FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Isoefficiency Analysis of Parallel QMR-Like Iterative Methods and its Implications on Parallel Algorithm Design

H. Martin Bücker

KFA-ZAM-IB-9604

Januar 1996
(Stand 11.01.96)

This work was supported by the Graduiertenkolleg “Informatik und Technik”, RWTH Aachen, D–52056 Aachen, Germany.

Isoefficiency Analysis of Parallel QMR-Like Iterative Methods and its Implications on Parallel Algorithm Design*

H. Martin Bücker

Central Institute for Applied Mathematics
Research Centre Jülich (KFA)
D – 52425 Jülich, Germany
tel.: +49 2461 61 6753
fax: +49 2461 61 6656
e-mail: m.buecker@kfa-juelich.de

January 11, 1996

Abstract

A specific problem arising out of electrostatics is taken as an example to demonstrate the process of, firstly, transforming a physical problem into a mathematical model and, secondly, its numerical solution by generating a system of linear equations via finite difference approximations. The resulting nonsymmetric sparse linear system is solved by a class of iterative methods that is defined by taking the *Quasi-Minimal Residual* (QMR) method as a typical member. A performance model called *isoefficiency concept* is used to analyze the behavior of such methods implemented on parallel distributed memory computers with two-dimensional mesh topology. The isoefficiency concept is employed to compare two different mappings of data to processors as well as to give hints how QMR-like iterative methods should be designed with respect to parallel computing.

1 Introduction

A lot of scientific and engineering problems are solved by making use of numerical linear algebra algorithms. The solution of systems of linear equations constitutes one of the basic tools among these algorithms. Nowadays, realistic problems are becoming increasingly larger and more difficult to solve while at the same time new parallel hardware architectures along with modern mathematics deliver the potential of ever higher performances. Unfortunately, traditional algorithms used on conventional serial machines inevitably do not achieve the same degree of efficiency on parallel computers. This is the reason for designing new parallel algorithms offering more separate independent duties that can be treated simultaneously.
The problems to be tackled these days arise from a variety of disciplines. There is now hardly any area of science and engineering that does not use the computer as a working instrument. The diversity of problem components as well as their hardness require an interdisciplinary approach of mathematicians, physicists, computer scientists, and engineers. However, concerted actions of different disciplines are almost always arduous and troublesome for several reasons, e.g., the desire to maintain the discipline's autonomy or their distinct technical jargon. The goal of this paper is to

*This work was supported by the Graduiertenkolleg "Informatik und Technik", RWTH Aachen, D–52056 Aachen, Germany.

bridge the gap between two disciplines: numerical analysis and computer science. The contents of the first are rarely taught in courses of the second. Computer science is generally not participating in processing of numerical data.
Finding "answers" to realistic problems often involves at least the development of computationally feasible models for physical systems, designing algorithms for solving issues arising in the modeling process, and matching these algorithms to adequate computer architectures. By taking a specific physical problem as an example, Sect. 2 derives a mathematical model and demonstrates its numerical solution resulting in a system of linear equations with nonsymmetric sparse coefficient matrix. That part of the paper is intended to address an audience that is not familiar with elementary numerical methods. Those readers well-informed about discretizing partial differential equations by finite difference approximations can turn directly to Sect. 3 that presents a class of iterative methods applicable to the system derived in the previous section. The class is defined by a typical member: the quasi-minimal residual (QMR) method developed by Freund and Nachtigal [10]. It is shown how these iterative methods can be implemented on parallel computers with distributed memory. Two different mappings of data to processors are investigated and communication times on parallel computers with two-dimensional mesh topology are given for each kind of operation occuring in the iterative methods. Section 4 puts these communication times into a performance model called *isoefficiency analysis* that tries to analyze the *scalability* of a parallel algorithm implemented on a parallel architecture, i.e., its ability to achieve performance proportional to the number of processors. The model can serve helping to design new parallel algorithms which is shown in Sect. 5.

2 Continuous Problems Solved Discretely

The solution of many problems from science or engineering requires the formulation of a mathematical model, typically systems of differential equations, either ordinary or partial. If explicit closed-form solutions are not possible—like in most "real-world" applications—the numerical solution on digital computers demands replacing continuous problems by discrete correspondences. The collection of theories, techniques, and tools applicable to such approximations is vast. This section presents a physical sample problem and its mathematical model. For discretization, a common technique that can be found in almost any elementary numerical textbook [21, 13] is used. The result is a system of linear equations with nonsymmetric coefficient matrix.

Mathematical Model

Consider the task of determining the electric field $\vec{\mathbf{E}}$ produced by a charge density ρ in a medium of dielectric permeability ε. The theory of electromagnetic fields answers such questions by examining a set of equations called *Maxwell's equations*. The above problem falls into a particular class of the general theory. In this *electrostatic* case, it is well-known [17] that there is no interaction between electric and magnetic phenomena. Thus, by ignoring those of Maxwell's equations determining magnetic quantities, the electric field can be computed from only two of these equations in

addition with a third equation describing properties of the medium, namely

$$\operatorname{curl} \vec{\mathbf{E}} = \mathbf{0} \ , \tag{1}$$

$$\operatorname{div} \vec{\mathbf{D}} = \rho \ , \tag{2}$$

$$\vec{\mathbf{D}} = \varepsilon \, \vec{\mathbf{E}} \ , \tag{3}$$

where $\vec{\mathbf{D}}$ denotes the electric displacement. Equation (1) shows that the electric field $\vec{\mathbf{E}}$ can be expressed in terms of a scalar potential V satisfying

$$\vec{\mathbf{E}} = -\,\mathbf{grad}\, V \ . \tag{4}$$

Hence, by eliminating $\vec{\mathbf{E}}$ from the above set of equations, the problem of finding the electric field is reduced to determining its scalar potential. Inserting (3) and (4) into (2) yields

$$-\operatorname{div}(\varepsilon \, \mathbf{grad}\, V) = \rho \ ,$$

and finally, using rules for differential operators leads to

$$\varepsilon \, \Delta V + \mathbf{grad}\, \varepsilon \cdot \mathbf{grad}\, V = -\rho \ . \tag{5}$$

Assume that the domain of the problem is the unit square $0 \le x, y \le 1$ and ε and ρ are given functions of x and y. Furthermore, suppose Dirichlet boundary conditions, i.e.,

$$V(x,y) = f(x,y) \ , \qquad (x,y) \quad \text{on boundary} \ , \tag{6}$$

where f is a given function of x and y. A schematic illustration of the above two-dimensional problem from electrostatics is depicted in Fig. 1.

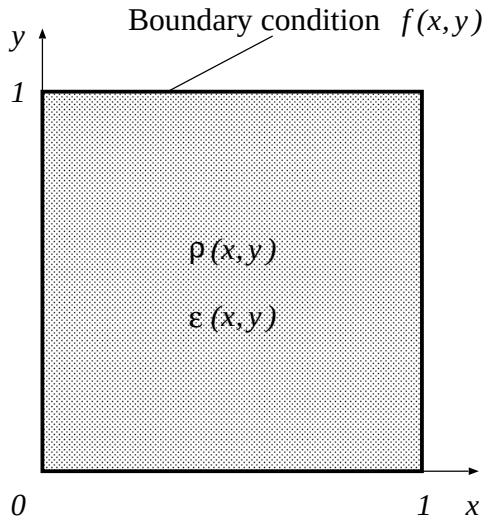

Figure 1: Electrostatic sample problem

In summary, using Maxwell's equations (1) and (2) as well as (3) that describes properties of the medium, the physical problem is converted into a mathematical model consisting of a linear partial differential equation of second order (5) with corresponding boundary conditions (6).

