TASK QUARTERLY 2 No 4 (1998), 611–731

METHODS OF SOLVING OPERATOR EIGENPROBLEMS IN PARALLEL DISTRIBUTED MEMORY SYSTEMS AS APPLIED IN ELECTROMAGNETICS[1]

MICHAŁ REWIEŃSKI

Faculty of Electronics, Telecommunications and Computer Science
Technical University of Gdansk
Narutowicza 11/12, 80-952 Gdansk, Poland
mrewiens@task.gda.pl

Abstract: This study presents numerical methods of solving operator eigenproblems, focusing on their applications in electromagnetics. The discussion concentrates on the analysis of new iterative algorithms or modifications of the existing ones, which are capable of finding a few eigenvalues from the point spectrum of a non-symmetric operator. The salient feature of the considered methods is a low computational cost and memory complexity as compared to alternative solutions. This paper also presents implementations of the investigated algorithms in parallel distributed memory systems, based on the message-passing parallel programming model and providing portable parallel eigensolvers. The discussion of the applied designs of the parallel algorithms is supported by the presentation of the results of performance tests in selected distributed memory environments, including scalable parallel supercomputer systems and networks of workstations. The results of these tests confirm high efficiency of the eigensolvers in the considered parallel environments. In this study attention is also drawn to the question of the applicability of the eigensolvers to problems of modelling of electromagnetic fields in dielectric waveguides. The results of numerical tests validating the methods in these applications determine the scope of problems which may be most effectively solved using the specific eigensolvers.

Keywords: operator eigenproblems, parallel numerical methods, computational electromagnetics, boundary value problems

1. Introduction

1.1 Motivation and background

The rapid development of the methods of functional analysis has brought about important changes in the mathematical treatment of a huge number of both theoretical and engineering problems from various disciplines of science. With revolutionary changes occurring in physics, related in general to the success of wave mechanics and the theory of quanta, the operator formalism has gained very special importance. Solving operator eigenproblems, i.e. finding eigenvalues and eigenfunctions of given operators and investigating spectral properties of entire classes of operators have become a central issue addressed by the mathematical physics. The classical eigenproblems formulated in physics, such as the famous

[1] This paper is entirely based on the author's M. Sc. Thesis

Schrödinger equation, have served to solve numerous problems including e.g. finding eigenfrequencies of the electron waves in an atom or determining energy eigenstates in various quantum systems. The extremely dynamic progress in the mathematical methods of theoretical physics has resulted in adopting the operator formalism to many scientific fields beyond quantum physics, including classical optics, solid state physics and the theory of electromagnetic field. Throughout the years the mathematical formulations involving integral or differential operator equations and eigenproblems have become dominant in such disciplines as the theory of optical resonators and lasers or the theory of guided electromagnetic waves ([1], [2]) and have served to solve such problems as investigating fluctuations of amplitude of laser oscillations or describing modes of electromagnetic field in dielectric waveguides.

Application of the operator formalism in a broad spectrum of research areas of science and engineering has generated the necessity of further intensive developments in the methods of solving operator eigenproblems. It has turned out that the analytical methods of operator calculus, which were initially most widely used, are capable of dealing only with the simplest systems and operators. Consequently the growing need for solving more complex problems formulated in terms of operator equations has resulted in the development of approximate, numerical methods of functional analysis.

Several classical techniques, including variational calculus [3] or basic iterative methods, have been investigated and successfully applied to various operator problems. Soon, a clear division in the development of approximate methods of functional analysis has emerged, related to investigation of either methods valid for general linear operators or methods which could only be applied to finite-dimensional linear operators, namely the numerical methods of linear algebra.

The latter research area has gained a very special importance with the advent of computers, which enabled one to model physical systems of an unprecedented scale of complexity. A very large number of novel numerical algorithms of solving eigenproblems for finite-dimensional linear operators (matrix operators) have been developed. At the same time many classical approaches toward solving eigenproblems for matrix operators had to be rejected or at least revised due to the problems with their application in computer-based calculations, related e.g. to the numerical instability of the algorithms which was causing unacceptable accumulation of errors while using floating-point arithmetics. Still, by the mid-eighties, the developments in scientific computing enabled one to solve routinely matrix operator eigenproblems of order a hundred or, at most, a few hundreds using available algorithms and hardware platforms. At the same time the efforts to standardize the linear algebra computer algorithms were undertaken and resulted in the development of portable libraries of subroutines, such as EISPACK (the package of Fortran77 routines for solving symmetric and non-symmetric matrix operator eigenproblems), BLAS (Basic Linear Algebra Subprograms) or LAPACK (Linear Algebra Package), providing very efficient implementations of various

algorithms from numerical linear algebra. This successful standardization has greatly contributed to broadening application of the numerical methods of linear algebra in scientific modelling and popularization of computer modelling in general.

The success of the numerical methods of linear algebra applied to solving eigenproblems of finite-dimensional linear operators has caused a rapid development of strategies or techniques of discretization (finite-dimensional mapping) of infinite-dimensional linear operators. In this way the methods of numerical linear algebra could have been applied to solving eigenproblems for a much broader class of operators, including e.g. all integral and differential operators playing a crucial role in many scientific and engineering applications. While investigating the finite-dimensional mapping methods several problems had to be taken into account, including quality of the approximation of a given infinite--dimensional linear operator by a finite matrix operator and the cost of the applied finite representation. The issues directly related to the cost of this representation are: 1) the size of the problem for the emerging finite-dimensional operator and 2) the characteristics of the representation which often determines a method to be applied to solve the discrete problem. The problem of finding efficient, cost reducing finite-dimensional mapping strategies for certain operators or classes of operators is continually one of the most up-to-date problems in modern scientific computing and has a colossal impact on the scope of possible applications of the specific numerical algorithms to solving operator eigenproblems.

The introduction of parallel, multiprocessor computer architectures in the recent years has caused another revolution in scientific computing, affecting also the approach towards numerical solving of operator and matrix eigenproblems. The computational power of multiprocessor systems, especially scalable parallel distributed memory systems, offering now the peak performance of order of gigaflops or even teraflops could have been efficiently exploited only if the algorithms had made use of the characteristics of these systems and had taken into account various additional design problems. These additional problems which have been found to have a substantial impact on the efficiency of an algorithm executed in a parallel environment refer mainly to balancing the workload and minimizing data interchange across a large number of processors. In order to deal with these issues many existing sequential algorithms have had to be redesigned and many new, inherently parallel methods have had to be introduced. In this way designing parallel algorithms which could be efficiently implemented in scalable parallel systems has become one of extremely important and challenging issues, making numerical methods once again a field of intensive research.

In the mainstream of the current research efforts aiming at designing more efficient (parallel) numerical algorithms one may encounter the dynamically developing field of computational electromagnetics which deals with numerical techniques suitable for electromagnetic applications. This research field has emerged as a response to growing computational needs generated by the

electromagnetic community and related to modelling more complex electromagnetic systems. Moreover computational electromagnetics has started to address issues in numerical modelling requiring application of non-standard numerical tools and methods which resulted in developing specialized versions of general algorithms e.g. for solving operator eigenproblems. An overview of currently investigated methods of computational electromagnetics applied to problems of modelling electromagnetic fields in waveguiding structures may be found in [4]. Different approaches towards computational problems of electromagnetics aiming at decreasing the numerical cost and memory complexity of the algorithms are presented in the quoted paper and include e.g. finding more efficient, cost reducing finite-dimensional mapping strategies for the investigated systems and operators or developing parallel solvers for the considered electromagnetic problems.

1.2 Scope and goal of this work

This study tries to join the mainstream of current investigations in the fields of computational electromagnetics and numerical methods. The approach towards designing numerical algorithms presented in this work concentrates on choosing cost-efficient solutions which offer low computational cost and storage requirements as compared to orthodox algorithms. Consequently, the priority is given to iterative methods and finite-dimensional mapping techniques which provide most efficient schemes of solving operator eigenproblems. Although the reduction in computational cost is often achieved at a price of limiting generality of the proposed algorithms, the designs presented in this study try to remain suitable for solving relatively broad classes of computational problems. An important approach towards reducing computation time, also widely discussed in this study, is the strategy of parallelization, which aims at obtaining scalable algorithms. In this context another main point addressed within this study clearly emerges, i.e. designing algorithms suitable for solving large-scale operator eigenproblems.

Having briefly presented basic ideas and approaches applied in this study let us now outline its scope. This work focuses on a class of iterative algorithms capable of finding one or several eigenvalues from the point spectrum of a non-symmetric linear operator. The iterative algorithms have been selected so that they can be applied to large-scale problems, with matrices of the order of thousands or more, as opposed to the methods based solely on matrix transformations (direct methods) whose applicability is limited to relatively small problems. Moreover, in most of the scientific or engineering problems it is necessary to find only one or a few eigenvalues e.g. those with the largest real part or the largest modulus, rather than the entire point spectrum of the operator. In this context the iterative methods seem to be the only suitable tool.

Another aspect of solving operator eigenproblems lying within the scope of this work is the question of finite-dimensional mapping of infinite-dimensional operators. Although the solutions presented in this work are general and may be applied to operators with different domains, special attention is dedicated to problems involving

two-dimensional fields. The main reason for this choice is that these functions have an application in the modelling of electromagnetic waveguiding structures, being of the author's particular interest. Moreover, the non-symmetric operators and eigenproblems appearing in the theory of electromagnetic fields and waveguides have served as main examples used to validate the presented algorithms.

Last but not least, the implementor's point of view has also been thoroughly discussed in this work with the central question of the design of parallel algorithms, considered jointly with selected programming paradigms and system architectures. The work has focused on distributed memory parallel systems which, due to the scalability of the architecture, seem to be best suited for "grand challenge" computational problems requiring the largest available memory and processing resources. Although the major attention is dedicated to massively parallel processing with parallel supercomputers, some consideration is also given to "poor man's' supercomputers" , namely the networks of workstations in order to discuss the portability of both designs and implementations of the parallel solvers. Referring to programming paradigms, this study concentrates on the most popular one — the message-passing programming model, offering both greatest versatility in parallel design as well as high performance.

With the scope of interest presented above the following main goals of this work clearly emerge:

- Present iterative methods which can be used to solve eigenproblems for a general class of non-symmetric linear operators and discuss the role of some original modifications of these methods.
- Propose discretization schemes for infinite-dimensional operators, with a special attention paid to differential operators arising in electromagnetics.
- Present some new methods of solving operator eigenproblems based on the discussed iterative processes and discretization strategies and highlight their advantages and limitations.
- Describe the parallel designs of the presented methods and discuss their applicability in distributed memory parallel systems.
- Investigate the aspects of numerical complexity of the algorithms and their efficiency in given parallel environments.
- Validate the algorithms by showing their application to solving boundary value problems arising in the theory of electromagnetic waves and assess the scope of problems which may be most effectively solved using the specific eigensolvers.

1.3 Section outline

This work starts with an introduction (in Section 2) of some basic concepts concerning operator eigenproblems and presentation of operators arising in selected electromagnetic applications, followed by the description of the two iterative

algorithms of solving operator eigenproblems. The description of the algorithms is based on the papers by Sorensen [11], Jabłoński [16], [17] and Mrozowski [18]. The author's extensions to the presented algorithms involve describing deflation procedures in the Iterative Eigenfunction Expansion Method and discussing matrix formulation for this method (Appendix A). Section 3 presents methods of discretization of infinite-dimensional linear operators, based on the Finite Difference technique and the Method of Moments representation. The discussed design of the fast method of calculating inner product for the operators and functions applying the Method of Moments representation has been first proposed by Mrozowski [18]. This section also presents the two-dimensional analogues of theorems published in a book by Briggs and Henson [33] and concerning estimation of numerical errors in calculation of Fourier coefficients using the Discrete Fourier Transform. Section 4 presents a general discussion of issues concerning programming aspects in distributed memory systems, based on the material from a book by Foster [29]. Section 5 concentrates on describing original parallel designs and / or implementations of the algorithms of solving operator eigenproblems. Also a complete description of a new eigensolver (based on the IRAM algorithm and implicit representation of the input operator) is given. Section 6 discusses application of the previously described eigensolvers to the problems of modelling dielectric waveguides with arbitrary permittivity profiles and shows the results of tests validating the algorithms. It also presents an original modification of the eigensolver using implicit representation of the input operator, which extends its applicability to modelling dielectric waveguides with discontinuous, rectangular permittivity profiles. Section 7 presents a collection of the results of performance tests in selected parallel distributed memory systems which concludes the analysis of the proposed parallel eigensolvers, confirming their high efficiency and scalability in these environments.

2. Algorithms of solving matrix and operator eigenproblems

This begins with presenting some concepts related to operator eigenproblems and describing briefly eigenvalue problems arising in electromagnetics. Later on selected methods of solving operator eigenproblems are discussed. The considerations concentrate on the main points of the algorithms putting aside to the following sections the questions referring to specific features of the operators and their domains, be they finite- or infinite-dimensional.

2.1 Operator eigenproblems — basic concepts

Given a normed complete linear space (Banach space) X with complex scalars and a bounded linear operator $\mathbf{A} \in B(X, X)$ the eigenproblem (eigenvalue or spectral problem) of this operator is defined by the equation:

$$\mathbf{A}v = \lambda v \tag{1}$$

where $\lambda \in C$ is in eigenvalue and $0 \neq v \in X$ is a corresponding eigenfunction (right eigenfunction) of the operator $\mathbf{A}$. If X is a finite-dimensional space, the

linear operator **A** may be represented in the form of a matrix. In this case its eigenfunctions are most frequently called eigenvectors.

If a scalar product $((\cdot,\cdot): X \times X \to C)$ is defined in the space X, then X becomes a Hilbert space and an adjoint operator **A*** may be associated with the initial operator **A**. By definition the operator **A*** satisfies the following condition:

$$\forall_{x\in X}\forall_{y\in X}(\mathbf{A}x, y) = (x, \mathbf{A}^* y) \tag{2}$$

If $\mathbf{A} = \mathbf{A}^*$ then the operator **A** is called self-adjoint (symmetric). It may easily be found that if the operator **A** is symmetric (self-adjoint) then all its eigenvalues are real. Correspondingly, if the operator is non-symmetric (non-self adjoint) its point spectrum σ_p (the set of its eigenvalues) may contain complex values.

The following eigenproblem may be associated with the operator **A***:

$$\mathbf{A}^*\tilde{v} = \tilde{\lambda}\tilde{v} \tag{3}$$

The functions $\tilde{v}$ are called left eigenfunctions of the operator **A**, as opposed to its right eigenfunctions, defined in equation (1). An important relation joins the point σ_p of the operators **A** and **A***:

$$\sigma_P(\mathbf{A}^*) = \sigma_P^*(\mathbf{A}) = \{\lambda^* : \lambda \in \sigma_P(\mathbf{A})\} \tag{4}$$

Moreover the left and right eigenfunctions satisfy the following orthogonality relation:

$$(\lambda_i - \tilde{\lambda}_j^*)(v_i, \tilde{v}_j) = 0 \tag{5}$$

As it is seen, unless $\lambda_i = \tilde{\lambda}_j^*$, the left and right eigenfunctions of an operator are orthogonal.

2.2 Operator eigenproblems in electromagnetics

Having described some basic issues concerning operator eigenproblems let us now turn to a brief presentation of the operators arising in electromagnetics. These operators will also be discussed in Section 6 which concentrates on validation of the proposed algorithms of solving operator eigenproblems in electromagnetic applications.

Operator eigenproblems are found in various research areas of computational electromagnetics, including the theory of electromagnetic waveguides being of the author's particular interest. Let us consider a dielectric waveguide, shown in Figure 1 which is homogeneous in the z direction and has an arbitrary electrical permittivity profile Figure 1 which is homogeneous in the z direction and has an arbitrary electrical permittivity profile $\varepsilon(x, y)$ in its cross-section ($x - y$ plane). The transverse magnetic field in this structure may be modelled by the following equation derived from the Maxwell's equations:

$$\nabla_t^2 \vec{H}_t + k_0^2 \varepsilon(x,y)\vec{H}_t + \frac{1}{\varepsilon(x,y)}\left[\nabla_t \varepsilon(x,y) \times (\nabla_t \times \vec{H}_t)\right] = \beta^2 \vec{H}_t \tag{6}$$

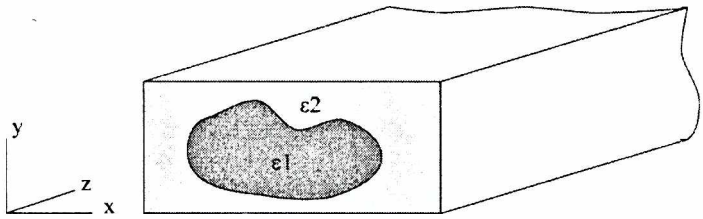

Figure 1. *Schematic of a dielectric waveguide, homogeneous in the z direction and having an arbitrary permittivity profile ε (x, y) in the x – y plane.*

where $\nabla_t^2(\cdot)=\left(\frac{\partial}{\partial x},\frac{\partial}{\partial y}\right)(\cdot), \vec{H}_t=\left(H^x(x,y),H^y(x,y)\right)$ is the transverse magnetic field, k_0 is the wavenumber in the free space, $\varepsilon(x,y)$ is the permittivity profile in the $x-y$ plane and β is a propagation constant.

The computational problem which arises at this point is finding the propagation constant and the form of the transverse magnetic field. In mathematical terms this problem may be viewed as an eigenproblem of the linear operator **T**:

$$\mathbf{T}(\cdot)=\nabla_t^2(\cdot)+k_0^2\varepsilon(x,y)(\cdot)+\frac{1}{\varepsilon(x,y)}\left[\nabla_t\varepsilon(x,y)\times\left(\nabla_t\times(\cdot)\right)\right] \tag{7}$$

with the transverse magnetic field $\vec{H}_t$ as an eigenfunction and β^2 as an eigenvalue to be found. It may be noticed that the operator **T** is a non-symmetric *vector* operator. If the term involving partial derivatives of the permittivity profile is leaved out one obtains the following *scalar* non-symmetric operator $\tilde{\mathbf{T}}$:

$$\tilde{\mathbf{T}}(\cdot)=\nabla_t^2(\cdot)+k_0^2\varepsilon(x,y)(\cdot) \tag{8}$$

The eigenproblem of the above operator provides a simplified model of the waveguiding structure presented in Figure 1.

As it may be seen from the above description, the right eigenfunctions of the considered operator **T** (or $\tilde{\mathbf{T}}$) have got a well-defined physical meaning as they describe the transverse magnetic field in a dielectric waveguide. The question is whether the left eigenfunctions of the operator **T** ($\tilde{\mathbf{T}}$) may also be described in some physical terms. The main problem which has to be solved at this point is deriving an adjoint operator **T*** ($\tilde{\mathbf{T}}$ *) and an associated eigenproblem. This issue is broadly discussed in a report by Przybyszewski et al. [5]. The results show that the left eigenfunctions of the operator **T** ($\tilde{\mathbf{T}}$) are given by the formula: $\hat{i}_z\times\vec{E}_t=\left(-E^y,E^x\right)$, where $\vec{E}_t=\left(E^y,E^x,0\right)$ is the transverse electric field and $\hat{i}_z$

is a unit vector in the z direction. The most important consequence of the fact that the left eigenfunction of the operator **T** is described by the electric field in a waveguide is that this left eigenfunction may be derived directly from the right eigenfunction without the necessity of solving an adjoint operator eigenproblem. This may be achieved by using the following formulas derived from the Maxwell's equations:

$$E^x = \sqrt{\frac{\mu_0}{\varepsilon_0}} \frac{1}{k_0 \varepsilon} \left(\frac{-1}{\beta} \frac{\partial}{\partial y} \left(\nabla_t \cdot \vec{H}_t \right) + \beta H^y \right) \tag{9}$$

$$E^y = \sqrt{\frac{\mu_0}{\varepsilon_0}} \frac{1}{k_0 \varepsilon} \left(\frac{1}{\beta} \frac{\partial}{\partial x} \left(\nabla_t \cdot \vec{H}_t \right) - \beta H^x \right) \tag{10}$$

where μ_0 is the permeability of the free space, ε_0 is the permittivity of the free space and other symbol have the same meaning as above.The fact that the left eigenfunctions may be so easily derived from right eigenfunctions for the considered operator **T** will be applied in the algorithm described in Section 5.5, which uses both eigenfunctions during its iterative process.

2.3 Overview of the algorithms of solving operator eigenproblems

Having presented basic concepts concerning eigenproblems and their applications in electromagnetics, in this section we shall briefly outline different approaches towards numerical solving of operator eigenproblems before discussing in detail two algorithms being the main object of this study.

We shall start this overview with iterative methods of solving operator eigenproblems which currently provide the only efficient strategy for finding eigenvalues in large-scale eigenproblems. The first method to be mentioned is a very well known Power Method (the simple iteration method) [6] which is not only the simplest but also the most important iterative algorithm for solving operator eigenproblems due to its numerous implications for modern iterative eigensolvers. The numerical methods being the main subject this study originate precisely in the Power Method which serves as a basis for the iterative processes. Given the operator **A** the steps of the basic version of the Power Method are given as follows:

ALGORITHM 1: *The Power Method.*

STEP 0: Choose an initial function v_1 such that $\|v_1\| = 1$, assume $k = 1$.
STEP 1: Iterate:
  STEP 1.1: Calculate $w_{k+1} = \mathbf{A}v_k$.
  STEP 1.2: Normalize: $v_{k+1} = w_{k+1}/\|w_{k+1}\|$.
  STEP 1.3: $k := k + 1$.

The main feature of the above method is that it converges to the eigenfunction corresponding to the dominant eigenvalue (the eigenvalue with the largest modulus) of the operator **A**. In the case of a symmetric operator **A** with its eigenfunctions

forming an orthonormal basis in the operator's domain it is easy to show (cf. [7]) that:

$$\lim_k \frac{\left\|\mathbf{A}^k v_1\right\|}{\left\|\mathbf{A}^{k-1} v_1\right\|} = \lim_k \left\|\mathbf{A} v_{k-1}\right\| = \left|\lambda_{\max}\right| \tag{11}$$

where λ_{max} is the dominant eigenvalue of the operator $\mathbf{A}$. The other important feature of the above algorithm is that the information on the operator $\mathbf{A}$ is passed to the iterative process only via the $\mathbf{A}v_k$ operation which allows one to apply any kind of implicit representation of the input operator. The main drawback of the above simple method is that it is able to find only a single, dominant eigenvalue and eigenfunction of the operator. Still, the functionality of this algorithm may be extended if deflation and shifting techniques are applied within the iterative process [6] allowing one to find other eigenvalues of the input operator.

Another important aspect of the Power Method is that during the iterative process a *Krylov subspace* K_m is being constructed:

$$K_m = Span\left\{v_1, \mathbf{A}v_1, \ldots, \mathbf{A}^{m-1} v_1\right\} \tag{12}$$

At this point it should be noted that the Power Method exploits only the last two functions from the basis of the Krylov subspace K_m shown above. This fact provided a basis for the development of the *iterative subspace methods* which exploit the whole Krylov subspace in order to achieve quicker convergence than in the Power Method. These algorithms, which may be used to solve eigenproblems both for infinite-dimensional linear operators and finite-dimensional matrix operators, are currently the most dynamically developing field of research in numerical analysis. The most representative examples of modern iterative subspace methods are the Lanczos method (for symmetric operators), the Arnoldi method (non-symmetric case) or the Davidson algorithm ([8]) (originally designed for symmetric matrices). In these highly effective methods the problem, defined usually for a sparse or structured matrix operator of very large dimension, is reduced to a much smaller dense matrix operator problem. This smaller problem may then be solved by any of the standard techniques used for dense matrix operators. Due to the structure of the three mentioned algorithms they are normally used to find several eigenvalues from a spectrum of a given operator. Another numerical method which to a certain extent contains the Power Method is the Iterative Eigenfunction Expansion Method (IEEM) (described in detail in one of the following sections) which may be used to solve non-symmetric operator and matrix eigenproblems.

Apart from the algorithms which are capable of solving eigenproblems for general symmetric or non-symmetric linear operators, a huge number of algorithms designed to deal with matrix operators have been developed, contributing to a rapid progress in the numerical linear algebra. Historically the development of the methods of solving matrix eigenproblems starts with symmetric eigenproblems. One of the first algorithms dealing with symmetric matrices was the Jacobi method that used orthogonal matrix transformations to find eigenvalues and eigenvectors.

This classical algorithm was gradually replaced in applications by other techniques e.g. Householder reduction and, above all, by the QR method [6]. The above methods are still common in many applications as they are able to find all the eigenvalues of a given matrix. Nevertheless, the relatively high numerical cost which reaches roughly $O(n^3)$ computations significantly reduces the size of the problem which may be solved by these algorithms using available computer systems. The quest for finding more efficient algorithms solving matrix eigenvalue problems caused the development of more specialized methods aimed at solving problems for either dense or sparse, symmetric or non-symmetric matrices as well as banded and highly regular matrix operators. Broad presentations of the algorithms used to solve matrix eigenproblems may be found in the book by Saad [9] (especially sparse matrices) or the book by Golub and van Loan [6].

2.4 The Arnoldi method

This section presents the Arnoldi method which belongs to a class of iterative subspace algorithms capable of approximating a few eigenvalues and the corresponding eigenvectors of a general square matrix. In a classical approach ([10]) the applicability of this technique was strongly limited due to a potentially unbounded growth in storage as well as the lack of numerical stability of the iterative process resulting e.g. in a loss of orthogonality of the eigenvectors. These problems have been successfully solved by Sorensen [11] who proposed a modification of the initial Arnoldi algorithm called the Implicitly Restarted Arnoldi Method (IRAM). Exploiting the analogy between the Arnoldi process and the QR iteration the IRAM provides an iterative scheme which has a fixed memory complexity if the number of eigenvalues to be sought is pre-specified. The other advantage of the method is that it preserves the orthogonality of the Arnoldi basis in the Krylov subspace (compare the previous section) if the number of the eigenvalues to be found is not too large.

The Implicitly Restarted Arnoldi Method was found to be a highly efficient tool for solving eigenproblems, capable of reducing both storage requirements and the computation time for a very wide class of large structured non-symmetric matrices in different fields of applications. (cf. [12], [13]) The problem which was found to occur with the IRAM (presented later on in the work) is the significant increment in the number of update iterations with the increasing size of the input matrix.

2.4.1 The Arnoldi factorization

In the approach proposed by Sorensen ([11]) the Arnoldi factorization may be treated as a truncated reduction of a given square matrix $\underline{\underline{A}}$ to a form of an upper Hessenberg matrix. This operation is performed in an iterative process and the k-th step of the factorization may be described by the following formula (cf. [6]):

$$\underline{\underline{A}}\,\underline{\underline{V}}_k = \underline{\underline{V}}_k\,\underline{\underline{H}}_k + \underline{f}_k\,\underline{e}_k^T \tag{13}$$

where

$\underline{\underline{A}}$ is the input $n \times n$ matrix,

$\underline{\underline{H}}_k$ is a $k \times k$ upper Hessenberg matrix ($k < n$),

$\underline{\underline{V}}_k$ is an $n \times k$ matrix whose columns are Arnoldi vectors,

$\underline{f}_k$ is a residual vector of size n, satisfying the relation $\underline{\underline{V}}_k^T \underline{f}_k = 0$.

The idea of Arnoldi factorization is illustrated in Figure 2. From equation (13) it may been noticed that the process is a truncation of the complete reduction to the Hessenberg form and if the vector $\underline{f}_k$ becomes zero the eigenvalues of the Hessenberg matrix will equal the eigenvalues of the given matrix $\underline{\underline{A}}$. The columns of the matrix $\underline{\underline{V}}_k = [\underline{v}_1, \underline{v}_2, \ldots, \underline{v}_k]$ constructed in the Arnoldi process form an orthonormal basis in the Krylov subspace K_k:

$$K_k = Span\left\{\underline{v}, \underline{\underline{A}}\,\underline{v}, \underline{\underline{A}}^2\,\underline{v}, \ldots, \underline{\underline{A}}^{k-1}\,\underline{v}\right\}$$

where $v \in R^n$ ($v \in C^n$). The basis $\{v_i\}_{i=1,\ldots k}$ is formed in k iterations of the basic Arnoldi algorithm, which may be implemented in a few ways, including the most common, known as the Arnoldi Modified Gram Schmidt algorithm. The steps of this algorithm are given as follows (cf. [9]):

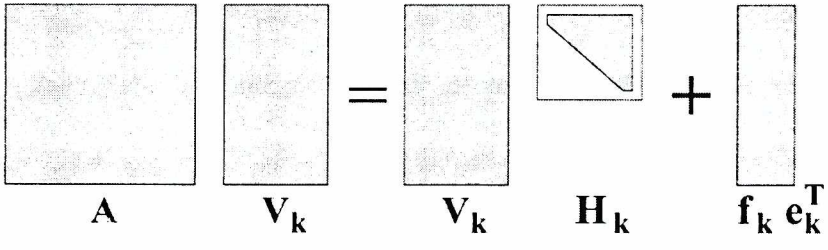

Figure 2. *The schematic of the Arnoldi factorization*

ALGORITHM 2: *Arnoldi-MGS.*

STEP 0: Choose an initial vector $\underline{v}_1$ such that $\|\underline{v}_1\|_2 = 1$
STEP 1: Iterate: For $j = 1, 2, \ldots, k$ do:

STEP 1.1: $\underline{w} := \underline{\underline{A}}\,\underline{v}_j$

STEP 1.2: For $i = 1, 2, \ldots, j$ do:

STEP 1.2A: $h_{ij} = (\underline{w}, \underline{v}_i)$,

STEP 1.2B: $\underline{w} = \underline{w} - h_{ij}\,\underline{v}_i$

STEP 1.3: $h_{j+1,j} = \|\underline{w}\|_2$

STEP 1.4: $\underline{v}_{j+1} = \underline{w} / h_{j+1,j}$

where h_{ij} are the elements of the upper Hessenberg matrix $\underline{\underline{H}}$.

It has to be noted that during the factorization process the information on the input matrix ($\underline{\underline{A}}$) is passed to the algorithm only via the matrix-vector product $\underline{\underline{A}}\underline{v}_j$. This is an extremely important feature since $\underline{\underline{A}}$ does not have to be known explicitly.

2.4.2 Polynomial filters in the Arnoldi method

In the basic Arnoldi algorithm presented in the previous section two main problems appear. The first is an undefined number of iterations k necessary to obtain a desired accuracy of the eigenvalues (estimated by calculating the residual norm) which leads to unbounded memory complexity. This problem may be solved by restarting the iterative process after a given, fixed number of iterations with a suitably updated initial vector $\underline{v}_1$ [9]. The other problem is forcing the algorithm to converge to the eigenvalue from the desired part of the matrix spectrum. This may be achieved by "filtering-out" the "unwanted" eigenvalues at each restart of the algorithm using e.g. polynomial filtering, as proposed by Sorensen [14]. In this technique, after initial k steps of the basic Arnoldi algorithm, additional p iterations are performed. Next, $k+p$ eigenvalues are found as the eigenvalues of the upper Hessenberg matrix $\underline{\underline{H}}_k$ (the Ritz values) and p "unwanted" eigenvalues are then being filtered out using the implicit shift algorithm with an appropriate filtering polynomial. The algorithm is then restarted with an updated initial vector $\underline{v}_1$ and the subsequent p basic Arnoldi iterations are being performed.

2.4.3 Numerical and memory complexity of the algorithm

As it has already been told in the previous sections, the Implicitly Restarted Arnoldi Method (IRAM) demonstrates a fixed memory complexity. If the number of the eigenvalues to be found equals k, the number of additional eigenvalues to be computed is p and the input matrix size is n then, denoting $l = k + p$, the algorithm requires $n \cdot O(l) + O(l^2)$ storage. Lehoucq et al. suggest ([15]) that p should equal k in order to obtain an efficient algorithm with good convergence rate. Then the memory complexity equals $n \cdot O(k) + O(k^2)$. If one keeps in mind that k is much smaller than n, it results that the IRAM itself requires very little storage. (Obviously some extra storage may be required to perform the matrix-vector product $\underline{\underline{A}} \cdot \underline{v}$ operation, but it should not exceed n^2.)*

The numerical complexity may only be assessed for a single update in a p-step IRAM algorithm. If the cost of the matrix-vector product (step 1.1 of the factorization) is excluded then the complexity equals $O(p^2 n)$. If one assumes that

* Other authors [13] indicate that the choice $p = k$ may not be the optimal one and propose a choice of the value of p as a function of the problem dimension n in order to obtain quicker convergence. In this case the theoretical complexity becomes a function of n, still in the applications presented in [13] it does not result in high memory requirements.

$p = k$ ($p = O(k)$) then the numerical cost becomes of $O(k^2 n)$. Once again, as the number of eigenvalues k to be found is normally much smaller than the problem size n, a linear complexity is obtained. Obviously the cost of the operation of matrix-vector product may be significantly higher, reaching $O(n^2)$ in the worst case and result in a quadratic overall complexity. Still, as shown later in this work, this cost may be reduced if the matrix is sparse or does not have to be represented explicitly by all its elements.

2.5 The Iterative Eigenfunction Expansion Method

The Iterative Eigenfunction Expansion Method (IEEM) was first proposed in 1986 by Jabłoński [16], [17] and later improved by Mrozowski [18]. Originally it was presented and investigated as a method of solving eigenproblems for a certain class of differential operators being of special interest in the theory of electromagnetic waveguides. This section starts with the topic of the decomposition of a given input operator which is a main point of the IEEM. Later on, the original operator formulation is discussed and deflation techniques extending the functionality of the method are presented. The formulation of the method for finite-dimensional linear matrix operators which shows the relations between IEEM and the simple iteration method (the Power Method) is described in Appendix A.

