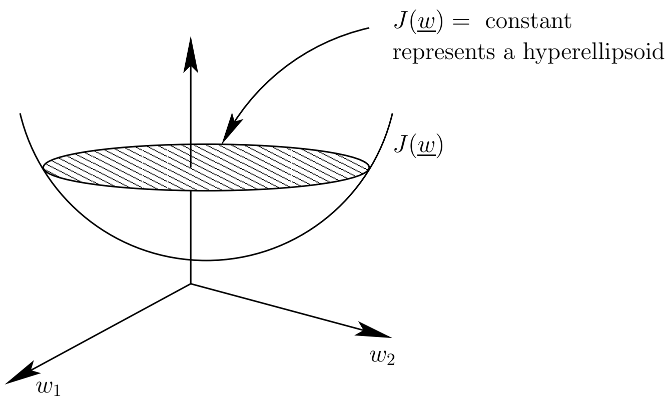


Figure 4: In general, $J(\underline{w}) =$ constant represents a hyperellipsoid; see (18).

Consider again the system $\underline{A}\,\underline{u} = \underline{f}$. Assume that we have an approximate solution $\underline{\tilde{u}}$ to $\underline{u}$. Then, the *residual*, $\underline{r}$, is defined as the difference between the right hand side, $\underline{f}$, and the approximate left hand side, $\underline{A}\,\underline{\tilde{u}}$, i.e.,

$$\underline{r} = \underline{f} - \underline{A}\,\underline{\tilde{u}}. \tag{20}$$

A residual equal to zero implies that $\underline{\tilde{u}} = \underline{u}$.

Assume that our initial guess is $\underline{\tilde{u}} = \underline{0}$ (e.g., we do not know anything about the solution). In this case, the initial residual is

$$\underline{r}_0 = \underline{f}. \tag{21}$$

Now, consider the vectors generated by performing the matrix-vector products

$$\underline{A}\,\underline{r}_0, \underline{A}^2\,\underline{r}_0, \ldots, \underline{A}^{m-1}\,\underline{r}_0. \tag{22}$$

If $\underline{A}$ is SPD, the vectors $\{\underline{r}_0, \underline{A}\,\underline{r}_0, \ldots, \underline{A}^{m-1}\,\underline{r}_0\}$ are linearly independent, and form a basis of a vector space of dimension m. The space spanned by these vectors are called a Krylov space and is denoted as $\mathcal{K}^{(m)}$, i.e.,

$$\mathcal{K}^{(m)} = \text{span}\{\underline{r}_0, \underline{A}\,\underline{r}_0, \ldots, \underline{A}^{m-1}\,\underline{r}_0\}.$$

We have earlier shown that

$$\underline{A}\,\underline{u} = \underline{f} \quad \Longleftrightarrow \quad \underline{u} = \arg \min_{\underline{w} \in \mathbb{R}^N} J(\underline{w}).$$

However, if we minimize $J(\underline{w})$, not over the entire space $\mathbb{R}^N$, but over the space $\mathcal{K}^m \subset \mathbb{R}^N$ (with $m < N$), the minimization will not, in general, give us the exact solution $\underline{u}$, but an approximate solution $\underline{u}^{(m)}$. This is the essence of the CG method, and the approximate solution $\underline{u}^{(m)}$ corresponds to the m'th iterate. In addition,

$$\underline{u}^{(m)} \to \underline{u} \text{ as } m \to N. \tag{23}$$

Often

$$\underline{u}^{(m)} \simeq \underline{u} \text{ for } m \ll N. \tag{24}$$

The convergence rate of the CG method depends upon the *condition number* of $\underline{A}$,

$$\kappa(\underline{A}) = \frac{\lambda_{\max}(\underline{A})}{\lambda_{\min}(\underline{A})}.$$

The number of iterations, $\mathcal{N}_{\text{iter}}$, scales as the square root of the condition number, i.e.,

$$\mathcal{N}_{\text{iter}} \sim \mathcal{O}\Big(\sqrt{\frac{\lambda_{\max}(\underline{A})}{\lambda_{\min}(\underline{A})}}\Big). \tag{25}$$

We now summarize the conjugate gradient iteration for the solution of a system of algebraic equations, $\underline{A}\,\underline{u} = \underline{f}$, where $\underline{A}$ is symmetric and positive definite:

The conjugate gradient algorithm:

```
Set $\underline{u}^0 = \underline{0}$,   $\underline{r}^0 = \underline{f}$
For $m = 1, 2, \ldots$
    $\beta_m = \dfrac{(\underline{r}^{m-1})^T \underline{r}^{m-1}}{(\underline{r}^{m-2})^T \underline{r}^{m-2}}$        $(\beta_1 \equiv 0)$
    $\underline{p}^m = \underline{r}^{m-1} + \beta_m \underline{p}^{m-1}$        $(\underline{p}^1 \equiv \underline{r}^0)$
    $\alpha_m = \dfrac{(\underline{r}^{m-1})^T \underline{r}^{m-1}}{(\underline{p}^m)^T \underline{A}\underline{p}^m}$
    $\underline{u}^m = \underline{u}^{m-1} + \alpha_m \underline{p}^m$
    $\underline{r}^m = \underline{r}^{m-1} - \alpha_m \underline{A}\underline{p}^m$
end
```

Basic operation	**Expression**	per iteration	$\mathcal{N}_{op}^{basic}$
Matrix-vector product	$\underline{y} = \underline{A}\,\underline{x}$	1	depends on $\underline{A}$
Innerproduct	$\sigma = \underline{x}^T \underline{y}$	2	$\simeq 2N$
Scalar-vector multiplication	$\underline{y} = \sigma \underline{x}$	3	N
Addition of two vectors	$\underline{z} = \underline{x} + \underline{y}$	3	N

Table 1: Basic operations in the conjugate gradient algorithm. Here, N is the vector length and $\mathcal{N}_{op}^{basic}$ is the number of floating point operations per single basic operation.

In Table 1, we list the basic operations at each conjugate gradient iteration. In addition to these basic operations, we also need to check for convergence. Since the length of the residual vector is readily available, we can easily check if $(\underline{r}^T \underline{r})^{1/2} < \varepsilon$, where ε is a tolerance (or a stopping criterion) specified by the user.

The memory requirement depends on how we represent $\underline{A}$. If we never store the matrix explicitly, the memory requirement, $\mathcal{M}$, typically scales with the problem size, N.