To avoid multiple subscripts in the next subsection, we introduce the redundant symbols a, b, c, and d as functions of x and y. They characterize given aspects of the dielectric permeability ε as well as of the charge density ρ and are defined by

$$a := \varepsilon \ , \qquad b := \varepsilon_x \ , \qquad c := \varepsilon_y \ , \qquad d := -\rho \ ,$$

where the subscripts denote partial derivatives. Thus, the partial differential equation (5) becomes

$$a\,(V_{xx} + V_{yy}) + bV_x + cV_y = d \ . \tag{7}$$

Discretization

To solve the problem numerically, we use a common approach based on *finite differences*. The idea behind finite difference methods in two dimensions is to approximate the behavior of a function in a continuous plane with its behavior on a regularly spaced finite set of points in the plane. Therefore, a mesh of grid points is imposed on the unit square with spacing h between the points in both the horizontal and the vertical directions. Two groups of grid points can be distinguished with respect to their location in the domain. The first group consists of *boundary* grid points. These grid points are on the boundary of the domain, here, the sides of the unit square. All other grid points fall into the second group and are called *interior* grid points. These interior grid points are given by

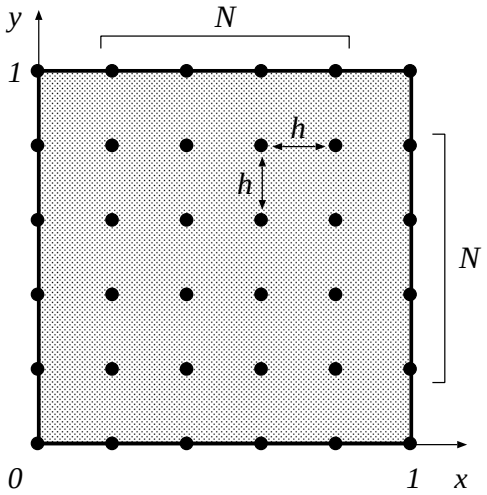

Figure 2: Mesh of grid points with $N = 4$

$$(x_i, y_j) = (ih, jh) \ , \qquad i, j = 1, 2, \ldots, N \ ,$$

with $(N + 1)h = 1$ whereas the coordinates of the boundary grid points satisfy $x_0 = y_0 = 0$ and $x_{N+1} = y_{N+1} = 1$. Figure 2 demonstrates this discretization for $N = 4$. Note that this discretization results in N^2 interior grid points.
At an interior grid point (x_i, y_j), the partial derivatives V_x, V_y, V_{xx}, and V_{yy} are approximated by the *centered finite difference* approximations

$$\begin{aligned}
V_x(x_i, y_j) &\doteq \frac{1}{2h}\Big[V(x_{i+1}, y_j) - V(x_{i-1}, y_j)\Big] \ , \\
V_y(x_i, y_j) &\doteq \frac{1}{2h}\Big[V(x_i, y_{j+1}) - V(x_i, y_{j-1})\Big] \ , \\
V_{xx}(x_i, y_j) &\doteq \frac{1}{h^2}\Big[V(x_{i+1}, y_j) - 2V(x_i, y_j) + V(x_{i-1}, y_j)\Big] \ , \\
V_{yy}(x_i, y_j) &\doteq \frac{1}{h^2}\Big[V(x_i, y_{j+1}) - 2V(x_i, y_j) + V(x_i, y_{j-1})\Big] \ ,
\end{aligned}$$

where $i, j = 1, 2, \ldots, N$. In the sequel, we denote a given function evaluated at (x_i, y_j) by subscripts separated by a comma, i.e., we use $a_{i,j}$ rather than $a(x_i, y_j)$ and analogously in the context with b, c, and d. If we insert the above finite difference approximations into the differential equation (7), we obtain at all interior grid points (x_i, y_j)

$$\begin{aligned}
&\left(\frac{a_{i,j}}{h^2} + \frac{c_{i,j}}{2h}\right) V(x_i, y_{j+1}) + \left(\frac{a_{i,j}}{h^2} - \frac{c_{i,j}}{2h}\right) V(x_i, y_{j-1}) \\
&\qquad + \left(\frac{a_{i,j}}{h^2} - \frac{b_{i,j}}{2h}\right) V(x_{i-1}, y_j) + \left(\frac{a_{i,j}}{h^2} + \frac{b_{i,j}}{2h}\right) V(x_{i+1}, y_j) \\
&\qquad\qquad - \frac{4a_{i,j}}{h^2} V(x_i, y_j) \doteq d_{i,j} \ , \qquad i, j = 1, 2, \ldots, N \ .
\end{aligned}$$

The *exact* solution $V(x_i, y_j)$ of the differential equation must satisfy these relations in the approximative sense "$\doteq$" at all interior grid points. We now turn this process around and use equality "$=$" in place of approximation "$\doteq$" hoping to find an

accurate estimate to the exact solution. That means we try to find numbers $V_{i,j}$ that satisfy the equations

$$\begin{aligned}\left(\frac{a_{i,j}}{h^2}+\frac{c_{i,j}}{2h}\right)V_{i,j+1}+\left(\frac{a_{i,j}}{h^2}-\frac{c_{i,j}}{2h}\right)V_{i,j-1}\\ +\left(\frac{a_{i,j}}{h^2}-\frac{b_{i,j}}{2h}\right)V_{i-1,j}+\left(\frac{a_{i,j}}{h^2}+\frac{b_{i,j}}{2h}\right)V_{i+1,j}\\ -\frac{4a_{i,j}}{h^2}V_{i,j}=d_{i,j}\ , \qquad i,j=1,2,\ldots,N\ , \quad (8)\end{aligned}$$

at all interior grid points and fulfill

$$V_{i,j}=f_{i,j}\ , \qquad i,j \quad \text{on boundary}\ , \quad (9)$$

at all boundary grid points, where $f_{i,j}$ is used to denote the evaluation of f at a boundary grid point (x_i, y_j). Then, we can consider the numbers $V_{i,j}$ to be approximations to the exact solution $V(x_i, y_j)$. Equations (8) relate the approximation at an interior grid point (x_i, y_j) to the approximations at its four immediate neighbors in the north, south, west, and east. This relationship is illustrated in Fig. 3.
Notice the different meaning of subscripts for *given* functions and the function that one is looking for by solving the partial differential equation. The subscripts of a given function, e.g., in $f_{i,j}$, are used to denote the evaluation of that function at a grid point, i.e., $f_{i,j} := f(x_i, y_j)$. On the other hand, the notation $V_{i,j}$ indicates numbers that satisfy (8) and (9) and that can afterwards be considered to approximate the exact solution, i.e., $V_{i,j} \approx V(x_i, y_j)$.

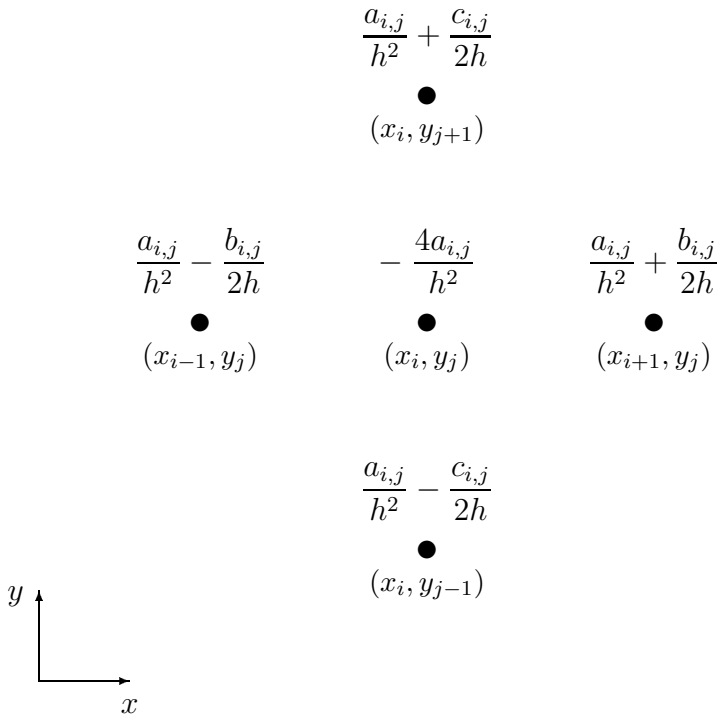

Figure 3: Approximation at an interior grid point (x_i, y_j) is related to the approximations at its four immediate neighbors with coefficients according to equations (8)

Matrix-Vector Form

Equations (8) give a system of linear equations in the variables $V_{i,j}$. Some of them, to be more precise those $V_{i,j}$ at the boundary grid points, are directly given by the boundary conditions (9). Therefore, (8) reduces to a system of N^2 linear equations in N^2 unknowns corresponding to the interior grid points. To write this system in matrix-vector form, the interior grid points are numbered in a row-major fashion from left to right and from bottom to top. Figure 4 shows this numbering called *natural ordering* for $N = 4$.