2.5.1 Decomposition of the operator

Given an operator $\mathbf{T}$, defined over a certain Hilbert space X, it may be represented in a following form:

$$\mathbf{T} = \mathbf{L} - \mathbf{F} \tag{14}$$

where the operator $\mathbf{L}$ is a symmetric (self-adjoint) operator with a discrete spectrum and its eigenvalues form an orthogonal basis in the given operator's domain X. The above decomposition is known from the perturbation theory [19], with the operator $\mathbf{F}$ being a "small" (in a sense of operator norm) perturbation of the operator $\mathbf{L}$. However, in the iterative eigenfunction expansion method, the perturbation operator $\mathbf{F}$ does not have to be "small" but only relatively compact (in an appropriate domain) with respect to $\mathbf{L}$. This relaxed assumption constitutes a significant generalization as compared to the classical perturbation technique, although, as proved by Jabłoński in [16], it is sufficient to construct an iterative process which converges to an eigenvalue of the operator $\mathbf{T}$. Below the original operator formulation of the simple iterative process capable of finding a single eigenvalue of the input operator is described.

2.5.2 Operator formulation of the iterative process

We start the description of IEEM with the formulation suitable for solving eigenproblems for general linear operators. In this original formulation which has been successfully applied to problems of electromagnetics (cf. [16], [17] and [18]) the concept of "eigenfunction expansion" clearly emerges.

Let us assume that the input operator $\mathbf{T}$ is decomposed as described in the

previous section (cf. equation (14)). Given the sets of known eigenvalues of the operator **L**, denoted as $\{\Lambda_i\}$ and the corresponding eigenfunctions $\{h_i\}$ forming an orthonormal basis in the Hilbert space X, any function u from X may be expanded in terms of these functions:

$$u = \sum_i f_i h_i \tag{15}$$

where f_i are the coefficients (Fourier coefficients) of the linear combination of $\{h_i\}$. Denoting as λ a certain eigenvalue of the operator **T** the following formula for the coefficients f_i may easily be derived:

$$f_i = \frac{(\mathbf{F}u, h_i)}{\Lambda_i - \lambda} \tag{16}$$

where h_i is an eigenfunction of the operator **L** corresponding to the eigenvalue Λ_i and $(\cdot,\cdot)$ denotes an inner product defined in the Hilbert space X. The above formula is a basis for the iterative process which may be defined as follows:

ALGORITHM 3: *IEEM.*

STEP 0: Choose an arbitrary initial function $u^{(0)}$ such that $\|u^{(0)}\| = 1$, $k = 0$
STEP 1: Compute $\lambda^{(0)}$ from the Rayleigh quotient:

$$\lambda^{(0)} = \frac{\left(\mathbf{T}u^{(0)}, u^{(0)}\right)}{\left(u^{(0)}, u^{(0)}\right)}$$

STEP 2: Iterate:

STEP 2.1: $$u^{(k)} = \sum_i \frac{\left(\mathbf{F}u^{(k-1)}, h_i\right)h_i}{\Lambda_i - \lambda^{(k-1)}}$$

STEP 2.2: $$u^{(k)} := \frac{u^{(k)}}{\left\| u^{(k)} \right\|}$$

STEP 2.3: Assuming $u^{(k)} = \sum_i f_i^{(k)} h_i$ compute the Rayleigh quotient as:

$$\lambda^{(k)} = \sum_i \Lambda_i \left| f_i^{(k)} \right|^2 - \sum_i f_i^{*(k)} \left(\mathbf{F}u^{(k)}, h_i\right)$$

STEP 2.4: $k := k + 1$.

As already explained in the previous section, if the operator **F** is relatively compact with respect to **L** then the iterative process presented above converges to an eigenvalue of the operator **T**. main advantage of the above method is a very fast convergence rate. In electromagnetic applications it has been found [18] that IEEM provides a basis for an extremely efficient eigensolver, offering very fast

convergence for an entire general class of differential operators investigated in this research area. Still, the main drawback of the original algorithm is that it may be used to find only one eigenvalue from the operator's point spectrum. Moreover, this eigenvalue has not been identified within the spectrum of the input operator. In other words it is not known which eigenvalue is being found in the method. Nevertheless, although no rigorous proof exists for this fact, it was found that IEEM applied to eigenproblems arising in dielectric waveguide modelling converges to the fundamental mode in a waveguide, which is of particular interest in this application area (cf. [18]).

2.5.3 Deflation techniques

As already mentioned the original algorithm described in the previous section may be used to find a single eigenvalue of the input operator. In order to be able to find more than one eigenvalue of the operator certain modifications have to be introduced to the basic Iterative Eigenfunction Expansion Method. These include the deflation techniques, i.e. the methods of modifying the operator's spectrum, so that the iterative process may converge to a different eigenvalue and the orthogonalization procedures assuring that the orthogonality is sustained between the appropriate left and right eigenfunctions (cf. equation (5)).

Wieland's deflation. The most widely known techniques of modifying the spectrum are based on the Wieland's deflation. Let $\{\lambda_i\}_i = \sigma_P(\mathbf{T})$ be the point spectrum of the operator T. Let λ_1 and u_1 be an eigenpair of the operator $\mathbf{T}$ found by a certain iterative process. The deflation procedure can be used to modify the spectrum of the operator by replacing λ_1 with another eigenvalue, e.g. $\lambda_1 - \alpha$, where $\alpha \in C$. This is done by modifying the operator itself:

$$\tilde{\mathbf{T}}(\cdot) = \mathbf{T}(\cdot) - \alpha u_1(\cdot, v) \tag{17}$$

where v is an arbitrary function such that $(u_1, v) = 1$. The spectrum of the operator $\tilde{\mathbf{T}}$ is:

$$\sigma_P(\tilde{\mathbf{T}}) = \{\lambda_1 - \alpha, \lambda_2, \lambda_3, \ldots\}$$

where $\lambda_1, \lambda_2, \lambda_3, \ldots$ are the eigenvalues of the operator $\mathbf{T}$. Moreover, the eigenfunction u_1 and all the left eigenfunctions $\{w_1\}_i$ of the operator $\mathbf{T}$ are preserved as corresponding right and left eigenfunctions of the operator $\tilde{\mathbf{T}}$. There are many possible choices for the function v, still the most popular ones are: 1) $v = u_1$. In this case the right eigenfunctions of the operator other than u_1 are not preserved; 2) $v = w_1$. In this case the right eigenfunctions are preserved. Still, the disadvantage of this choice is that it is necessary to know the left eigenvector w_1 corresponding to the eigenvalue λ_1^* of the adjoint operator $\mathbf{T}^*$. The left eigenfunction may be found by either explicit solution of an adjoint eigenproblem or, at a lower cost, by exploring the form of the operator and deriving left eigenfunctions directly from right eigenfunctions which is possible in some of the applications (compare Section 2.2).

Below we present the modified version of IEEM iteration assuming that

$v = w_i$. The algorithm is restarted after the subsequent eigenvalues are being found. If also assumed that $s - 1$ eigenvalues $\lambda_1, \ldots, \lambda_{s-1}$ (with the corresponding right and left eigenfunctions $\{v_1, \ldots, v_{s-1}\}$ and $\{w_1, \ldots, w_{s-1}\}$) have already been found the $k-th$ step of the iteration may be described by the following steps:

ALGORITHM 4: *IEEM-deflation.*

STEP 1: Compute the eigenfunction approximation

$$u^{(k)} = \sum_i \frac{\left(\mathbf{F}u^{(k-1)}, h_i\right) + \sum_{r=1}^{s-1} \alpha_r \lambda_r \left(u^{(k-1)}, w_r\right) f_{i,r}}{\Lambda_i - \lambda^{(k-1)}} h_i$$

STEP 2: Normalize:

$$u^{(k)} := \frac{u^{(k)}}{\left\| u^{(k)} \right\|}$$

STEP 3: Compute eigenvalue approximation:

$$\lambda^{(k)} = \sum_i \Lambda_i \left| f_i^{(k)} \right|^2 - \sum_i f_i^{*(k)} \left(\mathbf{F}u^{(k)}, h_i\right) + \sum_{r=1}^{s-1} \alpha_r \lambda_r \left(u^{(k)}, w_r\right) f_{i,r}$$

where $u^{(k)} = \sum_i f_i^{(k)} h_i$ and $u_r = \sum_i f_{i,r} h_i$ for $i = 1, \ldots, (s-1)$.

The main problem occurring within the procedure presented above is that at each restart of the algorithm the deflation term introduced to modify the operator's spectrum contains a numerical error. This error is related to the approximation of the eigenvalue, and, more importantly, to the approximation of the right and left eigenfunctions. After the subsequent restarts the errors from all previous computations will accumulate in the modified operator and this can be disastrous if the currently computed eigenvalue is poorly conditioned. Another problem which may immediately be seen is the numerical cost and storage requirements growing after each restart of the algorithm. These two drawbacks limit the applications of this technique and make it a tool capable of finding only a few eigenvalues and eigenfunctions from the operator's spectrum.

Orthogonal projections. In the previous section it has been noted that one of the key factors which limit the deflation technique is the error introduced by the approximations of right and left eigenfunctions (eigenvectors). This error demonstrates itself in the loss of orthogonality between right and left eigenfunctions. More precisely the right eigenfunction u_s ceases to be orthogonal to the left eigenfunctions $w_1, \ldots, w_{s-1}$.

This fact suggests that introducing a re-orthogonalization phase to the iterative algorithm may result in a reduction of the numerical error and improvement in the stability of the iterative process. The orthogonalization may be performed using the

modified Gram-Schmidt (MGS) algorithm [6] while calculating the subsequent approximation of the eigenfunction $u^{(k)}$. It should be noted that the orthogonalization may be performed every few iterations or even should not be performed at each iteration in order to allow the convergence of the method. At the same time it has to be stressed that the re-orthogonalization should be used jointly with the deflation procedures. Otherwise the iterative process will not converge at all or will converge to the eigenvalue which has been found as a first one.

2.5.4 Numerical and memory complexity of the algorithm

If an input matrix operator of size n is considered, then the numerical cost of a single iteration of the IEEM consists of the following items: 1) The cost of calculating the approximation of the eigenvector — This cost equals $O(n)$ if we assume that the values of scalar products $(\underline{\underline{F}}\underline{u}^{(k-1)}, \underline{h}_i)$ are known from the previous iteration. 2) The cost of normalization — $O(n)$ and 3) The cost of calculating the subsequent approximation of the eigenvalue — This cost is dominated by the complexity of calculating the matrix-vector product $\underline{\underline{F}}\underline{u}$, which may reach $O(n^2)$ in the worst case but may be significantly reduced (to $O(n \log(n))$ or $O(n)$) for certain types of matrices (sparse or structured matrices) or if an implicit representation of the matrix is used.

As it is seen the computational complexity of a single iteration of the IEEM largely depends on the numerical cost of calculating the matrix-vector product for a given matrix operator and may range from $O(n)$ to $O(n^2)$.

The storage requirements of the Iterative Eigenfunction Expansion Method may be extremely low, although once again they depend primarily on the form of matrix, e.g. for sparse diagonal or banded matrices and/or highly regular matrices the storage cost of the matrix may be reduced from $O(n^2)$ to $O(n)$. The same can be achieved if an implicit matrix representation is applied. The other memory requirements include the space needed to store eigenvalues of the operator $\mathbf{L}$ (n memory locations) and the subsequent approximations of the eigenfunctions of the operator $\mathbf{T}$ ($O(n)$ memory locations).

If the deflation procedure is applied the memory cost increases significantly and depends on the number of the eigenvalues to be found. If assumed that s eigenvalues are to be found and the deflation involves both right and left eigenvectors then the method will require: 1) In the worst case additional n^2 memory locations to store an operator adjoint to $\mathbf{T}$ which may be necessary to solve the adjoint eigenproblem; 2) Additional $2(s-1)n$ memory locations used to store formerly found left and right eigenvectors. The storage described in point 1) may be reduced to zero if the left eigenvectors may be derived directly from right eigenvectors avoiding the necessity of solving an adjoint eigenproblem.

The computational complexity of the IEEM iteration with an additional deflation procedure applied increases with the number of eigenvalues which have already been found. If assumed that s is the number of eigenvalues previously

found, the additional cost due to the application of deflation equals roughly $2sn$ operations. In this assessment the cost of calculating the left eigenvectors has not been taken into account, but it has to be kept in mind that it may even double the algorithm's execution time. Another item which increases the numerical cost of the algorithm is the re-orthogonalization phase whose complexity (for the MGS procedure) is of order $O(sn)$.

Summing up, both the memory complexity and the computational cost of the Iterative Eigenfunction Expansion Method depend considerably on the representation of the matrix operator and the cost of computing of the matrix-vector product. (An analogous situation occurs in the case of the Arnoldi (IRAM) method). Consequently, if the input matrix yields any kind of special structure, including a regular pattern of distribution of its non-zero elements, sparsity or specific representation these costs may be very significantly reduced. Nevertheless, the application of deflation procedures will inevitably increase both the storage requirements (even by a few times) and the numerical cost, not to mention the increment in the number of iterations required to obtain convergence for each next eigenvalue being sought.

2.6 Other methods of solving operator and matrix eigenproblems

Having presented the two algorithms of solving operator and matrix eigenproblems playing a central role in this study let us only mention some recent developments in this research area. The most important include modifications in the Davidson method leading to algorithms suitable for non-symmetric matrices, including the Jacobi-Davidson algorithm or the introduction of look-ahead strategy to two-sided Lanczos algorithms. The investigations also include the designs of algorithms which inherently assume parallel computations. An example for such method is the divide and conquer algorithm ([20]) with extensions exploiting the relationship between a certain matrix algebra and complex polynomials ([21]).

The detailed description of the methods outlined above is clearly far beyond the scope of this limited study and may be found in many excellent books, including classical book by Wilkinson and Reinsch [22], the monograph by Golub and van Loan [6] which broadly covers the questions of non-symmetric eigenproblems, the book by Saad [9] or the paper by van der Vorst and Golub [7] which presents a review of recent developments.

3. Cost reducing discretization of infinite-dimensional operators

The previous section described selected iterative methods of solving operator eigenproblems putting aside the questions of the form of the linear operator or its domain. In the case of a finite-dimensional domain and a general linear operator represented in the matrix form both the Arnoldi method and Iterative Eigenfunction Expansion Method give recipes ready for use in order to calculate numerically eigenvalues and eigenvectors of a given matrix. However, in the discussion of the computational complexity of both methods it was found that this complexity is determined primarily by the cost of calculating the matrix-vector product. This cost,

in turn, depends on representation (implicit or explicit) of the matrix and the form of the operator (sparse or dense matrix with regular or irregular distribution pattern of non-zero elements). Clearly little can be done to reduce the cost of the matrix-vector product if the given input finite-dimensional linear operator is already represented e.g. by a dense matrix. This situation is very different if initially one has an infinite-dimensional operator which is inherently unsuitable for any numerical treatment. The problem which appears is finding the discretization or finite-dimensional mapping of the operator so that it may be approximated in a finite space by a different linear operator. As various discretization methods exist, it means that one may control the form and representation of the emerging finite-dimensional operator and consequently influence (reduce) the numerical cost of performing the $\mathbf{A}v$ operation, where $\mathbf{A}$ should be understood as a finite approximation of the initial operator and v should as perceived as a corresponding representation of the function from the operator's domain.

The problem of defining a finite-dimensional mapping refers not only to operators but also to the functions belonging to the operator's domain. There is a great variety of finite representations of functions, with an emphatic majority based on expansions in terms of a chosen set of basis functions. Obviously, even a short description of the most popular functional bases lies far beyond the scope of this work. Nevertheless, some general classes of representations may be distinguished, starting from simple representations based on regular or irregular sampling of a function in its domain to the *entire domain expansions, entire subdomain expansions or domain subdivision expansions* in which accuracy of the representation depends correspondingly on the number of expansion terms or (in the third case) the number of subdomains or sampling points within the domain. (The Finite Difference (FD) discretization method presented later on in this section belongs clearly to the domain subdivision methods, while the Method of Moments or Discrete Fourier Transform (DFT) based representation (also discussed later in this section) are entire domain expansion methods.) Apart from different finite mappings of functions also various operator representations may be chosen which gives rise to a number of numerical procedures. If, for instance, the operator projection is achieved by calculating scalar products, as in the Method of Moments (the Galerkin Method) then for different representations of functions various methods are obtained e.g. the Finite Element Method (FEM) with a resulting sparse operator matrix having usually an irregular distribution of non-zero elements or the collocation (or point matching) technique with a resulting sparse or dense matrix. The discussion of functional expansion and discretization techniques may be found in a number of books — cf. [23], [24], [25], [1], [26].

The rest of this section concentrates on two finite-dimensional mapping methods in which very different and to a certain extent opposing approaches towards approximating both functions and operators are applied. Although these discretization methods produce approximate finite-dimensional operators with entirely different properties, the specifics of both representations enable one to 1)

reduce the cost of performing the $\mathbf{A}v$ operation, which has a substantial impact on the efficiency of numerical solving of given eigenvalue problems; 2) efficiently implement operations involving the discretized operators in parallel distributed memory environments.

Before describing in detail the two selected discretization methods we shall discuss some aspects of finite-dimensional mapping which play an important role if the algorithms involving discretized operators and functions are to be implemented in scalable parallel systems. The general conclusions drawn from this discussion substantiate to a certain extent the choice of the operator discretization methods applied in the parallel eigensolvers presented in Section 5 which implement selected iterative algorithms of solving non-symmetric operator eigenproblems in distributed memory environments.

3.1 Discussion of discretization aspects in scalable parallel systems

Before presenting some finite-dimensional mapping strategies let us make a few remarks on the mutual relations between the efficiency of parallel matrix computations and the choice of a finite-dimensional mapping technique for a given operator. The discussion will concentrate on the general issues concerning data locality and closely related computation locality postponing the detailed description of the specific parallel designs and implementations in selected parallel systems to the following sections.

Designs of the numerical methods performing various matrix calculations, such as computing the matrix-vector product, matrix-matrix product or deriving matrix transposition, are determined primarily by the form of the input matrix operator. Application of these algorithms in the environment with multiple processing elements (PEs) requires developing suitable mapping techniques of both data and computations to the processors in order to achieve the main goal of parallel processing, i.e. minimization of the total execution (wall-clock) time. Although these mapping techniques certainly depend on the representation of the input matrix operators and the specifics of the matrix computations to be parallelized, the basic two strategies will certainly be applied: 1) Place the computational tasks on different processing elements in order to enhance concurrency, 2) Place the computational tasks which make use of the same data on the same processor to increase the locality.

These strategies may sometimes turn out very conflicting which would require trade-offs in design of the mapping techniques. At the same time an inadequate exploitation of any of these strategies will usually reduce or even eliminate the gain in performance of the numerical algorithms implemented in a parallel environment. This fact is particularly true for scalable parallel systems where often a large number of processing elements is involved in the computations.

Let us consider the simplest, still up to now the most important parallel mapping technique, i.e. the static domain decomposition technique. In this mapping method all the data (e.g. matrix or vector elements or a grid in the spatial domain) as well as computational tasks are distributed among the processing elements in an

fixed manner. In the method the properties of the domain being decomposed determine whether a computational task may be efficiently mapped to the available processing elements. Concentrating on the techniques of distributing matrix operators let us discuss the specifics of parallel decomposition for some classes of matrices:

1. *Matrices with a block structure.* In this case an ideal locality of data and computations (within a single PE or a group of PEs) may be achieved if all the elements of a given block in the matrix are local to a single processor or a group of processors. The most favorable case occurs if the number of PEs equals or divides the number of blocks of the matrix and all the blocks have equal sizes. Then all data (e.g. necessary to perform the matrix transposition) may be stored locally and the amount of computations may be perfectly balanced across the processors. The problems with balancing the computations will occur if the sizes of the blocks are not equal and/or the number of processors does not correspond directly to the number of the matrix blocks. In this case the assignment of matrix blocks to PEs has to take into account the numerical complexity of the operations performed on each block in order to obtain balancing of the workload. (Still, the workload balancing may cause an imbalance in the local storage requirements.) The question that emerges is: Which operators may be discretized to produce an operator matrix with a block structure? The first group of such operators are scalar operators acting on multidimensional vector fields. Separating the field components in a finite-dimensional representation may give a block structure of the resulting matrix. The other group of operators may be defined as operators modelling short-range, local interactions in a number of disjoint subsystems. Applying e.g. the Finite Difference (FD) discretization may then result in a block-structured matrix or a banded matrix.

2. *Banded matrices* also have a very favorable structure while investigating their parallel distribution using domain decomposition method. In most cases the amount of non-local data which is used by the processing elements is of order $O(b^2)$ or $O(b)$ (depending on the mapping and computational task), where b is the matrix bandwidth. If the bandwidth is small relatively to the matrix size then the emphatic majority of necessary data is stored locally by each PE and most of the computations involve only local data. Banded matrices are frequently obtained by using the Finite Difference (FD) discretization scheme. The FD technique has also the advantage of producing a highly regular matrix with an even distribution of its non-zero elements. This has a very positive impact on workload balancing which may be easily achieved by applying regular domain decomposition.

3. *Sparse, non-banded matrices* are the class of matrices which may be encountered if the Finite Element Method (FEM) is used to discretize the operator's domain. Although the matrices are usually sparse, the irregular distribution of their non-zero elements may result in problems while seeking for an efficient parallel

mapping using static domain decomposition. The first problem is that potentially large amount of non-local data has to be used by each processing element in order to perform parallel matrix operations. One of the solutions to this situation is designing specific procedures of accessing or communicating non-local data in order to avoid bottlenecks and reduce the number of non-local data accesses. The other problem is the irregular non-zero element distribution which may cause an imbalance in the workload across the PEs. Summing up, in the case of sparse, non-banded matrices the static parallel domain decomposition schemes may turn out unsuitable if high performance in a scalable parallel execution environment is to be achieved.

4. *Dense matrices* appear when entire domain or entire subdomain expansion discretization techniques are used. (The example of such technique - the Method of Moments representation will be described in one of the following subsections.) The parallel decomposition of dense matrices may potentially result in very large amount of non-local data which has to be accessed by the processing elements while performing such operations as e.g. matrix transposition. There is usually little that can be done to avoid a great deal of computations involving non-local data. Still, in order to maintain high level of parallel performance one may increase the computation time involving solely local data as compared to the time spent on accessing or using non-local data by applying appropriate scaling of the problem size. Unfortunately this cannot be done if the complexity of operations involving non-local data is higher than the numerical cost of the local computations. The positive feature while dealing with dense matrices is that the workload balance may be achieved by applying a simple regular domain mapping scheme.

Summing up, the characteristics of different types of matrices obtained in various methods of finite-dimensional mapping of linear operators may affect positively or negatively the performance of parallel algorithms involving operating on distributed matrices. Within the limits of the static domain decomposition parallel mapping techniques the positive features of matrices to be distributed include block structure, sparsity of the matrix, relatively narrow matrix bandwidth, while the negative ones include irregular non-zero element distribution in sparse matrices or dense non-zero element packing. Some of these negative factors may even exclude the static domain decomposition technique if an efficient parallelization of a given computational problem is to be achieved.

In this case different parallel mapping techniques have to be applied. At this point the following mapping schemes may be mentioned:

— *load balancing algorithms* which include: *probabilistic load balancing or cyclic mapping* — the static methods which exploit structure of the computations and data to distribute the domain of computations and may be used e.g. to problems involving matrices with an irregular distribution of non-zero elements or irregular distribution of computations; *dynamic load balancing* in which the

parallel mappings change during execution of the algorithm — this method may be applied e.g. in the multigrid algorithms (cf. [27], [28]).

— *task scheduling algorithms* which explore the potential for *functional* parallel decomposition of the computational tasks and may be applied to obtain parallel mapping of problems with FEM-based discretization, multigrid approach, etc.

A much broader discussion of parallel mapping strategies with various case study presentations may be found in a book by Foster [29] or the teaching materials from the Edinburgh Parallel Computing Centre [30] (in which mainly static domain decomposition techniques are described).

This section presented general issues concerning parallel mapping techniques of discrete matrix operators and related computational tasks. In the above description some potential problems occurring during parallel mapping of different classes of matrices obtained during discretization of linear operators were discussed. In the above approach we tried to answer whether a suitable parallel mapping may be found for a given type of matrix. Still, these general guidelines may be applied in a somewhat inverse approach. This second approach consists of exploring the possible parallel mapping techniques for a given parallel system architecture *before* choosing discretization and finite representation scheme for a given input linear operator. In this way the finite-dimensional representations of operators which will not fit any efficient parallel mapping technique may be immediately excluded.

As already mentioned the following sections discuss two different finite-dimensional mapping techniques in which very different and to a certain extent opposing approaches towards approximating both functions and operators are applied. Although these discretization methods produce approximate finite-dimensional operators with entirely different properties, the specifics of both representations enable one to reduce the cost of calculations involving these discrete operators.

3.2 The Finite Difference discretization

The Finite Difference (FD) method is one of the simplest and very commonly used algorithms of operator discretization. In this method the functions from the domain of the given operator are represented either as simple sets of values sampled over a certain region or as expansions with simple (usually piecewise linear) expansion functions defined over rectangular subdomains. As already mentioned this method belongs to a class of domain subdivision expansions which become more accurate with a growing number of subdomains or sampling points.

The FD method is most frequently used to discretize various differential operators, e.g. those involving Laplace operator, and consists in substituting the differentials by the finite-dimensional difference operators. The finite difference operators may yield various forms, starting from simple 2-point stencils valid for approximating the first order derivatives in one dimension to complex multipoint stencils used to obtain higher accuracy or deal with higher order derivatives,

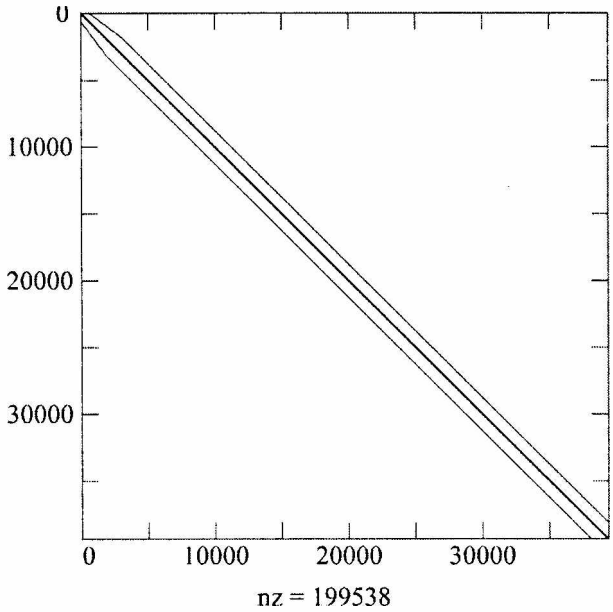

Figure 3. *Distribution of non-zero elements in operator matrix obtained using the FD discretization.*

directional derivatives and so forth.

The FD procedure applied to linear differential operators inevitably results in a finite-dimensional operator represented by explicitly computed elements of its matrix. This is an important feature of this approach affecting both memory and computational complexity of the eigensolvers based on the FD technique. The common feature of all the matrices generated by the finite difference scheme is a highly sparse structure and a usually very regular pattern of distribution of non-zero elements. Moreover, these matrices usually yield very large dimensions in modern applications that equal the number of sampling points (which is of order $10^2 - 10^3$ or more in every spatial dimension).

An example of a structure of the operator matrix obtained using the FD discretization has been shown in Figure 3. The Figure presents the distribution of non-zero elements in the matrix approximating the following second order non-symmetric differential operator (introduced in Section 2.2):

$$\mathbf{A}v = \nabla_t^2 v + \frac{1}{\varepsilon(x,y)}\left[\nabla_t \varepsilon(x,y) \times (\nabla_t \times v)\right]$$

where $\nabla_t(\cdot) = \left(\frac{\partial}{\partial_x}, \frac{\partial}{\partial_y}\right)(\cdot), \varepsilon(x,y)$ is a fixed, arbitrary function defined over two-dimensional space and v is an appropriate two-dimensional vector field defined over a 2D spatial domain.

This matrix shown in Figure 3 has a dimension of approximately 40000, which

corresponds to a discretization of a 2D vector field $\vec{v} = (v_x, v_y)$ over a 200 × 100 regular spatial grid and the number of the non-zero matrix elements equals approximately 200000. Although the matrix is non-symmetric it has a highly regular structure with 95% of its elements located on 5 diagonals: 0 (main diagonal), +2, −2, +199, −199. These five diagonals reflect the 5-point finite difference stencils replacing the appropriate derivatives. At this point it should be noted that the bandwidth of the discussed matrix depends substantially on the ordering of the elements of vector functions, obtained from discretizing the vector field $\vec{v} = (v_x, v_y)$. With an inappropriate ordering of elements one may obtain a matrix with a substantially increased bandwidth (or even a non-banded matrix). In our example the bandwidth equals approximately 400 and is minimal for the applied ordering, which puts first all the elements of the v_x field component before all the elements of the v_y field component. Still, the bandwidth could easily be increased if the elements of the v_x and v_y field components are mixed.

In any case the resulting matrix is sparse. Consequently, it may be noted that, although the dimension of the matrix n is large, the memory requirements are not of order $O(n^2)$ but of order $O(n)$ and the matrix may be stored in one of the sparse matrix storage formats, e.g. Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) which save memory and enable efficient handling of sparse matrices using specifically designed numerical procedures (cf. the description of the SPARSKIT numerical library — [31]). In the above example the storage requirements may be further reduced if the five diagonals are stored separately and solely the irregularly located elements are stored using e.g. the CSR format. Another optimization may be achieved if the equal values of matrix elements associated with five-point finite difference operators are excluded from the stored elements and included implicitly only while calculating e.g. a matrix-vector product. If regularities of the matrix are exploited a more efficient algorithm for calculating the matrix-vector product may be designed, with the numerical complexity approaching the cost of performing $5n$ multiplications and additions.

The above example shows the possible optimizations due to specific form of the matrix obtained in the Finite Difference discretization. — It is found that although the size of matrix is inevitably large both the memory and numerical complexities may be kept linear. The following section presents an opposite approach in which a dense operator matrix is obtained and consequently a different technique has to be applied to lower the cost of both the matrix storage and the computation of the matrix-vector or matrix-matrix product.

3.3 Method of Moments formulation

The method of finite-dimensional mapping described in this section is based on the representation of an operator by its products with chosen basis functions spanning a given functional space. This approach is known from the Method of Moments or its most important version — the Galerkin method.

Let us start the description of this discretization method with discussing the finite representation of functions. Let the domain X of a given operator $\mathbf{T}$ be a functional Hilbert space with a properly defined scalar product and $\{h_i\}_{i=1}^{\infty}$ be a complete orthonormal set of functions in the space X. According to the definition of completeness every function $u \in X$ is convergent to its Fourier series, being the expansion of u in terms of the basis functions:

$$\lim_n \left\| u - \sum_{i=1}^{n} (u, h_i) h_i \right\| = 0 \tag{18}$$

Consequently any function from the space X is represented by a sequence of the Fourier coefficients $\{f_i\}_{i=1}^{\infty} = \{(u, h_i)\}_{i=1}^{\infty}$. Truncating this sequence to a finite number of terms gives a wanted finite mapping of the function u:

$$\underline{u} = [(u, h_1),\ (u, h_2),\ \ldots,\ (u, h_n)]^T \tag{19}$$

The method of discretization of the operator $\mathbf{T}$ immediately follows from the above representation of the functions. Defining the elements of the $n \times n$ matrix $\underline{\underline{T}} = [c_{ij}]_{i,j=1}^{n}$ as:

$$c_{ij} = (\mathbf{T}h_j,\ h_i) \tag{20}$$

we obtain a finite-dimensional linear operator being a mapping of the operator $\mathbf{T}$ which has the following property:

$$\underline{\underline{T}}\,\underline{u} = \mathbf{T}u = [(\mathbf{T}u, h_1), (\mathbf{T}u, h_2), \ldots, (\mathbf{T}u, h_n)]^T \tag{21}$$

As already mentioned, the representation of the operator involving the matrix of scalar products given by the equation (20) is used by the method of moments in which this matrix is constructed explicitly. Unfortunately this may bring about a series of negative effects. Firstly, the matrix (20) may be dense and its explicit storage may require n^2 memory locations. Secondly, the matrix-vector product $\underline{\underline{T}}\,\underline{u}$ can involve $O(n^2)$ operations which may cause the computation time in the methods which are based on this representation to blow up for the increasing problem size n. (This effect is widely known e.g. from the Galerkin method.)

The question which emerges is whether it is possible to find an orthonormal basis (a complete set of functions) in the Hilbert space X such that either the storage cost of the discretized operator or the cost of calculating the discussed matrix-vector product may be significantly reduced even if the matrix $\underline{\underline{T}}$ (cf. equation (21)) is dense. The answer is positive for a certain class of functional Hilbert spaces chosen for the operator's domain. This wide class, being the most important one in a variety of application fields, may be defined as the space of square integrable

functions defined over a bounded region Ω, namely the $L_2(\Omega)$ space, with the scalar product defined as follows:

$$(u,v)=\int uv^* d\Omega \tag{22}$$

If, without significant loss of generality, we shall limit our discussion to the case of the L_2 space defined over a two-dimensional bounded rectangular region $\Omega = ([0, b] \times [0, a]) \subset R^2$ then the orthonormal bases which have the desired feature are the trigonometric complete sets of functions:

$$h_{ij}^x = A_{ij} \sin\left(\frac{i\pi x}{b}\right)\cos\left(\frac{j\pi y}{a}\right) \tag{23}$$

or

$$h_{ij}^y = B_{ij} \cos\left(\frac{i\pi x}{b}\right)\sin\left(\frac{j\pi y}{a}\right) \tag{24}$$

where A_{ij} and B_{ij} are properly defined normalization constants. If we consider e.g. a two-dimensional vector field $u = (u^x, u^y)$, where $u^x, u^y \in L_2([0, b] \times [0, a])$, then u may be represented e.g. by the following series:

$$u^x = \sum_{ij} c_{ij}^x h_{ij}^x$$

$$u^y = \sum_{ij} c_{ij}^y h_{ij}^y$$

If the above series are truncated then the emerging finite approximation of the function u is a vector of the Fourier coefficients:

$$\begin{aligned}\underline{u} &= \left[c_{11}^x, c_{12}^x, \ldots, c_{mn}^x, c_{11}^y, c_{12}^y, \ldots, c_{mn}^y\right] \\ &= \left[\left(u^x, h_{11}^x\right), \left(u^x, h_{12}^x\right), \ldots, \left(u^x, h_{mn}^x\right), \left(u^y, h_{11}^y\right), \left(u^y, h_{12}^y\right), \ldots, \left(u^y, h_{mn}^y\right)\right]\end{aligned} \tag{25}$$

The most significant observation about the above vector is that it is simply a vector of samples of the two-dimensional Fourier transform of the function $u = (u^x, u^y)$. Consequently, keeping in mind that the inner products are given by the integrals (22), the approximations of the vector elements may be numerically found using the two-dimensional Discrete Fourier Transforms (DFTs). In turn, the two-dimensional DFTs may be very efficiently computed by applying the Fast Fourier Transform (FFT) algorithm, proposed first by Cooley and Tukey (cf. [32]). The following section shows how this observation may be used to reduce both cost of calculating the $\underline{\underline{T}}\,\underline{u}$ product as well as cost of storing the operator matrix by applying the implicit, instead of explicit, matrix representation.