Note that, in exact arithmetic, the conjugate gradient iteration gives the exact solution after N iterations. In fact, this is why this method initially only was considered as a direct method and not as an iterative method.

5 Implementation issues

In the following, we turn our attention to the implementation of the conjugate gradient method for the solution of (6). In particular, we will consider a parallel implementation of the method.

5.1 Choice of data structure

Figure 5 depicts the data structure used to represent all the field variables (e.g., the solution u, the given right hand side f, etc.). Here, the first variable, i, defines the x-coordinate, while the second variable, j, defines the y-coordinate. All the field variables are represented as two-dimensional arrays in our program. However, this visual interpretation will be rotated since the first index represents the row and the second represents the column of our two-dimensional arrays. The data representation we have chosen here is referred to as a *local* data representation: we need two indices (i and j) to define a single value of a scalar field variable.

If the matrix $\underline{A}$ in our program was explicitly formed, we would have to use a global vector representation for our field variables. One possibility is shown in Figure 6. Here, we number the columns before the rows, which means that, using the programming language C, the elements in the vector $\underline{u}$ will be stored contiguously in memory. Again, we emphasize that we will *never* explicitly contruct the matrix $\underline{A}$.

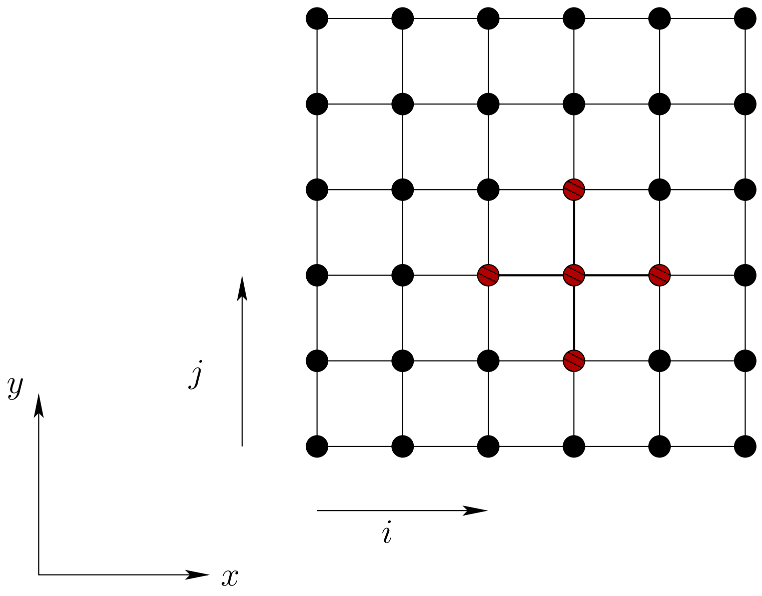


Figure 5: Local data representation of a field variable. The first index, i, runs along the x-axis and the second index, j, along the y-axis.

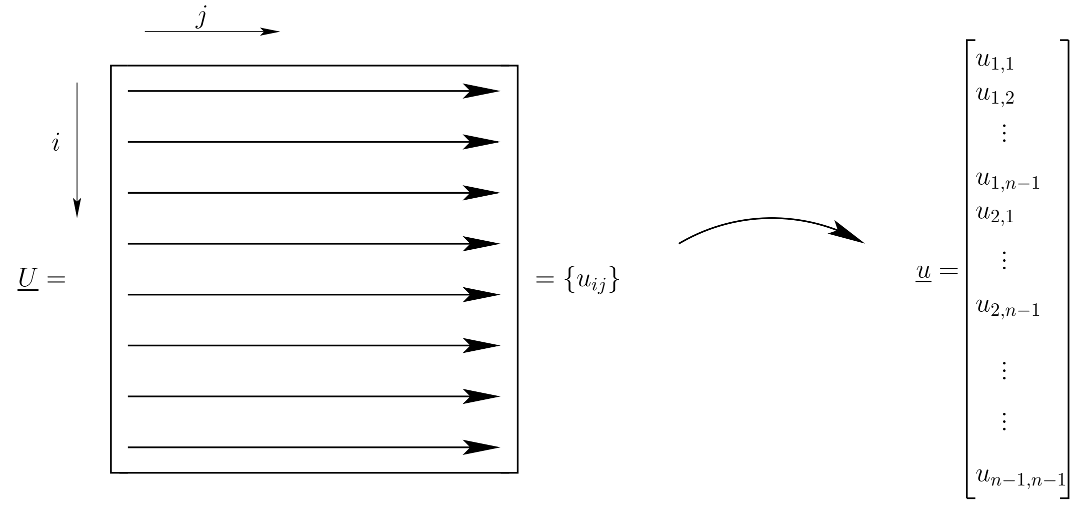


Figure 6: One possible choice of mapping between a local data representation (a two-dimensional array) and a global data representation (a vector). The index i indicates the row, and the index j the column.

5.2 Parallel processing

When we implement numerical algorithms on multiple processors, we need to choose a partitioning of our problem into smaller subproblems. A natural way to do this for our problem is to partition the domain Ω into smaller subdomains. We can do this in several ways. One alternative is to decompose the domain Ω along one of the coordinate axes. Another alternative is to decompose the domain along both coordinate axes. Here, we will choose the former alternative. We split the domain along the x-axis into P subdomains, and associate a single subdomain with each processor $p = 0, \ldots, P-1$. Within each subdomain we use a local data representation as explained above.

Figure 7 shows two such subdomains and their associated grids. As we can see, the five-point finite difference stencil applied to the points along the interface between two subdomains will require access to the data residing on the neighboring processor. Hence, an *operator evaluation* of the type $\underline{w} = \underline{A}\,\underline{v}$ will require nearest neighbor communication. In addition, the two innerproducts in each conjugate gradient iteration will require global communication (vector reduction).