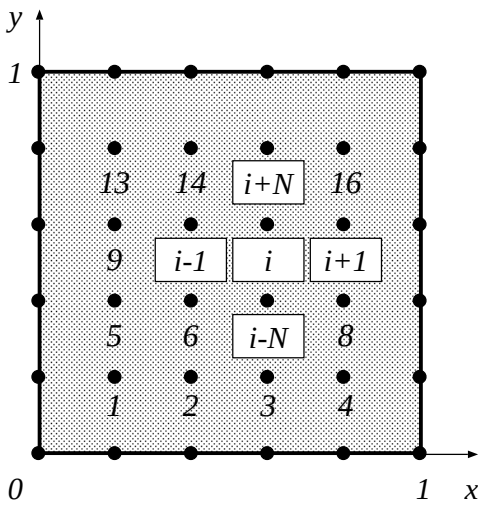

Figure 4: Natural ordering with $N = 4$

In the following, we focus on the interior grid points alone and change the notation of their subscripts by using a one-dimensional labeling corresponding to the natural ordering rather than the two-dimensional labeling that was inspired by grid points in the plane. Take the variables $V_{i,j}$ as an example and keep in mind that the given functions $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, $d_{i,j}$, and $f_{i,j}$ are treated similarly. In the above discussion, the values at the interior grid points, e.g., of the bottom row, were denoted by $V_{1,1}$, $V_{2,1}$, $V_{3,1}, \ldots, V_{N,1}$, whereas the new one-dimensional labeling refers to them as V_1, V_2, $V_3, \ldots, V_N$. The latter labeling corresponds to the natural ordering and will be used in the sequel.

The interior grid points can be partitioned into two groups: Those that are immediately adjacent to at least one boundary grid point, and those that are not immediately adjacent to any boundary grid points. Consider an interior grid point that falls into the second group. As Fig. 4 shows, such a point i has four immediate neighbors in the north, south, west, and east labeled with $i + N$, $i - N$, $i - 1$, and $i + 1$, respectively. Hence, equations (8) become

$$\left(\frac{a_i}{h^2} + \frac{c_i}{2h}\right) V_{i+N} + \left(\frac{a_i}{h^2} - \frac{c_i}{2h}\right) V_{i-N} + \left(\frac{a_i}{h^2} - \frac{b_i}{2h}\right) V_{i-1} + \left(\frac{a_i}{h^2} + \frac{b_i}{2h}\right) V_{i+1} - \frac{4a_i}{h^2} V_i = d_i \ , \quad (10)$$

where i is an interior grid point that is not immediately adjacent to any boundary grid point. Therefore, if we write the linear system in matrix-vector form, the ith row of the coefficient matrix has five nonzero entries: One entry in the principal diagonal, two entries each immediately adjacent to the principal diagonal, and two entries each N positions apart from the principal diagonal.

The equations corresponding to interior grid points falling into the first group, i.e., that are immediately adjacent to at least one boundary grid point, look slightly different from (10). For such interior grid points, those terms of the equations that are related to boundary grid points can be thought of as constants because these values are given by the boundary conditions. If we put these constants to the right hand side, the related terms of the equations vanish on the left hand side. Hence, the rows of the coefficient matrix corresponding to interior grid points of the first

group have fewer nonzero entries. Take the interior grid point at the bottom left corner as an example. Since this point is labeled with $i = 1$, the corresponding equation determines the first row of the coefficient matrix. The terms related to V_{i-1} and V_{i-N}, i.e., the west and south immediate neighbor, are given by the boundary conditions. For this point, the left hand side of the equation analogous to (10) only consists of three terms related to V_{i+1}, V_{i+N}, and V_i. Thus, the first row of the coefficient matrix has three nonzero entries. Similarly, the interior grid points that are immediately adjacent to exactly one boundary grid point produce rows of the coefficient matrix having four nonzero entries.

Using the natural ordering, the matrix-vector form of the linear system given by (8) has a coefficient matrix whose nonzero entries are structured according to the symmetric pattern depicted in Fig. 5.

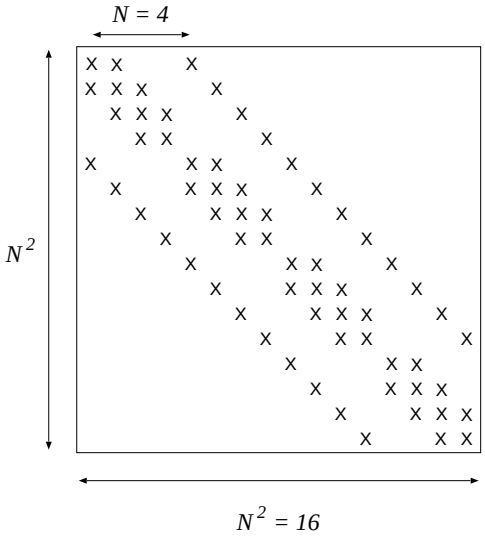

Figure 5: Pattern of nonzeros with $N = 4$

Remark 2.1. Regardless of the size of N, the coefficient matrix of the system defined by (8) has at most five nonzero entries per row. If N is of moderate size, say $N = 10^3$, then the coefficient matrix consists of $N^2 = 10^6$ rows giving approximately $5 \cdot 10^6$ nonzero entries out of $N^4 = 10^{12}$ total matrix entries. Therefore, the coefficient matrix is very *large* and *sparse*. The meaning of "large" depends upon the type of computer that is used for the numerical solution and, moreover, changes rapidly with the progress of computer technology. The meaning of "sparse" is somewhat vague too. A common definition is due to Wilkinson who called a matrix "sparse" *whenever it is possible to take advantage of the number and location of its nonzero entries*. In this paper, we assume that "sparse" means the usage of an appropriate storage scheme such that vector and matrix operations can be computed efficiently. More precisely, given a matrix $\mathbf{A} \in \mathbb{R}^{N^2 \times N^2}$ and a vector $\mathbf{x} \in \mathbb{R}^{N^2}$, then the number of arithmetic multiplications and additions needed to compute $\mathbf{Ax}$ is proportional to $5N^2$ as opposed to N^4 that a conventional matrix-vector multiplication with a dense matrix would require.

Large sparse matrices arise in a variety of ways besides the numerical solution of partial differential equations, e.g., in Markov chain models and many-body problems where the interaction only between nearby particles is taken into consideration. Therefore, we are not concerned with algorithms specifically-tuned to the symmetric matrix pattern of Fig. 5. Instead, we use the above discussion as an example showing the origin of an arbitrary nonsymmetric linear system. The principal concepts of Sect. 4 where a performance model is presented apply to arbitrary matrix patterns as well provided that slight modifications are introduced.

Remark 2.2. Although the coefficient matrix derived above has a symmetric pattern, the matrix itself is generally nonsymmetric. Figure 3 schematically shows the five-point finite difference approximation formula whose coefficients build the matrix rows. The nonsymmetry of these coefficients leads to a nonsymmetric matrix. Note that in the special case, where $b \equiv c \equiv 0$ and a is constant, the coefficients of the five-point finite difference approximation formula are symmetric resulting in a symmetric coefficient matrix. Physically speaking, this means that the dielectric permeability ε is homogeneous, i.e., $\mathbf{grad}\, \varepsilon = \mathbf{0}$, and the problem reduces to a

Poisson problem. Note further that the charge density $\rho = -d$ and the boundary condition f do not affect the symmetry of the coefficient matrix, in any manner.

Remark 2.3. For the numerical solution of the partial differential equation (7), centered finite difference approximations that take five grid points into account are used. Neither it is forbidden to consider more grid points, say nine, nor it is necessary to deal with finite difference approximations that are centered. Another topic of finite differences is their error behavior. If we decrease the spacing h between the grid points, i.e., we increase the number of grid points to obtain a finer grid, do the numbers $V_{i,j}$ approximate the exact solution $V(x_i, y_j)$ more accurately, and if so at what rate? We refer the reader to the vast literature on finite differences for answers to not only those questions.

Remark 2.4. In the above discussion, the natural ordering is used to label the interior grid points and to put the linear system into matrix-vector form. There are many different numbering schemes resulting in permutated coefficient matrices, e.g., red-black, multi-colored, diagonal, minimum-degree, and nested dissection ordering, to name just a few. The principal aim of all orderings is to make the subsequent solution process fast and numerically stable. During a solution process, e.g., Cholesky factorization, a zero-element may turn into a nonzero. This phenomenon is called *fill-in*. Assuming that matrix operations and storage schemes exploit the sparsity of the coefficient matrix, fill-in increases storage requirements as well as the number of arithmetic operations. As can be seen from some examples, ordering has an influence on fill-in. So, in addition to providing numerical stability, another goal of ordering is to minimize fill-in. Unfortunately, minimizing fill-in is generally an intractable combinatorial problem [22, 12] such that heuristics have to be used. A further objective comes along with ordering in the context of parallel processing. An ordering should maximize the number of independent tasks of the subsequent solution process. Note that maximizing parallelism and minimizing fill-in usually are contradictory jobs. The reader is referred to [18, 15] and the references therein for more details on orderings suitable for parallel processing.