3.3.1 Calculation of the scalar products

According to the previous section computing of the $\underline{\underline{T}}\underline{u}$ product may be viewed as calculating the inner products (in the finite space) of $(\mathbf{T}u, h_{ij}^{x})$ and $(\mathbf{T}u, h_{ij}^{y})$ with the given vector of Fourier coefficients (25). This kind of approach enables one to develop a procedure of computing the $\underline{\underline{T}}\underline{u}$ product which does not require the explicit storage of the *dense* matrix $\underline{\underline{T}}$. Thanks to this both memory and computational cost may be reduced. The discussed operation may be performed in the following steps:

1. Using the given Fourier coefficients (refeqfour) calculate the values of the function u for a discrete set of points from the Ω spatial domain by computing a two-dimensional *backward* FFT.
2. Calculate the values of the $\mathbf{T}u$ function at the gridpoints of the domain Ω using the previously calculated values of u.

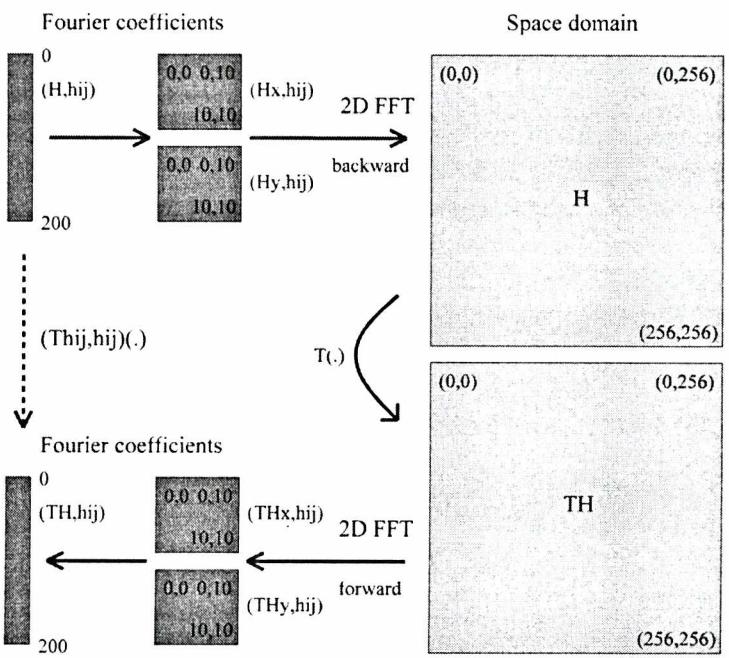

Figure 4. *Calculation of the matrix-vector product for the DFT domain operator formulation.*

3. Compute the inner products (the Fourier coefficients) $(\mathbf{T}u, h_{ij}^{x})$ and $(\mathbf{T}u, h_{ij}^{y})$ by performing a two-dimensional *forward* FFT.

The above scheme has been illustrated in Figure 4 if the function $u \equiv H$ is a two-dimensional vector field. In the Figure, the function $H = (H^x, H^y)$ is represented in the DFT domain by 200 Fourier coefficients, which corresponds to 10 expansion functions in every spatial direction for both H^x and H^y. (Referring to equations (23)

and (24) the indices run as follows: $i = 1, 2, \ldots, 10$ and $j = 0, 1, \ldots, 9$.) Then the discrete values of the function H are computed using two 2D backward FFTs. In the example we obtain two 256×256 arrays of samples of the function H in the 2D spatial domain. Then the operation $\mathbf{T}$ on H gives a matrix of samples of the $\mathbf{T}H$ function which is subsequently transformed using forward 2D FFTs to obtain the desired Fourier coefficients. One should note at this point an important relation which joins the discussed discretization (based on finite expansion series) and the Finite Difference (FD) method. In Step 2 of the above scheme one calculates the values of the $\mathbf{T}u$ function at the discrete gridpoints in entirely the same way as in the FD method. The additional Steps (1 and 3) are required to move back and forth between the spatial domain and the DFT domain.

Another aspect of this computation shown in Figure 4 is the difference in the dimensions of the matrices of Fourier coefficients (10×10) and the matrices in the spatial domain (256×256). In the presented scheme the function is oversampled in the spatial domain, which means that a reduced number of Fourier coefficients is calculated using more samples than necessary. This allows one to compute the first e.g. 100 Fourier coefficients with greater accuracy, while omitting all the other, containing larger (and often very serious) numerical error. The issue of estimating the numerical error of the Fourier coefficients is discussed in the following section.

The other question is: What is the numerical complexity of the algorithm? If K_x and K_y denote the lengths of the Fast Fourier Transforms, i.e. the number of sample points in the spatial domain in the x and y directions, respectively and the numbers of expansion functions used to represent the functions in the DFT domain equal N_x and N_y in the respectful directions then the cost of performing the steps 1 and 3 in the calculation of the matrix-vector product equals $O(4 N_x K_y \log K_y + 4 K_y K_x \log K_x)$. (The cost of performing a one-dimensional FFT of the length N is $O(N \log N)$ — cf. [33]). Denoting $K = K_x \cdot K_y$ and $N = N_x \cdot N_y$, this cost may be estimated at a level $O(K \log K)$ since the length of the FFTs (the number of sampling points in the spatial domain) should be proportional to the number of expansion functions. The next issue is estimating the cost of the $\underline{\underline{T}}\underline{u}$ product, where the function u is represented by N samples in the spatial domain. It is hard to evaluate this cost in the case of a general linear operator, still if only differential operators are considered (just as in the previous section) then the cost is given by $O(N)$. It is now seen that the overall cost of calculating the matrix-vector product in this representation equals $O(K \log K)$.

One may ask what are the advantages of this representation as compared to the FD finite-dimensional mapping in which the matrix-vector product could be calculated within the linear time cost. Clearly, in the DFT representation only the step 2 involves a similar number of computations as the entire matrix-vector product in the FD discretization. The advantage of the DFT representation may be seen if one compares the dimensions of the resulting operators. The size of the vectors in the DFT domain equals K which, due to the oversampling, is usually considerably smaller than the vector size resulting from the FD mapping. (e.g. For

the FFT length which equals 256, the number of applied expansion functions usually equals 20,40 or at most 60. Consequently in two dimensions the problem size equals e.g. 3600 in the DFT space as compared to approximately 66000 in the spatial domain.) So, the DFT representation usually reduces the problem size which has a very significant impact on the execution time the program solving the eigenvalue problem. Summing up, if the DFT representation is applied, the extra time spent on calculating in matrix-vector products is then regained by spending less time on solving the eigenproblem.

Reffering to the memory complexity, this method needs relatively little space to be able to calculate the matrix-vector product. The memory requirements include the space necessary to store the samples of the function $u = (u_x, u_y)$ in the spatial domain whose size equals $2N_xN_y = 2N$ and the space needed to perform Fourier transforms. In the Winograd version of the FFT algorithm (cf. [34] or [35]) the extra workspace needed to perform the one-dimensional transform will equal approximately $3N_x$ ($3N_y$). If the space needed to store the input/output vectors of Fourier coefficients is taken into account then the overall memory complexity equals: $O\left(2K + 2N + 6\sqrt{N}\right)$ (assuming that $N_x = O(N_y)$). In this estimation the cost of storing the operator matrix $\underline{\underline{T}}$ is not taken into account as the implicit storage is assumed. Clearly, if this matrix is stored explicitly, the memory requirements may increase dramatically.

3.3.2 Estimation of the numerical error in DFT integration

One of the difficult questions while dealing with the Discrete Fourier Transform is estimating the numerical errors introduced to the Fourier coefficients obtained in the computations and the quality of approximation of the input function by a finite Fourier series. The detailed discussion of various aspects of DFT error estimation may be found in the book by Briggs and Henson [33]. This section extends the discussion of Briggs and Henson to the case of functions defined over a two-dimensional rectangular region applying some of the results presented in the quoted

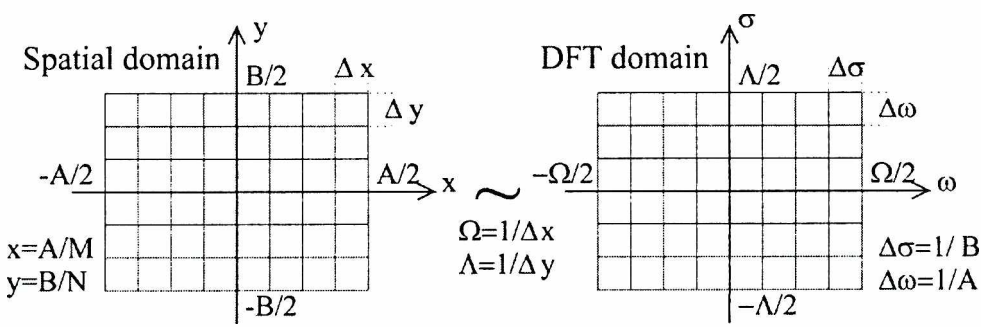

Figure 5. *The illustration of the spatial 2D domain and the DFT domain with the corresponding reciprocity relations.*

reference. The following discussion focuses on compactly supported functions, that is, the functions which vanish except a compact (bounded) region in the 2D space.

For such functions the numerical error in calculation of the Fourier coefficients while using DFT may easily be found if the relation between the DFT and the Fourier transform are explored. The following Fourier transform is associated with a given function $H_x \in L_2([-A/2, A/2] \times [-B/2, B/2])$:

$$\hat{H}_x(\omega,\sigma) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} H_x \exp(-2i\pi\omega x - 2i\pi\sigma y)dxdy \tag{26}$$

(In order to simplify the derivations the complex, exponential version of the transform has been applied in this section). If the function H_x has a compact support, (e.g. it vanishes outside the rectangular region $[-A/2, A/2] \times [-B/2, B/2]$) a simple relation joins the Fourier transform and the coefficients of the Fourier series of a periodic extension of the function H_x:

$$c_x^{mn} = \frac{1}{AB}\hat{H}_x(\omega_m,\sigma_n) \tag{27}$$

where $\omega_m = m/A$ and $\sigma_n = n/B$ define the discrete gridpoints in the Fourier transform domain. Apart from a continuous Fourier transform the function H_x also has a Discrete Fourier Transform. Denoting $H_x^{mn} = H_x(x_m, y_n)$, with $x_m = mA/M$ and $y_n = nB/N$, the vector $(H_x^{mn})_{mn}$ where $m = (-M/2+1), \ldots, M/2$ and $n = (-N/2+1), \ldots, N/2$ defines the given function at the discrete gridpoints in the spatial domain. The DFT of the function H_x is then given by the formula:

$$D\left(H_x^{st}\right)_{mn} = F_x^{mn} = \frac{1}{MN}\sum_{s=-M/2+1}^{M/2}\sum_{t=-N/2+1}^{N/2} H_x^{st}\exp\left(-i\frac{2\pi ms}{M} - i\frac{2\pi nt}{N}\right) \tag{28}$$

where $m = -M/2+1, \ldots, M/2$, $n = -N/2+1, \ldots, N/2$. The relation between the spatial domain and the DFT domain, together with reciprocity relations, has been shown in Figure 5.

In order to estimate $|F_x^{st} - c_x^{st}|$ — the error of computing the Fourier coefficients by using the DFT, the Poisson summation formula has to be applied. Briggs and Henson [33] give a derivation of this formula in one-dimensional case. Using an analogous approach results in the following 2D Poisson summation formula:

$$\sum_{j=-\infty}^{\infty}\sum_{k=-\infty}^{\infty}\hat{H}_x\left(\omega - \frac{j}{\Delta x}, \sigma - \frac{k}{\Delta y}\right) = \Delta x\Delta y\sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty} H_x^{mn}\exp(-i2\pi x_m\omega - i2\pi y_m\sigma) \tag{29}$$

where $\Delta x = A/M$ and $\Delta y = B/N$ (cf. Figure 5).

By applying the relations (27), (28), (29) one gets a formula which may be applied to estimate the error (which is in fact the error due to aliasing):

In order to estimate the above error, it is necessary to find a bound for the

$$\left|F_x^{st} - c_x^{st}\right| = \left|\sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} c_x^{s+iM,t+jN} - c_x^{st}\right| \tag{30}$$

Fourier coefficients c_x^{st}. In other words the main issue is estimating the rate of decay of the coefficients with the increasing indices s and t. This can be done if some additional assumptions are made about the function H_x defined over the region $\Omega = [-A/2, A/2] \times [-B/2, B/2]$. We assume that: 1) H_x has a finite number of discontinuities (i.e. discontinuity points or planes) in the region Ω, 2) H_x is differentiable (except the discontinuity points or planes), 3) For any curves lying on the surface $\partial H_x(x, y)/\partial x$ or $\partial H_x(x, y)/\partial y$ these curves are piecewise monotone. Note, that the condition 3) is in fact not too restrictive and is satisfied by virtually all functions which arise in applications. Under these assumptions the 2D analogue of the theorem presented in [33] (p. 187) may be written:

$$\left|c_x^{st}\right| \le \frac{C}{|s|^2 |t|^2} \qquad s \neq 0,\ t \neq 0 \tag{31}$$

$$\left|c_x^{0t}\right| \le \frac{C_1}{|t|^2} \qquad t \neq 0 \tag{32}$$

$$\left|c_x^{s0}\right| \le \frac{C_2}{|s|^2} \qquad s \neq 0 \tag{33}$$

where C, C_1 and C_2 are constants independent of s or t (but they depend on the number of discontinuity points or planes and the extreme values of H_x and its derivative at Ω). Now we are ready to estimate the error (30) using the above bounds together with formula (30):

$$\left|F_x^{st} - c_x^{st}\right| \le \frac{\tilde{C}_1}{|s|^2 N^2} + \frac{\tilde{C}_2}{|t|^2 M^2} + \frac{C_3}{N^2 M^2} \qquad t \neq 0,\ s \neq 0 \tag{34}$$

$$\left|F_x^{s0} - c_x^{s0}\right| \le \frac{C_4}{M^2} + \frac{C_5}{M^2 N^2} \qquad s \neq 0 \tag{35}$$

$$\left|F_x^{0t} - c_x^{0t}\right| \le \frac{C_6}{N^2} + \frac{C_7}{M^2 N^2} \qquad t \neq 0 \tag{36}$$

In the numerical tests we have compared the values of Fourier coefficients computed analytically with those computed using Discrete Fourier Transform. The tests included calculation of the Fourier coefficients of the matrix of a chosen differential operator **T** represented by the scalar products (compare formula (20)) in the DFT domain. The analytical values were calculated by explicit computation of appropriate values of sines and cosines combined accordingly to form of the input operator. In this way the matrix of Fourier coefficients was explicitly created. While using DFT, the calculation of the coefficients was based on computing the matrix-vector product for a given input vector, which involved calculating backward and

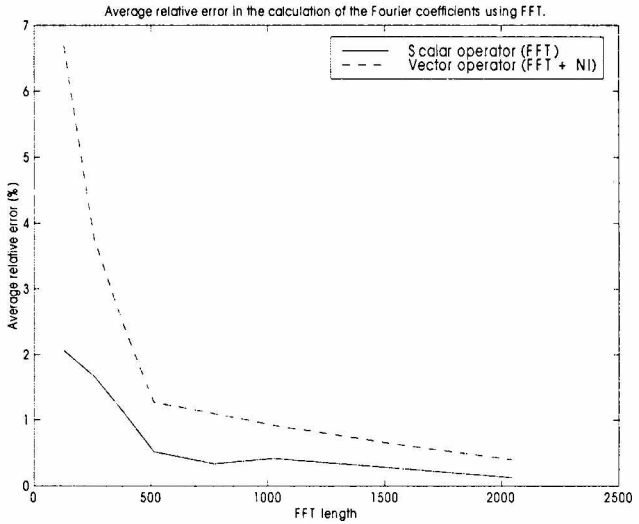

Figure 6. *Average error in calculating the Fourier coefficients using the DFT for two different input operators: the vector operator, given by formula (7) and the scalar operator, given by formula (8).*

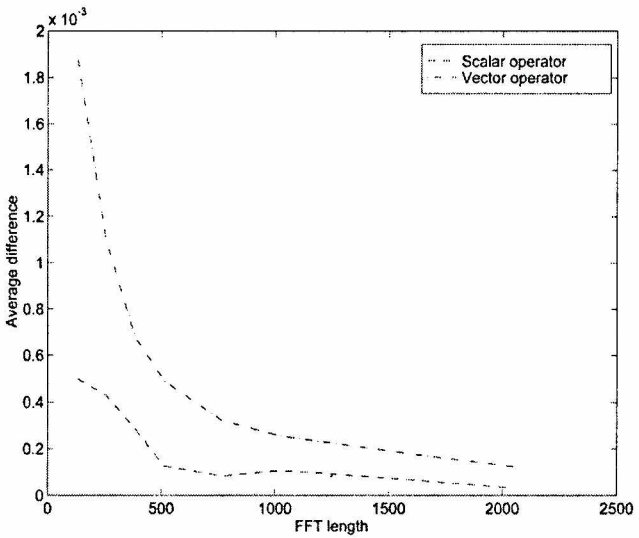

Figure 7. *Average absolute error in calculating the Fourier coefficients using the DFT for two different input operators: the vector operator, given by formula (7) and the scalar operator, given by formula (8)*

forward Fourier transforms, just as described in Section 2.1. The comparisons of the Fourier coefficients were made in two kinds of tests:

1. In the first approach we have calculated the DFT-based matrix-vector product for the input vector which had only one (e.g. i-th) non-zero element. The outcome of the product was simply the i-th column of the operator matrix, which could immediately be compared to the analytically computed values of the

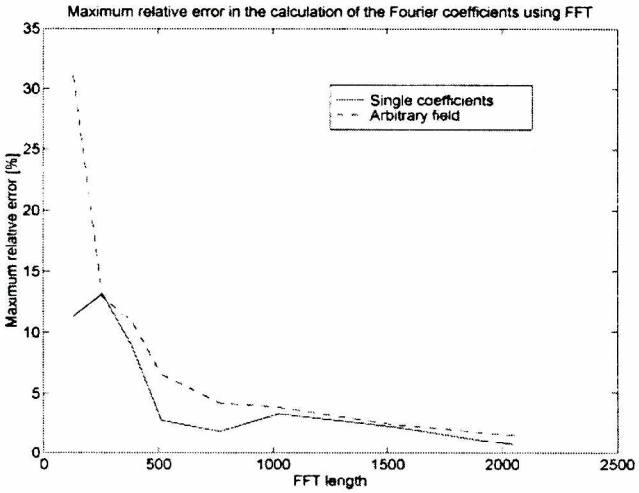

Figure 8. *Maximum error in computing Fourier coefficients using the DFT for two different input vector fields.*

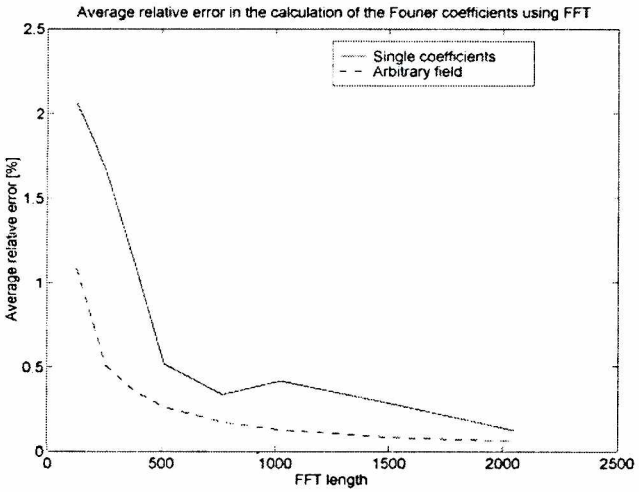

Figure 9. *Average error in computing Fourier coefficients using the DFT for two different input vector fields.*

corresponding Fourier coefficients.

2. In the second approach the DFT-based matrix-vector product was calculated for an arbitrary input vector. Consequently, the outcome had to be compared with outcome of the matrix-vector product for the analytically derived operator matrix and the same input vector. This test was intended to investigate the quality of the DFT approximation for a more realistic case.

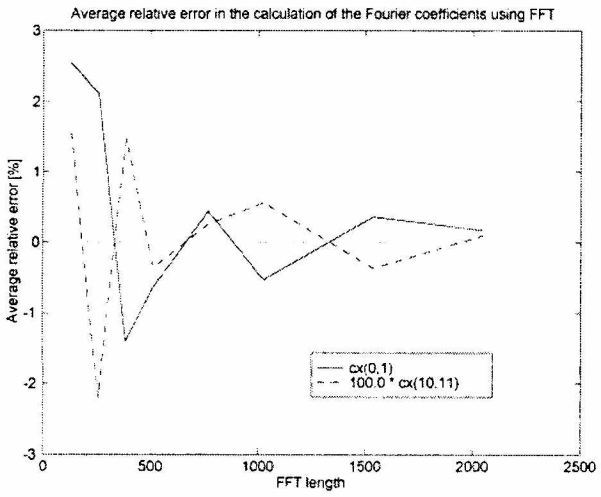

Figure 10. *Errors in computing the Fourier coefficients using the DFT for two chosen coefficients.*

In the tests the errors for the first 100 coefficients (with indices ranging from 0 to 9 or 1 to 10 in every spatial dimension) were calculated for different lengths of the Fourier transform, ranging from 128 to 2048 (both in x and y dimensions). Figure 6 shows the average relative errors in calculating the first hundred coefficients for different lengths of the DFTs. The average errors were computed as arithmetic mean of the absolute values of the relative errors. The DFT lengths were equal in both directions and were changed simultaneously. The Figure presents the results for two different operators: for the first one only FFT is used to compute the matrix-vector product; for the other one a hybrid algorithm using FFT and numerical integration (IRAM-FFT-NI), described in Section 6.1.1 is applied. Referring to absolute error, Figure 7 shows a graph presenting average absolute errors (computed as arithmetic mean of absolute values of differences between Fourier coefficients computed analytically and computed with FFT). In the Figure an approximately quadratic decrease (for N and $M<1024$) in the average absolute error is observed which stays in accordance with the estimations (36). (As $s << M$ and $t << N$ the estimations give a quadratic decrease of the error with the increasing M or N). The decrement in the average absolute error becomes slower (approximately linear) for larger FFT lengths which may be due to the increasingly important error due to floating-point arithmetics computations.

Figure 8 and 9 show a comparison of the results for the two different types of tests described above. Figure 8 presents a maximum relative error observed for the first hundred Fourier coefficients. As it is seen this error is higher in the case of an arbitrary input field than for the field with only one non-zero component. Still, if the average error is considered (Figure 9) the situation is contrary. The average relative error for an arbitrary input field is continually smaller. Moreover, the

approximately quadratic decrement in this error observed with the growing DFT length appears to be very stable. It is an optimistic result which shows that for general input the DFT-based algorithm may produce an outcome with a very predictable size of the error. The graph shown in Figure 10 presents the values of the relative error (in this case the original signs of the errors were maintained) for two different Fourier coefficients — one of them is a low order coefficient ($c_x^{0,1}$) and the other one is a higher order coefficient.($c_x^{9,10}$). In both cases the error decreases approximately in a quadratic order.

An interesting observation is that the relative error for the high order coefficient is approximately by a hundred times smaller than the corresponding error for the low order coefficient.

The general conclusion which may be drawn from the above tests is that the approximately quadratic decrement in the relative error is observed with the increasing (in both dimensions) length of the 2D Discrete Fourier Transform for N and $M < 1024$. Referring to the values of the relative errors, the average error for the first hundred coefficients stays at a level of a few percent for the transform length that equals 128. If the acceptable level of error equals 0.5 % then the DFT length should be increased to at least 1000. The results refer only to the first 100 coefficients and with more coefficients taken into account while representing a given operator in the DFT domain the average relative error will inevitably increase, so that longer transforms will be necessary to obtain the desired level of numerical error.

4. Characteristics of the distributed memory systems

Having presented the numerical algorithms being in the scope of interest of this study we will now discuss some issues in high performance computing which are crucial to the process of designing parallel algorithms, concentrating on the relations between scalable parallel system architectures and programming paradigms. We shall also briefly describe the specific features of parallel programming environments and answer how they influence the implementation of the algorithms expected to deal efficiently with large scale scientific and engineering computations.

4.1 Massively Parallel Processing

In the recent years the term of "Massively Parallel Processing" has gained a tremendous popularity among the users of hi-end computer systems who perform highly demanding numerical computations as it comprises the realized hopes for a platform suitable for large-scale simulations. The technology of parallel distributed memory supercomputers (including the virtual shared memory architectures), provided the only truly scalable environment offering computation speedup of tens or hundreds times with an adequate increment in memory storage capabilities.

At the same time many questions referring to programming techniques or data handling in distributed memory environments had to be resolved. The development of programming methods in the scalable parallel distributed memory systems has

led to defining certain dominant programming models, described later on. The growing understanding of the scalable parallel architectures also resulted in setting forth the key factors influencing the performance in distributed memory systems and creating a number of highly efficient parallel numerical libraries and parallel programming environments. The following sections discuss these issues in more detail, presenting the relations between the distributed memory architectures and parallel programming aspects.

4.2 Programming paradigms and parallel data decomposition

The formulation of new programming models or paradigms has always been closely related to the developments in computer system architectures, now classified by the Flynn's taxonomy which is based on the distinction of systems with single and multiple data and instruction processing streams (cf. Figure 11). The progress in computer technology brought about the dominance of the Multiple Instruction Multiple Data (MIMD) systems. This in turn resulted in rapid development of parallel programming techniques in MIMD systems. Two of them have gained a particular importance and have recently become dominant in parallel programming of MIMD systems. These are the data parallel programming and the message-passing programming.

Starting with the data parallel programming one has to mention that this model originates in vector supercomputers where programs applied highly efficient vector or matrix operations which inherently involved parallel data processing and distribution. The data parallel model assumes that a programming language offers intrinsic functions and mechanisms which enable the processors (processes or computational threads) to operate on global data. Consequently, this paradigm was a basic programming model in virtually all Single Instruction Multiple Data (SIMD) systems, such as architectures based on transputer matrices e.g. Connection Machine CM-200. In these machines a program defined the operations performed on global data, without specifying the data interchange scheme among the

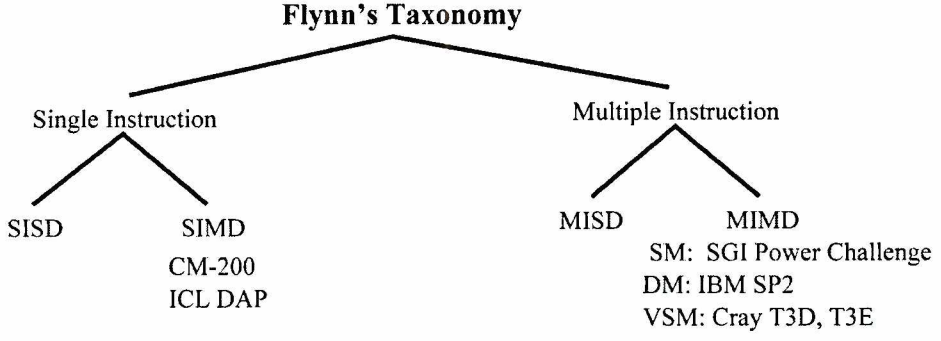

Figure 11. *Flynn's taxonomy. In the scheme: S = Single, M = Multiple, I = Instruction stream and D = Data stream. Within the MIMD machines the following architectures may be distinguished: Shared Memory (SM), Virtual Shared Memory (VSM) and Distributed Memory (DM) systems.*

processors and so the parallelism was obtained by the parallel data distribution. One thing has to be stressed about the data parallel paradigm in SIMD machines: the model assumes here that continuous synchronization occurs between the processors during parallel data processing. The situation is different in the case of modern MIMD systems, where the processors are said to be "loosely synchronized" with the same operations being performed on analogous data approximately at the same time. This constitutes one of the generalizations of the data parallel programming model on MIMD machines. The other important generalization is the possibility of defining local, private data whose distribution is handled by the compiler and which allows e.g. replicating some calculations on all the processors involved in the computations.

The data parallel programming model as described above seems to be inherently attributed to computer systems equipped with the global storage e.g. vector supercomputers or true shared memory systems. This is no longer a valid point of view while some novel parallel system architectures gain growing importance. These architectures include the *virtual shared memory* systems being in fact distributed memory scalable parallel systems equipped with efficient global addressing software and hardware mechanisms, which favour them from other MIMD systems for use with the data parallel programming paradigm. (The examples of such architecture is Cray T3E parallel system described in Section 7.1.2.)

The example of a parallel programming language which applies the data parallel paradigm is the High Performance Fortran (HPF). (An overview of the characteristics of this language may be found in [36] or [37]. One of the HPF implementations is described in [38].) In High Performance Fortran, which is a superset of Fortran77 and Fortran90, the parallelism of matrix and vector operations is obtained solely by defining parallel data distribution. HPF may be therefore considered a high level parallel programming language offering a simple platform for writing parallel codes which frees the programmer from issues concerning optimization of non-local data access patterns or design specifics of parallel matrix operations.

The other popular parallel programming model is the message-passing programming which is often considered a low-level programming paradigm. Indeed, this programming style offers great freedom in the design of a parallel algorithm at the price of the necessity of dealing with various programming issues concerning e.g. the design of interprocessor communication schemes. The message-passing programming paradigm assumes that the parallel computation takes place in an environment of interconnected multiple processing elements with each element having its own local memory. This model further assumes that there is no globally addressable memory (as in the data parallel paradigm) so that only one processing element may directly access its local memory. As it is seen this programming model is inherently attributed to distributed memory parallel MIMD systems, although it may also be ported to shared memory architectures. The other fundamental assumption about the model is that processors (processes, computational threads)

cooperate by explicit data exchange / communication using messages sent and received across the interconnection network. The process synchronization also occurs via message-passing.

In the message-passing model the programmer defines all details of the parallel design including e.g. the modes and the sequence in which the messages will be sent and received. (In the case of collective communication procedures the specific design schemes of message-passing may often be resolved on a lower level which explores the characteristics of a given parallel distributed memory environment in order to obtain higher efficiency.) The programmer is particularly responsible for designing correct communication e.g. avoiding deadlocks, livelocks or assuring the determinism in a parallel computation (unless specified otherwise). The necessity of dealing with all the above issues inevitably complicates the implementation of a message-passing based parallel program. On one hand this programming complexity is the main drawback of this paradigm and on the other hand it offers the programmer a free choice from a variety of solutions in order to make use of any potential parallelism enclosed in the problem being solved. Consequently the output parallel programs may result more efficient in a given parallel environment than their analogues constructed using the data parallel programming paradigm. The other fact is that the message-passing model allows one to implement various parallel data and computation mapping techniques (described in Section 3.1). While the data parallel model applies solely the static domain decomposition scheme, the message-passing programming may be used to implement e.g. functional parallel decomposition of a given problem or a dynamic load leveling scheme.

The two main standards which are most widely used in the distributed memory parallel systems and support the message-passing programming model (or rather provide a standard interpretation of this model) are the Message Passing Interface (MPI) standard specification and the Parallel Virtual Machine (PVM) communication system. The domination of these two programming instruments is a result of their versatility and portability which allows one to run the same codes on a variety of parallel systems ranging from massively parallel supercomputers to networks of workstations. A great number of publications is devoted to both MPI and PVM (cf. [29], [39], [40] (MPI), [41], [42] (PVM)) describing the capabilities and implementations of these message-passing systems.

4.3 Performance issues in parallel distributed memory systems

Some of the basic features and elements of parallel distributed memory systems have a decisive impact on the design and performance of parallel programs which are to be run in these environments. They are briefly described in the following items:

— *The cost of accessing non-local data significantly higher than the cost of accessing local data.* This very substantial feature of distributed memory systems is a result of using the message-passing mechanisms and protocols in order to transmit non-local data to the processor requesting a remote access by another processor. The message communication time which is composed of the communication startup time (related to the topology of a given network, network

protocol and the routing algorithms applied) and the transmission time (related to the physical bandwidth of the communication channel joining the units interchanging data) exceeds the analogous local memory access time by tens, hundreds (in the case of highly efficient interconnection networks in parallel supercomputer systems) or thousands times (in the case of standard networks connecting workstations). The situation is more balanced in virtual shared memory (VSM) systems where additional hardware circuitry supports software procedures handling non-local memory access requests (cf. [43]). Still, in any case the amount of inter-processor communication determines the performance of parallel programs in these systems and if this communication is not optimized the parallel bottleneck is inevitable. The parallel bottleneck demonstrates in the increment of the processors' idle time and the degradation of speed-up and efficiency with the increasing number of processors involved in computations.

— *Topology of the interconnection network.* The network topology may importantly influence the performance of collective communication operations and balancing of the execution times across the processing elements (PEs). In the case of a network of workstations the network topology may favour some processing elements which will result in a quicker communication between selected processing elements, in the imbalance in the execution time and eventually in a decrement in the overall efficiency. In the case of the interconnection networks applied in supercomputer systems, the uniform topologies of connections between the PEs are usually used. Nevertheless, in a certain topology some collective communication or reduction schemes may be favoured, e.g. algorithms involving only nearest neighbour (systolic) communication or algorithms applying global broadcast operations. The specifics of the interconnection topology are widely exploited in the design and implementation of parallel communication and numerical libraries provided for use in given parallel systems.