Figure 8 shows in detail how the communication has been implemented in the context of performing an operator evaluation. The subdomain associated with each processor is defined as a two-dimensional grid, and the associated data structure for the field variables is a two-dimensional array. We define this array in such a way that it also includes the values associated with the grid points along the external boundary (even though these are not unknowns). In addition, we define the array in such a way that it can hold the additional "column" of values coming from the

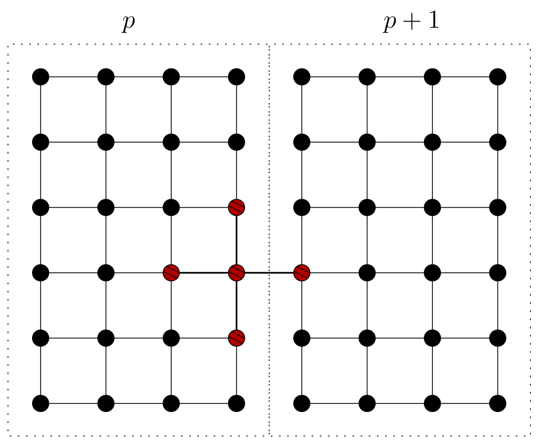


Figure 7: Adjacent subdomains couple via the five-point stencil.

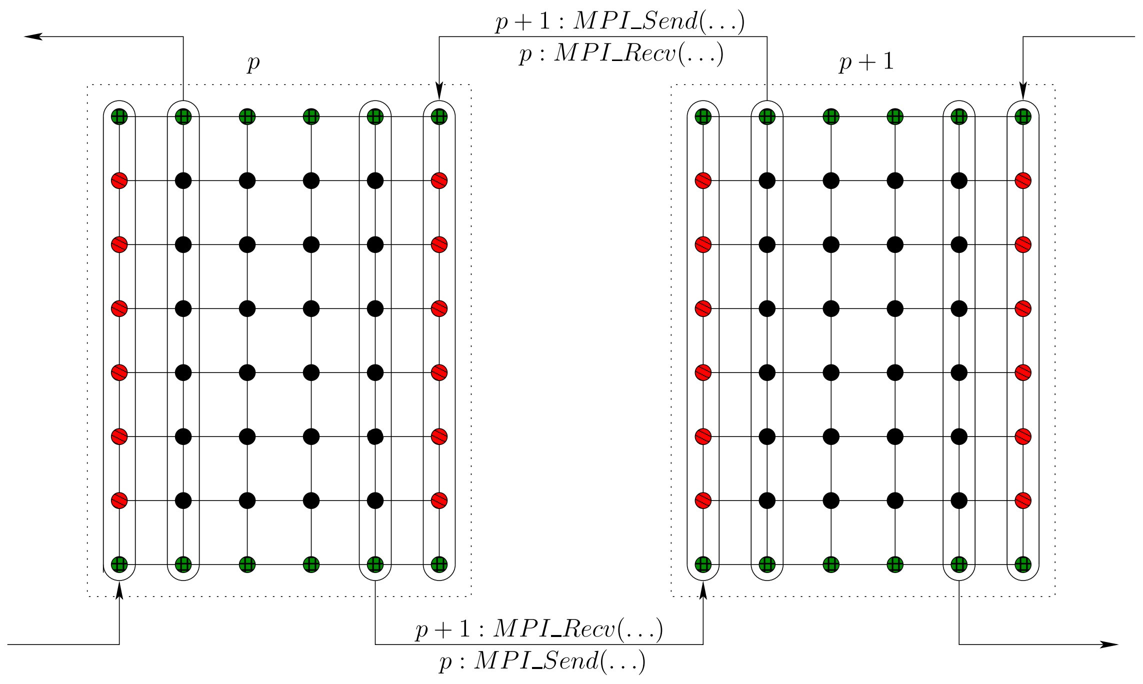


Figure 8: Nearest neighbor communication.

neighboring processor. In this fashion, the send and receive operations can proceed by pointing directly to the data structure with no need for extracting data to and from send and receive buffers. This approach also simplifies the programming somewhat.

The exchange of information between neighboring processors proceeds using a *red black ordering* of the processors. The processors in a send mode are colored red, while the processors in receive mode are colored black. Figure 9 depicts the send and receive operations necessary to accomplish the exchange. We use a blocking communication mode, and hence, a send operation always has to be matched by a receive operation on the neighboring processor. After the send and receive operations have been successfully completed, the program proceeds with the next instuction on each processor. If we have more than two processors, it will take four send/receive operations to complete the exchange of boundary data.

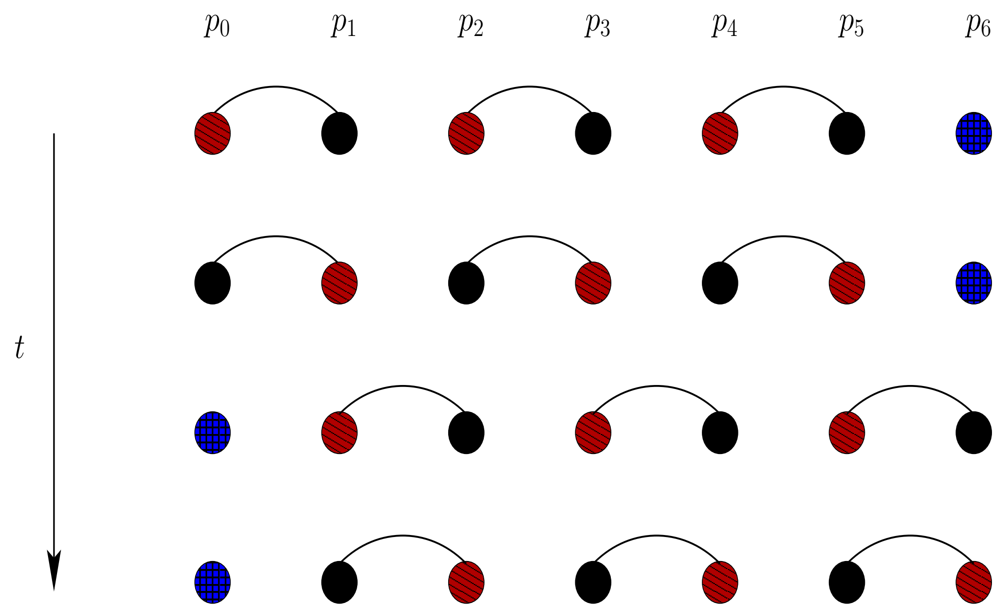


Figure 9: The communication between the processors. Red indicates send mode, black indicates receive mode, and blue indicates idle. If we use more than two processors, it will take four send/receive operations to complete the exchange of boundary data.

After the communication phase has finished, the operator evaluation (or matrix-vector product) can proceed without any communication by applying the five-point stencil to all the *internal* points in each subdomain; this approach will automatically then result in the correct couplings to the values along the boundary of each subdomain.