3 Parallel QMR-Like Iterative Methods

There are two broad categories for the solution of systems of linear equations. *Direct methods* are based on a factorization of the coefficient matrix. A well-known example is Gaussian elimination. Alternatively, *iterative methods* compute successive approximations to obtain more accurate solutions at each step. The wide range of techniques in the field of iterative methods is surveyed in [1]. This section briefly states only those issues of iterative methods that are necessary for the performance model presented in the next section. More precisely, we concentrate on the operations involved in a special class of iterative methods and show how mapping of data to processors influences their communication times on distributed memory machines.

QMR-Like Iterative Methods

The quasi-minimal residual method (QMR) developed by Freund and Nachtigal [10] is an iterative technique that is applicable to the nonsymmetric linear system of the previous section. Recall that we want to solve

$$\mathbf{A}\mathbf{x} = \mathbf{b} \ , \qquad \text{where} \quad \mathbf{A} \in \mathbb{R}^{N^2 \times N^2} \qquad \text{and} \quad \mathbf{x}, \mathbf{b} \in \mathbb{R}^{N^2} \ .$$

Note that the coefficient matrix $\mathbf{A}$ is supposed to be sparse with nonzero elements symmetrically patterned as shown in Fig. 5. QMR belongs to a specific class of iterative methods called *Krylov subspace methods* [9]. The crucial part of any such method is the numerically stable computation of a basis for the subspace $\langle \mathbf{p}, \mathbf{A}\mathbf{p}, \mathbf{A}^2\,\mathbf{p}, \ldots, \mathbf{A}^n\,\mathbf{p} \rangle$, where $\mathbf{A}$ is the coefficient matrix and $\mathbf{p}$ is a suitable starting vector. During the iterative process, the following operations are performed: scalar operations, linear combinations of vectors, inner products, and matrix-vector products. QMR whose main loop is given in Fig. 6 consists solely of operations of the above kind where all inner products are marked with a bullet in front of the corresponding assignment. Note that the Euclidean norm of a vector $\mathbf{z}$ is defined by $\|\mathbf{z}\|_2 = \sqrt{\mathbf{z}^T\mathbf{z}}$ such that a Euclidean norm is an inner product followed by a scalar operation. In order to hide the mathematical background, we refer to methods that exclusively use scalar operations, linear combinations, inner products, and matrix-vector products as *QMR-like iterative methods*. We assume that the main loop of such a method consists of a constant number of these operations, i.e., methods like GMRES [20] whose number of inner products grows linearly with respect to the number of iteration steps are not considered here. Exploiting the sparsity of the coefficient matrix while calculating a matrix-vector product, cf. Remark 2.1, the execution time of the fastest known sequential algorithm to perform a single iteration step of a QMR-like iterative method is then linear with respect to the number of vector components

for $n = 1, 2, 3, \ldots$ **do**

- $\delta_n = \mathbf{w}_n^T \mathbf{v}_n$

  $\mathbf{p}_n = \mathbf{v}_n - \frac{\xi_n \delta_n}{\epsilon_{n-1}} \mathbf{p}_{n-1}$

  $\mathbf{q}_n = \mathbf{w}_n - \frac{\rho_n \delta_n}{\epsilon_{n-1}} \mathbf{q}_{n-1}$
- $\epsilon_n = \mathbf{q}_n^T \mathbf{A} \mathbf{p}_n$

  $\beta_n = \frac{\epsilon_n}{\delta_n}$

  $\widetilde{\mathbf{v}}_{n+1} = \mathbf{A}\mathbf{p}_n - \beta_n \mathbf{v}_n$

  $\widetilde{\mathbf{w}}_{n+1} = \mathbf{A}^T \mathbf{q}_n - \beta_n \mathbf{w}_n$
- $\rho_{n+1} = \|\widetilde{\mathbf{v}}_{n+1}\|_2$
- $\xi_{n+1} = \|\widetilde{\mathbf{w}}_{n+1}\|_2$

  $\vartheta_n = \frac{\rho_{n+1}}{c_{n-1}|\beta_n|}$

  $c_n = \frac{1}{\sqrt{1+\vartheta_n^2}}$

  $\eta_n = -\eta_{n-1} \frac{\rho_n c_n^2}{\beta_n c_{n-1}^2}$

  $\mathbf{d}_n = \eta_n \mathbf{p}_n + (\vartheta_{n-1} c_n)^2 \mathbf{d}_{n-1}$

  $\mathbf{x}_n = \mathbf{x}_{n-1} + \mathbf{d}_n$

  $\mathbf{v}_{n+1} = \frac{1}{\rho_{n+1}} \widetilde{\mathbf{v}}_{n+1}$

  $\mathbf{w}_{n+1} = \frac{1}{\xi_{n+1}} \widetilde{\mathbf{w}}_{n+1}$

endfor

Figure 6: QMR; cf. Alg. 7.1 of [11]

$$T_{\text{seq}} = cN^2\, t_{\text{a}} \ , \tag{11}$$

where c is a constant and t_{a} is the time required to perform an arithmetic operation on a given serial computer.

For the isoefficiency analysis carried out in the next section, it is necessary to determine an overhead. The purpose of the rest of this section is to derive communication times for each of the above four kinds of operations that are responsible for the overhead.

Remark 3.1. The symmetric pattern of the coefficient matrix implies that the operations $\mathbf{A}\mathbf{z}$ and $\mathbf{A}^T\mathbf{z}$ can be treated identically when deriving communication times for matrix-vector products. We therefore focus on the operation $\mathbf{A}\mathbf{z}$. For nonsymmetric patterns, a discussion of the operation $\mathbf{A}^T\mathbf{z}$ would easily generalize the concepts presented here.

Usually, iterative methods are combined with a suitable preconditioning technique to speed up convergence and to improve numerical stability. We will not discuss such techniques; those preconditioning techniques that themselves are limited to the above four kinds of operations are contained in our discussion.

Parallel Computation of Scalar Operations and Linear Combinations

We suppose throughout the paper that there is an integer p such that p divides N^2. Then, we decompose the vector space $\mathbb{R}^{N^2}$ as the Cartesian product of p lower-dimensional subspaces $\mathbb{R}^{N^2/p}$. Any vector $\mathbf{z} \in \mathbb{R}^{N^2}$ is decomposed as

$$\mathbf{z} = \begin{pmatrix} \widehat{\mathbf{z}}_1 \\ \widehat{\mathbf{z}}_2 \\ \vdots \\ \widehat{\mathbf{z}}_p \end{pmatrix} \in \mathbb{R}^{N^2} \quad , \qquad \text{where} \quad \widehat{\mathbf{z}}_i \in \mathbb{R}^{N^2/p}, \quad i = 1, 2, \ldots, p \ ,$$

is called a *block-component* of the vector $\mathbf{z}$. If p denotes the number of processors in a parallel computing environment with distributed memory, we assign the block-component $\widehat{\mathbf{z}}_i$ to the ith processor. On the assumption that scalars are stored at each processor, scalar operations and linear combinations of vectors are computed locally, i.e., the communication times of these operations satisfy

$$T_{\text{comm}}^{\text{scal oper}} = T_{\text{comm}}^{\text{lin comb}} = 0 \ . \tag{12}$$

Remark 3.2. The assumption that the order of the matrix N^2 is divisible by the number of processors p does not tend to restrict the numerical analyst in choosing his discretization in real applications at all. The number of grid points in one dimension N—and hence, the matrix order N^2—is typically very large in comparison to the number of processors available in today's parallel computing systems, i.e., $N^2 \gg p$. Therefore, if p does not divide N^2 the entire work to be done in $\mathbb{R}^{N^2}$ is generally dominated by the work of p processors each handling a subspace of dimension $N^2 \operatorname{div} p$. The remaining work due to a subspace of dimension $N^2 \mod p$ is negligible or can be suitably managed under load balancing considerations.