— *Relatively small local memory storage.* This fact is a consequence of distributing the memory resources across a number of processing elements. In distributed memory systems the memory usually equals about 128-256 MB per PE, while in the case of medium size shared memory systems the usual size of the global storage equals 4 GB. Consequently the designs of parallel algorithms have to assume balanced memory requirements for the processes to be executed on different PEs.

— *Local and global disk storage.* Different kinds of disk storage organization schemes are used in parallel distributed memory systems. In some designs the disk memory is attached locally to each processing element while in others there is only a single processing node that manages the entire disk storage system. In this case all the access requests have to be processed by this single node. In the case of tasks which have to use disk memory intensively or periodically it is advisable e.g. to distribute the disk read or write requests across the PEs in case

of many local disks or if all the processors (processes, computational threads) have to access a single disk storage system, the access requests should be distributed in time as to avoid bottlenecks.

The above points presented selected issues which have to be addressed while designing parallel programs in distributed memory parallel systems as they may greatly influence the performance of parallel programs. Typically, as many of these issues require conflicting design solutions, various trade-offs appear. Let us recall at this point the classical form of the Amdahl's law which imposes an upper bound on the speed-up S achieved by a program executed on P processors:

$$S \leq \frac{1}{\alpha + (1-\alpha)/P} \leq \frac{1}{\alpha} \tag{37}$$

where a is a fraction of the single-processor execution time of a given program, spent on operations which cannot be parallelized. In the original interpretation of the Amdahl's law the value of a was determined solely by the structure of the sequential program to be parallelized. Still, the process of parallelization involves modifications of a given sequential program which include e.g. introducing inter-processor communication. Consequently the value of a is modified (incremented) by these parallel overhead operations. In this context the value of α starts to depend on the applied parallelization strategy and also typically becomes a function of the number of processors P. At this point the role of the mentioned parallel design trade-offs, which aim at decreasing the value of α at least for a certain range of the number of applied processors P, becomes clear. A very typical parallel design trade-off arises e.g. when the amount of communication is being reduced by applying the replication of some calculations on all the processing elements. If the amount of replicated calculations becomes too large the problem of serial bottleneck appears. So, a kind of trade-off has to be applied in order to avoid both parallel and serial bottlenecks in parallel design of a given algorithm. The trade-offs also appear while dealing with distributed memory resources. For instance, some output data produced by a single PE may either be stored locally which saves inter-processor communication but increases imbalance in memory requirements per PE or may be distributed among all the PEs which requires communication but allows one to achieve better management of memory resources.

Many more examples could be given with different design solutions. Still, quite often the general solutions that could produce highly efficient parallel programs cannot be given due to the diversity of parallel distributed memory systems. In this case achieving efficient, scalable parallel programs often requires application of various programming tools available in particular systems offering highly optimized parallel solutions for many computational tasks.

4.4 Parallel programming environment - discussion of the available tools

In this section we would like to outline the role of parallel programming tools, especially parallel libraries in developing parallel codes and enhancing their

efficiency in scalable distributed memory environments. Let us address these issues in the following points:

— *Compilers*. The role of compilers in distributed memory systems depends primarily on the programming model applied to develop a target parallel program. If the data parallel programming paradigm is used then the role of the compiler is principal as managing the issues of non-local data access, process synchronization and communication is left to the compiler, while the programmer defines only a parallel data distribution pattern. In turn, if the message-passing programming model is applied, the role of the compiler is rather limited. As in this programming style all the inter-processor communication is handled by the library routines (e.g. MPI or PVM routines) and the entire parallel design is left to the programmer only a serial compiler is needed and its role is reduced to performing standard serial optimizations. Obviously in this case the role of the message-passing libraries becomes particularly important.

— *Inter-processor communication libraries*. Focusing on the two most important libraries, i.e. Parallel Virtual Machine (PVM) library and communication system and the implementations of Message Passing Interface (MPI) it is important to indicate some significant differences between these two parallel programming tools. First of all, PVM is mainly designed for use in parallel network environments. This fact has the following consequences: 1) PVM contains features which are particularly useful in the network environment e.g. the dynamic task creation or defining relative speeds of the processing elements forming the virtual machine. 2) The PVM communication routines are not optimized for use with any particular network topology. This second point indicates that if the portable versions of PVM library implementations are applied in parallel supercomputer systems, the capabilities of usually highly efficient interconnection networks in these systems may not be fully exploited. This especially refers to collective communication procedures whose performance may be significantly enhanced if the characteristics of the network topology are taken into account in their design. Although there exist implementations of PVM library for massively parallel distributed memory systems (e.g. the PVMe library for IBM SPx [44]), this library offers very little in the supercomputing environment as compared to the MPI library. In turn, MPI appears to be specifically designed for use in supercomputer systems offering a great variety of communication modes and types, including various collective communication or global reduction operations. In this case the role of implementation which exploits the network capabilities (e.g. the capabilities of switches applied to connect the processing elements or specifics of the protocols applied in a given interconnection network) becomes crucial and consequently the optimized communication libraries supplied by the hardware vendors may provide much higher performance than the analogous portable implementations. Lastly, one has to mention that the original MPI standard assumes a static

communication system. Therefore, if sophisticated parallel systems involving dynamic task creation or parallel input / output operations are to be developed, some newer solutions and tools should be applied e.g. PVM v. 3.4 [45] or MPI-2 [46]. Summing up, all the above facts have to be taken into account while designing and implementing a parallel application. In the process of porting parallel codes to specific parallel systems the appropriate choice of tools, including parallel inter-processor communication libraries, should be made which may result in a serious improvement of the parallel performance.

— *Parallel numerical libraries.* The role of parallel numerical libraries in the process of designing computational applications to be run in distributed memory parallel environments cannot be underestimated. With the development of parallel algorithms in such application areas as basic linear algebra, differential equations, FFT and signal processing or eigenproblem analysis various parallel numerical libraries emerged. Among the most important portable libraries covering a large number of basic algorithms used in linear algebra one has to mention PBLAS (Parallel Basic Linear Algebra Subroutines) and ScaLAPACK (Scalable Linear Algebra Package) which extends PBLAS. Both the libraries depend also on the BLACS (Basic Linear Algebra Communication Subprograms) (cf. [47]) library which serves as a base for inter-processor communication and provides a parallel programming interface. Apart from public, portable implementations of the numerical libraries also a number of native implementations associated with given parallel systems have been created. The most widely known libraries from this group, implementing many BLAS and LAPACK routines, are: LibSci (provided by Cray Research) [48], Parallel Engineering and Scientific Subroutine Library (PESSL [49]) (IBM) or NAG library (from Numerical Algorithms Group [50]). The libraries cover many application areas in scientific computations and provide its users with interfaces to different inter-processor communication systems: e.g. BLACS or HPF (in case of PESSL - cf. [49]), PVM or MPI (in case of NAG — cf. [50]). Apart from these most widely known products many smaller parallel programming libraries which cover more specific areas of applications are available, including PARPACK library (widely described in the following section) or PIM (Parallel Iterative Methods package used to solve large systems of linear equations using conjugate-gradient approach [51]). (A much broader overview of available parallel numerical libraries may be found in the report [52].) The principal role of parallel numerical libraries is facilitating the process of development of numerical solvers in parallel environments. While the parallel libraries provide the ready designs and implementations of numerical algorithms, they also offer ready parallel data distribution schemes. Consequently, if we choose a certain numerical library, we also have to accept the available parallel interface which can be more or less suitable for our parallel system as well as incorporate the parallel data distribution or mapping model supported by the library. This

important aspect of using parallel numerical libraries has to be taken into account as, although in most cases it simplifies the process of design, it also limits the possible parallelization strategies.

5. Implementation of the operator eigensolvers in parallel distributed memory systems

This section describes parallel design and implementation of the algorithms of solving operator eigenproblems presented in Section 2. The implementations of the iterative solvers are discussed jointly with the parallel designs of matrix-vector products for the discretization schemes discussed in Section 3, as to provide the reader with a description of complete methods which can immediately be used to solve eigenproblems for a wide class of operators.

Three parallel solvers will be presented in this section:

— *IRAM-FD solver*, based on the IRAM iterative process and the Finite Difference (FD) discretization of the input operator.

— *IRAM-FFT solver*, based on the IRAM iterative process and the implicit discrete representation of the operator, applied jointly with the FFT algorithms to enhance the efficiency of the method.

— *IEEM-FFT solver*, providing a parallel implementation of the IEEM-FFT algorithm, described in [18] and based on the Iterative Eigenfunction Expansion Method presented in Section 2.

The base for the implementation of the two first parallel solvers (IRAM-FD and IRAM-FFT) is the PARPACK library, described in the following section and offering portable parallel implementation of the IRAM iterative algorithm, ready for use in distributed memory systems. The tasks which have to parallelized independently include:

— The matrix-vector product operation for the matrix operator discretized using the FD mapping technique.

— Two-dimensional backward and forward Fast Fourier Transforms.

— The matrix-vector product in the IRAM-FFT algorithm which requires calculation of the appropriate inner products, involving computation of 2D FFTs.

— The basic iteration of the IEEM-FFT algorithm, involving computation of the inner products (as described in the previous item) and global norms (requiring inter-processor communication in a parallel implementation).

5.1 The P_ARPACK library — The Arnoldi solver for MPP platforms

This section presents a description of the Parallel ARnoldi PACKage (P_ARPACK) — a portable library implementing the Implicitly Restarted Arnoldi Method (IRAM) for distributed memory parallel systems.

The P_ARPACK software has been developed at Rice University (cf. [53]) and provides a versatile package of Fortran77 subroutines for solving either symmetric

or non-symmetric, real or complex matrix eigenproblems. The important feature of the Arnoldi algorithm which has been exploited in the design of the library routines is that the method does not require any explicit form of the input operator matrix to be used. Instead, all the information on the considered operator is passed via the matrix-vector product. This has been used by introducing the *reverse communication* interface. On one hand, this interface enables the subroutines that perform the Arnoldi algorithm iteration to be independent of the input matrix storage format and, on the other hand, it makes the user of P_ARPACK free to choose the most appropriate method of computing the matrix-vector product for a specified input matrix operator. The general framework of a parallel program calling P_ARPACK routines in a reverse communication loop is shown in Figure 12 and constitutes a basis for the solvers presented in the following sections.

The central point of the presented program is a call to the `pdnaupd()`

```
c ----------- Parameter selection for pdnaupd() ----------
c
   comm = MPI_COMM_WORLD ! Set the communicator
   call MPI_Comm_size(comm,     ! Determine the number of
          nprocs, ierr)         ! processors used
   n     = N          ! size of the problem
   nev   = NEV        ! number of eigenvalues to be computed
   ncv   = NCV        ! number of orthogonal columns of V
   nloc  = n/nprocs   ! Determine local size of the problem
   bmat  = 'I'        ! standard eigenvalue problem
   which = 'LM'       ! find eigenvalues with largest magn.
   tol   = 1.e-8      ! set the desired accuracy
   ido   = 0          ! first call to reverse communication
   info  = 1          ! resid contains the initial vector
   do 100 i = 1, nloc ! initialize resid as a vector
      resid(i) = 1.d0 ! with 1's as all elements
 100 continue
   iparam(1) = 1      ! exact shifts with respect to H
   iparam(3) = 1000   ! maximum number of updates
   iparam(7) = 1      ! Mode set to 1
c
c -------------- Reverse communication loop --------------
c
 200 continue
       call pdnaupd(comm, ido, bmat, nloc, which, nev,
   &            tol, resid, ncv, v, ldv, iparam,
   &            ipntr, workd, workl, lworkl, info)
c
       if (ido .eq. -1 .or. ido .eq. 1) then
c        Compute matrix-vector product: A*v
         call Av(nloc, workd(ipntr(1)), workd(ipntr(2))
c
         go to 200 ! Loop back to call pdnaupd() again
       endif
c ------------------------------------------------------------
```

Figure 12. *Calling* `pdnaupd()` *P_ARPACK subroutine, solving a non-symmetric real eigenproblem in a reverse communication loop.*

subroutine which implements the IRAM algorithm for a non-symmetric real eigenproblem. This call is preceded by the initialization of various parameters defining the problem, including: `n` — the global problem size, `nloc` — the local problem size for a given processor (process), `nev` — the number of eigenvalues to be found, `bmat` — type of the eigenproblem (standard/generalized), `which` – – which part of the operator spectrum is to be considered (e.g. eigenvalues with the largest real part or the largest modulus). The `info` parameter determines whether an initial vector v_1 will be submitted. If `info=1` the initial vector is stored in the `resid` parameter. Otherwise, the initial vector is random. The `tol` parameter determines the stopping criterion for the Arnoldi factorization. The algorithm stops if the condition:

$$\|Au_i - u_i\lambda_i\|_2 \leq tol \cdot |\lambda_i| \tag{38}$$

is satisfied for all λ_i. Parameters `iparam(1)` - `iparam(8)` define various options of the algorithm including the maximum number of Arnoldi updates allowed or types of shifts used in the polynomial filtering process. A detailed description of all the parameters of P_ARPACK routines may be found in [15].

Another important design feature of the P_ARPACK library is the possibility of applying the Single Program Multiple Data (SPMD) programming style, regarded the most efficient and transparent in the parallel message-passing programming. This programming technique allows one to write a single code (such as shown in Figure 12) to be executed on all the processors. Once again the reverse communication interface to the P_ARPACK subroutines allows the user to choose a convenient parallelization strategy for the matrix-vector product operation.

Last but not least, the P_ARPACK library offers portability across a wide range of distributed memory parallel systems (including networks of workstations) by implementing its parallel routines using standard inter-processor communication libraries: the Message Passing Interface (MPI) ([39]) and the Basic Linear Algebra Communication Subprograms (BLACS) ([47]).

5.2 Parallel design of the Arnoldi factorization

Apart from knowing the functional characteristics of the routines implementing the Implicitly Restarted Arnoldi method in the P_ARPACK library, it is important to be conscious of the parallel design features of the basic Arnoldi factorization proposed in this library by Maschhoff and Sorensen ([53]).

If, once again, the formula for the Arnoldi factorization is examined:

$$\underline{\underline{A}}\ \underline{\underline{V_k}} = \underline{\underline{V_k}}\ \underline{\underline{H_k}} + \underline{f_k}\ \underline{e_k^T} \tag{39}$$

where the symbols have the same meaning as in Section 2.4.1, then the parallelization scheme illustrated in Figure 13 may be described as follows:

— the $k \times k$ upper Hessenberg matrix $\underline{\underline{H_k}}$ is replicated on every processor,

— the matrix $\underline{\underline{V_k}}$ is block-distributed across a one-dimensional processor grid,

— $\underline{f_k}$ and workspace are distributed accordingly,

— parallel data distribution in the input matrix $\underline{\underline{A}}$ is chosen by the user. Still, as the outcome of the matrix-vector product has to be distributed analogously

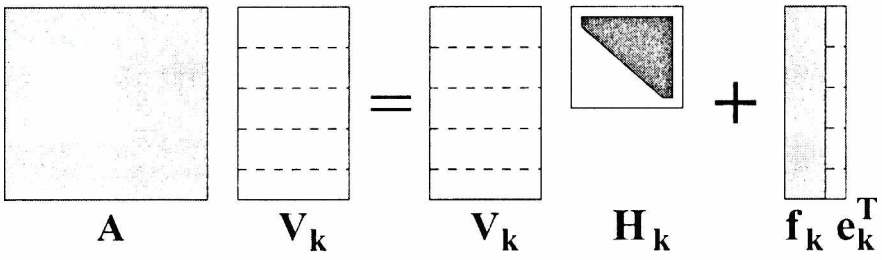

Figure 13. *Parallel block data distribution during the Arnoldi factorization applied in the P_ARPACK library.*

as the matrix $\underline{\underline{H_k}}$, the decomposition of the matrix will typically be commensurate with this block distribution.

According to the conclusions obtained in Section 2.4.3, the memory storage requirements for the applied data distribution equal $n_{loc}O(l) + O(l^2)$ per processor, where $n_{loc} \approx n/P$ (P equals the number of processors) and $l = k + p$ equals the sum of the number of eigenvalues to be found and to be filtered-out.

A crucial aspect of parallel implementation in distributed memory systems is the size of messages communicated between the processors during the execution of the algorithm. Referring to P_ARPACK and Arnoldi factorization there are only two communication points. One of them is computation of the norm of the distributed vector $\underline{f_k}$ and the other is the orthogonalization of $\underline{f_k}$ to $\underline{\underline{V_k}}$ using the MGS algorithm, where the global scalar products of a given vector with the columns of the matrix $\underline{\underline{V_k}}$ have to be computed. In the MPI implementation these global norms and sums are calculated using a global reduction procedure `MPI_Allreduce(.)`. For a single iteration in the Arnoldi factorization, the overall size of elements communicated across the processors is extremely low and is of order $O(Pk)$, where P denotes the number of processors and k equals the number of eigenvalues to be found.

A certain kind of trade-off may be observed in the parallelization strategy applied in the IRAM iteration. As all the operations on the upper Hessenberg matrix $\underline{\underline{H_k}}$ are replicated on each processor, the communication of the results is not needed. Nevertheless, this introduces some redundancy to the algorithm that may

lead to a serial bottleneck as the size k of the matrix increases. This may eventually cause the lack of scalability of the method.

According to the results obtained in Section 2.4.3 the numerical complexity of the parallel version of a single update in a p-step IRAM algorithm equals $O(p^2\, n_{loc})$ or (if $p = O(k)$) $O(k^2 n_{loc})$ per processor, where n_{loc} is the local dimension of the problem., in the above estimations the costs of performing the parallel matrix-vector operation and storing the operator matrix, which largely depend on the choice of finite-dimensional mapping method, have been excluded. This problem will be addressed in the following sections.

5.3 Parallel Arnoldi solver for FD operator discretization

This section presents two portable implementations of a parallel solver of a non--symmetric linear operator eigenproblem based the Implicitly Restarted Arnoldi Method (IRAM), for operators discretized using the Finite Difference method.

The implementations are based on calling the P_ARPACK library routines which perform the IRAM iteration. Consequently, according to the description of P_ARPACK from the previous section, the implementation of the solver follows the *reverse communication* scheme shown in Figure 12. Therefore, the implementation of the solver may concentrate on only two aspects: 1) Defining parallel data distribution, which includes distribution of the vectors and the discrete operator and 2) Implementing the parallel operation of matrix-vector product which corresponds to the applied parallel data distribution.

5.3.1 Implementation of the matrix-vector product in parallel

As discussed in Section 3.2 the matrix obtained in the FD mapping is a highly sparse matrix with very regular structure. Consequently, a simple parallel block data distribution scheme may be applied. In this distribution each of the processors

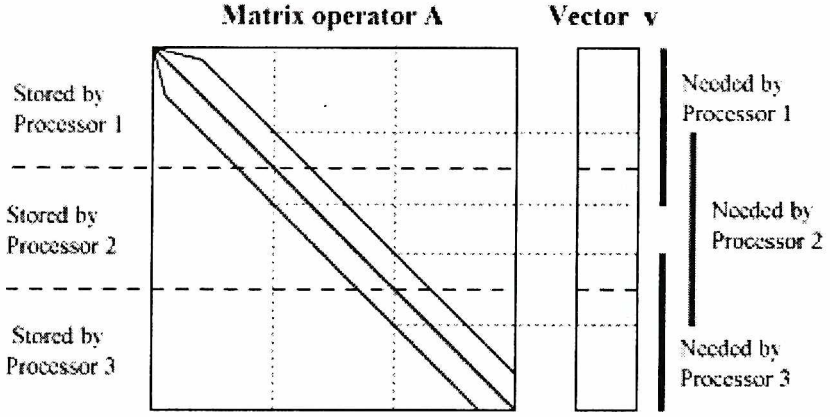

Figure 14. *Schematic of a block distribution among the processors of a quasi five-diagonal sparse matrix and the corresponding vector. The Figure shows that in order to calculate the matrix-vector product with such distribution the grayed parts of the vector* $\underline{y}$ *need to be communicated between the neighbouring processors (processes).*

(processes) stores a specific range of rows of the operator matrix and a corresponding range of elements of the input vectors. This has been illustrated in Figure 14.

The matrix presented in the Figure shows the discretized differential operator discussed as an example in Section 3.2. With most of the non-zero elements located on five diagonals, the presented distribution minimizes the inter-processor communication necessary to compute the matrix-vector product. The regions of the input vector $\underline{v}$ which have to be communicated between the pairs of neighbouring processors have been shown in the Figure as grayed regions. In our example, as the matrix (and the vector) size equaled 39700, the number of the vector elements to be communicated between each pair equaled approximately 400.

In order to investigate the importance of the matrix-vector product operation on the overall parallel performance of the solver this operation has been implemented using three different methods, including different formats for storing the operator matrix:

— In the first implementation all the non-zero elements of the local part of the matrix belonging to an appropriate processor are stored using the Compressed Sparse Row (CSR) format. In this representation the information on the regular quasi five-diagonal structure of the matrix is not used. In this case the overall number of memory locations needed to store the matrix equals $2 \cdot nnz + n + 1$, where n is the matrix dimension and nnz is the number of the non-zero matrix elements. In order to perform the matrix-vector product all the elements of the vector $\underline{v}$ are communicated, so that the entire vector is "known" to all the processors. In the case of the MPI-based implementation this communication may be performed by applying a single high level collective communication routine, such as `MPI_AllGather(.)`. Clearly the overall memory requirements have to be incremented by $P \cdot n$, where P is the number of processors used. Another issue which has to be addressed in a parallel implementation is the overall size of the messages communicated between the processors. In the considered case, this size is relatively high and equals approximately $(P-1)n$ elements. After the communication has been completed each of the processors calls a general purpose SPARSKIT library ([31]) routine `amux(.)` which calculates the matrix-vector product for the appropriate range of rows of the distributed matrix. As it is seen, the procedure presented above may be used to perform a parallel matrix-vector product for an arbitrary block-distributed sparse matrix. The two other implementations use the characteristics of the matrix to reduce both complexity of serial operations and storage, as well as the size of the inter-processor communication.

— In the second implementation a serial optimization is performed. If the matrix discussed in Section 3.2 is considered, then on each of the processors a hybrid type of storage may be applied. The five diagonals are stored separately in the $5 \times n$ matrix and the remaining 5% of the elements are stored in the CSR format. The resulting storage requirements are lower than in point 1 and (assuming that

$5n \approx 0.95nnz$) equal approximately $0.95nnz + 2 \cdot 0.05nnz + n + 1 = 1.05nnz + n + 1$ elements. Similarly as in the first implementation the entire vector $\underline{v}$ is communicated across all the processors. The algorithm of calculating the matrix-vector product is also a hybrid one and consists in computing the product of the matrix elements located on the five diagonals with the vector elements and computing the product of the remaining matrix elements with the vector using the `amux(.)` routine.

— In the third implementation the serial part of the computations and the matrix storage scheme remain the same as in the previous implementation. Instead, the inter-processor communication is significantly optimized. Once again the information on the structure of the matrix operator is used. In the considered example the band of the matrix (understood as the maximum difference between the row indices for non-zero elements located in the same column) equaled 400. Consequently, only 400 elements had to be communicated as to enable every processor to compute its part of the matrix-vector product (cf. Figure 14). In consequence, no collective communication routines are necessary and a simple two-step inter-processor communication scheme shown in Figure 15 may be used. This Figure shows how the necessary parts of the input vector $\underline{v}$ are communicated using a series of simple blocking send and receive procedures. The pseudo-Fortran77 code of this operation is shown in Figure 16. In our example the overall size of the data communicated between the processors decreases dramatically and equals approximately $(P - 1) \cdot 0.01n$ elements, as $400 = 0.01 \cdot 40000 \approx 0.01n$. This means the size of communication is reduced by a hundred times! Clearly, the presented scheme of communication may be applied to arbitrary banded matrices provided that $b < (2 \cdot n/P)$, where b is the

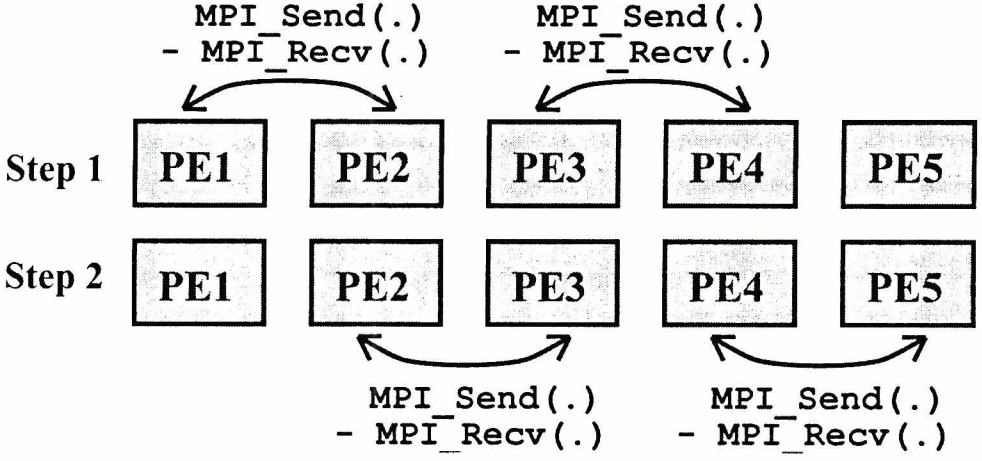

Figure 15. *Two-step communication scheme used in the parallel matrix-vector product calculation.*

bandwidth of the matrix of size n, block-distributed among P processors. If this condition is not satisfied more complicated schemes have to be applied involving not only pairs of neighbouring processors.

```
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)

if (mod(rank,2) .eq. 0) then
   if(rank .lt. (size-1)) then
      call MPI_Recv() # receive data from 'rank+1'-th processor
      call MPI_Send() # send data to 'rank+1'-th processor
   end if
   if(rank .gt. 0) then
      call MPI_Recv() # receive data from 'rank-1'-th processor
      call MPI_Send() # send data to 'rank-1'-th processor
   end if
else
   if(rank .gt. 0) then
      call MPI_Send() # send data to 'rank-1'-th processor
      call MPI_Recv() # receive data from 'rank-1'-th processor
   end if
   if(rank .lt. (size-1)) then
```

Figure 16. *Pseudo-code involving MPI calls showing the optimized inter-processor communication scheme during the matrix-vector product operation.*

```
      call MPI_Send() # send data to 'rank+1'-th processor
      call MPI_Recv() # receive data from 'rank+1'-th processor
   end if
end if
```

The implementations discussed above used Message Passing Interface (MPI) as an inter-processor communication platform and exploited the MPI interface provided within the P_ARPACK library. These programs may be ported to various parallel systems, still it is recommended to use them in scalable distributed memory supercomputer systems. The main reason is that the vendor supplied implementations of the MPI standard provide library functions, containing high-level collective communication and global reduction routines which are highly optimized for use in specific system architectures. Therefore, while designing and implementing the parallel solver using MPI, part of the complexity may be hidden in high-level inter-processor communication routines without loosing any efficiency.

5.3.2 PVM-BLACS implementation

Apart from the MPI-based parallel functions, the P_ARPACK library also provides routines supporting Basic Linear Algebra Communications Subprograms (BLACS) inter-processor communication platform. Availability of the BLACS version of this parallel library extends its functionality and enables its users to apply high level (as compared to MPI) designs of communication schemes in their parallel solvers. Generally speaking, using BLACS gives simpler and faster

implementation at the price of giving up the versatility offered by MPI.

The BLACS version of the P_ARPACK library not only does allow one to implement the entire solver using this communication library but also provides means to use concurrently different inter-processor communication platforms on the implementation level. This somehow non-standard use of BLACS-P_ARPACK library has been presented in the rest of this section.

The parallel MPI-based solver described in the previous section may also be implemented using two communication platforms simultaneously. Firstly, the BLACS interface to P_ARPACK routines may be applied and secondly Parallel Virtual Machine (PVM) (cf. [42]) library functions can be used to implement the solver, including the routine for calculating the parallel matrix-vector product.

```
program parpackfd

include 'fpvm.h'                    # Include the PVM header file

call pvmfmytid (mytid)              # Enroll in the virtual machine

call pvmfjoingroup ('solver', me) # Join a group of processes

if ((me .eq. 0) .and. (NPROCS .gt. 1)) then
    tids1(0) = mytid
    call pvmfspawn ('parpackfd', PVMDEFAULT, '*' ,    # Spawn
    NPROCS-1, tids1(1), info)                         # processes
end if
#
# Synchronize all the processes
#
call pvmfbarrier ('solver' , NPROCS, info)
#
# Communicate the array of tids among all the processes
#
if (me .eq. 0) then
    call pvmfinitsend (PVMDEFAULT, info)
    call pvmfpack (INTEGER4, tids1(0), NPROCS, 1, info)
    call pvmfbcast ('solver', 1, info)
else
    call pvmfrecv (-1, 1, info)
    call pvmfunpack (INTEGER4, tids1(0), NPROCS, 1, info)
end if
#
# Initiate pvm processes in the BLACS domain
#
if (NPROCS .eq. 1) then
call SETPVMTIDS (NPROCS, mytid)
else
call SETPVMTIDS (NPROCS, tids1(0))
end if
#
call BLACS_PNFO (mypinfo, nproc)
#
```

```
#
# Get the BLACS context
#
call BLACS_GET (0, 0, context)
#
# Initiate the block data distribution by rows
#
call BLACS_GRIDINIT (context, 'R', nproc, 1)
#
# Obtain information on the distribution
#
call BLACS_GRIDINFO (context, nprow, npcol, myprow, mypcol)
#
# Synchronize all the processes
#
call pvmfbarrier ('solver', NPROCS, info)
#
# Call the main solver routine
#
########################################
#
call solver (me, context, NPROCS, tids1)
#
########################################
#
# Leave PVM group and the virtual machine
#
```

Figure 17. *This fragment of Fortran77 code presents main points of an SPMD program in which both PVM and BLACS communication subsystems are initialized to be used jointly in an arbitrary parallel solver (called by the* `solver(.)` *subroutine).*

```
call pvmflvgroup ('solver', info)
call pvmfexit (info)

stop
end
```

The combination of the two communication platforms requires constructing a certain kind of a "wrapper" for the parallel numerical solver in which both systems are coherently initialized. Such general construction has been shown in Figure 17 which presents an SPMD (Single Program Multiple Data) parallel Fortran77 code. In the Figure, the call to the `solver(.)` routine, which performs all numerical calculations, is preceded by several initialization steps:

1. The first processor (process) enrolling in the virtual machine creates the group of processes named "solver" and spawns a given number of processes (`NPROCS-1`).
2. All the spawned processors (processes) executing the program enroll in the Parallel Virtual Machine and join the "solver" group.
3. After these steps, all the processes are synchronized using `pvmfbarrier(.)`.
4. The array of tids containing task ids of all the members of the "solver" group is

communicated to all the processes from process 0. In this way, all the processes may identify other processes involved in the parallel solver.

5. Following is the BLACS initialization which starts with a call (executed by all the processes) to the `SETPVMTIDS(.)` BLACS library function. (The `SETPVMTIDS(.)` routine belongs to "unofficial" functions of the BLACS library and is available only in the PVM-BLACS implementation of this library.) This function establishes processes with provided task ids as processes which take part in the BLACS communication.

6. The next important step is obtaining the BLACS context by the processes by calling the `BLACS_GET(.)` routine. The BLACS context (an equivalent of the MPI intra-communicator) establishes a "communication universe" for the processes involved in the solver.

7. The following step (optional at this point) defines the parallel data distribution type used by the solver, by calling the `BLACS_GRIDINFO(.)` routine. In the presented program the block distribution by rows has been applied (cf. Figure 17).

Afterwards the `solver(.)` routine is called with parameters defining the number of processes involved in the computation, their tids related to ordinal numbers in the "solver" PVM group (and commensurate with the ordinal numbers of the processes established by BLACS), the BLACS communication context and the ordinal number of the process calling the routine. This set of parameters is sufficient to establish a coherent communication using jointly BLACS and PVM.

The implementation of the actual parallel Arnoldi solver using BLACS and PVM is entirely analogous to its MPI implementation. In the calls to P_ARPACK library routines the MPI communicator is replaced by the BLACS context and in the implementation of the matrix-vector product (the optimized version) the same communication scheme from Figure 15 is applied with `pvmfsend(.)` and `pvmfrecv(.)` function calls and additional data packing and unpacking routines replacing the MPI blocking send and receive functions.

The main advantage of the presented implementation is that it extends the functionality of P_ARPACK which originally does not have a PVM version of its library routines. Consequently, using the scheme shown in Figure 17, the programs which applied P_ARPACK routines can also make use of the capabilities of the Parallel Virtual Machine communication system. As PVM remains the most popular library for parallel network computing, the P_ARPACK based solvers implemented in PVM may be efficiently ported to the environment of networks of workstations (NOWs). In this way, many of the PVM features, specifically oriented for use in parallel network environment, may be exploited to improve performance of the solvers in the network systems. Although BLACS is a static system and consequently the P_ARPACK routines may be used with a constant number of processors (processes), the reverse communication interface to these routines enables one to introduce dynamic process creation to the solvers implemented in PVM while computing in parallel e.g. the matrix-vector product. In this way, if the

time spent on computing the $\underline{\underline{A}}v$ product is suitably longer than the time spent in the P_ARPACK routines, then the design involving dynamic process creation may produce in some cases more efficient solvers, fully exploiting the potential of a given network environment.

5.3.3 Numerical and memory complexity of the method

The memory complexity of the parallel solver may be easily derived if the results of the discussion from the previous sections are applied. In the case of a general sparse matrix the overall memory requirements per processor equal the sum of the storage needed by the solver $O(k^2 n_{loc})$ and the storage size used by the matrix in the CSR format $O(2 \cdot nnz_{loc} + n_{loc} + 1)$, where nnz_{loc} is the number of non-zero elements in the locally stored part of the matrix and $n_{loc} \approx n/P$, where P is the number of processors. It may be seen that a very undesirable situation will occur if the non-zero matrix elements are not distributed evenly in the matrix. In this case the memory requirements will vary very significantly among the processors.