6 Performance analysis

We will now do a theoretical analysis of the performance of the proposed algorithms. In the following, we assume that double precision is used in the implementation. We denote by τ_A the time to perform a single floating-point operation. Furthermore, we assume that the interconnect for our multiprocess computer has the following characteristics: the time to send a message with k bytes, $\tau_C(k)$, can be approximated as

$$\tau_C(k) = \tau_S + \gamma \cdot k \quad . \tag{26}$$

Here, τ_S is a fixed startup time, and γ is the inverse bandwidth.

Let us first estimate the solution time on a single processor. As discussed earlier, the matrix-vector-product evaluation is implemented in such a way that storing and operating on matrix elements which are known to be zero is avoided. Let us assume that it takes M conjugate gradient iterations to reach convergence to a given tolerance.

The number of degrees-of-freedom is $N \approx n^2$. Using the five-point stencil, the matrix-vector product evaluation requires five multiplications and four additions per grid point. The global matrix-vector product evaluation $\underline{w} = \underline{A}\,\underline{p}^m$ therefore requires approximately $9N$ operations. In addition, we have three operations of the type "multiplication of vector with a scalar"; these require $3N$ multiplications (see Table 1). We also have three operations of the type "addition of two vectors"; these require $3N$ additions. Finally, we have two innerproducts which require $4N$ multiplications and additions. Hence, the number of floating-point operations per iteration is

$$\mathcal{N}_{op} \;\approx\; 9N + 3N + 3N + 4N = 19N.$$

The total solution time on a single processor can be estimated as

$$T_1 \approx M\,\mathcal{N}_{op}\,\tau_A = 19N\,M\,\tau_A\,. \tag{27}$$

We now consider a multi-process implementation using the MPI message passing library. We decompose the $n \times n$ grid into P approximately equal subgrids of size $n \times m_2$, i.e., $m_2 \approx n/P$. We assign one subgrid to each processor; see Section 5 for more details.

The global matrix-vector product and the two innerproducts per conjugate gradient iteration require communication. For the matrix-vector product evaluation, point-to-point communication using MPI_Send and MPI_Recv is appropriate. For the two innerproducts, we need the global reduction operation MPI_Allreduce. The communication cost per conjugate gradient iteration can then be estimated as

$$T_{comm,1} = 4\tau_C(8n) + 2(\tau_A + \tau_C(8))\log_2(P)$$

The factor four in the first term comes from the fact that it takes four send/receive operations to complete the exchange of boundary data (assuming more than two processors); see Section 5.2. On a typical computer, $\tau_A \ll \tau_C(8)$, and $\tau_C(8) \approx \tau_S$. Hence, the communication cost can be simplified to read

$$T_{comm,1} = 4\tau_C(8n) + 2\tau_S \log_2(P)$$

The total communication cost for M iterations is then $T_{comm} = M\, T_{comm,1}$.

The solution time on P processors can then be estimated as

$$T_p = \frac{T_1}{P} + T_{comm}, \tag{28}$$

while the speedup can be estimated as

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{\frac{T_1}{P} + T_{comm}} = \frac{P}{1 + P \cdot \frac{T_{comm}}{T_1}} \quad . \tag{29}$$

Since both T_{comm} and T_1 are proportional to M, the speedup is independent of the number of iterations, M (as expected). Inserting for T_{comm} and T_1, we get

$$S_p \approx \frac{P}{1 + P(\frac{4\tau_C(8n)+2\tau_S \log_2(P)}{19n^2\tau_A})} \quad . \tag{30}$$

We remark that, for a *fixed* number of processors P, T_{comm} will be bounded as $\mathcal{O}(n)$ as n increases, while T_1 will scale as $\mathcal{O}(n^2)$. Hence, the term T_{comm}/T_1 will scale as $\mathcal{O}(\frac{1}{n})$. In conclusion, it will be easier to get good speedup as the problem size increases for a given number of processors.

7 Numerical results

7.1 Correctness

We will now verify the numerical algorithm by performing a convergence test. To this end, we solve the Poisson problem on the domain $\Omega = (0,1) \times (0,1)$:

$$-\nabla^2 u = f \;\text{ in } \Omega, \tag{31}$$

$$u = 0 \;\text{ on } \partial\Omega. \tag{32}$$

We *choose* a known solution to the problem (31)-(32), $u(x,y) = e^x \sin(\pi x)\sin(2\pi y)$, and *derive* the right hand side of (31) to be $f(x,y) = -((1-5\pi^2)e^x \sin(\pi x) + 2\pi e^x \cos(\pi x))\sin(2\pi y)$. In Table 2 we list the maximum error for different values of n. We see that when n is doubled, the error decreases by a factor of four, which shows the expected second order behaviour of the finite difference scheme; see (5). We observe that the error does not depend on the numbers of processors used. In Figure 10 we present the same data as a convergence plot.

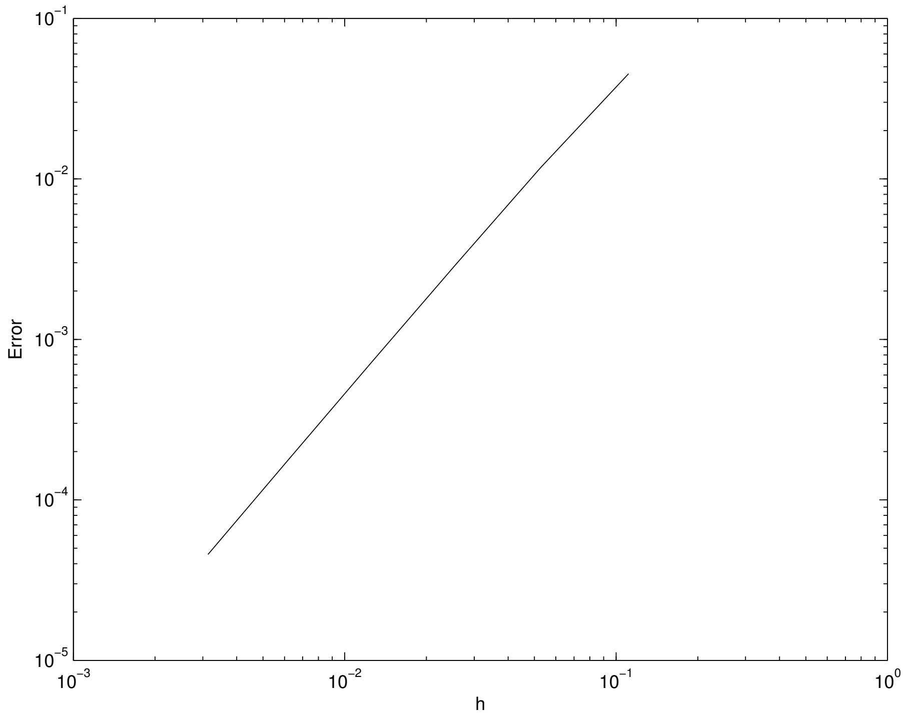


Figure 10: Convergence-plot for the two-dimensional Poisson-problem.