Parallel Computation of Inner Products

Assume that two vectors, $\mathbf{y}$ and $\mathbf{z}$, are distributed over p processors in the way just described. The inner product of these vectors is determined by

$$\mathbf{y}^T\mathbf{z} = \sum_{i=1}^{p} \widehat{\mathbf{y}}_i^T \widehat{\mathbf{z}}_i$$

and can be computed in two phases. The first phase of the parallel computation of an inner product is executed by all processors $i = 1, 2, \ldots, p$ simultaneously without communication. In this phase, a processor i calculates the inner product of the block-component $\widehat{\mathbf{y}}_i^T \widehat{\mathbf{z}}_i$. In the second phase, these p partial results have to be added by a global sum operation involving communication. Conceptually, the communication pattern used in the second phase of the evaluation of an inner product is known as a *reduction*. A reduction starts with a different value on each processor, here $\widehat{\mathbf{y}}_i^T \widehat{\mathbf{z}}_i$, and ends with a single value on each processor that is the result of applying any associative operation, here addition, on all the starting values. Thus, determining the communication time of an inner product is equivalent to finding the communication time of reducing the values $\widehat{\mathbf{y}}_i^T \widehat{\mathbf{z}}_i$.
Due to its global structure, a reduction involves communication times that strongly depend upon the way how processors are interconnected. Unless otherwise stated, we assume a two-dimensional mesh of $\sqrt{p}$ processors in both dimensions with wrap-around connections. Furthermore, it is assumed that a processor can send data on

only one of its ports at a time. Receiving data is only permitted on one port at a time as well. However, a processor can send and receive data simultaneously—either on the same port or on separate ports. Let us further assume that *cut-through routing* is used. In this routing scheme, the time to completely transfer a message of length l between two processors that are d connections away is given by $t_{\mathrm{s}} + lt_{\mathrm{w}} + (d-1)t_{\mathrm{h}}$, where $t_{\mathrm{s}}, t_{\mathrm{w}}$, and t_{h} are defined as follows. The startup time t_{s} is the time required to initiate a message transfer at the sending processor. This time occurs only once for every single message. The per-word time t_{w} is the time each word takes to traverse two directly-connected processors, i.e., t_{w} is equal to $1/b$ where b is the bandwidth of the communication channel. (It is supposed that an element of $\mathbb{R}$ requires one word of storage such that l corresponds to the number of elements of $\mathbb{R}$ that are transferred.) Finally, the per-hop time t_{h} is the time needed by the header of a message to travel between two directly-connected processors.
Under the above scenario, it is known [18] that the communication time of reducing messages each containing l words is

$$T_{\mathrm{comm}}^{\mathrm{red}} = 2\Big[(t_{\mathrm{s}} + lt_{\mathrm{w}})\log p + 2t_{\mathrm{h}}(\sqrt{p}-1)\Big] \ , \tag{13}$$

where the reduction is supposed to be performed in two phases as follows. Note that in the computation of an inner product, the messages to be reduced consist of a single word, namely $\widehat{\mathbf{y}}_i^T\widehat{\mathbf{z}}_i$, such that $l = 1$. In the first phase called *single-node accumulation*, the values $\widehat{\mathbf{y}}_i^T\widehat{\mathbf{z}}_i$ stored as partial results at every processor are combined through addition, and accumulated at a single destination processor. The second phase consists of reversing the directions and sequence of messages sending the final result $\mathbf{y}^T\mathbf{z}$ from this single processor to all other processors. The second phase is known as a *single-node broadcast*. To simplify the below discussion, we use

$$T_{\mathrm{comm}}^{\text{inn prod}} \approx 2\Big[(t_{\mathrm{s}} + t_{\mathrm{w}})\log p + 2t_{\mathrm{h}}\sqrt{p}\Big] \tag{14}$$

as an approximation of the communication time of an inner product hereafter.

Remark 3.3. Equation (14) serves as an input to the isoefficiency analysis of the next section. Although the discussion in this paper is confined to the special architecture described above, the isoefficiency analysis is easily adopted to other kinds of parallel architectures by considering corresponding communication times. For example, Gupta et al. [14] analyze the hypercube by using $T_{\mathrm{comm}}^{\text{inn prod}} = 2t_{\mathrm{s}}\log p$ instead of (14) and the fat-tree by simply ignoring any communication times incurred by inner products. Their work—like our discussion—assumes that reduction is performed by a single-node accumulation followed by a single node broadcast. But, hypercube algorithms for basic communication operations have been extensively investigated [7, 19] and it is known [18] that there is a faster way to perform reduction on a hypercube. Accustomizing the communication pattern of an all-to-all broadcast, the communication time of a reduction can be improved by a factor of two. Note that there is an even faster method useful for long messages to be reduced that is based on splitting messages into smaller parts; see [2] for details. Finally, we remark that for a two-dimensional mesh, the communication times of basic communication operations increase at most by a factor of four in the absence of wraparound connections [18].

Parallel Computation of Matrix-Vector Products

In this subsection, data is considered to be distributed over processors in two different ways. For both of these distributions, communication times of matrix-vector products are derived. The different results are used in the next section in order to compare these two distributions with respect to their effect on a certain model of parallel QMR-like iterative methods.

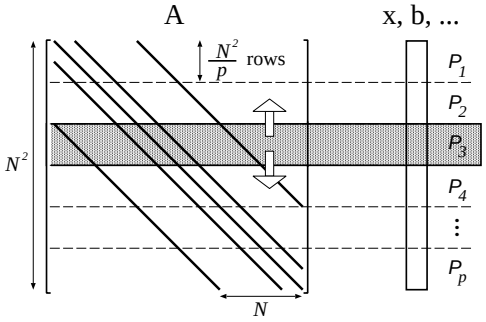

Figure 7: Simple stripe partitioning. A processor exchanges messages of length N to two neighbors.

The first mapping of data to processors is fairly straightforward and easy to implement. The coefficient matrix with its rows numbered from 1 to N^2 is partitioned among the processors as shown in Fig. 7. If we refer to N^2/p subsequent rows as *stripes* the matrix consists of p stripes that are stored at consecutive processors. So, the ith processor stores the nonzero elements of the ith stripe, i.e., processor P_i holds data of matrix rows numbered from $(i-1)\,N^2/p+1$ to $i\,N^2/p$. All vectors are distributed correspondingly. Note that the distribution of vectors in this partitioning scheme is exactly the one described above. We refer to this distribution of matrix and vectors as *simple stripe partitioning*.

If techniques that exploit the sparsity of the matrix are used, the communication time of a matrix-vector product depends on the structure of the matrix. Here, the structure of the matrix is such that while multiplying it with a vector, the ith component of the vector has to be multiplied with matrix elements of rows numbered $i, i+1, i-1, i+N$, and $i-N$. Thus, each component of the vector is required at locations that are at a distance of 1 and N on either side. The data transfer to perform a matrix-vector product depends upon the number of vector components each processor is storing. If we assume that the number of vector components per processor, N^2/p, is greater than or equal to the maximal distance that data has to travel during a matrix-vector product, N, then the required communication is accomplished by directly-connected processors. Thus, if $N \geq p$, a processor P_i communicates with P_{i-1} and P_{i+1} while performing a matrix-vector product. A communication step consists of exchanging N vector components at the boundaries of each processor. The communication time spent in a matrix-vector product therefore is

$$T_{\text{comm}}^{\text{mat-vec}} = 2(t_{\text{s}} + N t_{\text{w}} + t_{\text{h}}) \ , \qquad \text{if simple stripe partitioning}\,. \tag{15}$$

Note that the assumption $N \geq p$ is usually justified in parallel processing applications; cf. Remark 3.2.

Compared to the first partitioning scheme presented above, the second mapping of data to processors is less comprehensive and more difficult to implement. In contrast to the first partitioning scheme where the communication time of a matrix-vector product is simply derived by analyzing the mapping of matrix rows to processors, the second partitioning scheme is more easily investigated by considering the distribution of grid points to processors. We present both representations of the second partitioning scheme in Fig. 8. The top of this figure shows the partitioning of matrix rows, whereas the partitioning of grid points is depicted on the bottom.

As in simple stripe partitioning, each processor here is storing the nonzero elements

of N^2/p matrix rows. But they are not subsequently numbered throughout a processor. The N^2 rows of the coefficient matrix are packaged into $N\sqrt{p}$ stripes each consisting of $N/\sqrt{p}$ consecutive matrix rows. These stripes are cyclically distributed over $\sqrt{p}$ subsequent processors each. To see where this partitioning scheme stems from, consider the finite difference grid and recall that each interior grid point exactly contributes one matrix row. If each processor stores a square subsection of $N/\sqrt{p}$ interior grid points in both dimensions this partitioning of grid points corresponds to the partitioning of matrix rows just described. The vector components are mapped to processors such that a processor stores the ith component only if the ith matrix row is stored at this processor. We refer to this scheme as *cyclic stripe partitioning*.