The numerical complexity of the parallel solver estimated for a single update in a p-step IRAM algorithm consists of the cost of performing the Arnoldi factorizations which equals $O(p^2 n_{loc})$ and the cost of calculating p matrix-vector products with the sparse matrix stored in the CSR format. The latter cost equals $O(p \cdot nnz_{loc})$. Consequently, the overall (per processor) cost of a single update in the iterative solver equals $O(p^2 n_{loc} + p \cdot nnz_{loc})$. Once again, this shows that the workload imbalance may result from a non-uniform distribution of non-zero elements in the input sparse matrix.

The size of messages communicated among the processors is determined primarily by the data sent and received during the matrix-vector product. In the case of a general sparse matrix with a highly irregular distribution of non-zero elements this size may be as high as $O(p(P-1)n)$ for a single update in a p-step IRAM algorithm. If the matrix is a banded one with a bandwidth b then (assuming that $b < (2n/P)$) this message size is reduced to $O(p(P-1)b)$ and becomes independent of the problem size n.

5.4 Parallel Arnoldi solver with implicit discrete representation of the operator

This section presents a parallel program which exploits the Method of Moments representation of both functions and the input operator in order to solve the given operator eigenproblem using the Implicitly Restarted Arnoldi Method, implemented in the P_ARPACK library. The salient feature of this discrete representation is that the operator matrix is stored implicitly, resulting in reduced storage requirements and allowing much more efficient implementation of the matrix-vector product operation.

The description concentrates on presenting the MPI implementation of the solver which may be ported to various distributed memory systems and deals with real non-symmetric eigenproblems of operators whose domain are 2D vector fields

defined over two-dimensional space. This choice of the operator domain is guided by numerous applications in which such vector fields play an important role. The chosen 2D domain also generates some non-trivial issues concerning parallel design of the algorithm which make it an interesting research subject.

Analogously as in the previous section, the implementation of the solver is based on the reverse communication scheme (presented in Figure 12) in which calls

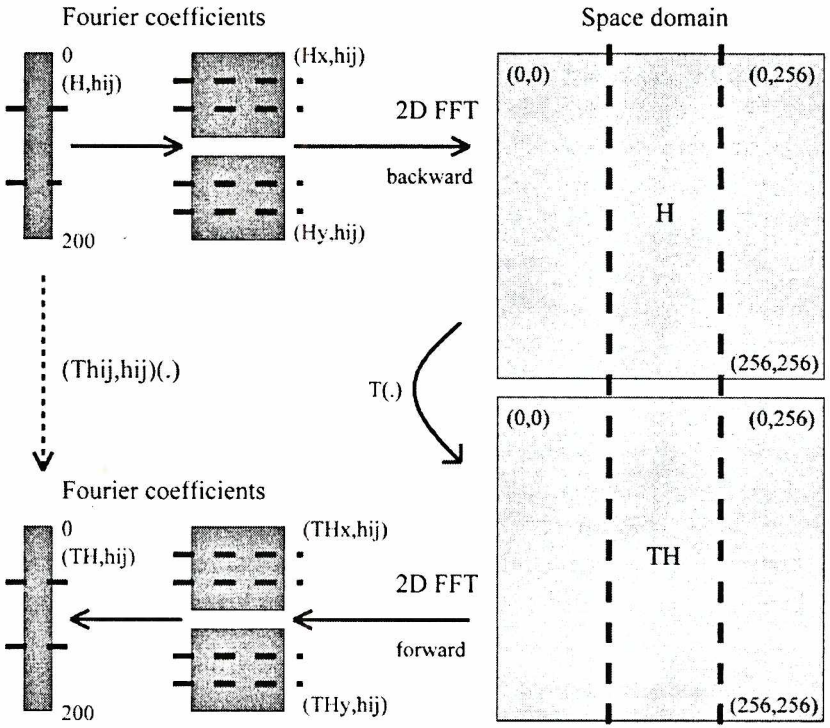

Figure 18. *Schematic of parallel data distribution in matrix-vector product design for the DFT-based operator and function representation. The dashed lines mark the block data distribution pattern among the processors.*

to P_ARPACK library routines (mainly the `pdnaupd()` routine) performing the Arnoldi factorization are followed by calls to user-supplied routines calculating the matrix-vector product.

The following section presents the parallel implementation of the matrix-vector product jointly with the description of the parallel distribution of the elements of the input vector.

5.4.1. Parallel implementation of the matrix-vector product using two-dimensional Fast Fourier Transform

Assuming that the domain of the given linear operator **T** is the space of 2D vector fields $\vec{H} = \left(H^x, H^y\right)$ where $H^x, H^y \in L_2([0, b] \times [0, a])$ the following

representation for the functions in this domain has been defined in Section 3.3:

$$\begin{aligned}\underline{H} &= \left[c_{11}^x, c_{11}^y, c_{12}^x, c_{12}^y, \ldots, c_{mn}^x, c_{mn}^y\right] \\ &= \left[\left(H^x, h_{11}^x\right), \left(H^y, h_{11}^y\right), \left(H^x, h_{12}^x\right), \left(H^y, h_{12}^y\right), \ldots, \left(H^x, h_{mn}^x\right), \left(H^y, h_{mn}^y\right)\right]\end{aligned} \tag{40}$$

where $\underline{H}$ is a finite representation for the vector field $\vec{H} = \left(H^x, H^y\right)$, c_{ij}^x and c_{kl}^y are Fourier coefficients defined by appropriate inner products and $\{h_{ij}^x\}$ and $\{h_{kl}^y\}$ form orthonormal bases in the $L_2([0, b] \times [0, a])$ functional space.

As described in Section 3.3, calculating the matrix-vector product in the case of the discussed representation may be performed using an efficient method which dramatically reduces the computational cost of this operation, as compared to the classical approach used in the Galerkin Method (GM). In this unorthodox approach the operation of calculating matrix-vector product involves three steps: 1) calculating the backward 2D FFTs, 2) calculating the $\mathbf{T}\vec{H}$ product in the spatial two-dimensional domain and 3) calculating forward 2D FFTs. This has been illustrated in Figure 18.

This Figure also shows the main idea of parallelization of this matrix-vector product, which is based on block-distributing (by rows) of the input elements of the vector $\underline{H}$, given by the equation (40). In other words, each processor stores a range of rows of the matrices of coefficients $[c_{ij}^x]$ and $[c_{ij}^y]$. The number of rows stored by each processors is balanced, as to assure a similar workload for all the processors. After completing the computation of the matrix-vector product each processor stores the same range of rows of the Fourier coefficient matrices for the $\mathbf{T}\vec{H}$ field.

In the Figure 18 it may also be noted that after computing the two-dimensional backward FFTs, the elements of the matrices H^x and H^y are block distributed by columns and not by rows. This is the effect of the parallel design of the two-dimensional FFT algorithm. Let us look in more detail at the parallel algorithm of computing the backward two-dimensional FFT. The schematic of this operation has been shown in Figure 19. The computation involves three steps:

1. As the matrices (from which only one was shown for simplicity) of the Fourier coefficients are distributed by rows, each processor computes a backward one-dimensional FFT in the x-direction for a locally stored range of rows.

2. In order to perform the backward one-dimensional FFT in the y-direction the processors need to have access to a full range of coefficients from specified columns. Consequently, a parallel transposition of the distributed matrices obtained after completing the backward FFTs in the x-direction has to be performed. This operation involves mainly the inter-processor communication, as each processor has to send $(P - 1)$ blocks of the locally stored part of the matrix and has to receive also $(P - 1)$ different blocks from other processors. In the MPI implementation of the solver this operation may be performed by using a high-level collective communication routine `MPI_Alltoall(.)` (or

`MPI_Alltoallv(.)` for non-equal sizes of the transmitted matrix blocks) which sends from all the processors to all the processors the specified blocks of data. Clearly, this operation may also be performed by using simple send and receive operations by scheduling these operations appropriately. Still, if the high-level message-passing routine is applied, the programming complexity is passed to the library implementation. Another advantage of such approach is that we may achieve better performance if a native implementation of the MPI library which optimizes collective communication routines for a specific interconnection network topology is applied in a given testing platform. In the actual implementation this approach has been successfully used, producing

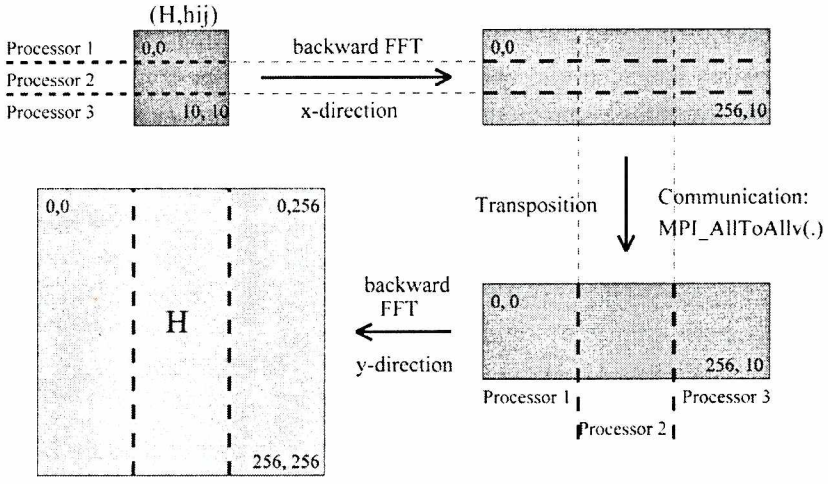

Figure 19. *Idea of the parallel backward two-dimensional FFT algorithm design. The scheme of performing a forward 2D FFT is entirely analogous.*

a highly efficient parallel routine as shown in Section 7.

3. After the transposition each processor computes a one-dimensional FFT in the y-direction for a locally stored range of columns.

This completes the parallel operation of computing the two-dimensional FFT. One may ask whether the elements of the output matrices should be block distributed among the processors by rows rather than by columns. The answer is negative. The main reason is that there is no need to perform an extra transposition operation (which involves a considerable amount of inter-processor communication) in order to obtain a parallel block distribution by columns. As may be seen from Figure 18, after computing the backward 2D FFT and performing the $\mathbf{T}\vec{H}$ operation a forward 2D FFT is performed. During the forward 2D FFT the parallel block distribution by rows is restored. The forward FFT involves analogous steps as those shown in Figure 19, namely: 1) Computing one-dimensional FFTs in the y-direction, 2) Performing a parallel matrix-transposition using the `MPIAlltoallv(.)` routine,

3) Computing one-dimensional FFTs in the x-direction.

By reversing the order of computing one-dimensional FFTs, two unnecessary (and costly) transposition operations are avoided. The numerical tests performed by the author comparing the two versions of the algorithm for computing the matrix-vector product — the one described above and the older serial implementation which performed additional transpositions (applied e.g. in [18]) show that for a single-processor execution the first algorithm was by about 30 % faster than the second serial algorithm. Even the overheads due to initiating the MPI communication and additional computations needed to establish the parallel data distribution scheme did not prevent the parallel algorithm from running faster on one processor than the second algorithm. This fact implies that the execution times of the solver, given in [18] may be further reduced by up to 30 %.

So far nothing has been told about the operation $\mathbf{T}\vec{H}$ performed in the spatial domain during the matrix-vector product. This step is entirely dependent on the form of the operator $\mathbf{T}$. Still, in many applications this operation may be completed in a linear time with respect to $N = N_x \cdot N_y$, where N_x and N_y denote the FFT lengths in the x- and y-directions respectively. Before discussing in detail the memory and computational complexities of the presented version of the parallel matrix-vector product computation let us present a special case of the algorithm which results important in certain application fields.

5.4.2 Special case of the FFT-based matrix-vector product implementation

Let $\mathbf{T}$ be a given infinite-dimensional linear operator and its domain is the same space of 2D vector fields as defined in the previous section. It is assumed that this operator may be decomposed (analogously as in the IEEM method - compare Section 2.5.1) as follows:

$$\mathbf{T} = \mathbf{L} - \mathbf{F} \tag{41}$$

where the operator $\mathbf{L}$ is a self-adjoint adjoint operator. Furthermore it is assumed that $\mathbf{L}$ is a scalar operator and its eigenfunctions are the functions from the trigonometric bases $\{h_{ij}^x\}$ and $\{h_{kl}^y\}$ (which form the orthonormal bases in the discussed functional space) and the corresponding eigenvalues $\{\Lambda_{ij}^x\}$ and $\{\Lambda_{kl}^y\}$ are known.

Let $\vec{v} = \left[v_x, v_y\right]^T \in X$ be an eigenfunction of the operator $\mathbf{T}$ corresponding to the eigenvalue λ:

$$\mathbf{T}\vec{v} = \mathbf{L}\vec{v} - \mathbf{F}\vec{v} = \lambda\vec{v} \tag{42}$$

The above equation may be written in the following form:

$$\mathbf{\Pi}_x \cdot \mathbf{L}\vec{v} - \mathbf{\Pi}_x \cdot \mathbf{F}\vec{v} = \lambda\mathbf{\Pi}_x\vec{v} \tag{43}$$

$$\mathbf{\Pi}_y \cdot \mathbf{L}\vec{v} - \mathbf{\Pi}_y \cdot \mathbf{F}\vec{v} = \lambda\mathbf{\Pi}_y\vec{v} \tag{44}$$

where $\mathbf{\Pi}_x([q_x, q_y]^T) = q_x$ and $\mathbf{\Pi}_y([q_x, q_y]^T) = q_y$. Denoting $\mathbf{\Pi}_x \cdot \mathbf{F} = \mathbf{F}_x$ and $\mathbf{\Pi}_y \cdot \mathbf{F} = \mathbf{F}_y$ and taking into account that $\mathbf{L}$ is a scalar operator the above equations may be rewritten as:

$$\mathbf{L}v_x - \mathbf{F}_x\vec{v} = \lambda v_x \tag{45}$$

$$\mathbf{L}v_y - \mathbf{F}_y\vec{v} = \lambda v_y \tag{46}$$

Expanding the field $\vec{v}$ in the series of the h_{ij}^x and h_{ij}^y functions:

$$\vec{v} = [v_x, v_y]^T = \left[\sum_i c_i^x h_i^x, \sum_i c_i^y h_i^y\right]^T \tag{47}$$

and inserting the above expansion into equations (45) and (46) gives:

$$\sum_{ij} \Lambda_{ij}^x c_{ij}^x h_{ij}^x - \mathbf{F}_x\vec{v} = \lambda \sum_{ij} c_{ij}^x h_{ij}^x \tag{48}$$

$$\sum_{ij} \Lambda_{ij}^y c_{ij}^y h_{ij}^y - \mathbf{F}_y\vec{v} = \lambda \sum_{ij} c_{ij}^y h_{ij}^y \tag{49}$$

Taking the inner product of both sides of the above equations by h_{kl}^x and h_{kl}^y correspondingly one gets:

$$c_{kl}^x \Lambda_{kl}^x - \left(\mathbf{F}_x\vec{v}, h_{kl}^x\right) = \lambda c_{kl}^x \tag{50}$$

$$c_{kl}^y \Lambda_{kl}^y - \left(\mathbf{F}_y\vec{v}, h_{kl}^y\right) = \lambda c_{kl}^y \tag{51}$$

Consequently, if the already discussed finite approximation of the operator $\mathbf{T}$ is applied the elements of the emerging operator matrix are given by the following formulas:

$$\left(\mathbf{T}h_{ij}^x, h_{kl}^x\right) = -\left(\mathbf{F}_x h_{ij}^x, h_{kl}^x\right) \quad (i,j) \neq (k,l) \tag{52}$$

$$\Lambda_{ij}^x - \left(\mathbf{F}_x h_{ij}^x, h_{kl}^x\right) \quad (i,j) = (k,l) \tag{53}$$

$$\left(\mathbf{T}h_{ij}^y, h_{kl}^y\right) = -\left(\mathbf{F}_y h_{ij}^y, h_{kl}^y\right) \quad (i,j) \neq (k,l) \tag{54}$$

$$\Lambda_{ij}^y - \left(\mathbf{F}_y h_{ij}^y, h_{kl}^y\right) \quad (i,j) = (k,l) \tag{55}$$

Consequently the modified steps of the parallel FFT-based matrix-vector product $\underline{\underline{T}}\,\underline{H}$, where $\underline{\underline{T}}$ is a finite representation of the operator $\mathbf{T}$ and $\underline{H}$ denotes the finite representation of the field $\vec{H}$, are given as follows:

1. Given the vector of Fourier coefficients c_{ij}^x and c_{kl}^y compute two 2D backward FFTs in order to obtain H_x and H_y,
2. Compute $\mathbf{F}_x\vec{H}$ and $\mathbf{F}_y\vec{H}$

3. Compute two 2D forward FFTs in order to obtain $\left(\mathbf{F}_x\vec{H},\ h_{ij}^x\right)$ and $\left(\mathbf{F}_y\vec{H},\ h_{kl}^y\right)$
4. Compute $\tilde{c}_{ij}^x = \Lambda_{ij}^x c_{ij}^x - \left(\mathbf{F}_x\vec{H},\ h_{ij}^x\right)$ and $\tilde{c}_{kl}^y = \Lambda_{kl}^y c_{kl}^y - \left(\mathbf{F}_y\vec{H},\ h_{kl}^y\right)$

Note that an additional step 4 of the above algorithm does not involve any communication, as all the necessary data is stored locally be each processor. Therefore this step is "perfectly parallel".

5.4.3 Numerical and memory complexity of the method

In this section the numerical and memory complexity of a single p-step update of the IRAM algorithm involving the FFT-based matrix-vector product will be investigated. Applying the results from Section 2.4.3 and Section 3.3.1 one may estimate the overall memory storage needed by the parallel solver as the sum of the storage needed by the IRAM procedure ($(N/P)\cdot O(k)+O(k^2)$) and the memory required in the matrix-vector product computation $\left(O\left(2K/P+2N/P+6\sqrt{K}\right)\right)$, where P is the number of processors, k is the number of eigenvalues to be found ($p = O(k)$), $K = K_x \cdot K_y$, where K_x and K_y denote the FFT lengths in the x- and y-direction respectively and N is the problem size. ($N = N_x \cdot N_y$, where N_x and N_y denote the number of expansion functions used to represent the functions in the respectful spatial dimensions.)

The numerical complexity of a single update involves the time cost of performing the Arnoldi factorizations ($O(p^2N/P)$) and the cost of computing p matrix-vector products which equals $pO(K/P \log K)$. The overall cost is given by the formula:

$$\frac{1}{P}O\left(p^2N + pK\log K\right) = \frac{1}{P}O\left(k^2N + kK\log K\right) \tag{56}$$

with all the symbols having the same meaning as above.

Another aspect which has to be addressed is the size of messages communicated in the algorithm which in this case is dominated by the size of the messages communicated during the matrix-vector product computation. In a single matrix-vector product the communication occurs during the two transposition operations. The size of the communicated data equals $O((K_xN_y + K_yN_x)(P-1)/P)$ elements. Consequently in a k-step IRAM algorithm the communication size equals:

$$O\left(p(K_xN_y + K_yN_x)(P-1)/P\right) = O\left(p\sqrt{KN}(P-1)P\right) \tag{57}$$

assuming that $K_y = O(K_x)$ and $N_y = O(N_x)$.

5.5 Parallel Iterative Eigenfunction Expansion Method with the FFT integration

This section presents a parallel algorithm solving operator eigenvalue problems based on the Iterative Eigenfunction Expansion Method described in Section 2.5.3. The finite dimensional mapping of the input operator is based on the implicit

operator representation as discussed in Section 3.3. In this representation calculation of the scalar products may be implemented using Fast Fourier Transform, resulting in the reduced numerical cost. Such parallel implementation has already been presented in the previous section for the case of operators whose domain is the appropriately defined space of two-dimensional vector fields. As all the considerations concerning computing the scalar products (the matrix-vector product) and the discussion of the parallel design of the computations and parallel data distribution are entirely the same as in the previous sections (5.4.1 and 5.4.2) we shall now limit to presenting the explicit steps of the IEEM method in the basic version and in the modified version, which applies the deflation techniques (Compare Section 5.4).

5.5.1 Parallel implementation of the basic algorithm

As discussed in Section 2.5 the basic IEEM is capable of finding a single eigenvalue from the point spectrum of a given operator **T**. We will assume that the domain of the input operator is the L_2 space defined over a two-dimensional bounded rectangular region $\Omega = ([0, b] \times [0, a]) \in R^2$ and the appropriate decomposition of the operator (as discussed in Section 2.5.1) may be applied. In this case all the results from the previous sections (especially the Section 5.4.2) may be described by the following steps ([26]):

ALGORITHM 5: *IEEM-FFT.*

STEP 1: Applying the Fourier coefficients:

$$\left[c_{11}^{x(k-1)}, c_{11}^{y(k-1)}, c_{12}^{x(k-1)}, c_{12}^{y(k-1)}, \ldots, c_{mn}^{x(k-1)}, c_{mn}^{y(k-1)}\right]$$

obtained in the previous iteration, apply the parallel procedure involving 2D backward and forward FFTs to compute the following inner products:

$$\left(\mathbf{F}_x \vec{H}^{k-1}, h_{ij}^x\right) \quad \left(\mathbf{F}_y \vec{H}^{k-1}, h_{kl}^y\right)$$

where $\vec{H}^{(k-1)} = \left(H^{x(k-1)}, H^{y(k-1)}\right)$

and $H^{x(k-1)} = \sum_{ij} c_{ij}^{x(k-1)} h_{ij}^x$, $H^{y(k-1)} = \sum_{kl} c_{kl}^{y(k-1)} h_{kl}^y$

STEP 2: Compute the new values of Fourier coefficients $c_{ij}^{x(k)}$ and $c_{kl}^{y(k)}$:

$$c_{ij}^{x(k)} = \frac{\left(\mathbf{F}_x \vec{H}^{k-1}, h_{ij}^x\right)}{\Lambda_{ij}^x - \lambda^{(k-1)}}$$

$$c_{kl}^{y(k)} = \frac{\left(\mathbf{F}_y \vec{H}^{k-1}, h_{kl}^y\right)}{\Lambda_{kl}^y - \lambda^{(k-1)}}$$

STEP 3: Normalize the Fourier coefficients:

$$g_{ij}^{x} = \frac{c_{ij}^{x(k)}}{\left\| \vec{H}^{(k)} \right\|_2}$$

$$g_{kl}^{y} = \frac{c_{kl}^{y(k)}}{\left\| \vec{H}^{(k)} \right\|_2}$$

STEP 4: Compute the k-th approximation of the eigenvalue λ:

$$\lambda^{(k)} = \sum_{ij} \left(\Lambda_{ij}^{x} \left| g_{ij}^{x} \right|^2 + \Lambda_{ij}^{y} \left| g_{ij}^{y} \right|^2 \right) - \sum_{ij} g_{ij}^{x*} \left(\mathbf{F}_x \vec{H}^{k-1}, h_{ij}^{x} \right) - \sum_{ij} g_{ij}^{y*} \left(\mathbf{F}_y \vec{H}^{k-1}, h_{ij}^{y} \right)$$

Assuming that a parallel block distribution by rows of the matrices of Fourier coefficients $[c_{ij}^{x(k)}]$ and $[c_{kl}^{y(k)}]$ has been applied, the following inter-processor communication has to be performed during the iteration. Apart from inter-processor communication arising in Step 1 of the method additional communication is performed in Steps 3 and 4. The communication in Step 3 is necessary to compute the global norm of the field $\vec{H}^{(k)}$. In the MPI implementation this operation involves a single call to the `MPI_Allreduce(.)` routine. After this call all the processors know the value of the squared global norm. The size of messages sent in this step is very small and equals about $2P$ elements. An entirely analogous situation occurs in Step 4 where also a global sum has to be computed and an MPI global reduction procedure is used to this end. Summing up, the intensive communication appears solely in Step 1 of the presented algorithm.

Applying the results obtained in the Section 5.4.3, one may easily assess the complexity of the parallel version of the IEEM-FFT method. The memory complexity equals: $O\left(2K/P + 2N/P + 6\sqrt{K}\right)$, where P is the number of processors used, $K = K_x \cdot K_y$ is a product of the FFT transform lengths in respectful directions and $N = N_x \cdot N_y$ is the problem size. The computational complexity of a single iteration consists of computing the inner-products which involves $O(K/P \log K)$ operations and the complexity of the steps 2-4 which is linear ($O(N/P)$). The overall computational complexity may be estimated at the level $O(K/P \log K)$. The size of messages is roughly the same as in the previous method and equals:

$$O\left(\frac{P-1}{P} \sqrt{KN} \right)$$

5.5.2 Implementation of the deflation procedures

This section presents a modification of the IEEM algorithm described in the previous section. In this algorithm the deflation procedures, introduced in Section

2.5.4, are incorporated in the iteration loop allowing one to find a few eigenvalues from a given operator's spectrum. It is assumed that $(s-1)$ eigenvalues (together with corresponding right and left eigenvectors) have already been found in the previous stages of the algorithm. This assumption requires an additional comment. Generally speaking finding left eigenfunctions (eigenvectors) of a given operator may become a problem which is equally complex as the problem of finding right eigenvalues, as it may require solving the adjoint eigenproblem. In this case the complexity of the presented method may even double. Still, in some applications, including some problems arising in electromagnetics, computation of left eigenvalues may be performed in a different way, as described in Section 2.2. If this is the case, then deriving left eigenvalues is a fairly inexpensive operation which only slightly increases the computational complexity of the method. With the above assumptions the steps the k-th iteration of the algorithm are summarized in the following points:

ALGORITHM 6: *IEEM-FFT-deflation.*

STEP 1: Applying the Fourier coefficients:

$$\left[c_{11}^{x(k-1)}, c_{11}^{y(k-1)}, c_{12}^{x(k-1)}, c_{12}^{y(k-1)}, \ldots, c_{mn}^{x(k-1)}, c_{mn}^{y(k-1)}\right]$$

obtained in the previous iteration, apply the parallel procedure involving 2D backward and forward FFTs to compute the following inner products:

$$\left(\mathbf{F}_x \vec{H}^{k-1}, h_{ij}^x\right) \quad \left(\mathbf{F}_y \vec{H}^{k-1}, h_{kl}^y\right)$$

where $\vec{H}^{(k-1)} = \left(H^{x(k-1)}, H^{y(k-1)}\right)$, $H^{x,y(k-1)} = \sum_{ij} c_{ij}^{x,y(k-1)} h_{ij}^{x,y}$.

STEP 2: For $r = 1, \ldots, (s-1)$ compute the following scalar values:

$$t_r = \alpha_r \lambda_r \left(E_r, H^{(k-1)}\right)$$

where α_r are the deflation coefficients, λ_r previously computed eigenvalues of the input operator and E_r are the corresponding left eigenvectors.

STEP 3: Compute the new values of Fourier coefficients $c_{ij}^{x(k)}$ and $c_{kl}^{y(k)}$:

$$c_{ij}^{x(k)} = \frac{\left(\mathbf{F}_x \vec{H}^{k-1}, h_{ij}^x\right) + \sum_{r=1}^{s-1} t_r c_{r,ij}^x}{\Lambda_{ij}^x - \lambda^{(k-1)}}$$

$$c_{kl}^{y(k)} = \frac{\left(\mathbf{F}_y \vec{H}^{k-1}, h_{kl}^y\right) + \sum_{r=1}^{s-1} t_r c_{r,kl}^y}{\Lambda_{kl}^y - \lambda^{(k-1)}}$$

STEP 4: Normalize the Fourier coefficients:

$$g_{ij}^x = \frac{c_{ij}^{x(k)}}{\left\|\vec{H}^{(k)}\right\|_2}$$

$$g_{kl}^{y} = \frac{c_{kl}^{y(k)}}{\left\| \bar{H}^{(k)} \right\|_2}$$

STEP 5: Once again, for $r = 1, \ldots, (s-1)$ compute the following scalar values:

$$t_r = \alpha_r \lambda_r \left(E_r, H^{(k)} \right)$$

where α_r are the deflation coefficients, λ_r are previously computed eigenvalues of the input operator and E_r are the corresponding left eigenvectors.

STEP 6: Compute the k-th approximation of the eigenvalue λ:

$$\lambda^{(k)} = \sum_{ij} \left(\Lambda_{ij}^{x} \left| g_{ij}^{x} \right|^2 + \Lambda_{ij}^{y} \left| g_{ij}^{y} \right|^2 \right) - \sum_{ij} g_{ij}^{x*} \left[\left(\mathbf{F}_x \bar{H}^{k-1}, h_{ij}^{x} \right) + \sum_{r=1}^{s-1} t_r c_{r,ij}^{x} \right] - \sum_{ij} g_{ij}^{y*} \left[\left(\mathbf{F}_y \bar{H}^{k-1}, h_{ij}^{y} \right) + \sum_{r=1}^{s-1} t_r c_{r,ij}^{y} \right]$$

After the convergence of the above iterative process is obtained, the s-th left eigenvector has to be computed. This may be done by either solving the adjoint problem or, if possible, deriving this vector from the right vector (as shown in Section 2.2). After the left eigenvector is found it has to be normalized as to obtain the following relation:

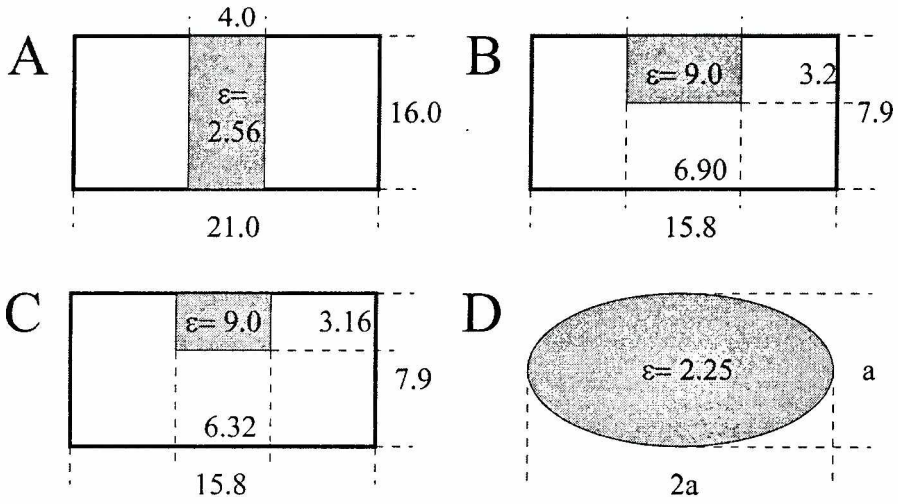

Figure 20. Cross-sections of the waveguiding structures used during numerical tests. In the Figure all dimensions are given in millimeters.

$$\left(\vec{H}_s, \vec{E}_s\right) = 1$$

In order to enhance the stability of the iterative process involving the deflation procedure the orthonormalization procedure (MGS algorithm) is introduced to the iteration after Step 3 (and is executed every few iterations). In this procedure is the vector field $\vec{H}_s^{(k)}$ is re-orthonormalized with respect to $(s-1)$ left eigenvectors $\vec{E}_1, \ldots, \vec{E}_{s-1}$.

6. Application and validation of the algorithms

This section presents application of the algorithms presented above to solving operator eigenproblems arising in electromagnetics or more specifically in the theory of electromagnetic waveguides. The eigensolvers are used to find propagation constants in selected dielectric waveguides, shown in Figure 20. The tested structures include a slab waveguide (A) and image guides (B,C) with discontinuous permittivity profiles as well as an elliptical guide (D) with a continuous permittivity profile $\varepsilon(x, y)$.

The numerical tests presented within this study include validation of four algorithms of solving operator eigenproblems: 1) IRAM-FD algorithm, 2) IRAM-FFT algorithm, 3) IEEM-FFT-deflation algorithm and 4) IEEM-FFT-NI algorithm. The last algorithm is a modification of the IEEM-FFT algorithm which extends applicability of this method to modelling waveguides with discontinuous permittivity profiles and will be presented in the following section.

The results obtained using the mentioned algorithms are then compared to the results produced by 1) the Transverse Resonance Method (TRM) which is regarded one of the most accurate algorithms for finding propagation constants, suitable while dealing with certain relatively simple waveguiding structures; 2) the Galerkin Method (GM) in which the operator is represented by the appropriate inner products, as described in Section 3.3. Although the idea behind the representation of the operator is the same for the Galerkin Method and for the IRAM-FFT and IEEM-FFT algorithms, the two extremely important differences occur: 1) In GM the operator matrix is stored explicitly as compared to the implicit storage applied in the two latter algorithms and 2) For the Galerkin Method the analytical integration is used to compute matrix elements (Fourier integrals) as opposed to the FFT integration applied in the two other algorithms. In the case of the eigensolver implementing the Galerkin Method used in comparative tests the eigenproblem for the generated operator matrix has been solved using the QR algorithm.

Before presenting the results of numerical tests let us address a few issues concerning operators which arise in numerical modelling of electromagnetic waveguides.

6.1 Modelling electromagnetic waveguides using operator formulation

The forms of differential operators, derived from Maxwell's equations, which model electromagnetic fields in dielectric waveguides have already been discussed in Section 2.2. The first of the mentioned operators was a vector differential operator:

$$\mathbf{T}(\cdot)=\nabla_t^2(\cdot)+k_0^2\varepsilon(x,y)(\cdot)+\frac{1}{\varepsilon(x,y)}\left[\nabla_t\varepsilon(x,y)\times(\nabla_t\times(\cdot))\right] \tag{58}$$

where $\nabla_t^2(\cdot)=\left(\frac{\partial}{\partial x},\frac{\partial}{\partial y}\right)(\cdot)$, k_0 is the wavenumber in the free space and $\varepsilon(x,y)$ is the permittivity profile of a waveguide in the $x-y$ plane. The simplified, scalar version of the above operator was given by the formula:

$$\tilde{\mathbf{T}}(\cdot)=\nabla_t^2(\cdot)+k_0^2\varepsilon(x,y)(\cdot) \tag{59}$$

The domain of both operators is defined as the space of 2D vector fields $\vec{H}=\left(H^x,H^y\right)$, where H^x and H^y are square integrable functions defined over a bounded rectangular region containing the cross-section of the examined waveguiding structure. In the presented formulation, the eigenfunctions of the operator $\mathbf{T}\left(\tilde{\mathbf{T}}\right)$ correspond to the transverse magnetic field and its eigenvalues correspond to squared propagation constants β^2.