$P=1$			$P=4$		
n	Error	M	n	Error	M
10	$4.51 \cdot 10^{-2}$	10	10	$4.51 \cdot 10^{-2}$	10
20	$1.17 \cdot 10^{-2}$	24	20	$1.17 \cdot 10^{-2}$	24
40	$2.93 \cdot 10^{-3}$	56	40	$2.93 \cdot 10^{-3}$	56
80	$7.32 \cdot 10^{-4}$	116	80	$7.32 \cdot 10^{-4}$	116
160	$1.83 \cdot 10^{-4}$	234	160	$1.83 \cdot 10^{-4}$	234
320	$4.57 \cdot 10^{-5}$	465	320	$4.57 \cdot 10^{-5}$	465

Table 2: The maximum absolute error for different values of n. For these measurements, $P = 1$ (left) and $P = 4$ (right) is used. M denotes the number of iterations.

7.2 Scaling of the numerical algorithm

We now present some timing results in order to verify that our implementation gives the expected scaling. From Section 6, we expect the number of floating point operations to scale as $\mathcal{O}(n^2)$ per iteration (recall that $N \approx n^2$). The number of iterations, M, can be estimated as follows: $\underline{A}$ in our problem represents the discrete Laplace operator, and it can be shown that

$$\begin{aligned} &\lambda_{\min}(\underline{A}) \sim \mathcal{O}(1), \\ &\lambda_{\max}(\underline{A}) \sim \mathcal{O}(1/h^2) \sim \mathcal{O}(n^2). \end{aligned}$$

Hence, according to (25), we expect $M \sim \mathcal{O}(n)$. In summary, we expect the total simulation time, T, to scale as $\mathcal{O}(n^3)$.

In Table 3 we present some results concerning the scalability of the algorithm. Here, we have also divided the solution time T with n^3 in order to report $\tilde{T} = T/n^3$. The reason for this is that we expect the solution time to scale as $\mathcal{O}(n^3)$, and hence, $\tilde{T} = T/n^3$ should be approximately constant. In Figure 11 we have also plotted $\tilde{T} = T/n^3$ as a function of n.

Notice the change in the behaviour for $n \sim 1000$; the reason for this is related to the memory hierarchy. In the conjugate gradient algorithm we need to store four $n \times n$-arrays, namely the numerical solution, $\underline{u}$, the residual, $\underline{r}$, the search direction, $\underline{p}$, and the temporary variable, $\underline{w}$. We use double precision, which implies 8 bytes per floating point number. Thus, for $n = 1000$ we need $1000 \cdot 1000 \cdot 8 \cdot 4$bytes $=$ 32Mbytes. We also know that the size of the L2-cache on `gridur` (the earlier supercomputer at NTNU) is 8 Mbytes. Hence, for $P = 4$ and $n < 1000$, the L2-cache is large enough for all our data. The interpretation of all these results is that the algorithm scales as $\mathcal{O}(n^3)$, but that the *effective* value of τ_A depends on whether we can fit all the data in L2-cache or not.

7.3 Speedup and parallel efficiency

In Tables 4, 5 and 6 we have speedup results. The same data is displayed in Figure 13. The results are as expected for $n = 250$ and $n = 500$ with the parallel efficiency starting to decrease earlier for smaller values of n. For $n = 1000$, however, we see that we get better than perfect speedup for $n \geq 8$. Again, this is most likely due to the effect of the memory hierarchy. If a fixed problem cannot fit in L2-cache on a single processor, but can fit in L2-cache when several processors are used, we may experience superlinear speedup.

Acknowledgement. The figures, the implementation and the numerical tests were all done by doctoral student Tormod Bjøntegaard at the Department of Mathematical Sciences at NTNU. His help is greatly appreciated.

n	P	T	$\tilde{T} = \frac{T}{n^3}$
250	4	0.845	$5.41 \cdot 10^{-8}$
500	4	6.28	$5.63 \cdot 10^{-8}$
750	4	23.5	$5.57 \cdot 10^{-8}$
900	4	41.6	$5.71 \cdot 10^{-8}$
1000	4	73.6	$7.36 \cdot 10^{-8}$
1100	4	104	$7.81 \cdot 10^{-8}$
1250	4	216	$1.11 \cdot 10^{-8}$
1500	4	366	$1.08 \cdot 10^{-8}$
1750	4	579	$1.08 \cdot 10^{-8}$
2000	4	880	$1.10 \cdot 10^{-8}$

Table 3: Timing results on `gridur` (in seconds) for $P = 4$.