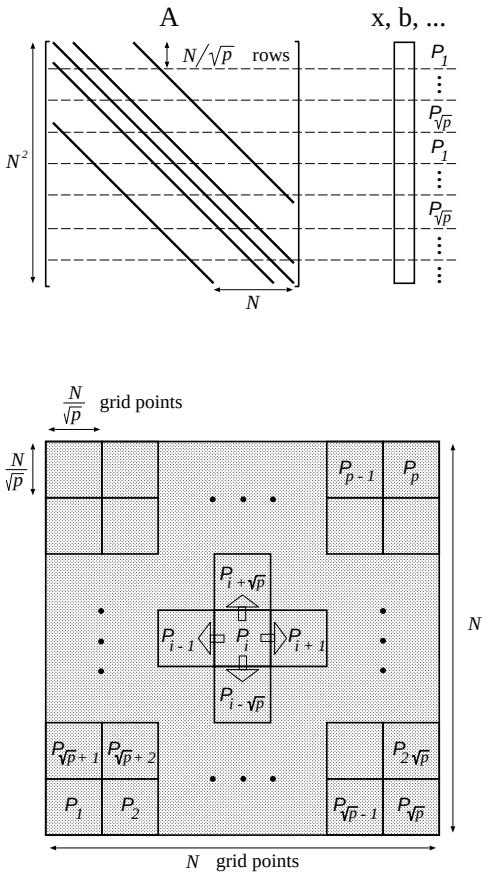

Figure 8: Cyclic stripe partitioning in terms of matrix rows (top) and grid points (bottom). A processor exchanges messages of length $N/\sqrt{p}$ to four neighbors.

Note that in simple stripe partitioning as well as in cyclic stripe partitioning, each processor stores N^2/p matrix rows and the corresponding vector components. The two schemes differ in which rows a processor is storing. But this difference does not affect the above communication times derived for linear combinations and inner products, i.e., (12) and (14) hold for both partitioning schemes.

The representation in terms of the finite difference grid can be used to derive the communication time of a matrix-vector product using cyclic stripe partitioning. A processor P_i that is not located at the grid boundary needs to exchange data with each of his four directly-connected neighboring processors $P_{i+\sqrt{p}}$, $P_{i-\sqrt{p}}$, P_{i-1}, and P_{i+1}. Processors located at the grid boundary communicate with less than four neighboring processors. Each data transfer consists of exchanging $N/\sqrt{p}$ vector components. Hence, the communication time to perform a matrix-vector product is

$$T_{\text{comm}}^{\text{mat-vec}} = 4\left(t_{\text{s}} + \frac{N}{\sqrt{p}}t_{\text{w}} + t_{\text{h}}\right) \quad , \qquad \text{if cyclic stripe partitioning} \, . \tag{16}$$

4 Isoefficiency Analysis

This section presents a concept modeling the scalability of a parallel algorithm on a parallel computer. The concept is used to analyze a single iteration step of a parallel QMR-like iterative method.

Isoefficiency Concept

Ignoring problems arising from the introduction of hierarchical caches, sequential algorithms traditionally are evaluated in terms of their execution times. The sequential execution time is usually expressed as a function of a free variable called

problem size. In this paper, we are concerned with QMR-like iterative methods. Since it is not known in advance how many iteration steps a method needs to converge, we do not consider the whole algorithm until its termination but consider a single iteration step and take N, the number of grid points in one dimension, as the problem size. As mentioned in the previous section, the execution time of the fastest known sequential algorithm to perform a single QMR-like iteration is

$$T_{\text{seq}}(N) = cN^2\, t_{\text{a}} = \Theta(N^2) \ , \tag{17}$$

where c is a constant and t_{a} is the time required to perform an arithmetic operation*. Things are more difficult in the context with parallel computing. The execution time of a parallel algorithm T_{par} depends not only on the problem size N but also on the number of processors p and the particular architecture on which it is implemented. Kumar et al. [18] have introduced an *isoefficiency* concept relating the execution time of the fastest known sequential algorithm to the number of processors needed to maintain a fixed efficiency while taking into account both, parallel algorithm and architecture. The goal is to evaluate the *scalability* of a parallel algorithm on an architecture, i.e., its ability to achieve performance proportional to the number of processors.

For the motivation of the isoefficiency concept, we briefly state the conventional definitions of speedup and efficiency. The *speedup* S is defined as the ratio of the time to solve a problem on a single processor using the fastest known sequential algorithm to the time required to solve the same problem on a parallel computer, $S = T_{\text{seq}}/T_{\text{par}}$. The *efficiency* E is defined as the ratio of the speedup to the number of processors, $E = S/p$. The optimal speedup is equal to p and the corresponding efficiency is equal to one. The isoefficiency concept is motivated by the following two observations:

- With a fixed problem size—and thus, with a fixed execution time of the fastest known sequential algorithm—the speedup does not increase with increasing number of processors but tends to saturate. Hence, efficiency decreases.
- With a fixed number of processors, the speedup increases by executing the algorithm on larger problems, i.e., by increasing the problem size and T_{seq} respectively. Thus, efficiency increases.

Therefore, one can expect to keep efficiency constant by allowing T_{seq} to grow properly with increasing number of processors. The rate at which T_{seq} has to be increased with respect to the number of processors to maintain a fixed efficiency can serve as a measure of scalability.

Algorithm implementations on real parallel computers do not achieve optimal speedup. For example, data communication delays and synchronization are reasons for nonoptimal speedup. All causes of dropping the theoretically ideal speedup are called overhead and the *total overhead function* is formally defined as

$$T_{\text{over}}(N,p) = p\, T_{\text{par}}(N,p) - T_{\text{seq}}(N) \ ,$$

i.e., that part of the total time spent in solving a problem summed over all processors $p\, T_{\text{par}}$ that is not incurred by the fastest known sequential algorithm T_{seq}. So,

*The notation $f(N) = \Theta\big(g(N)\big)$ is used to denote the fact that the function g is an asymptotically tight (upper and lower) bound for the function f. More formally, a function $f(N)$ belongs to the set $\Theta\big(g(N)\big)$ if there exists positive constants c_1, c_2, and N_0 such that the inequalities $c_1|g(N)| \le |f(N)| \le c_2|g(N)|$ hold for all $N \ge N_0$.

the efficiency can be expressed as a function of the total overhead and the execution time of the fastest known sequential algorithm

$$E = \frac{S}{p} = \frac{T_{\text{seq}}(N)}{p\,T_{\text{par}}(N,p)} = \frac{T_{\text{seq}}(N)}{T_{\text{seq}}(N) + T_{\text{over}}(N,p)} = \frac{1}{1 + T_{\text{over}}(N,p)/T_{\text{seq}}(N)} \quad . \tag{18}$$

Given T_{over} as well as T_{seq}, this equation determines the efficiency of a parallel algorithm on an architecture for a chosen problem size N and a number of processors p. But this equation can be interpreted in another way. For fixed T_{seq} and N respectively, efficiency decreases with increasing p because T_{over} usually increases with p. Keeping p fixed and provided that T_{over} grows slower than $\Theta(T_{\text{seq}})$, the efficiency increases with increasing T_{seq} and N respectively. Thus, efficiency can be maintained at a desired value if p and T_{seq} are both increased appropriately. If such behavior is possible, i.e., the ratio $T_{\text{over}}/T_{\text{seq}}$ remains fixed, the combination of an algorithm and the parallel architecture on which it is implemented is called a *scalable parallel system.*

The rate with respect to p at which T_{seq} has to be increased to keep efficiency constant is used to asses the quality of a scalable parallel system. For example, if T_{seq} has to be increased as an exponential function of p to maintain efficiency fixed, the system is poorly scalable. A system is highly scalable if one only has to linearly increase T_{seq} with respect to p. Such growth rates can be calculated from (18) or equivalently from

$$T_{\text{seq}}(N) = \frac{E}{1-E}\, T_{\text{over}}(N,p) \quad , \tag{19}$$

where E is the desired efficiency to be maintained. Rather than deriving a growth rate of T_{seq} with respect to p yielding an *isoefficiency function* [18], we are concerned with analyzing how the problem size N has to be increased with respect to p to keep the efficiency from dropping. The task is therefore to solve (19) for N as a closed function of p.

Isoefficiency Analysis of a Single QMR-Like Iteration Step

We carry out the isoefficiency analysis for a single QMR-like iteration step. To calculate growth rates from (19), we need to know T_{seq} and T_{over} of a single QMR-like iteration step. The execution time of the fastest known sequential algorithm is given by (17). The total overhead is solely due to communication times, i.e., $T_{\text{over}} = p\,T_{\text{comm}}$. Since (12) shows that scalar operations and linear combinations of vectors incur no communication times, the total overhead is of the form

$$T_{\text{over}} = sp\,T_{\text{comm}}^{\text{inn prod}} + mp\,T_{\text{comm}}^{\text{mat-vec}} \quad , \tag{20}$$

where it is assumed that s inner products and m matrix-vector products are computed in a single QMR-like iteration step. Notice that for the algorithm given in Fig. 6, the relations $s = 4$ and $m = 2$ hold.