Let us now consider problems which arise if some kind of numerical treatment is to be applied to solving an eigenproblem of a differential operator **T** given by the formula (58). The initial issue which has to be addressed is finding the finite-dimensional mapping of the operator **T**. One of the immediate choices is the Finite Difference (in this case the Finite Difference Frequency Domain (FDFD)) mapping technique. This case has already been discussed in Section 3.2 and it has been found that the resulting operator matrix is a sparse, banded matrix with a highly regular non-zero element distribution. (In the discussion we have applied data referring to the discretization of the operator modelling a dielectric waveguide with a discontinuous, rectangular permittivity profile (cf. structures A-C in Figure 20).) Consequently in this case the parallel solver implementing the IRAM-FD algorithm described in Section 5.3 may be applied to find eigenvalues of the input operator. The following section presents eigenvalues computed using this solver for selected waveguiding structures.

The other finite-dimensional representation — the Method of Moments representation with implicit operator storage, discussed within this study (Section 3.3) may also be applied. Still, in this case it is necessary to find out whether this representation is suitable for all the operators in the form given by Equation (58). As described in the previous sections, in the considered finite mapping technique the operator is represented by certain inner products - the Fourier coefficients, such as $(\mathbf{T}h_{ij}^x,\ h_{kl}^y)$. These coefficients are in fact 2D definite integrals whose values are computed numerically by using the Discrete Fourier Transform. Using the DFT we calculate approximate values of these integrals using a regular grid of samples of the integrated 2D function. The numerical error in the integration depends on the class of the integrated function. If the form of the operator **T** given by formula (58)

is examined, one may note the term $\nabla_t \varepsilon(x,y)$. If the permittivity profile is a C^1 class function then the operation **T** on an arbitrary function from the C^2 class results in a continuous function. In this case one may expect that the DFT-based representation of the operator will provide an adequate finite approximation of the operator. In fact, as shown in the section presenting results validating the discussed numerical methods, in the case of continuous permittivity profiles the solvers applying the DFT-based approach produce correct results.situation may change drastically if the permittivity profile has a discontinuity. This situation may case is considered in the following subsection.

6.1.1 Discontinuous permittivity profiles and the DFT representation.

In the case of a waveguide shown in Figure 21 the permittivity profile $\varepsilon(x,y)$ is a discontinuous function which is given by the following formula:

$$\varepsilon(x,y)=(\varepsilon-1)h(x_2-x)h(x-x_1)h(y_2-y)h(y-y_1)+1 \tag{60}$$

where $h(x)$ denotes the Heaviside function. Only the derivatives in a generalized sense exist for $\varepsilon(x,y)$:

$$\frac{\partial}{\partial x}\varepsilon(x,y)=(\varepsilon-1)h(x_2-x)h(x-x_1)h(y_2-y)h(y-y_1)(\delta(x-x_1)-\delta(x_2-x)) \tag{61}$$

where $\delta(x)$ denotes the Dirac distribution.

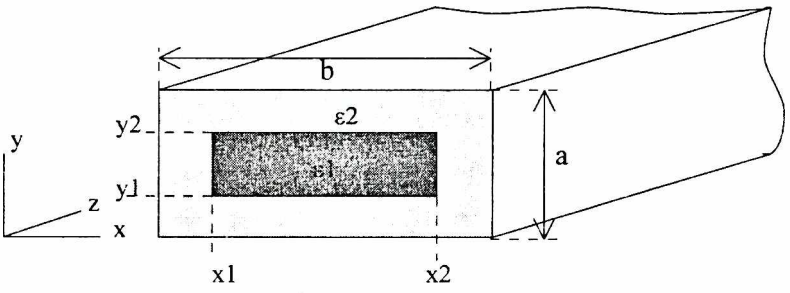

***Figure 21**. Schematic of a dielectric waveguide with a rectangular, discontinuous permittivity profile ε (x, y).*

Obviously applying "sampling" to a distribution is impossible. Consequently calculating the Fourier integrals by using the Discrete Fourier Transform does not have any correct mathematical meaning in this case, which results in severe numerical errors which are indeed observed in many applications (including the currently discussed one) if this kind of approach is applied.

The solution which may be proposed is a modification of the solvers which use the discussed implicit representation of operators (e.g. IRAM-FFT or IEEM-FFT algorithm). The modification refers to the method of calculating the matrix-vector product which implicitly contains the form of the operator. The proposed method is

a hybrid algorithm which uses both DFT (FFT) and numerical integration to calculate the matrix-vector product. The method starts with decomposing the initial operator **T** given by the equation (58):

$$\mathbf{T} = \mathbf{L} - \mathbf{F} \tag{62}$$

where:

$$\mathbf{L}(\cdot) = \nabla_t^2(\cdot) + k_0^2 \varepsilon(x,y)(\cdot) \tag{63}$$

$$\mathbf{F}(\cdot) = \frac{1}{\varepsilon(x,y)}\left[\nabla_t \varepsilon(x,y) \times (\nabla_t \times (\cdot))\right] \tag{64}$$

Denoting as $\mathbf{F}_x$ and $\mathbf{F}_y$ the projections of the vector operator $\mathbf{F}$ into x- and y- directions, the inner products $\left(\mathbf{F}_x \vec{H}_t, h_{ij}^x\right)$ and $\left(\mathbf{F}_y \vec{H}_t, h_{kl}^y\right)$ for the structure shown in Figure 21 are given by the following 1D integrals:

$$\left(\mathbf{F}_x \vec{H}_t, h_{ij}^x\right) = \int_{x_1}^{x_2} \frac{2(\varepsilon-1)}{(\varepsilon+1)} \left[\left(\nabla_t \vec{H}_t(x,y)\right) h_{ij}^x(x,y)\right]_{y=y_2}^{y=y_1} dx$$

$$\left(\mathbf{F}_y \vec{H}_t, h_{kl}^y\right) = \int_{y_1}^{y_2} \frac{2(\varepsilon-1)}{(\varepsilon+1)} \left[\left(\nabla_t \vec{H}_t(x,y)\right) h_{kl}^y(x,y)\right]_{x=x_2}^{x=x_1} dy$$

where the term $\dfrac{2(\varepsilon-1)}{\varepsilon+1}$ is obtained while integrating the permittivity profile, under the assumption that the Heaviside function is given by the formula:

$$h(x) = \begin{cases} 0 & x < 0 \\ 0.5 & x = 0 \\ 1 & x > 0 \end{cases} \tag{65}$$

The above linear integrals may be computed using any standard method of numerical integration. If we denote by $\mathbf{L}_x$ and $\mathbf{L}_y$ the projections of the operator **L** (cf. equation (63)) then the steps of the hybrid algorithm calculating the inner products $\left(\mathbf{T}_x \vec{H}_t, h_{ij}^x\right)$ and $\left(\mathbf{T}_y \vec{H}_t, h_{kl}^y\right)$ may be given as follows:

1. Given the Fourier coefficients $\{c_{ij}^x\}$ and $\{c_{kl}^y\}$ compute the values of the vector field $\vec{H}_t = \left(H^x, H^y\right)$ in the spatial domain by applying backward 2D FFTs.
2. Applying numerical integration (NI) compute the following inner products:

$$g_{ij}^x = \left(\mathbf{F}_x \vec{H}_t, h_{ij}^x\right) \qquad g_{kl}^y = \left(\mathbf{F}_y \vec{H}_t, h_{kl}^y\right)$$

3. Derive the Fourier coefficients ($\mathbf{L}_x H_x$, h_{ij}^x) i ($\mathbf{L}_y H_y$, h_{kl}^y) using the 2D FFT

Table 1. *Comparison of the normalized propagation constants β/k_0 calculated for a slab guide (structure A in Figure 20) with a discontinuous permittivity profile. In the tests the frequency f equalled 15 GHz, the number of expansion functions used equalled 10 in every spatial direction, the FFT length equalled 256 in both directions, NEV = 4, NCV = 20 (IRAM-FFT).*

TRM [18]	*IEEM-FFT*	*GM*	*IRAM-FFT*
1.2353	1.2352	1.2344	1.2339
—	1.0756	1.0813	1.0818
—	1.0622	1.0648	1.0641
4.1570–01	4.1563–01	4.1146–01	4.1412–01

algorithm.

4. Compute the final Fourier coefficients:

$$\left(\mathbf{T}_x \vec{H}_t, h_{ij}^x\right) = \left(\mathbf{L}_x H_x, h_{ij}^x\right) + g_{ij}^x$$

$$\left(\mathbf{T}_y \vec{H}_t, h_{kl}^y\right) = \left(\mathbf{L}_y H_y, h_{kl}^y\right) + g_{kl}^y$$

where $\mathbf{T}_x$ and $\mathbf{T}_y$ denote the projections of the initial operator $\mathbf{T}$ onto x- and y-dimensions.

Apart from the obvious advantage of being able to deal with discontinuous permittivity profiles, the above algorithm also has a very substantial drawback of increasing the computational complexity of the matrix-vector product algorithm by $O\left(\sqrt{K}N\right)$ (in the worst case this means the increment of the complexity to $O(K^{3/2})$, as compared to $O(K \log K)$), where K is the product of the DFT lengths in the x- and y-dimensions and N is the problem size (the product of the number of expansion terms used to approximate functions in the x- and y-dimensions).

Application of the third considered algorithm — the Iterative Eigenfunction Expansion Method (IEEM) to solving the eigenproblem of the differential operator $\mathbf{T}$ given by equation (58) arises if the following decomposition is applied:

$$\mathbf{T} = \tilde{\mathbf{L}} - \tilde{\mathbf{F}} \tag{66}$$

where:

$$\tilde{\mathbf{L}}(\cdot) = \nabla_t^2(\cdot) + k_0^2(\cdot) \tag{67}$$

$$\tilde{\mathbf{F}}(\cdot) = -k_0^2\left(\varepsilon(x,y)+1\right)(\cdot) - \frac{1}{\varepsilon(x,y)}\left[\nabla_t \varepsilon(x,y) \times \left(\nabla_t \times (\cdot)\right)\right] \tag{68}$$

One should note that the above decomposition is different from the one applied previously (cf. equation (62)). By selecting this kind of decomposition we assure that the operator $\tilde{\mathbf{L}}$ is self-adjoint in the considered functional space and its

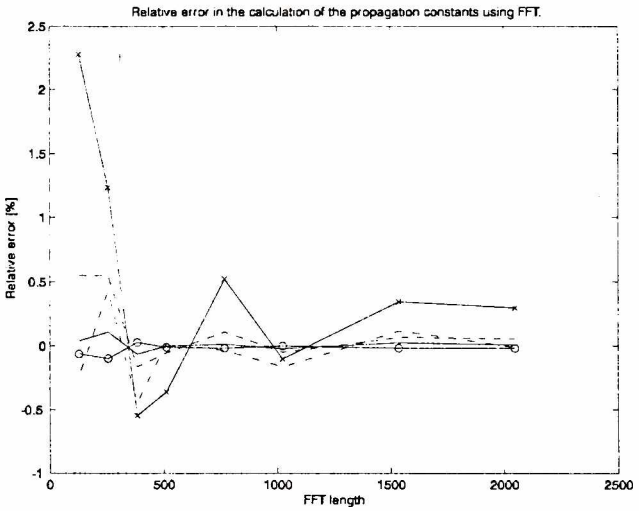

Figure 22. *Relative difference in the values of normalized propagation constants β/k_0 computed for a slab guide (structure A in Figure 20) using the IRAM-FFT solver and the Galerkin Method (GM) as a function the length of the FFT applied in the IRAM-FFT algorithm. During the tests the frequency f equalled 15 GHz, the number of expansion functions used equalled 10 (in every spatial direction), NEV = 5, NCV = 20 (IRAM-FFT). The errors were computed for the first 5 eigenvalues found by the methods.*

eigenfunctions form a trigonometric orthonormal basis in this space. Although it has not been proven that the operator $\tilde{\mathbf{F}}$ is relatively compact with respect to operator $\tilde{\mathbf{L}}$ which is a sufficient condition to ensure the convergence of the IEEM method (compare Section 2.5), the numerical tests show that the iterative process converges to the eigenvalues of the input operator **T**.

6.2 Validation of the algorithms

Having presented main features of operators used to model magnetic fields in selected waveguiding structures and specifics of application of the discussed algorithms to solving eigenproblems of these operators we may now turn to the presentation of the results of numerical tests.

We start with the presentation of the numerical results with a comparison of the values of normalized propagation constants β/k_0 calculated for a simple slab waveguide (cf. structure A in Figure 20) using four different algorithms. Although the number of expansion functions used in discrete representations of functions and operators is very small and equals 10 in every spatial direction (x and y) (for IEEM-FFT, IRAM-FFT and GM) a very good convergence of results is obtained. (cf. Table 1) It may easily be found that the relative differences between the corresponding eigenvalues do not exceed 1%. Moreover, although the vector operator **T** (cf. formula (58)) has been used in these tests, application of a simpler, scalar operator $\tilde{\mathbf{T}}$ (cf. formula (59)) gives entirely analogous results. This effect is due to a very simple structure of the modelled waveguide.

Table 2. *Comparison of the normalized propagation constants β/k_0 computed using the modified IRAM-FFT algorithm (IRAM-FFT-NI) and the Galerkin Method. The structure used in the tests was an image guide (structure C in Figure 20). Other test parameters: $f = 15$ GHz, $NEV = 8$, $NCV = 40$. The relative error was computed using the formula: $E = 100|\beta_{IRAM} - \beta_{GM}|/|\beta_{GM}|$.*

GM 20×20	*IRAM-FFT-NI 20×20 FFT length 2048*	*Relative error [%]*
2.3110	2.3132	0.10
1.5947	1.5951	0.03
8.5624–01	8.5142–01	0.56
5.1314–01+2.7998–01i	5.1403–01+2.7986–01i	0.16
5.1314–01–2.7998–01i	5.1403–01–2.7986–01i	0.16
6.9140–01i	7.0192–01i	1.52
1.1278i	1.1276i	0.02
1.2571i	1.2568i	0.03

A different series of tests performed for the same simple waveguiding structure compared the values of normalized propagation constants computed with the Galerkin Method and the IRAM-FFT algorithm for different lengths of the Fourier Transforms, applied in the IRAM-FFT method and ranging from 128 to 2048 in every direction. The results of these tests are shown in Figure 22. As one could expect the relative differences between the computed eigenvalues become smaller with the increasing FFT length. This means that the approximations of inner products computed using the FFT algorithm approach the values of the inner products computed analytically in the Galerkin Method with application of a more refined dicretization grid (determined by the FFT length).

If a more complex waveguiding structure is considered, e.g. an image guide with a discontinuous permittivity profile (structure C in Figure 20) then substantial problems appear with algorithms which use the DFT-based discretization scheme, i.e. the IRAM-FFT algorithm and the IEEM-FFT algorithm. The eigenvalues found e.g. by the IRAM-FFT algorithm for the vector operator **T** (compare equation (58)) differ significantly (by 10-20%) from the corresponding eigenvalues found using the Galerkin Method. This fact is due to the effects described in detail in Section 6.1.1. The situation improves considerably if a modification of the IRAM-FFT algorithm (denoted as IRAM-FFT-NI algorithm), described in Section 6.1.1 is applied. The results of the comparison between the GM and IRAM-FFT-NI algorithm are presented in Table 2 and Figure 23 and show that the computed eigenvalues stay very close to each other (especially for lower-order modes). The results confirm the usefulness of the investigated IRAM-FFT-NI algorithm in modelling structures with discontinuous rectangular permittivity profiles, such as the tested image guide (cf. structure C in Figure 20).

Although obviously the IRAM-FFT-NI algorithm, instead of the IRAM-FFT

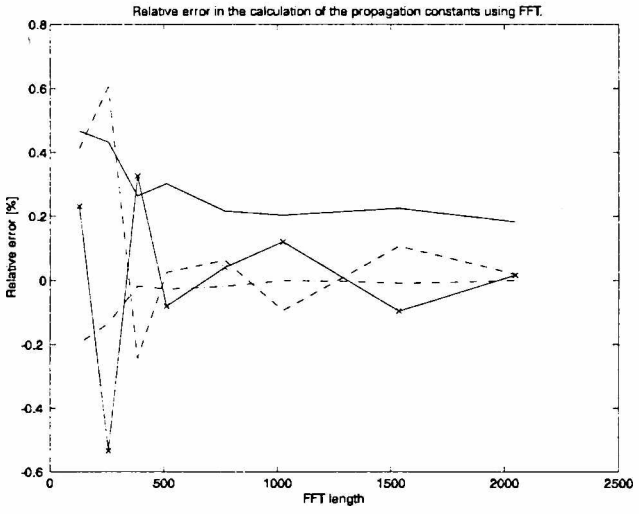

Figure 23. *Comparison of selected real normalized propagation constants β/k_0 computed using the modified IRAM-FFT algorithm (IRAM-FFT-NI) and the Galerkin Method for different FFT lengths applied. The structure used in the tests was an image guide (structure C in Figure 20). Other test parameters: $f = 15$ GHz, $NEV = 8$, $NCV = 40$. The relative error was computed using the formula: $E = 100(\beta_{IRAM} - \beta_{GM})/(\beta_{GM})$.*

method, should be used to model waveguides with discontinuous permittivity profiles, the scope of applicability of the basic algorithm does not limit to investigating only the simplest structures. The IRAM-FFT algorithm may be effectively used to model waveguiding structures with continuous permittivity profiles. The numerical tests involving computation of propagation constants for an elliptical waveguide with continuous permittivity profiles (structure D in Figure 20) were performed applying the IRAM-FFT algorithms. The tested structure was an elliptical waveguide with the semiaxes ratio 2/1 and the permittivity profile given by the function:

$$\varepsilon(x,y) = \varepsilon_0\left[1 - \left((x/2a)^2 + (y/a)^2\right)^{\alpha/2}\right] \tag{69}$$

An open structure was modelled by taking the screening walls sufficiently far away from the guide (at the distance of 20a from the centre of the waveguide). The results presented in Table 3 show a comparison of the propagation constants (for different profile exponents alpha) computed using IEEM (not IEEM-FFT!) algorithm [18] and obtained by the author using the IRAM-FFT method. The Table shows non-dimensional normalized propagation constants Z computed from the following formula:

$$Z = \frac{\beta^2/k_0^2 - 1}{\varepsilon - 1} \tag{70}$$

In the tests the normalized frequency V, given by formula: $V = k_0 \cdot 2a \cdot \sqrt{\varepsilon - 1}$

Table 3. *A comparison of the normalized propagation constants Z computed in the IEEM method and the IRAM-FFT algorithms for the elliptical waveguide with a continuous permittivity profile (structure D in Figure 20), for different permittivity profile exponents alpha. In the tests: V = 3, NEV = 1, NCV = 20, FFT length = 256,number of expansion function used to represent functions = 75 (in every spatial direction).*

α	*IEEM [18]*	*IRAM-FFT*	*Difference [%]*
2	0.4894	0.4907	0.27
4	0.6254	0.6258	0.06
6	0.6740	0.6742	0.03
8	0.6976	0.6978	0.03
10	0.7112	0.7114	0.03

Table 4. *Comparison of the normalized propagation constants β/k_0 computed using the IRAM-FD algorithm and the Galerkin Method. The structure used in the tests was an image guide (structure C in Figure 20). Other test parameters: f = 15 GHz, NEV = 8, NCV = 40 (IRAM-FFT). The relative error was computed using the formula: $E = 100|\beta_{IRAM} - \beta_{GM}|/|\beta_{GM}|$.*

IRAM-FD (200×100)	*GM 20×20*	*Relative error [%]*
2.3058	2.3110	0.23
1.5840	1.5947	0.67
8.0878–01	8.5624–01	5.54
4.9398–01+3.2770–01i	5.1314–01+2.7998–01i	1.41
4.9398–01–3.2770–01i	5.1314–01–2.7998–01i	1.41
7.3406–01i	6.9140–01i	6.17
1.1297i	1.1278i	0.17
1.2639	1.2571i	0.54

Table 5. *Comparison of the values of the propagation constants computed using the IRAM-FFT algorithm and the IEEM-FFT algorithm using the deflation techniques with additional re-orthonormalization performed every 20 iterations. The structure modelled was an image guide (structure B in Figure 20). The 4-th eigenvalue has not been found in the case of the IEEM-FFT algorithm due to the lack of convergence of the iterative process. Other test parameters: FFT length = 256, number of expansion functions = 7 in every direction, NEV = 4, NCV = 20, f = 15 GHz.*

IRAM-FFT	*IEEM-FFT-deflation*	*Difference [%]*
2.5408	2.5408	0.00
1.7205	1.7207	0.01
1.6973	1.6967	0.04
4.3770e–01	—	—

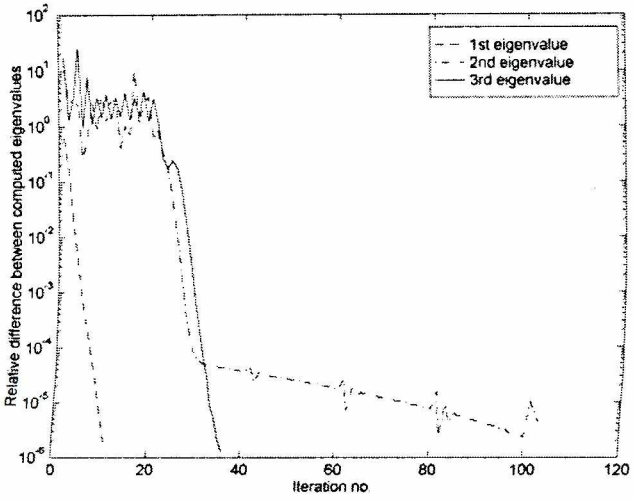

Figure 24. *Convergence profiles for the IEEM-FFT method applying deflation procedures with additional re-orthonormalization performed every 20 iterations. The relative difference was computed for the two most recently found approximations of eigenvalues. The structure modelled was an image guide (structure B in Figure 20) parameters: FFT length = 256, number of expansion functions = 7 in every direction, f = 15 GHz, convergence criterion = 1.0e-06.*

equalled 3. The Table shows almost a perfect agreement between the obtained results, confirming that the IRAM-FFT algorithm may be successfully applied to deal with the discussed class of structures.

The next series of tests aimed at comparing the IRAM-FD algorithm and the Galerkin Method. Table 6.2 shows the normalized propagation constants β/k_0 computed using the two considered algorithms for an image guide with a discontinuous permittivity profile (structure C in Figure 20). One may note that the differences in computed eigenvalues are significantly larger than in the case of the comparison between GM and IRAM-FFT-NI (cf. Table 2), although especially for modes far from cut-off these differences stay at an acceptable level. This may easily be explained if one takes into account that entirely different discretization strategies are applied in the compared methods. (In the case of the comparison between IRAM-FFT-NI and GM the finite-dimensional mappings in both methods were based on the same concept of representing operators by appropriate inner products.) In fact we do not know which propagation constants are computed with greater accuracy, as we do not know whether e.g. the 200 × 100 FD discretization grid provides a more accurate finite representation of the input operator than the 20 × 20 matrix of Fourier coefficients also representing the same operator. Still, a simple conclusion which may be drawn is that the IRAM-FD method provides one of acceptable approaches towards modelling of the considered waveguiding structures.

The last group of tests, presented in Table 5, shows some preliminary results obtained using the IEEM-FFT algorithm with deflation and re-orthonormalization

procedures applied. In the tests the propagation constants for an image guide with a discontinuous permittivity profile (structure B in Figure 20) were computed as eigenvalues of the scalar operator $\tilde{\mathbf{T}}$ (cf. equation (59)). The comparison shows almost perfect accordance with the results obtained using the IRAM-FFT algorithm. The problem which has been found to occur with the IRAM-FFT-deflation algorithm is the lack of convergence of the iterative process while trying to find the fourth eigenvalue. Although many different choices of parameters concerning deflation and re-orthogonalization procedures were made the situation did not improve. Still, the process of re-orthonormalization was found to play a particularly important role in stabilizing the iterative process in the algorithm, as excluding this phase from the algorithm resulted in the lack of convergence to any of the higher-order propagation constants. Figure 24 shows convergence profile for the discussed test using th IRAM-FFT-deflation method with additional re-orthogonalization. One may note that the fastest convergence occurs if the first eigenvalue is being sought. The convergence to higher-order eigenvalues usually involves more iterations, although this depends largely on the starting point of the iterative process.

7. Numerical results — performance of the parallel solvers

The previous section concentrated on assessing the scope of applications of the discussed numerical eigensolvers within the theory of electromagnetic waveguides. This part of the study focuses on performance and scalability of the solvers in distributed memory parallel systems, including both scalable supercomputer systems as well as networks of workstations. The tests presented in the following sections aim at determining whether the considered parallel eigensolvers may be efficiently applied in the mentioned systems and which factors affect their performance in these environments.

7.1 Characteristics of the hardware test platforms

We start with a brief description of the characteristic features of distributed memory systems which served as testing platforms for the proposed parallel solvers, presented in Section 5. Three environments are described: IBM Scalable Power2 (SP2) Parallel System, Cray T3E and a network of workstations with a special attention dedicated to the potential impact of the specifics of system architectures on the performance of parallel programs based on the message-passing programming paradigm.

7.1.1 IBM Scalable Power2 Parallel System

The IBM SP2 Parallel System installed in the Academic Computer Centre TASK in Gdansk, which has been used as one of the platforms for numerical tests, is a fully scalable distributed memory system which consists of 15 processing nodes and a high performance interconnection network. (A maximum of 8 nodes may have been used to run a parallel task in this system.) Starting from the processing nodes in the considered system there are 14 "thin" nodes and 1 "wide" node, each

node equipped with a POWER2 processor having a 66.7 MHz clock and a peak performance of 267 Mflops, 64 MB (128 MB in case of a wide node) of the local RAM memory and local disk storage. The total peak performance of the described system equals about 4 Gflops.

All nodes have 128 Kbytes of level 1 data cache and 32 Kbytes of instruction cache. The POWER2 superscalar processor has a four-way set-associative dual-ported cache with load and store pipes controlled by two fixed-point processors. It is possible to have cache with 32-byte data paths, so in total four double precision words can be loaded and four stored in one clock cycle. 15-20 clock cycles are required to recover from a cache miss. POWER2 has also two floating point processors each having a pipeline able to do a multiply and add in two cycles. Summing up, effectively two multiplies and two adds can be performed by the CPU each cycle provided that loads and stores can be appropriately scheduled by the compiler [54].

The other important element of the SP2 system, greatly affecting the performance of parallel applications, is the interconnection network. This interconnect, named High Performance Switch (HPS) is a low latency switching network capable of sustaining high transmission bandwidth. HPS may connect 16 nodes and can handle up to 128 communication threads between every pair of nodes. The bandwidth equals about 40 Mbytes per second in the bidirectional transmission and the latency equals less than 40 microseconds.

Table 6. *Network performance for MPI and MPIL based message communication in the IBM SP2 parallel system [54].*

Library	*Network latency [μ s]*	*Transfer rate [MB/s]*
MPI (ip)	656	6.26
MPI (us)	71	34.77
MPL (ip)	270	8.38
MPL (us)	44	35.20

The above hardware designs are supported by software solutions which enhance the performance of distributed parallel programs. The Communication System Support (CSS) should be mentioned in the first place. The CSS is a set of software layers that support communications through the High Performance Switch and includes interfaces with protocols that can be used for inter-processor communication using HPS. Two protocols are supported by CSS: 1) The Internet Protocol (IP). If this protocol is used to communicate through the HPS the IP messages are wrapped in the HPS protocol, so that applications using IP protocol can transparently apply HPS to achieve high-speed communication and data transfer. 2) The user space (US) protocol. This protocol is used in the message-passing library subroutines provided with IBM AIX Parallel Environment to develop high-performance parallel programs. Communication operations are directly executed

from the user space without any system call (bypassing the TCP/IP layers) which enhances the communication performance — compare Table 6.

The already mentioned IBM's Parallel Environment is a set of programming tools supporting parallel distributed applications. It includes parallel libraries: MPI, MPL and PVMe, especially tuned for use with the SP2 system nodes and the High Performance Switch interconnection network to achieve better parallel performance. The Parallel Environment also provides various tools which support compiling, running, monitoring and debugging parallel programs, e.g. compiler scripts (such as `mpxlf` — message-passing Fortran compiler) which automatically link in the message-passing libraries and provide environment variables allowing one to control the run-time environment.

Table 7. *All-to-all communication performance on a 32-wide node IBM SP2. m is the message size in bytes [55].*

m [bytes]	*MPI: Time [s]*	*HPF: Time [s]*
128	0.001	0.670
1K	0.016	1.236
128K	0.139	58.360
1M	0.784	355.714

Summing up, the IBM SP2 parallel system is a very typical distributed memory machine which provides both hardware and software support for distributed parallel computing. The programming model which emerges as a dominant one in this system is the message-passing programming due to optimized message-passing libraries (MPI, MPL) offering truly high-level parallel performance as compared to alternative solutions. This may be very clearly seen if we quote some performance data from the paper by Klepacki [55]. The results of the tests shown in Table 7 give the execution time in the IBM SP2 system for the all-to-all global communication routine implemented using Message Passing Interface and High Performance Fortran. The performance of MPI implementation is strikingly high as compared to the HPF version of the program. This indicates that if intensive collective communication takes place in a given parallel program then using the low-level message-passing programming model will result in a tremendous improvement in performance.

7.1.2 Cray T3E parallel system

The Cray T3E is a scalable virtual shared memory (VSM) parallel system. The term "virtual shared memory" means that although the memory is distributed across the processing nodes, the system provides a global, shared address space of up to 2048 processors over a three-dimensional torus topology interconnection network. node of the T3E system contains an Alpha 21164 processor running at 300 MHz

(450 MHz) clock, system control chip, local memory and network router. Torus links provide a raw bandwidth of 600 MB/s in each direction, with payload bandwidths ranging from 100 to 480 MB/s after protocol overheads (cf. [43]). The input / output is based on the GigaRing channel with sustained bandwidths of 267 MB/s for every four processors.

The Alpha 21164 processor is capable of issuing four instructions per clock cycle (with one floating point add and one floating point multiply) which gives it a peak performance of 600 Mflops (being more than twice as much as for the POWER2 processor). It contains the 8Kb level 1 data and instruction caches and 3-way associative 96 Kb level 2 unified cache. The local memory ranges from 64 Mb

Table 8. *Network performance for MPI, PVM and Shmem based message communication in the Cray T3E parallel system. This data has been published in [43].*

Library	*Network latency [μs]*	*Bandwidth [Mbyte/s]*
sma (Shmem)	1	350
PVM	11	150
MPI	14	260

to 2 Gb and the transfer rate from memory to processor equals about 1 Gb/s. There is no board level cache in the T3E nodes. Instead stream buffers are applied enhancing the local memory bandwidth.

The nominal latency of the network in the Cray T3E system equals 1 μs. Still, if using various message-passing libraries the effective latency is usually much larger due to overheads e.g. associated with buffering. Table 8 shows the effective network latency and the asymptotic bandwidth for different message-passing libraries. The most impressive result is the minimal latency offered by sma (shared memory access) Cray library which handles a simple one-sided communication. In the case of both PVM and MPI libraries the latencies are significantly higher. Still, in all the cases the bandwidth is similar. At this point a comparison can be made between the discussed Cray T3E and IBM SP2 systems. In the case of MPI communication the latency is about 5 times higher for the IBM SP2 and the bandwidth is about 7.5 times larger for the Cray T3E. In both aspects Cray T3E provides a significantly more efficient interconnection network.

Summing up, the Cray T3E system offers a highly efficient environment for parallel distributed programming offering an extremely low latency and high bandwidth network as well as high performance processing units. Although the globally addressable memory space, accessible through the calls to Shmem library routines is not used directly in the message-passing programming model, the native implementations of message-passing libraries available in the Cray T3E systems make use of this extremely efficient communication mechanism. It is clearly seen that the T3E system shows a significant superiority in terms of both the network performance and the computational capabilities of the processing nodes (600

MFlops as compared to 267 MFlops per processing node) as compared to the IBM SP2 system.

The actual Cray T3E system installation in the Interdisciplinary Centre for Mathematical and Computational Modelling at the University of Warsaw which has served as a platform for performance tests presented in the following sections consists of 32 processing nodes (with a maximum of 24 processing elements available for a single parallel task), with each computational node equipped with 128 MB of local RAM memory.

Table 9. *Approximate message startup times (t_s) and transmission rates (per four byte word) (t_w) in inter-processor communication for selected parallel systems. Data published in [29].*

Parallel system	$t_s[\mu s]$	$t_w[\mu s]$
IBM SP2	40	0.11
Intel Paragon	121	0.07
Workstations on Ethernet	1500	5.0
Workstations on FDDI	1150	1.1

7.1.3 Networks of workstations

The third distributed memory system which has served as a hardware platform for numerical tests is a cluster of 6 workstations connected through the Asynchronous Transfer Mode (ATM) based network. The workstations used are the Silicon Graphics Indy systems equipped with the R4400 RISC processors. The workstations are connected via a standard ATM switch.

Although obviously the performance of the workstations is lower in comparison to the performance of processing nodes in the IBM SP2 or Cray T3E, the main differences between NOWs and distributed memory supercomputer systems show up in specifics of the interconnection network. Generally speaking, NOWs may be characterized as systems in which there is a very significant imbalance between the processing capabilities and network communication efficiency. In other words, both high latency and relatively narrow bandwidth cause that intensive inter-processor communication which may occur in distributed parallel programs cannot be handled efficiently. Table 9 shows the comparison of network parameters between parallel supercomputers (IBM SP2, Intel Paragon) and networks of workstations. One may note that the network latency for NOWs may be almost 40 times higher than for supercomputer interconnects. This situation does not improve significantly if a more advanced network technology is used (e.g. FDDI). The transmission rates also vary significantly and are lower by an order of magnitude for the networks of workstations. Clearly, the result is the lack of scalability of such network systems, so that the number of workstations which can be connected in order to obtain high speedup and efficiency of the parallel programs is rather limited. An interesting discussion of a distributed memory network system based on computers with Intel

Pentium Pro processors connected through Fast Ethernet and its assessment for different parallel application programs may be found in the paper [56].