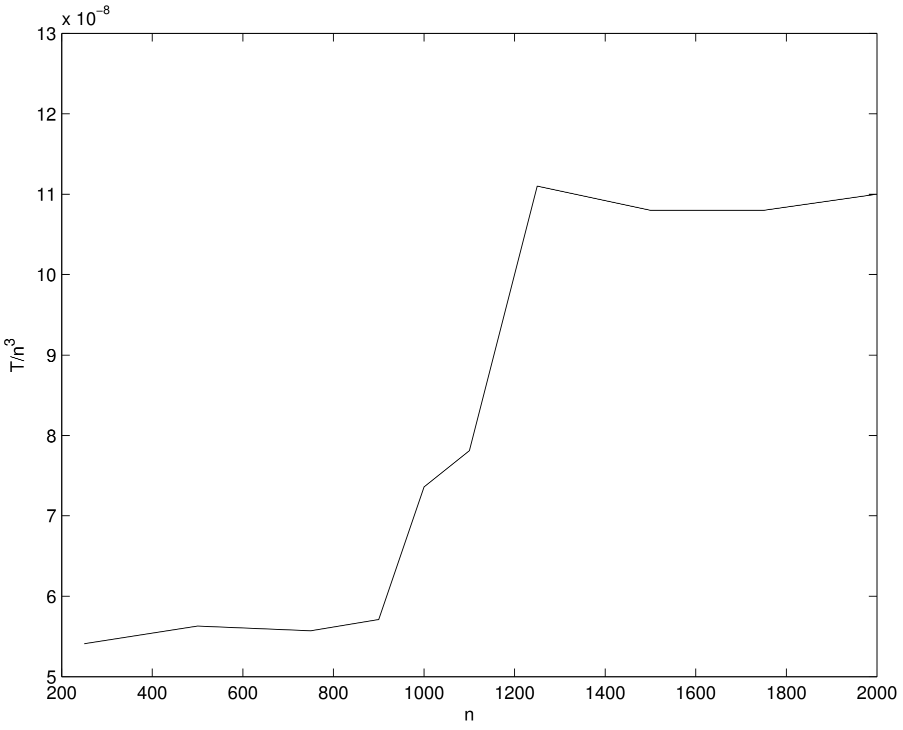


Figure 11: Solution time on `gridur` scaled with $1/n^3$ for $P = 4$.

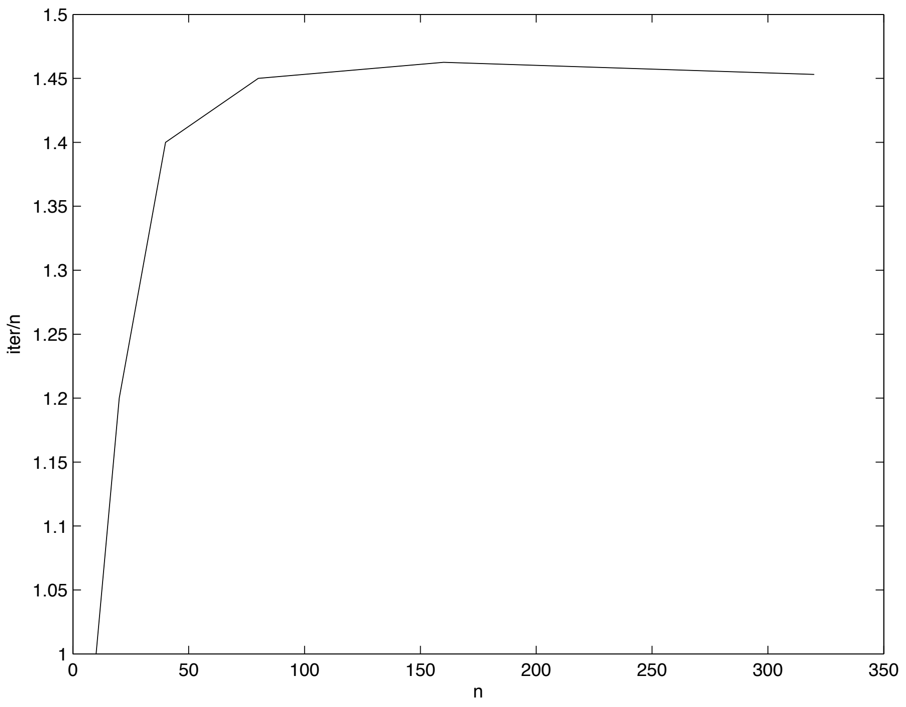


Figure 12: A plot of the number of conjugate gradient iterations, divided by n, as a function of n. These results confirm the expected result that the number of iterations scales as $\mathcal{O}(n)$.

n	P	T_1	T_p	$S_p = \frac{T_1}{T_p}$	$\eta_p = \frac{S_p}{p}$
250	1	3.17	3.17	1	1
250	2	3.17	1.64	1.93	0.97
250	4	3.17	0.845	3.75	0.94
250	8	3.17	0.560	5.66	0.71
250	16	3.17	0.442	7.17	0.45
250	32	3.17	0.334	9.49	0.30

Table 4: Timing results (in seconds), speedup, and parallel efficiency for $n = 250$. All the timing results were obtained on `gridur`.

n	P	T_1	T_p	$S_p = \frac{T_1}{T_p}$	$\eta_p = \frac{S_p}{p}$
500	1	31.1	31.1	1	1
500	2	31.1	12.8	2.43	1.21
500	4	31.1	8.48	3.67	0.92
500	8	31.1	4.78	6.51	0.81
500	16	31.1	2.38	13.07	0.82
500	32	31.1	1.52	20.46	0.64

Table 5: Timing results (in seconds), speedup, and parallel efficiency for $n = 500$. All the timing results were obtained on `gridur`.

n	P	T_1	T_p	$S_p = \frac{T_1}{T_p}$	$\eta_p = \frac{S_p}{p}$
1000	1	379	379	1	1
1000	2	379	174	2.18	1.09
1000	4	379	99.4	3.81	0.95
1000	8	379	27.5	13.78	1.72
1000	16	379	13.9	27.27	1.70
1000	32	379	8.40	45.12	1.41

Table 6: Timing results (in seconds), speedup, and parallel efficiency for $n = 1000$. All the timing results were obtained on `gridur`.