The last section introduced two different mappings of data to processors, simple stripe partitioning and cyclic stripe partitioning. These two schemes are now compared using an isoefficiency analysis. For both schemes, the communication time of an inner product is given by (14). The communication time of a matrix-vector product is different in each of the two partitioning schemes according to (15) and (16).

Inserting (14)–(16) into (20) and grouping terms of different orders of magnitude with respect to p yields

$$T_{\text{over}} = \begin{cases} 2mt_{\text{w}}Np + 2m(t_{\text{s}} + t_{\text{h}})p + 2s(t_{\text{s}} + t_{\text{w}})p\log p + 4st_{\text{h}}p^{3/2} \\ \qquad\qquad \text{if simple stripe partitioning} \ , \\ 4mt_{\text{w}}N\sqrt{p} + 4m(t_{\text{s}} + t_{\text{h}})p + 2s(t_{\text{s}} + t_{\text{w}})p\log p + 4st_{\text{h}}p^{3/2} \\ \qquad\qquad \text{if cyclic stripe partitioning} \ . \end{cases} \tag{21}$$

Since (21) shows that, for fixed p, the total overhead function T_{over} grows slower than $\Theta(T_{\text{seq}}) = \Theta(N^2)$, a single QMR-like iteration step implemented on a two-dimensional mesh is a scalable parallel system. Therefore, it is possible to calculate growth rates for fixed efficiency from (19) by inserting (17) and (21). It is not hard to solve (19) for N as a closed function of p. The task is just to solve a quadratic equation with respect to N. However, we will apply a different technique that easily shows the asymptotic growth rate and that is applicable to more complex isoefficiency analyses as well. If the overhead function consists of multiple additive terms, we solve (19) for N considering each term of the overhead function individually. The term of the overhead function that requires N to grow at the highest rate with respect to p determines the overall asymptotic behavior of the parallel system.
We illustrate this technique for each of the two data mappings beginning with simple stripe partitioning. Using only the first term of (21), we need to solve

$$cN^2\,t_{\text{a}} = \frac{E}{1-E}\,2mt_{\text{w}}Np$$

for N and get

$$N = \frac{2mt_{\text{w}}E}{c\,t_{\text{a}}(1-E)}\,p = \Theta(p) \ .$$

Considering all remaining terms of (21), the task is to solve

$$cN^2\,t_{\text{a}} = \frac{E}{1-E}\Big[2m(t_{\text{s}} + t_{\text{h}})p + 2s(t_{\text{s}} + t_{\text{w}})p\log p + 4st_{\text{h}}p^{3/2}\Big]$$

for N. Since none of the terms on the right hand side depends on N the asymptotic rate is determined by the last term that has the highest order of magnitude with respect to p. Hence,

$$N = \sqrt{\frac{4st_{\text{h}}E}{c\,t_{\text{a}}(1-E)}}\,p^{3/4} = \Theta(p^{3/4}) \ . \tag{22}$$

To keep the efficiency fixed at a value E, the problem size N has to grow as $\Theta(p)$ considering only the first term of the overhead function and as $\Theta(p^{3/4})$ examining all remaining terms. The asymptotically higher of the two rates is responsible for the overall asymptotic behavior $N = \Theta(p)$.
For cyclic stripe partitioning, the consideration of only the first term of the overhead function gives

$$cN^2\,t_{\text{a}} = \frac{E}{1-E}\,4mt_{\text{w}}N\sqrt{p}$$

which leads to

$$N = \frac{4mt_{\mathrm{w}}E}{c\,t_{\mathrm{a}}(1-E)}\,\sqrt{p} = \Theta(\sqrt{p}) \ .$$

Using all remaining terms yields the same result as in (22) that gives the overall asymptotic behavior for cyclic stripe partitioning. To summarize the above analysis for both mappings of data to processors, the problem size has to be increased as

$$N = \begin{cases} \dfrac{2mt_{\mathrm{w}}E}{c\,t_{\mathrm{a}}(1-E)}\,p & = \Theta(p) & \text{if simple stripe partitioning ,} \\ \sqrt{\dfrac{4st_{\mathrm{h}}E}{c\,t_{\mathrm{a}}(1-E)}}\,p^{3/4} & = \Theta(p^{3/4}) & \text{if cyclic stripe partitioning} \end{cases} \tag{23}$$

to maintain the efficiency fixed at a desired value E. If we compare the two different mappings of data to processors in the isoefficiency concept, we conclude that cyclic stripe partitioning is superior to simple stripe partitioning. To keep the efficiency fixed, the number of grid points in one dimension N only has to be increased as $\Theta(p^{3/4})$ using cyclic stripe partitioning whereas it takes a higher growth rate of $\Theta(p)$ implementing simple stripe partitioning. Thus, cyclic stripe partitioning is asymptotically more scalable.

Remark 4.1. In this paper, we calculate growth rates of the problem size N required to keep the efficiency fixed as the number of processors p increases. Since the execution time of the fastest known algorithm T_{seq} is a function of the problem size N, it is possible to perform an isoefficiency analysis with respect to T_{seq} instead of N. The growth rate of T_{seq} required to keep efficiency constant as p increases is called *isoefficiency function*. Using (17) the isoefficiency function corresponding to (23) is given by

$$T_{\mathrm{seq}} = \begin{cases} \dfrac{1}{c\,t_{\mathrm{a}}}\left(\dfrac{2mt_{\mathrm{w}}E}{1-E}\right)^2 p^2 & = \Theta(p^2) & \text{if simple stripe partitioning ,} \\ \dfrac{4st_{\mathrm{h}}E}{1-E}\,p^{3/2} & = \Theta(p^{3/2}) & \text{if cyclic stripe partitioning .} \end{cases} \tag{24}$$

Unfortunately, Kumar et al. [18] refer to T_{seq} as the "problem size". Since this terminology may lead to confusions with the conventional definition of problem size as a free parameter of the input size, we avoid their terminology.

5 Implications on Parallel Algorithm Design

The aim of the isoefficiency analysis given in the last section is to provide insight in the general behavior of parallel QMR-like iterative methods, not to give detailed performance predictions with high accuracy. It is therefore justified to consider asymptotic growth rates. In this section, it is examined how the isoefficiency analysis helps in designing improved parallel algorithms.

Minimizing communication times

Two different mappings of data to processors are compared above within the isoefficiency concept. Here, we focus on cyclic stripe partitioning that is asymptotically

more scalable than simple stripe partitioning. To maintain efficiency at a desired value E, (23) reveals that the number of grid points in one dimension N has to be increased as $\Theta(p^{3/4})$ with leading coefficient of the highest-order term

$$\sqrt{\frac{4st_{\mathrm{h}}E}{c\,t_{\mathrm{a}}(1-E)}}\ .$$

In this expression, t_{a} and t_{h} are given hardware parameters of the parallel architecture, the time required to perform an arithmetic operation and the per-hop time respectively. The constant c is defined in (11) by the fastest known sequential algorithm to perform a single iteration step and cannot be tuned either. The only parameter that can serve to decrease the leading coefficient from the parallel algorithm designer's point of view is s, the number of inner products calculated in a single iteration step. Since one cannot expect to save any inner product in a parallel algorithm compared to a serial method, the straightforward idea is to minimize communication times due to inner products.
Suppose that all s inner products of a single iteration step are independent of each other. Note that this assumption is generally not true. For example, there are only two independent inner products out of $s = 4$ in the QMR method depicted in Fig. 6. The computation of ρ_{n+1} is independent from the calculation of ξ_{n+1}. But computing ϵ_n depends via $\mathbf{p}_n$ and $\mathbf{q}_n$ on the value of δ_n. The question is whether the isoefficiency analysis shows an advantage if all s inner products are independent. In this case, all s inner products can be computed simultaneously as follows. The s inner products of the block-components are computed first without any communication. These values are globally added by a vector reduction meaning that the vector formed by these s values is reduced componentwise. This procedure is implemented by a single (vector) operation using the communication pattern of a (scalar) reduction with messages of length s. Hence, the expression to subsequently calculate s inner products that was used in the last section, namely

$$sT_{\mathrm{comm}}^{\text{inn prod}} = 2s\Big[(t_{\mathrm{s}} + t_{\mathrm{w}})\log p + 2t_{\mathrm{h}}\sqrt{p}\,\Big]$$