7.2 Performance of the solvers

Having briefly described some characteristics of distributed memory parallel systems which have served as hardware platforms for numerical tests let us now present the results of performance tests starting with the description of the details concerning methods of time measurement, compilation options, libraries used and run-time environment.

The main goal of the performance tests has been determining a few basic parameters characterizing investigated parallel programs, including the speed-up, efficiency and scalability. Both speed-up and efficiency have been computed as relative speed-up ($S_{relative}$) and efficiency ($E_{relative}$) which are given by the following formulas:

$$S_{relative} = \frac{T_1}{T_P} \tag{71}$$

$$E_{relative} = \frac{S_{relative}}{P} \tag{72}$$

where T_1 is the execution time on one processor of the parallel program and T_P is the execution time on P processors of the same parallel program. The above quantities are called relative because they are defined with respect to the parallel algorithm executing on a single processor. The absolute speed-up and efficiency are obtained if the time T_1 in the equations (71) and (72) is taken as the execution time on one processor for the best-known algorithm.

Calculating speed-up and efficiency involve measurements of the execution time of programs running in a parallel environment. The following general rules and timing procedures have been applied:

— The measured execution time was the *user time* and not the wall-clock time.

— The following routines have been used to measure the user time: 1) In the IBM SP2 and Cray T3E systems the `times()` routine has been used. This function gives the number of clock cycles already used by a given process. 2) In the SGI Indy systems the `etime` subroutine has been used to measure the user time.

— The time T_P from equations (71) and (72) has been calculated as follows: 1) The mean execution time (user time) T_P^i for all the P processes involved in the computation has been calculated for every measurement. 2) T_P has been calculated as a mean value of the T_P^i. The number of measurements has typically equalled from 2 to 5. Only two measurements were performed if the values obtained in both tests differed insignificantly which indicated that the measurements were exact and reliable.

The following compilers and compilation options have been used in the testing platforms:

— The `xlf` IBM Fortran compiler version 3 (jointly with the `mpxlf` script providing support for message-passing programs) available with AIX 4.1 operating system has been used to compile the codes in the IBM SP2 system. The following optimization options have been tested: `-O2 -qarch=pwr2; -O3; -O3 -qarch=pwr2; -O3 -qhot -qarch=pwr2` which perform different levels of optimization.

— The `cf90` Cray Fortran 90 compiler has been used in the T3E system. The optimization options considered included: `-scalar1; -scalar2; -scalar3; -O2; -O3`. It resulted that for the tested parallel programs there are only minor differences in performance if using `-O2, -O3, -scalar2 or -scalar3` option. A significant decrease in performance was noted if `-scalar1` or `-g, -g 1` options were applied.

— The `f77` SGI Fortran compiler available with IRIX 6.2 operating system has been used to compile codes for the SGI Indy workstations. Among the tested optimization options the simple `-O3` optimization resulted the most efficient both for the codes and supporting libraries.

A number of message-passing and numerical libraries have been used jointly with the parallel solver codes written in Fortran 77. These include:

— *MPI library*. Only native, vendor provided implementations of the MPI standard were used during the tests, including the IBM's MPI implementation available with AIX 4.1 operating system and the Cray Research MPI available within the CrayLibs package for the Unicos/mk operating system.

— *PVM library*. The library used in the tests was the public domain, portable implementation PVM v. 3.3 (cf. [41]) available from NETLIB, compiled for the SGI Indy workstations.

— *BLACS library*. We have used a portable implementation of the PVM-BLACS v. 1.1 available from the NETLIB repository. The BLACS library has been compiled for the SGI Indy workstations.

— *P_ARPACK library*. The parallel Arnoldi package is a library implemented by Maschhoff and Sorensen (cf. [53]) and available from `ftp.caam.rice.edu`. The P_ARPACK library depends on a few other libraries: MPI (or BLACS), BLAS and LAPACK (version 2). Although P_ARPACK is distributed with necessary BLAS and LAPACK (version 2) routines, the subroutines from native implementations of BLAS or LAPACK functions were used, whenever possible. Consequently the codes which used P_ARPACK were also linked to the ESSL library (IBM SP2) or LibSci (Cray T3E) library.

— *ESSL library*. The Engineering and Scientific Subroutine Library provides implementations of many computational subroutines including BLAS routines optimized for the POWER2 processor architecture (`-lesslp2`).

— *LibSci library* is a library of computational routines, implementing some of the LAPACK and BLAS functions optimized for use in the Cray T3E system.

— *FFTPACK library*. The FFTPACK library is a portable package of Fortran77 subprograms written by Paul Swarztrauber ([57]) for calculating one-dimensional Fast Fourier Transforms. The implementation is based on the Winograd version of the FFT algorithm. The library routines have been used to implement the code performing two-dimensional backward and forward Fourier transforms.

— *SPARSKIT library*. The parallel solver based on the FD finite-dimensional applied the `amux(.)` subroutine from the SPARSKIT library to compute the matrix-vector product for a sparse operator matrix. In the implementation a portable public-domain SPARSKIT code has been used ([31]).

Let us briefly present the run-time environment provided by the three discussed testing platforms:

— *IBM SP2 parallel system*. In this testing platform the Load Leveler job scheduling system and the Parallel Operating Environment (POE) has been used to run the parallel codes. An important feature of POE is that it allows the user to define which communication subsystem is to be used for inter-processor communication (Ethernet or High Performance Switch) and which communication protocol will be applied (Internet Protocol (IP) or User Space (US)). In the performance tests only the HPS communication subsystem has been used, as applying Ethernet resulted in totally unreliable performance results. Another important fact which appeared was the necessity of placing the executables in the local (`/tmp`) disks of every node involved in the parallel computation. This operation prevented the use of NFS-mounted disks which offered a rather unpredictable response during the run-time as it has been observed that if the executables were not copied to local disks the measured times varied by up to a 100 % which made the results clearly unacceptable. Such behaviour of the SP2 system has also been reported by Allan ([54]). This implies that the executables should always be copied to local disks of the processing nodes in order to obtain a stable execution time. A broader description of the issues concerning the IBM SP2 parallel run-time environment may be found in the report [58].

— *Cray T3E*. In the Cray T3E system the MPI parallel programs were run using the standard mpirun command. Scripts containing this command were submitted to the Network Queuing System (NQS). Each time before running the parallel program its code was copied to the `/tmp` local file system.

— *Network of workstations*. This platform served for tests of the PVM implementation of the IRAM (Arnoldi) solver using the FD mapping of the input operator. The parallel jobs were run from the PVM console after creating a virtual machine by adding all available hosts.

7.2.1 Internal scalability of the P_ARPACK routines

The first group of tests aimed at determining the performance and internal scalability of selected P_ARPACK library routines. The influence of different compiler optimization options and libraries linked in the parallel codes on the performance of P_ARPACK has also been examined. The hardware platform for all the tests described in this section was the IBM SP2 parallel system.

We investigated mainly the `pdnaupd()` P_ARPACK subroutine which performs the IRAM iteration for the non-symmetric real problems. The input operator with eigenvalues to be found was the square diagonal matrix with random elements between 0 and 1 located on the diagonal. Four of the diagonal elements were incremented by 1.1 which allowed them to be easily found by the `pdnaupd()` routine. The matrix had the size of 160.000 and was block-distributed among the

Table 10. *Performance of* `pdnaupd()` *routine for IP and US protocols, N = 160000, NEV = 4, NCV = 20, the number of Arnoldi iterations = 4, Compiler directive:* `mpxlf -O2 -qarch=pwr2 xxx.f`*; (portable BLAS). All times are given in seconds.*

Number of nodes	*Time (IP)*	*Speedup (IP)*	*Time (IP)*	*Speedup (US)*
1	51.05	1.00	51.17	1.00
2	25.59	1.99	26.78	1.91
4	17.64	2.89	20.10	2.55
8	7.71	6.62	9.83	5.20

Table 11. *Performance of* `pdnaupd()` *routine for IP and US protocols, N = 160000, NEV = 4, NCV = 20, number of Arnoldi iterations = 4, Compiler directive:* `mpxlf -O3 -qarch=pwr2 xxx.f` *(portable BLAS). All times are given in seconds.*

Number of nodes	*Time (IP)*	*Speedup (IP)*	*Time (US)*	*Speedup (US)*
1	72.96	1.00	77.25	1.00
2	38.52	1.89	39.96	1.93
4	25.33	2.88	25.43	3.04
8	12.73	5.73	12.91	5.98

Table 12. *Performance of* `pdnaupd()` *routine for IP and US protocols, N = 160000, NEV = 4, NCV = 20, number of Arnoldi iterations = 4, Compiler directive:* `mpxlf -O3 -qarch=pwr2 xxx.f -o xxx -lesslp2`*, (ESSL BLAS). All times are given in seconds.*

Number of nodes	*Time (IP)*	*Speedup (IP)*	*Time (US)*	*Speedup (US)*
1	72.27	1.00	77.03	1.00
2	38.26	1.89	39.72	1.94
4	21.82	3.31	29.23	2.63
8	13.35	5.41	13.09	5.88

processors. The `pdnaupd()` routine was called with the following parameters: NEV = 4 (number of requested eigenvalues), NCV = 20 (number of columns of the matrix $\underline{\underline{V}}_k$ — cf. equation (39)) and WHICH = 'LM' (eigenvalues with the largest magnitude were to be found). The characteristics of the problem presented above were chosen so that it was independent of the changing problem size and the number of Arnoldi update iterations remained constant in every case.

The code of the parallel solver for the operator described above was compiled with different optimization options. (The P_ARPACK library was compiled with the same sets of options.) The BLAS routines needed by the P_ARPACK routines were provided in two implementations: 1) The standard, portable implementation; 2) The ESSL IBM's implementation. The different compilation and linkage options are given below:

— `mpxlf -O2 -qarch=pwr2 xxx.f -o xxx` (Table 10)

— `mpxlf -O3 -qarch=pwr2 xxx.f -o xxx` (Table 11)

— `mpxlf -O3 -qarch=pwr2 xxx.f -o xxx -lesslp2` (Table 12)

— `mpxlf -O3 -qhot -qarch=pwr2 xxx.f -o xxx -lesslp2` (Table 13)

Tables 10, 11, 12 and 13 show the results of performance tests for different compilation flags used while building both the library and the tested executable. In the Tables all the times are average *user times* given in seconds. The time measured in all the cases is the time spent in the Arnoldi iteration of the `pdnuapd` routine and should be considered as the total time needed by the IRAM algorithm to converge to the wanted solutions. During the tests different communication protocols of inter-processor communication have been used: the column "Time (ip)" shows the timings while the Internet Protocol has been used, while "Time (us)" gives times measured while the User Space protocol has been applied. In all the cases the number of implicit Arnoldi updates equalled 4 and did not change with the changing number of processors used.

The results of the tests show that for the considered size of the problem N = 160000, the total time spent in the Arnoldi iteration is shorter if the library is compiled with a lower level of optimization, i.e. the `-O2` flag and not `-O3` flag.

Table 13. *Performance of* `pdnaupd()` *routine for IP and US protocols, N = 160000, NEV = 4, NCV = 20, number of Arnoldi iterations = 4, Compiler directive:* `mpxlf -O3 -qhot -qarch=pwr2 xxx.f -o xxx -lesslp2`, *(ESSL BLAS). All times are given in seconds.*

Number of nodes	*Time (IP)*	*Speedup (IP)*	*Time (US)*	*Speedup (US)*
1	76.77	1.00	76.66	1.00
2	38.18	2.01	39.26	1.95
4	19.62	3.91	21.60	3.55
8	10.78	7.12	10.05	7.63

While using only the `-O3` flag the performance degrades by about 30%. This kind of situation has also been observed by Allan ([54]). Comparing the tests performed using the IP and US communication protocols it may be noted that the execution times (user times) are slightly higher. This fact may be explained by the US protocol characteristics where the calls to communication routines are made directly from the user space omitting the system calls which are used in the case of the IP communication. Nevertheless, the results given in the Tables show that applying US protocol results in a better speed-up if the `-O3` optimization flag has been used during compilation. This result may confirm that the US protocol provides a more efficient communication between the nodes of the SP2 system.

Table 13 shows results of performance tests if an additional compiler flag `-qhot` is used during the compilation of both P_ARPACK library and the solver code. This flag forces the compiler to determine whether or not to perform high level optimization (`-O3`) on specific loops in the program's code. Consequently different parts of the code as optimized with different optimization levels. Although the measured times are similar to those obtained only with `-O3` flag, the speed-up obtained in this compilation method is the highest, e.g. for 8 processors the speedup exceeds 7.00 (cf. Table 13), while in the former cases it does not reach 6.00 (cf. Tables 10, 11, 12).

The above tests also examined the influence of choosing an implementation of the BLAS library routines on the performance of the Arnoldi solver. Parallel ARPACK software uses a number of Basic Linear Algebra Subroutines including a matrix-vector product subroutine `Xgemv(.)` or a matrix by upper triangular matrix multiplication `Xtrmm(.)`. In the first test (Table 11) the routines from the portable BLAS version 2 implementation provided together with P_ARPACK have been used, while in the second test (Table 12) the implementations of BLAS subroutines from the IBM's ESSL library have been linked to P_ARPACK routines. The results show that the performance of the pdnaupd does not depend much on the

Table 14. *Scalability of the* `pdnaupd()` *routine. The size of the problem equals* $N = P \cdot 200000$*, where* P *is the number of processors,* $NEV = 4$*,* $NCV = 20$*, compiler directive:* `mpxlf -O3 -qhot -qarch=pwr2 xxx.f -lesslp2.`

Number of nodes	*Time (IP)*	*Efficiency*	*Time (US)*	*Efficiency*
1	97.29	1.00	95.81	1.00
2	97.47	1.00	93.64	1.02
3	125.80	0.77	96.53	0.99
4	143.92	0.67	99.99	0.96
5	134.11	0.72	95.92	1.00
6	110.43	0.88	96.95	0.99
7	138.25	0.70	97.92	0.98
8	136.95	0.71	100.98	0.95

Table 15. *Performance of* `pdnaupd()` *routine for IP and US protocols, $N = P \cdot 200000$, where P is the number of processors, NEV = 1, 8, 16, 32; NCV = 40, compiler directive:* `mpxlf -O3 -qhot -qarch=pwr2 xxx.f -lesslp2`. *All times are given in seconds.*

NEV	*Number of nodes*	*Time (IP)[s]*	*Efficiency*	*Time (US)[s]*	*Efficiency*
1	1	40.96	1.00	40.92	1.00
	2	41.05	1.00	40.98	1.00
	4	41.90	0.98	42.68	0.95
	8	47.00	0.87	43.27	0.94
8	1	83.11	1.00	82.90	1.00
	2	105.87	0.78	85.93	0.96
	4	100.78	0.82	86.68	0.96
	8	131.61	0.63	87.26	0.95
16	1	140.69	1.00	143.14	1.00
	2	141.33	0.99	146.62	0.98
	4	185.00	0.76	*	*
	8	240.11	0.58	147.02	0.97
32	1	390.62	1.00	389.61	1.00
	2	496.79	0.79	395.85	0.98
	4	503.99	0.77	399.45	0.97
	8	586.86	0.66	401.66	0.97

version of BLAS library used, as minor differences in the execution times are observed. Still, the authors of P_ARPACK ([53] and [59]) suggest that the native BLAS implementations should be linked to the codes whenever possible instead of those provided with the P_ARPACK software. Consequently in all the following tests only the IBM's proprietary ESSL library or the LibSci library in the Cray T3E system containing implementations of BLAS subroutines have been used.

The following series of tests measured the internal scalability of the `pdnaupd()` newline P_ARPACK library subroutine. The same solver program as in the previous tests has been used during the measurements. As it has already been remarked the input matrix in the parallel solver has been chosen so that its characteristics was independent of the size of the problem. More precisely, during the scalability tests with the `pdnaupd()` subroutine, the number of Arnoldi update iterations remained the same for a fixed number of eigenvalues to be computed (NEV = 4) and the changing size of the problem. The size of the problem was chosen to be $N = 200000$ for 1 processor and increased linearly with the number of processors. Both IP and US communication protocols have been used during the measurements and the codes were compiled using the following directive was:

`mpxlf -O3 -qhot -qarch=pwr2 xxx.f -o xxx -lesslp2`. The results of the tests are given in Table 14.

The results of scalability tests show that the efficiency of the `pdnaupd()` subroutine remains relatively high for the available number of processors involved in the parallel computation. It is well seen that application of the US protocol gives a considerable improvement in scalability and efficiency of the executed program. For the number of processors from 1 to 8 and the problem size $N = 200000$, 40000, ..., 1600000 the efficiency stays above the level of 95 %, while in case of the Internet Protocol it falls below 70 %. The reason for such behaviour is that for large problem sizes the amount of communication increases, so that both the network latency and the bandwidth (compare Table 6) start to play an important role during the run-time. It has to be noted that the above results stay in close accordance with the scalability results reported in [53] for tests performed in the Maui HPCC SP2 machine. The obtained scalability is high but obviously not perfect. This effect is due to a serial bottleneck caused by the algorithm in which the upper Hessenberg matrix and the calculations involving this matrix are replicated by all the processors during the Arnoldi factorization.

Another series of tests performed using the previously described solver intended to find out the dependence of the total execution time of the `pdnaupd()` P_ARPACK subroutine on the number of eigenvalues (NEV) to be computed. In the IRAM algorithm the increment of $NEV = k$ causes an increment in the memory storage requirements, the number of computations and the size of messages communicated among the processors. In the parallel implementation of ARPACK these factors may affect performance and scalability of the library routines. As mentioned before, the upper Hessenberg matrix is replicated on every processor and

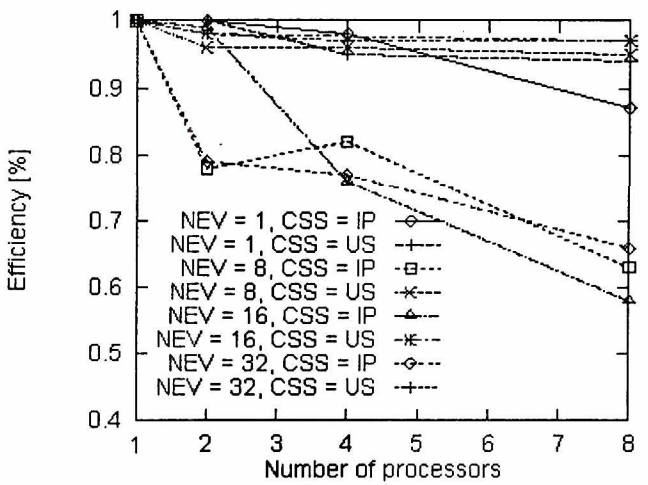

Figure 25. *Efficiency of the* `pdnaupd()` ***P_ARPACK subroutine. The size of the test problem equals*** $N = P \cdot 200000$. $NEV = 1, 8, 16, 32$; $NCV = 40$. *The number of Arnoldi update iterations remained constant for a fixed number of eigenvalues (NEV) to be computed.*

therefore may cause a serial bottleneck as its size increases. An increasing value of k also results in the increased communication costs during the re-orthogonalization phase where more global sums have to be computed and communicated using the global reduction operations.

Table 15 shows the timings obtained for the `pdnaupd()` subroutine. The size of the problem equalled $N = P \cdot 200000$, where P is the number of processors. The number of eigenvalues to be computed NEV = 1, 8, 16, 32 (NCV = 40). The number of Arnoldi update iterations remained constant for a fixed number of eigenvalues (NEV) to be computed. Once again two communication protocols were considered.

It may be noted that in the case of using the Internet Protocol, the efficiency decreases faster for larger values of NEV, e.g. for 8 processors the efficiency equals: 0.87 (NEV = 1), 0.63 (NEV = 8), 0.58 (NEV = 16). (The graph of the solver efficiency vs. the number of processors for different values of NEV is shown in Figure 25.) This effect has not been observed while applying the US protocol. A conclusion may be drawn that the degradation in the performance in the first case is mainly due to communication overhead and not a serial bottleneck caused by

Table 16. *Time spent on orthogonalization phase in the* `pdnaupd()` *routine. The test were performed using IP and US protocols, $N = P \cdot 200000$ where P is the number of processors; NEV = 1, 8, 16, 32; NCV = 40, compiler directive:* `mpxlf -O3 -qhot -qarch=pwr2 xxx.f -lesslp2`. *All times are given in seconds.*

NEV	*Number of nodes*	*Time (IP) [s]*	*Percent of total time*	*Time (US) [s]*	*Percent of total time*
1	1	19.72	48	19.75	48
	2	19.87	48	24.49	59
	4	27.78	53	25.49	59
	8	23.86	50	27.78	64
8	1	23.86	28	37.78	45
	2	53.51	50	39.32	46
	4	49.92	49	49.92	57
	8	70.61	54	53.34	61
16	1	54.80	38	55.71	39
	2	55.11	39	58.12	40
	4	81.19	44	*	*
	8	112.69	47	77.47	53
32	1	95.03	24	95.14	24
	2	135.80	27	94.13	24
	4	139.65	28	126.29	32
	8	173.65	29	136.61	34

replicating the $\underline{\underline{H}}_k$ matrix. If this had been the cause, the efficiency would have degraded for both communication subsystems.

The next Table (Table 16) shows the user times spent in the orthogonalization phase during the Arnoldi factorization for different numbers of eigenvalues to be computed. The data were obtained for the same parallel solver as in the previous

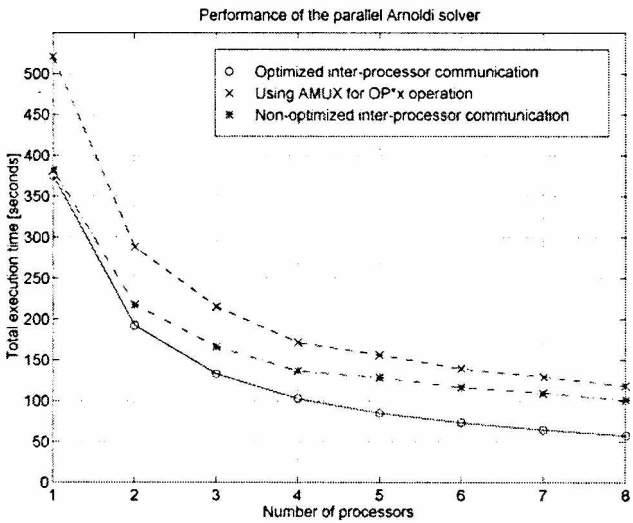

Figure 26. *Total execution time of the parallel IRAM-FD solver as a function of the number of processors for different methods of calculating the parallel matrix-vector (OP*x) product.*

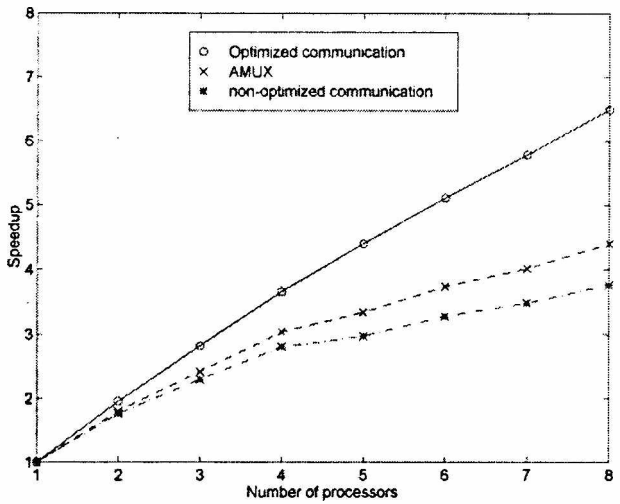

Figure 27. *Speedup in the total execution time of the parallel IRAM-FD solver as a function of the number of processors for different methods of calculating the parallel matrix-vector product.*

paragraph. During the orthogonalization phase global sums have to be computed and the number of these global reduction operations depends on the number of eigenvalues NEV to be computed. The test results show clearly that the percentage of time spent on orthogonalization increases with the increasing number of processors used (for a fixed value of NEV). This effect is due to the communication costs which appear during computation of global sums. Still, a positive fact which may be noted is that the percentage of time spent on orthogonalization decreases (for a fixed number of PEs) with the increasing value of NEV. Consequently this scaling effect may be exploited to reduce the influence of the poorly scalable orthogonalization operation on the overall performance of the solver.

Summing up the results obtained in this section one may conclude that the `pdnaupd` P_ARPACK routine shows high performance and scalability in the considered distributed memory environment. High efficiency is observed for different values of NEV and NCV, although it may potentially decrease if the percentage of time spent on the orthogonalization phase is too large as compared to the total execution time. The tests also show that the parallel performance may largely depend on the interconnection network parameters (particularly the network latency).

7.2.2 Parallel Arnoldi solver with FD operator discretization

The previous section aimed at assessing internal parallel performance of P_ARPACK routines implementing the IRAM iterative process. Consequently a trivial diagonal input matrix operator has been used in the tests. This section presents the results of performance tests of the parallel Arnoldi solver for the input operator which may be encountered in "real life" applications, discretized using the Finite Difference (FD) technique. The tests involved the parallel program (whose implementation was described in Section 5.3) applied to solving electromagnetic eigenproblems, as discussed in the previous Section. Although a specific application of the parallel algorithm is being tested the general characteristics of the parallel performance remain valid for the entire class of operators discretized using the Finite Difference mapping technique.

We start the discussion of the characteristics of the parallel IRAM-FD solver with the presentation of performance of the three versions of the algorithm (with three different methods of calculating the parallel matrix-vector product) described in Section 5.3.1.

The following tests were performed in the IBM SP2 system for the MPI implementation of the algorithm. During the tests the User Space (US) protocol was normally used to handle the inter-processor communication. The command used for compiling the P_ARPACK library and the solver code was: `mpxlf -O3 -qhot -qarch=pwr2 xxx.f -o xxx -lesslp2`. The essential input parameters defining the tests were as follows:

1. The size of the input matrix equalled $N = 39700$; the matrix was sparse with

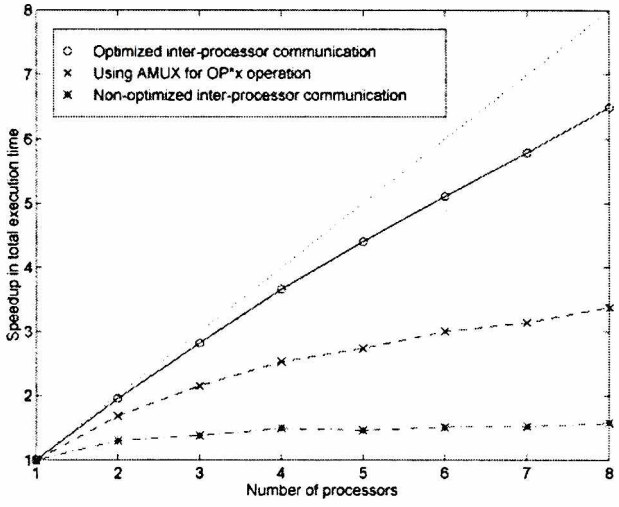

Figure 28. *Speedup in the execution time of the parallel matrix-vector (OP*x) product for different methods of calculating the parallel matrix-vector product as a function of the number of processors.*

199538 non-zero elements; among the non-zero elements 95 % were located in the five diagonals: 0 (main diagonal), +2, –2, +199, –199; the bandwidth of the matrix equalled 402.

2. NEV = 4 (number of eigenvalues to be found), NCV = 20 (number of additional eigenvalues to be filtered out)

3. The stopping criterion — the accuracy of computed eigenvalues equalled *tol* = 1.210-16.

Figure 26 shows total execution times of the parallel solver for three discussed methods of calculating the distributed, parallel matrix-vector product. In the first method the general routine `amux(.)` was used to calculate the matrix-vector (OP*x) product, the second method applied the optimized scheme of the serial calculations and the third method used the optimized version of the inter-processor communication. (compare Section 6.3.1) From the comparison of performance for one processor of the first two implementations (cf. "Using AMUX ..." and "Non-optimized ..." curves in Figure 26) is it well seen that exploiting the regularity of the matrix gives some drastic decrease in the total execution time of the solver. In this way the serial optimization has been performed.

Comparison of other two implementations which use different inter-processor communication patterns (cf. "Non-optimized ..." and "Optimized ..." curves in Figure 26) shows that for eight processors the optimized solver is almost twice as quick as the non-optimized one. More important differences between these two options of the algorithm are seen in Figure 27. The graph shows the speedup in the total execution time of the solver for different algorithms as a function of the number of processors applied. It may easily be inferred from the graph that, except

Table 17. Execution times of the parallel IRAM-FD solver (fully optimized version) while using two different communication protocols available in the IBM SP2 system. All the tests have been performed using the IBM's dedicated High Performance Switch (HPS) switching network.

Number of processors	*Time [seconds] IP protocol*	*Time [seconds] US protocol*
1	375.76	375.42
2	194.17	192.49
3	134.22	133.22
4	103.85	102.66
5	87.00	85.29
6	75.53	73.53
7	66.52	64.82
8	59.15	57.74

the algorithm applying the `amux(.)` routine, the time spent on calculating the matrix-vectoris dominated by the inter-processor communication. Consequently in the incase of non-optimized algorithm the inefficient communication between the nodes prevents the solver from speeding up for a larger number of processors. Only the optimized version of the solver is capable of making advantage of the additional computer power. In this case the program's speedup stays close to the ideal linear case.

Even more drastic differences between the implementations may be noticed in Figure 28 which shows speedups in the execution time of the matrix-vector product operation alone. As it is seen the speedup of optimized version closely approaches the ideal linear case, while the non-optimized implementation shows virtually no speedup with the increasing number of processors used. While comparing Figures 27 and 28 it may also be noted that the speedup of the parallel matrix-vector product computation is better than the speedup of the P_ARPACK routine `pdnaupd(.)`.

Another conclusion which may be drawn from the above results of performance tests is as follows: The optimization of inter-processor communication may be applied only if the input matrix is a banded one. If the input matrix is a result of discretization using the FEM technique then the distribution of non-zero elements is highly irregular producing a non-banded matrix. Consequently, in this case the parallel performance is expected to stay close to the performance shown by dashed lines in the graph in Figure 27. This shows the advantage of the FD technique over FEM if this simple static domain decomposition parallelization strategy is used.

Another aspect of parallel performance of the considered solver, specific to the IBM SP2 distributed memory system, is the influence of the communication protocol (Internet Protocol or User Space protocol) used to handle the inter-processor communication. In Table 17, the total execution time of the parallel

Table 18. *Chosen total execution times of the parallel IRAM-FD solver as a function of the number of processors involved in the computation. The tests were performed in the Cray T3E system.*

Number of PEs	*Total Time [s] NEV=4, NCV=20*	*Total Time [s] NEV=15, NCV=40*
1	225.26	631.42
2	118.27	326.33
4	59.38	180.10
8	30.70	90.56
16	16.15	51.05
24	12.52	37.99

solver as a function of the number of processors is shown for the two communication protocols applied. The results given in the Table show that there is only a minor difference in the performance while using the two different communication protocols. Still, the performance is always slightly better for the US protocol. Nevertheless, no dependence of the number of processors on the relative performance can be observed.

Let us now present the results of performance tests for the parallel IRAM-FD solver obtained in the Cray T3E parallel system. The set of input parameters used in the following series of tests was the same as in the tests performed in the IBM SP2 system. Additionally to the case where NEV = 4 (NEV — number of eigenvalues to be computed) and NCV = 20 (NCV — number of eigenvalues to be filtered out during the IRAM iteration) the tests for NEV = 15 and NCV = 40 were performed. Only the "fully optimized" version of the solver was tested. Both the P_ARPACK library and the solver code were compiled using the following directive: `f90 -O3 -X m xxx.f -o xxx -lsci -lmpi`.

Figure 29 shows the total execution time of the parallel Arnoldi FD solve for a different number of processors involved in the computation. For convenience, the same results for chosen numbers of processors have also been shown in Table 18. At this point a comparison can be made between the results obtained in the IBM SP2 system and the Cray T3E, shown in Tables 18 and 17. It may be calculated that for a single-processor execution (NEV = 4, NCV = 20) the program runs only 1.66 times faster on the Cray T3E, although the peak performance of a single node in the Cray T3E system is more than two times (2.25) higher as compared to the processing node of the SP2 system. The superiority of the Cray T3E system shows up in the parallel execution. For the IBM SP2 the speed-up in the execution time for 8 processors equals 6.50 while for the Cray T3E this factor equals 7.34. This clearly indicates that the interconnection network in the latter system is more efficient.