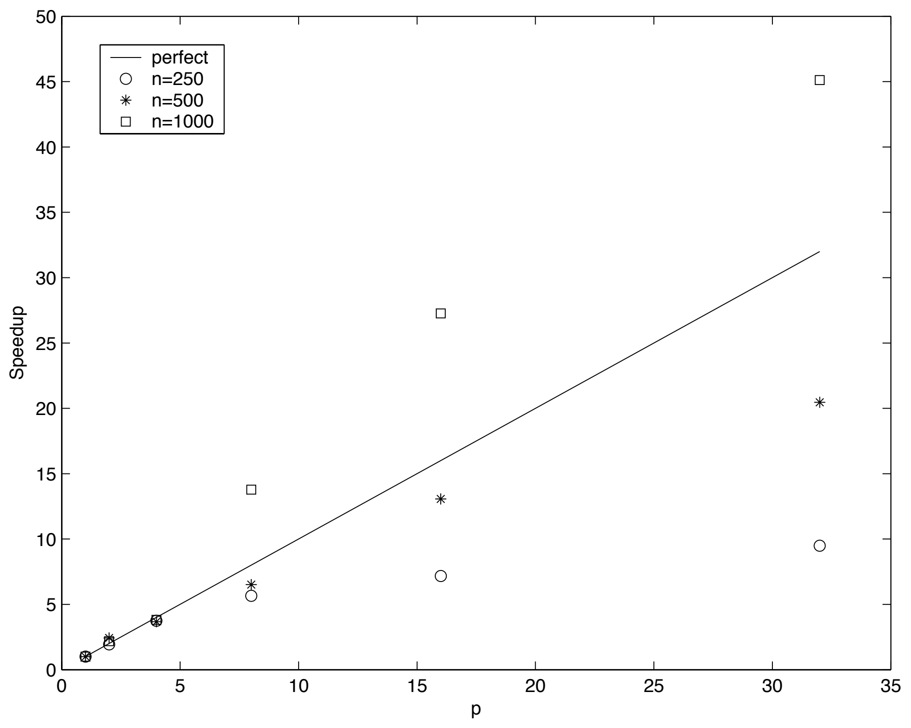


Figure 13: Speedup on `gridur` for different values of n. The solid line represents perfect (or linear) speedup, i.e., $S_p = p$.