can be replaced by

$$2\Big[(t_{\mathrm{s}} + st_{\mathrm{w}})\log p + 2t_{\mathrm{h}}\sqrt{p}\,\Big]\ , \tag{25}$$

where (13) with $l = s$ as well as the corresponding approximation leading to (14) are used. Inserting (25) into the overhead function (20) instead of $sT_{\mathrm{comm}}^{\text{inn prod}}$ and carrying out an isoefficiency analysis yields

$$N = \sqrt{\frac{4t_{\mathrm{h}}E}{c\,t_{\mathrm{a}}(1-E)}}\,p^{3/4}\ .$$

So, the result with independent inner products demonstrates the same growth rate $\Theta(p^{3/4})$ as with dependent inner products. But the leading coefficient of the highest-order term is decreased by a factor of $\sqrt{s}$. In the isoefficiency concept, it is therefore advantageous to design parallel QMR-like iterative methods such that all inner products are independent.
A further result of the isoefficiency analysis is the following. The leading coefficient of the highest-order term does not depend on m, the number of matrix-vector products in a single iteration step. Consequently, reducing communication times due to

matrix-vector products does not significantly help to improve scalability of parallel QMR-like iterative methods on two-dimensional mesh-based computers. Although highly-efficient implementations should consider this aspect too, it is more promising to think about making all inner products independent.

Remark 5.1. In this section, we concentrate on cyclic stripe partitioning because it is asymptotically more scalable than simple stripe partitioning. But let us consider simple stripe partitioning in this remark. Reflecting the result of simple stripe partitioning in (23) and transferring m a similar role as s plays above, one might wonder that the leading coefficient of the highest-order term can be decreased by reducing communication times of matrix-vector products. But this is not true. If we suppose m independent matrix-vector products we can replace

$$2m(t_{\mathrm{s}} + Nt_{\mathrm{w}} + t_{\mathrm{h}})$$

used in the analysis given in the last section by

$$2(t_{\mathrm{s}} + mNt_{\mathrm{w}} + t_{\mathrm{h}}) \ .$$

Repeating the isoefficiency analysis with this expression used in the overhead function (20) rather than $mT_{\mathrm{comm}}^{\text{mat-vec}}$ yields exactly the same result as in (23). Moreover, if we additionally assume that a single iteration step consists of s independent inner products the outcome of a corresponding isoefficiency analysis does not change either. So, simple stripe partitioning is not only asymptotically poor but it cannot be significantly improved by the independence of neither matrix-vector products nor inner products. Note that load balancing considerations are not considered in this paper.

Remark 5.2. Another way of improving parallel QMR-like iterative methods is trying to overlap all communication times with useful computation. This is usually managed by restructuring stable serial algorithms that is often more simpler than designing satisfactorily fast and stable parallel algorithms. Overlapping is not addressed here; see [8] and the references therein for details.

Remark 5.3. This paper models a single iteration step of parallel QMR-like iterative methods using the isoefficiency concept. Having fixed the main ideas of this paper, the report of Gupta et al. [14] came to the author's knowledge. The report carries out an isoefficiency analysis of the conjugate gradient method [16] for the solution of symmetric positive definite systems of linear equations. The conjugate gradient method falls into the class of those iterative methods that are investigated in this paper. For parallel computers with two-dimensional mesh topology and cyclic stripe partitioning, the report [14] derives the same asymptotic behavior. In contrast to this paper, the report [14] does not contain an analysis for simple stripe partitioning but it does consider truncated incomplete Cholesky preconditioning as well as other parallel computer architectures; cf. Remark 3.3.
Another model of parallel iterative methods is proposed in [4, 5, 6]. This work examines the running time of a single iteration step on parallel two-dimensional mesh-based computers. It is more general than the isoefficiency analysis given here and in [14] in the sense that it allows modeling of methods whose number of arithmetic operations in one iteration step is not constant, e.g., GMRES [20] is analyzed. Furthermore, overlapping communication with computation is permitted. But it does not take into account any communication times during a matrix-vector product. The overhead due to this kind of operation is simply ignored.

Both models are not intended to give accurate performance predictions; but they both show that the performance of parallel QMR-like iterative methods is severely influenced by the computation of inner products for a large number of processors. So, it is worthwhile designing algorithms where inner products can be calculated independently. For QMR, this is done in [3].

6 Summary

An electrostatic problem mathematically described by a linear partial differential equation of second order with Dirichlet boundary conditions is discretized using centered finite difference approximations. The resulting system of linear equations is put into matrix-vector form by natural ordering of the unknowns. For the solution of this linear system with nonsymmetric coefficient matrix, a class of iterative methods is analyzed that is defined by exclusively consisting of the following four kinds of operations: scalar operations, linear combinations of vectors, inner products, and matrix-vector products. The communication times for each of these four kinds of operations are derived on a parallel computer with two-dimensional mesh topology. Two different mappings of data to processors are compared by putting these communication times into the isoefficiency concept that tries to model scalability aspects. Out of these two mappings, the strategy distributing matrix rows cyclically to processors is shown to be the asymptotically more scalable one. Using this mapping strategy, performance can be improved by designing parallel algorithms such that all inner products are independent.

7 Acknowledgements

It is a pleasure to thank the organizing committee for providing an excellent workshop environment. Finally, I would like to express my gratitude to F. Hoßfeld, M. Sauren, C. Schelthoff, and P. Weidner. Their numerous and valuable comments significantly helped to improve this paper's quality.

References

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1993.

[2] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice–Hall, Englewood Cliffs, 1989.

[3] H. M. Bücker and M. Sauren. A Parallel Version of the Unsymmetric Lanczos Algorithm and its Application to QMR. Internal Report KFA–ZAM–IB–9605, Research Centre Jülich, Jülich, Germany, March 1996.

[4] E. de Sturler. A Parallel Variant of GMRES(m). Reports of the Faculty of Technical Mathematics and Informatics 91–85, Delft University of Technology, Delft, The Netherlands, 1991.

[5] E. de Sturler. A Performance Model for Krylov Subspace Methods on Mesh–based Parallel Computers. Technical Report CSCS–TR–94–05, Swiss Scientific Computing Center, CH–6928 Manno, Switzerland, May 1994.

[6] E. de Sturler and H. A. van der Vorst. Reducing the Effect of Global Communication in GMRES(m) and CG on Parallel Distributed Memory Computers. Technical Report 832, University of Utrecht, Utrecht, The Netherlands, October 1993.

[7] E. Dekel, D. Nassimi, and S. Sahni. Parallel Matrix and Graph Algorithms. *SIAM Journal on Computing*, 10(4):657–675, 1981.

[8] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel Numerical Linear Algebra. In *Acta Numerica 1993*, pages 111–197. Cambridge University Press, Cambridge, 1993.

[9] R. W. Freund, G. H. Golub, and N. M. Nachtigal. Iterative Solution of Linear Systems. In *Acta Numerica 1992*, pages 1–44. Cambridge University Press, Cambridge, 1992.

[10] R. W. Freund and N. M. Nachtigal. QMR: A Quasi–Minimal Residual Method for Non–Hermitian Linear Systems. *Numerische Mathematik*, 60(3):315–339, 1991.

[11] R. W. Freund and N. M. Nachtigal. An Implementation of the QMR Method Based on Coupled Two–Term Recurrences. *SIAM Journal on Scientific Computing*, 15(2):313–337, 1994.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness*. Freeman, San Francisco, 1979.

[13] G. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Academic Press, San Diego, 1993.

[14] A. Gupta, V. Kumar, and A. Sameh. Performance and Scalability of Preconditioned Conjugate Gradient Methods on Parallel Computers. Technical Report TR 92–64, Department of Computer Science, University of Minnesota, Minneapolis, MN – 55455, November 1992. Revised April 1994.

[15] M. T. Heath, E. Ng, and B. W. Peyton. Parallel Algorithms for Sparse Linear Systems. *SIAM Review*, 33(3):420–460, 1991.

[16] M. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[17] J. D. Jackson. *Classical Electrodynamics*. Wiley, New–York, second edition, 1975.

[18] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, 1994.

[19] O. A. McBryan and E. F. Van de Velde. Hypercube Algorithms and Implementations. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s227–s287, 1987.

[20] Y. Saad and M. H. Schulz. GMRES: A Generalized Minimum Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[21] H. R. Schwarz. *Numerische Mathematik*. B.G. Teubner, Stuttgart, 3. Auflage, 1993.

[22] M. Yannakakis. Computing the Minimum Fill–In is NP–Complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, 1981.