8 Appendix: Program listing

```
/*
This program will solve the Poisson problem on a rectangle
(0,Lx) x (0,Ly) by using p processors.
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

typedef double Real;


/* function prototypes */
Real *createRealArray(int n);
Real **createReal2DArray(int m, int n);
int mproc(int m_x, int proc, int myid);
void setgrid(Real *x, Real h_x, int myid, int proc,
                                int m_x, int m_x_p);
void force(Real **f, Real *x, Real h_y, int m_x_p, int m_y,
                                        Real Lx, Real Ly);
void u_exact(Real **u_e, Real *x, Real h_y, int m_x_p, int m_y,
                                            Real Lx, Real Ly);
void error(Real **u_e, Real **u, int m_x_p, int m_y,
                                 int myid, int proc);

Real dot(Real *a, Real *b, int N, int myid, int proc);
void add2s1(Real *a, Real *b, Real scalar, int N);
void add2s2(Real *a, Real *b, Real scalar, int N);
void mvpLaplace(Real **w, Real **p, Real h_x, Real h_y,
                int m_x_p, int m_y, int myid, int proc);
void cg(Real **u, Real **r, Real tol, int m_x_p, int m_y,
                  Real h_x, Real h_y, int myid, int proc);
```

```
main(int argc, char **argv)
{
  /* Parameters:
     n_x+1 - Number of discretization points in the x-direction.
     n_y+1 - Number of discretization points in the y-direction.
     m_x - number of degrees of freedom in the x-direction.
     m_y - number of degrees of freedom in the y-direction.
     m_x_p - number of degrees of freedom in the x-direction on this processor.
     m_x_p2 - m_x_p+2 (needed in the operateLaplace-routine).
     m_y2 - m_y+2 (needed in the operateLaplace-routine).
     tol - the tolerance used in the CG-algorithm.
     myid - The rank of this processor.
     proc - The total number of processors.
     h_x - The grid spacing in the x-direction.
     h_y - The grid spacing in the y-direction.
     Lx, Ly - Defines the 2D-domain (0,Lx)x(0,Ly).
     x - an array of the x-coordinates on this processor.
     u - a 2D-array of the numerical solution.
     u_e - a 2D-array of the exact solution in the discretization points.
     f - the right-hand side evaluated in the discretization points.
  */
  int n_x, n_y, m_x, m_x_p, m_y, m_x_p2, m_y2, myid, proc;
  Real Lx, Ly;
  Real h_x, h_y, tol, t1, t2;
  Real **u, **u_e, **f, *x;

  tol = pow(10,-8);
  Lx = 1.0;
  Ly = 1.0;

  n_x = atoi(argv[1]);
  n_y = atoi(argv[1]);
  m_x = n_x-1;
  m_y = n_y-1;

  h_x = Lx/(Real)n_x;
  h_y = Ly/(Real)n_y;
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &proc );
  MPI_Comm_rank( MPI_COMM_WORLD, &myid );

  if (myid == 0){
    t1 = MPI_Wtime();
  }

  m_x_p = mproc( m_x, proc, myid );

  m_x_p2 = m_x_p+2;
  m_y2 = m_y+2;

  x = createRealArray( m_x_p );
  u = createReal2DArray( m_x_p2, m_y2 );
  u_e = createReal2DArray( m_x_p2, m_y2 );
  f = createReal2DArray( m_x_p2, m_y2 );

  setgrid( x, h_x, myid, proc, m_x, m_x_p );

  force( f, x, h_y, m_x_p, m_y, Lx, Ly );

  cg( u, f, tol, m_x_p2, m_y2, h_x, h_y, myid, proc );

  if (myid == 0){
    t2 = MPI_Wtime();
    printf("t: %e\n", t2-t1);
  }

  u_exact( u_e, x, h_y, m_x_p, m_y, Lx, Ly );
  error( u_e, u, m_x_p, m_y, myid, proc );

  MPI_Finalize();
}
```

```
Real *createRealArray (int n)
{
  Real *a;
  int i;
  a = (Real *)malloc(n*sizeof(Real));
  for (i=0; i < n; i++) {
    a[i] = 0.0;
  }
  return (a);
}
```

```
Real **createReal2DArray (int n1, int n2)
{
  int i, n;
  Real **a;
  a    = (Real **)malloc(n1   *sizeof(Real *));
  a[0] = (Real  *)malloc(n1*n2*sizeof(Real));
  for (i=1; i < n1; i++) {
    a[i] = a[i-1] + n2;
  }
  n = n1*n2;
  for (i=0; i < n; i++) {
    a[0][i] = 0.0;
  }
  return (a);
}
```

```
/* This routine computes the number of degrees of freedom
   in the x-direction on this processor */
int mproc(int m_x, int proc, int myid){

  int m, rem;
  rem = fmod(m_x,proc);

  if (myid < rem){
    m = m_x/proc + 1;
  } else {
    m = m_x/proc;
  }
  return (m);
}
```

```
/* Calculates the x-coordinates for the grid points on this processor */
void setgrid(Real *x, Real h_x, int myid, int proc, int m_x, int m_x_p){

  int mypos, rem, i;
  rem = fmod(m_x,proc);

  if (myid < rem){
    mypos = myid*(m_x/proc+1)+1;
  } else {
    mypos = rem*(m_x/proc+1) + (myid-rem)*(m_x/proc)+1;
  }

  for (i=0;i < m_x_p; i++){
    x[i] = (mypos+i)*h_x;
  }
}
```

```
/* Calculates the right-hand side */
void force(Real **f, Real *x, Real h_y, int m_x_p, int m_y,
                                          Real Lx, Real Ly){

  int i,j;
  Real pi;

  pi = 4.*atan(1.);
  for (i=0; i < m_x_p; i++){
    for (j=0; j < m_y; j++){
      f[i+1][j+1] = -((1-5*pi*pi)*exp(x[i])*sin(pi*x[i])
                    +2*pi*exp(x[i])*cos(pi*x[i]))*sin(2*pi*h_y*(j+1));
    }
  }
}
```

```
/* Calculates the exact solution (Needed for convergence-tests) */
void u_exact(Real **u_e, Real *x, Real h_y, int m_x_p, int m_y,
                                              Real Lx, Real Ly){
  int i,j;
  Real pi;

  pi = 4.*atan(1.);
  for (i=0; i < m_x_p; i++){
    for (j=0; j < m_y; j++){
      u_e[i+1][j+1] = exp(x[i])*sin(pi*x[i])*sin(2*pi*h_y*(j+1));
    }
  }
}
```

```
/* Measures the maximum global error. */
void error(Real **u_e, Real **u, int m_x_p, int m_y, int myid, int proc){

  int i,j;
  Real localerror = 0, globalerror;

  for (i=0;i < m_x_p; i++){
    for (j=0; j < m_y; j++){
      if (fabs(u_e[i+1][j+1]-u[i+1][j+1]) > localerror){
         localerror = fabs(u_e[i+1][j+1]-u[i+1][j+1]);
      }
    }
  }
  if (proc != 1){
    MPI_Reduce(&localerror, &globalerror, 1, MPI_DOUBLE, MPI_MAX, 0,
                                                  MPI_COMM_WORLD);
  } else {
    globalerror = localerror;
  }
  if (myid == 0){
    printf("Global Error = %e\n",globalerror);
  }
}
```

```
/* a:= scalar*a + b */
void add2s1(Real *a, Real *b, Real scalar, int N){

  int i;
  for (i=0; i < N; i++){
    a[i] = scalar*a[i] + b[i];
  }
}



/* a:= a + scalar*b */
void add2s2(Real *a, Real *b, Real scalar, int N){

  int i;
  for (i=0; i < N; i++){
    a[i] = a[i] + scalar*b[i];
  }
}




/* Routine for calculating the global inner-product. */
Real dot(Real *a, Real *b, int N, int myid, int proc){

  int i;
  Real sum = 0, mysum = 0;
  for (i=0; i < N; i++){
    mysum += a[i]*b[i];
  }

  if (proc == 1){
    sum = mysum;
  } else {
    MPI_Allreduce( &mysum, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );
  }
  return (sum);
}
```

```
/* Calculates the global matrix-vector product, w=Ap. */
void mvpLaplace(Real **w, Real **p, Real h_x, Real h_y, int m_x_p, int m_y,
                                                        int myid, int proc){

  int tag=0,i,j;
  MPI_Status status;

  if (proc != 1){
    if (fmod(myid,2) == 0){
      if ( myid != (proc-1)){
        MPI_Send( p[m_x_p-2], m_y, MPI_DOUBLE, myid+1, tag, MPI_COMM_WORLD );
        MPI_Recv( p[m_x_p-1], m_y, MPI_DOUBLE, myid+1, tag, MPI_COMM_WORLD,
                                                                   &status );
      }
      if ( myid != 0){
        MPI_Recv( p[0], m_y, MPI_DOUBLE, myid-1, tag, MPI_COMM_WORLD,
                                                             &status );
        MPI_Send( p[1], m_y, MPI_DOUBLE, myid-1, tag, MPI_COMM_WORLD );
      }
    } else {
      if ( myid != 0){
        MPI_Recv( p[0], m_y, MPI_DOUBLE, myid-1, tag, MPI_COMM_WORLD, &status );
        MPI_Send( p[1], m_y, MPI_DOUBLE, myid-1, tag, MPI_COMM_WORLD );
      }
      if ( myid != (proc-1)){
        MPI_Send( p[m_x_p-2], m_y, MPI_DOUBLE, myid+1, tag, MPI_COMM_WORLD );
        MPI_Recv( p[m_x_p-1], m_y, MPI_DOUBLE, myid+1, tag, MPI_COMM_WORLD,
                                                                   &status );
      }
    }
  }
  for (i=1; i < (m_x_p-1); i++){
    for (j=1; j < (m_y-1); j++){
      w[i][j] = -(p[i+1][j]-2*p[i][j]+p[i-1][j])/(h_x*h_x)
                -(p[i][j+1]-2*p[i][j]+p[i][j-1])/(h_y*h_y);
    }
  }
}
```

```
/* Solve the linear system of algebraic equations using the CG-algorithm. */
void cg(Real **u, Real **r, Real tol, int m_x_p, int m_y,
                            Real h_x, Real h_y, int myid, int proc){
  int iter;
  Real rr, rr_old, res, beta, alpha, pw, **w, **p;

  w = createReal2DArray( m_x_p, m_y );
  p = createReal2DArray( m_x_p, m_y );

  rr = dot( r[0], r[0], m_x_p*m_y, myid, proc );
  res = sqrt( rr );

  iter = 1;
  while ( res > tol ){
    if ( iter == 1 ){
      beta = 0;
    } else {
      beta = rr/rr_old;
    }
    add2s1( p[0], r[0], beta, m_x_p*m_y );

    mvpLaplace( w, p, h_x, h_y, m_x_p, m_y, myid, proc );

    pw = dot( p[0], w[0], m_x_p*m_y, myid, proc );

    alpha = rr/pw;

    add2s2( u[0], p[0], alpha, m_x_p*m_y );
    add2s2( r[0], w[0], -alpha, m_x_p*m_y );

    rr_old = rr;
    rr = dot( r[0], r[0], m_x_p*m_y, myid, proc );
    res = sqrt(rr);
    iter++;
  }
  if (myid==0){ printf("iter=%d\n", iter); }
}
```