Figures 29 and 30 show the speed-ups in the execution time of the parallel calculation of the matrix-vector (OP*x) product and the total time used by the solver. It may be noted that the speed-up while calculating the matrix-vector product is almost perfect which is due to the form of the input operator matrix

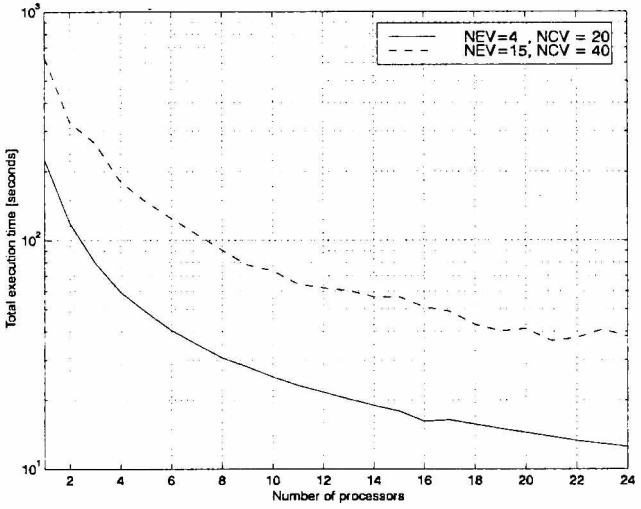

Figure 29. *Execution time of the parallel IRAM-FD solver as a function of the number of processors involved in the computation. The tests were performed in the Cray T3E system.*

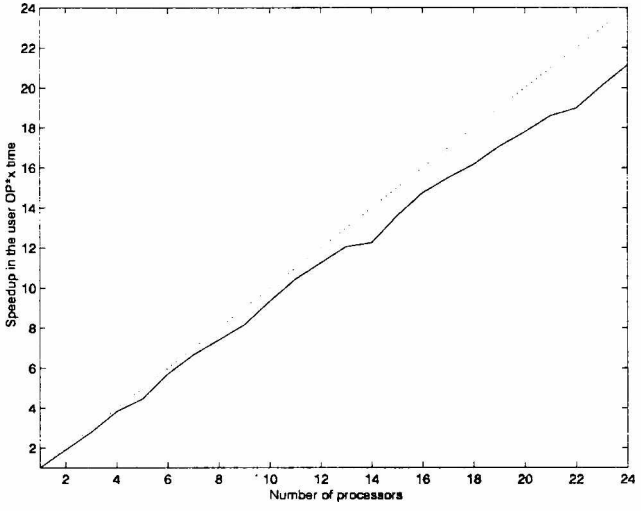

Figure 30. *Speedup in the execution time while calculating the matrix-vector (OP*x) product in the IRAM-FD solver in the function of the number of processors involved in the computation. The tests were performed in the Cray T3E system. The dotted line shows a perfect linear speed-up.*

obtained in the FD discretization. Analogously as observed in the IBM SP2 system the speed-up in total execution time is lower as compared to the speed-up for the matrix-vector product operation, still it reaches 18 for 24 processors applied which is a fairly good result.

Another thing which may be noted in Figure 31 is a relatively unstable performance of the solver for NEV = 15 and NCV = 40. This effect, which has also been observed in the IBM SP2 system, is due to a different number of both Arnoldi update iterations and number of matrix-vector operations performed during the

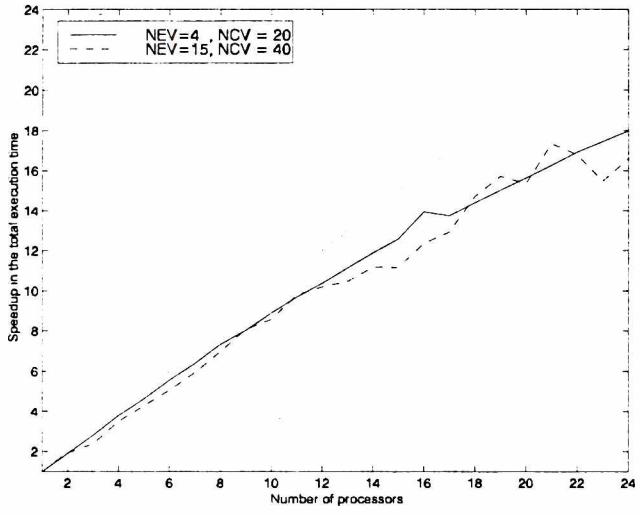

Figure 31. *Speedup in the total execution time of the parallel IRAM-FD solver as a function of the number of processors involved in the computation. The tests were performed in the Cray T3E system. The dotted line shows a perfect linear speed-up.*

execution of the algorithm with different number of processors applied. This phenomenon does not occur for the parameters NEV = 4 and NCV = 20. Generally speaking, it has been noted that a variable number of iterations occurs if the problem becomes more complex, i.e. more iterations are necessary to obtain the convergence of the Arnoldi process. Still, this does not explain the dependence of the number of iterations on the number of processors used. A possible explanation is that during the implicit updates and during the initial iteration of the Arnoldi factorization the vectors submitted to the iterative process are generated by each processor using only local data. In this case the global form of these vectors can be different for different numbers of processors applied. Consequently the starting point of the iterative process before each implicit restart may be different for different number of processors applied.

If now, for the case NEV = 15 and NCV = 40, we compute the speedups in computation time per single iteration of the algorithm we shall obtain much more stable results, presented in Figures 32 and 33, which show a true speed-up in computations due to parallelization. In fact the graph of the speed-up is almost identical to the case NEV = 4, NCV = 20.

The following graph (Figure 34) shows the percentage of total execution time of the IRAM-FD solver spent in the orthogonalization phase. These results cannot be clearly interpreted. The orthonormalization phase involves inter-processor communication so the percentage of time spent in this phase of the program should increase with the increasing number of processors. This happens only in the case NEV = 4, NCV = 20 while in the other case the effect is opposite. Probably the effect of scaling of the problem plays here the most important role. The only comment which can be made at this stage about these results is that the percentage of time spent in the orthonormalization phase does not change significantly with the

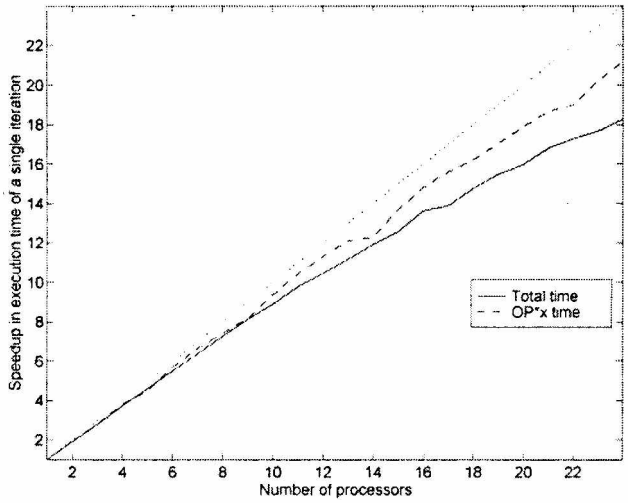

Figure 32. *Speedup in execution time of a single iteration of the IRAM-FD algorithm and the single matrix-vector product computation for the case NEV = 15, NCV = 40. The tests were performed in the Cray T3E system. The dotted line indicated the linear speed-up.*

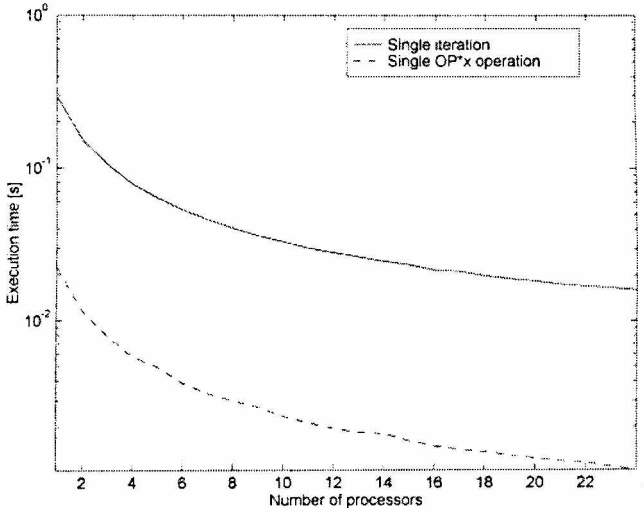

Figure 33. *Execution time of a single iteration of the IRAM-FD algorithm and the single matrix-vector product computation in the function of the number of processors applied for the case: NEV = 15, NCV = 40. The tests were performed in the Cray T3E system.*

number of processors which once again may confirm the efficiency of the interconnection network.

Another thing which has been investigated for the discussed parallel IRAM-FD solver is the load-balancing achieved for the applied parallel data distribution scheme. Figure 35 shows the difference of the execution times for the processors involved in a parallel computation. The investigated task was run on 24 processors and the differences (in per cent) are related to the execution time for the process 0. As it may be noted the load-balancing is almost perfect with the largest relative

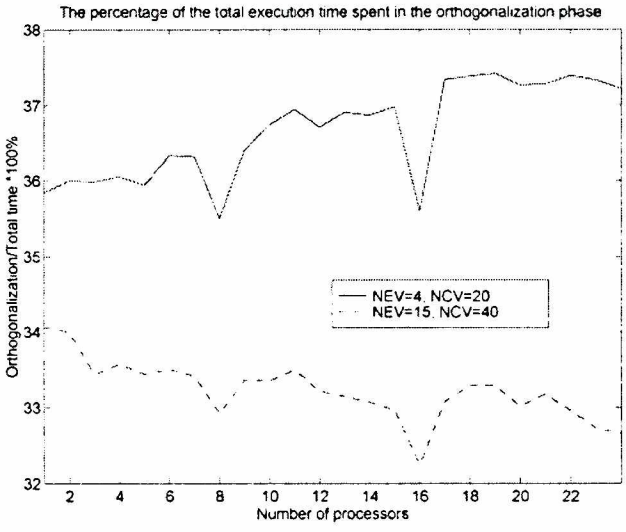

Figure 34. *The percentage of total execution time of the IRAM-FD solver spent in the orthogonalization phase as a function of the number of processors applied. The tests were performed in the Cray T3E system.*

Table 19. *Execution time and speedup in the parallel matrix-vector (OP*x) product calculation and the total execution time for the IRAM-FD parallel solver. The tests were performed in the cluster of SGI Indy Workstations connected via the ATM network.*

No. of nodes	*OP*x time [s]*	*Speedup*	*Total time [s]*	*Speedup*
1	118.76	1.00	2118.69	1.00
2	56.70	2.09	1024.81	2.07
3	41.02	2.90	762.09	2.78
4	28.81	4.12	526.27	4.03
5	27.89	4.26	503.71	4.21
6	26.69	4.45	443.22	4.78

difference in execution time equalling 0.4%. Consequently, it may be stated that an appropriate parallel data distribution scheme has been applied in the solver.

The last hardware platform used to test the Arnoldi-FD solver was a cluster of 6 SGI Indy workstations connected via ATM network. This time the implementation based on BLACS and PVM (cf. Section 5.3.2) was tested. Data showing both execution time and speed-up in the total computation time and matrix-vector product calculation time has been given in Table 19 and also presented in Figure 36. One may note that for 2 and 4 processors used, the obtained speed-up exceeds the linear speed-up. Once again this effect is due to the number of algorithm iterations changing with the number of processors. Still, the results show that in the range from 1 to 4 processors the parallel implementation of the solver

involving BLACS and PVM gives a good speed-up in the execution time and consequently the solver may be efficiently used in the network environment. Although the asymptotic behaviour of the parallel solver cannot be observed due to insufficient number of available workstations the results for 6 processors already indicate that a serious degradation in performance may occur if a larger number of workstations is used in the computations.

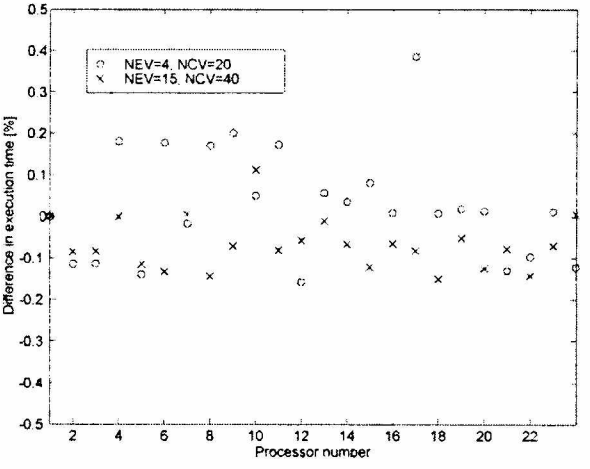

Figure 35. *The per cent variation in the execution times (as related to the execution time for process 0) on different processors involved in a parallel computation for the IRAM-FD solver. The tests were performed in the Cray T3E system.*

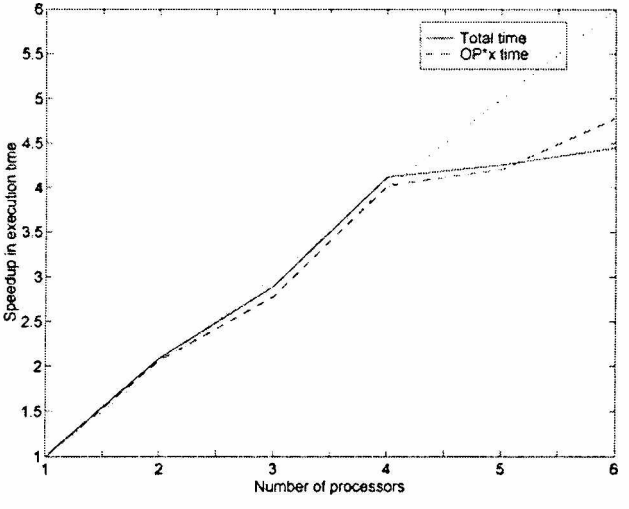

Figure 36. *Speedup in the parallel matrix-vector (OP*x) product calculation and the total execution time for the IRAM-FD parallel solver as a function of the number of processors involved in the computation. The tests were performed in the cluster of SGI Indy Workstations connected via the ATM network.*

Summing up, the results of performance tests for the parallel IRAM-FD solver obtained in different parallel platforms show a very good scalability of the solver which is mainly due to an efficient parallel design and implementation of the parallel matrix-vector product computation.

7.2.3 Parallel Arnoldi solver using implicit operator projection

The performance tests of the Arnoldi (IRAM) solver using implicit representation of operators (described in Section 5.4) have focused on measuring the speed-up achieved by the program in given parallel distributed memory environments. All the following tests have been performed in the two scalable parallel systems: the IBM SP2 and the Cray T3E.

The first Figure (Figure 37) shows the execution times for the IRAM-FFT solver (number of the eigenvalues to be found NEV = 8, number of eigenvalues to be filtered out NCV = 40) for different Fast Fourier Transform lengths and different number of expansion terms used to represent the functions in the input operator's domain. Analogous results are shown in Figure 38 for the tests performed in the Cray T3E system. It may be noted that usually the execution times are lower by about 1/3 for the Cray T3E system, analogously as observed in the previous section.

Figures 39 and 40 show the speed-ups in the total execution times for the IRAM-FFT solver in the IBM SP2 and Cray T3E platforms. It may be seen that the results for both platforms are entirely analogous. The best speed-up may be observed in the case when the number of expansion functions equals 256 in every direction and the FFT length equals 1024 (in both directions). This indicates that applying a larger, more complex problem gives better performance. In other words, the solver scales well with the problem size and complexity. The other positive

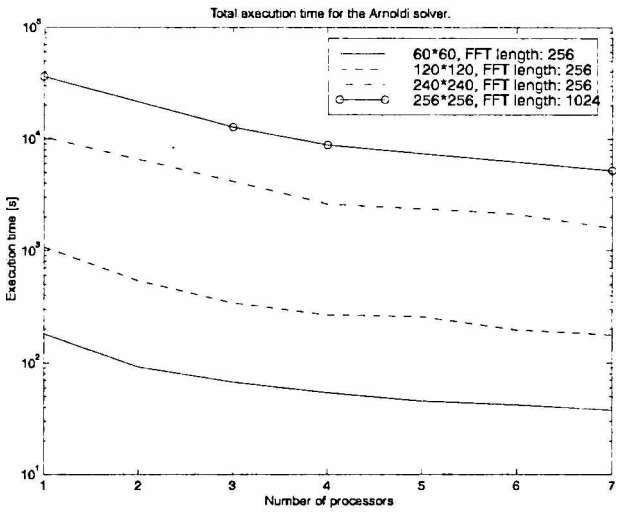

Figure 37. *Total execution time of the IRAM-FFT parallel solver as a function of the FFT length. The tests have been performed in the IBM SP2 system. (NEV = 8, NCV = 40).*

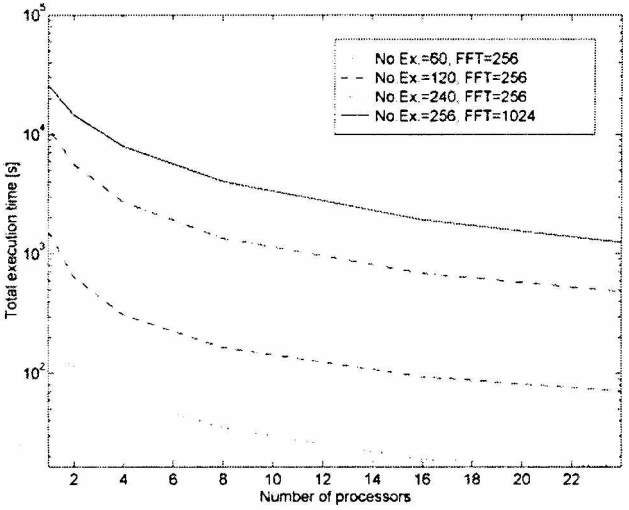

Figure 38. *Total execution time of the IRAM-FFT parallel solver as a function of the FFT length. The tests have been performed in the Cray T3E system.*

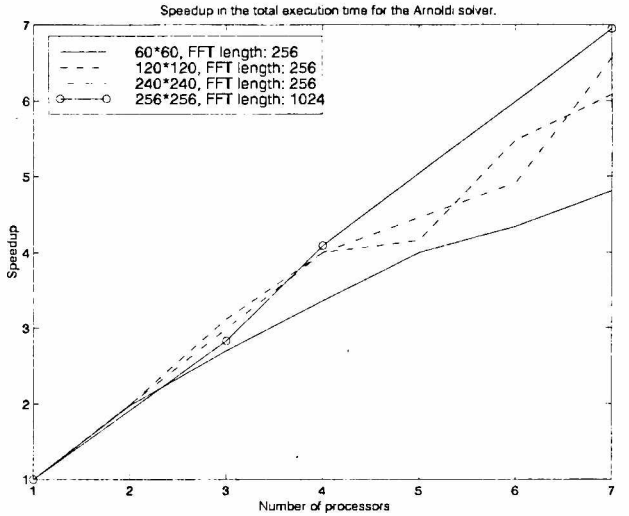

Figure 39. *Speed-up in the total execution time of the IRAM-FFT parallel solver vs. the number of processors. The tests have been performed in the IBM SP2 system.*

result which may be noted is that as the ratio between the number of expansion functions and the FFT length increases, the parallel performance also improves. This means that although the percentage of time spent on the matrix-vector product computation related to the total execution time increases and also the size of inter-processor communication during the parallel transposition operation becomes larger this does not cause a parallel bottleneck.

It may be noted in Figure 40 that the effect of a number of iterations changing with the number of processors also shows up in the IRAM-FFT solver resulting in

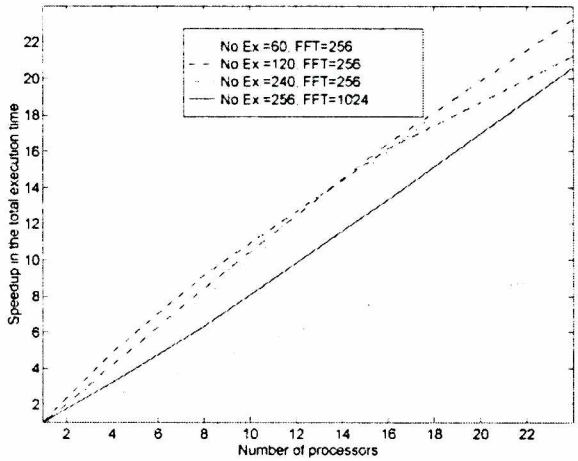

Figure 40. *Speed-up in the total execution time of the IRAM-FFT parallel solver vs. the number of processors. The tests have been performed in the Cray T3E system.*

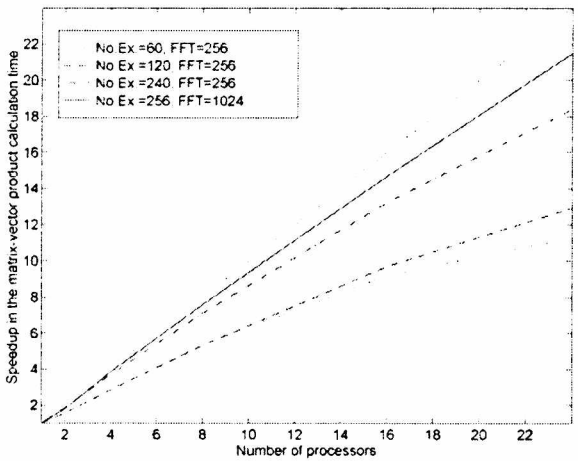

Figure 41. *Speedup in the total execution time of a single iteration of the IRAM-FFT parallel solver vs. the number of processors. The tests have been performed in the Cray T3E system.*

more than linear speedup for certain numbers of processors. If the "per iteration" speed-ups are calculated more stable parallel behaviour of the algorithm emerges, which has been shown in Figure 41. This Figure also shows that almost perfect speed-up in parallel computations is obtained in the case of FFT length = 1024 and the number of expansion function = 256.

Figures 42 and 43 show the speed-ups in the execution time of a pair of operations: a backward 2D FFT and a forward 2D FFT, as a function of the number of processors applied. The speed-ups were computed for the average time of a single operation. It may be noted from Figure 43 that although the speed-ups are high, they are lower from the total speed-up of the IRAM-FFT solver. This situation is opposite to the case of the IRAM-FD solver and can be a first signal that a parallel bottleneck may occur during parallel computation of the inner products,

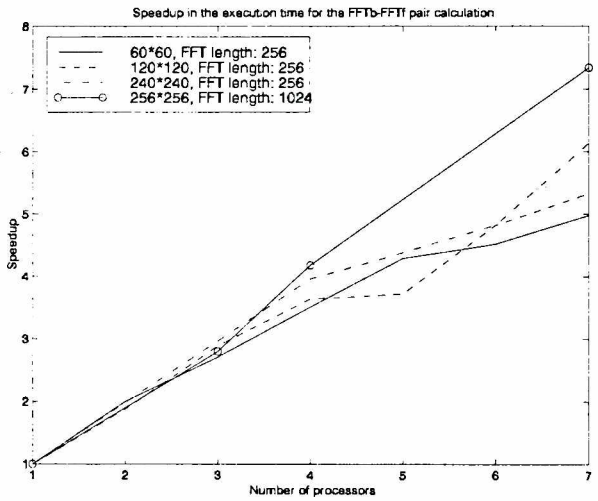

Figure 42. *Speed-up in the execution time of a pair of operations: a backward 2D FFT and a forward 2D FFT, as a function of the number of processors applied. The tests were performed in the IBM SP2 system.*

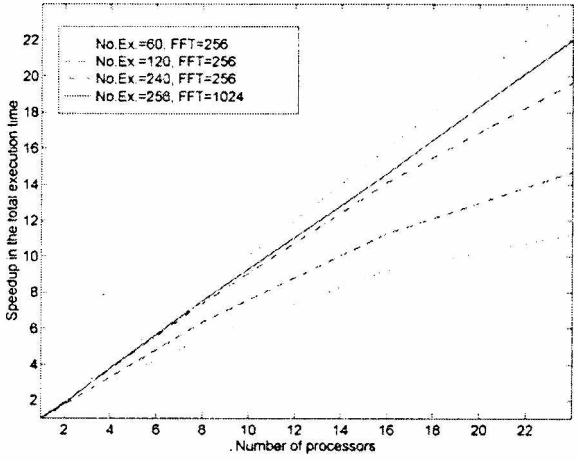

Figure 43. *Speed-up in the execution time of a pair of operations: a backward 2D FFT and a forward 2D FFT, as a function of the number of processors applied. The tests were performed in the Cray T3E system.*

involving 2D Fast Fourier Transforms, for a larger number of processors applied.

The Figure 44 shows a percentage of time spent on the orthogonalization phase during the execution of the parallel IRAM-FFT solver. The results show no dependence of the number of processors on this relative time which indicates that the time spent on inter-processor communication occurring in this procedure is entirely insignificant.

In Tables 20 and 21 the total execution times (for a single-processor execution) of the IRAM-FFT solver were shown for different number of expansion functions

Table 20. *The total execution time (for one processor) of the IRAM-FFT solver for different numbers of expansion functions used to approximate the 2D fields by the Fourier series. The tests were performed in the IBM SP2 system. (NEV = 15, NCV = 40, FFT length = 256)*

Number of expansion functions	*Size of the operator matrix*	*Total execution time [s]*
5 × 10	115	70.74
10 × 10	220	105.33
20 × 10	430	192.09
40 × 10	850	371.52
80 × 10	1690	892.69
160 × 10	3370	2216.21

Table 21. *The total execution time (for one processor) of the IRAM-FFT solver for different discretization grids (FFT lengths) used to approximate the 2D fields by the Fourier series. The tests were performed in the IBM SP2 system.*

DFT Length	*Total execution time [s]*
256 × 256	444.68
512 × 256	825.52
1024 × 256	1611.83
1024 × 512	4023.25
1024 × 1024	9235.12

Table 22. *Comparison of the execution times (on one processor) between the Galerkin Method (GM) and the IRAM-FFT solver. In case of the IRAM-FFT method: FFT length = 256, NEV = 4, NCV = 20. The tests were performed in the IBM SP2 system.*

Number of expansion functions:	*Time [s] GM:*	*Time [s] IRAM-FFT:*
10 * 10	1.61	8.85
20 * 20	304.92	19.94
30 * 30	4254.86	38.22

used and different FFT lengths. The results confirm a rather stable behaviour of the solver which shows up in the linear or linear-logarithmic type of time growth. This type of growth may be opposed to drastic time increment observed in the GM method (using the QR algorithm to find eigenvalues of the operator matrix) - compare Table 22. This last Table shows the substantial difference in performance of the classical method (the Galerkin Method) in which an explicit representation of

the input operator is applied producing a dense matrix and the proposed IRAM-FFT method which uses implicit operator representation.

Table 23. *The total execution time (for one processor) of the IEEM-FFT solver for different Discrete Fourier Transform lengths and a fixed number of expansion functions applied to represent the 2D fields (the number of expansion functions equalled 128 in every direction).*

FFT length	*Total execution time [s]*
256 × 256	0.58
512 × 256	1.28
512 × 512	2.99
1024 × 512	6.24
1024 × 1024	14.66
2048 × 1024	29.31
2048 × 2048	64.43

Table 24. *The total execution time and speed-up of the IEEM-FFT parallel solver vs. the number of processors. The tests have been performed in the Cray T3E system. (The number of expansion functions equalled 256 in every direction and the FFT length equalled 1024. The number of the solver iterations equalled 21 for the tested structure. The stopping criterion equalled 1e-06.)*

Number of PEs	*Time [s]*	*Speedup*
1	36.96	1.00
2	18.67	1.98
4	9.36	3.95
8	4.81	7.68
16	2.45	15.09
24	1.80	20.53

7.2.4 Parallel Iterative Eigenfunction Expansion Method with the FFT integration

This section shows some preliminary performance results for the solver implementing the IEEM-FFT method for parallel distributed memory systems, as described in Section 5.5. The code has been implemented in Fortran77 and MPI and has been tested in the Cray T3E system. The results of the tests are presented in Figure 45 and Table 24 showing very good performance of the parallel solver. These results of parallel IEEM-FFT solver are not surprising, as the implementation of this eigensolver is based mainly on the parallel implementation of the method of computing inner products using the 2D FFTs (cf. Section 5.4.1) which was found to scale very well. (The results showing performance of the parallel computation of a pair of a backward and forward 2D FFTs are presented in the previous section.) The other operations performed by the IEEM-FFT algorithm during its basic

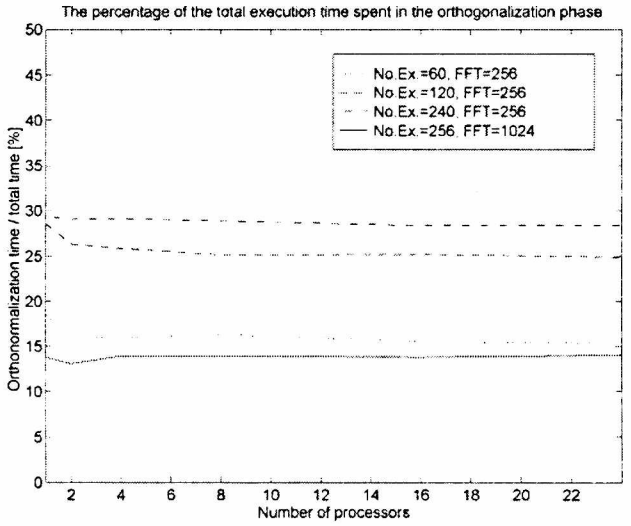

Figure 44. *The percentage of total execution time of the IRAM-FFT solver spent in the orthogonalization phase in the function of the number of processors applied. The tests were performed in the Cray T3E system.*

iteration are almost perfectly parallel with only minor inter-processor communication occurring, related to the computation of global vector norms. The other tests (involving single-processor execution), performed using the IEEM-FFT solver show the character of the execution time growth with the increasing FFT lengths — cf. Table 23. As it may be noted an almost perfect, linear growth of execution time is observed.

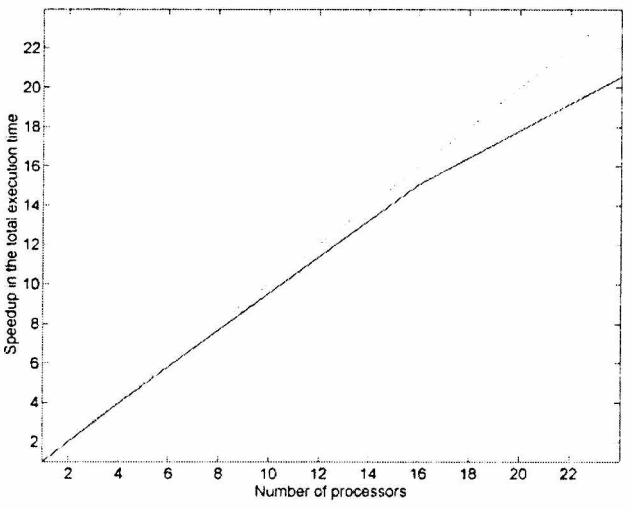

Figure 45. *Speed-up in the total execution time of the IEEM-FFT parallel solver vs. the number of processors. The tests have been performed in the Cray T3E system. (The number of expansion functions equalled 256 in every direction and the FFT length equalled 1024. The number of the solver iterations equalled 21 for the tested structure. The stopping criterion was 1e – 06.)*

Table 25. *Comparison of the number of updates, the number of matrix-vector product operations (OP*x) performed by the IRAM process and the execution times for IRAM-FFT and IRAM-FD solvers. In the case of IRAM-FFT algorithm both FFT length and the number of expansion functions used equalled 128. For IRAM-FD algorithm the discretization grid equalled 200 × 200 or 128 × 128. In all cases: NCV = 40 and NEV = number of eigenvalues to be found.*

NEV	*Problem size*	*No.of updates*	*No.of OP*x operations*	*Total time [seconds]*	*OP*x time [seconds]*
IRAM-FFT	(128×128)				
1	33024	142	2860	112.35	23.07
4	33024	168	5983	171.57	47.91
IRAM-FD	(200×200)				
1	79600	61	1240	80.27	5.56
4	79600	121	4264	191.24	19.10
IRAM-FD	(128×128)				
1	32512	34	700	18.42	1.31
4	32512	77	2688	50.02	5.03

7.2.5 Comparison of performance of the proposed eigensolvers

In this section we shall present a short comparison of execution times of the three basic algorithms discussed in this study: 1) IRAM-FFT, 2) IRAM-FD and 3) IEEM-FFT. In the comparison we investigate single-processor execution times for the considered methods, applied to solve the same problem (in physical terms). The problem consists of finding propagation constants in one of the waveguiding structures discussed in Section 6. The test parameters are as follows: 1) For all algorithms the stopping criterion equalled $1.2e-16$. 2) In the case of the IRAM-FD algorithm the 200×100 discretization grid has been used. 3) In the case of IRAM-FFT and IEEM-FFT methods the number of expansion functions equalled 40 in both x- and y- directions. The FFT lengths equalled 128, 256, 512 or 1024 in every direction. 4) In the case of IRAM-based algorithms the number of eigenfunctions to be filtered out equalled $20 - \mathrm{NCV} = 20$ and the number of eigenfunctions to be found (NEV) equalled 1, 2 or 4. With this choice of input parameters one may expect that the quality of approximations of eigenvalues computed using the discussed solvers will roughly be the same.

The graph shown in Figure 46 presents the execution times for the IRAM-FD algorithm and IRAM-FFT method (for different FFT lengths) as a function of the number of eigenvalues to be found. One may note that the IRAM-FFT algorithm is faster if the FFT length equals 128 or 256. Still, if FFT length equals 512 then the IRAM-FD algorithm appears to be twice as fast as the IRAM-FFT method. It should be stressed here that the number of update iterations of the IRAM process did not change at all with the changing FFT lengths in the IRAM-FFT algorithm.

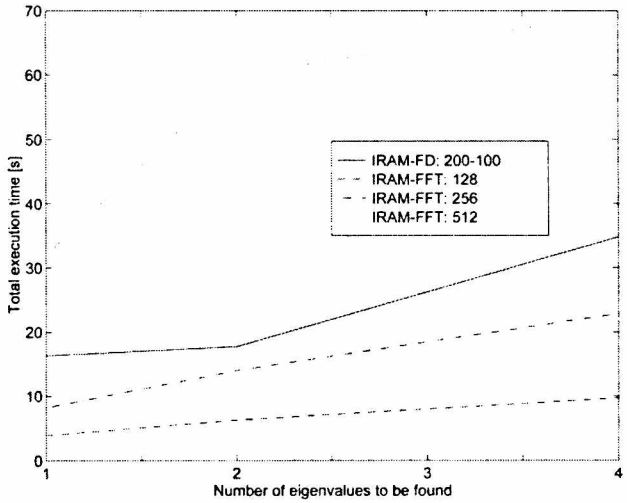

Figure 46. *Comparison of single-processor execution times of IRAM-FD and IRAM-FFT solvers for different number of eigenvalues to be found and different FFT lengths applied. The tests have been performed in the Cray T3E system.*

It means that the growth in execution time while changing the FFT length is due solely to the increasing execution time of calculating the inner products. Another interesting observation can be made about the presented results. It is apparent that the execution time grows faster for the IRAM-FD algorithm than for the FFT-based algorithm with the growing number of eigenvalues to be found. It is not known whether this tendency is stable or for what range of parameters it occurs, as the computational complexity of the IRAM-FD solver is lower than the cost of FFT-

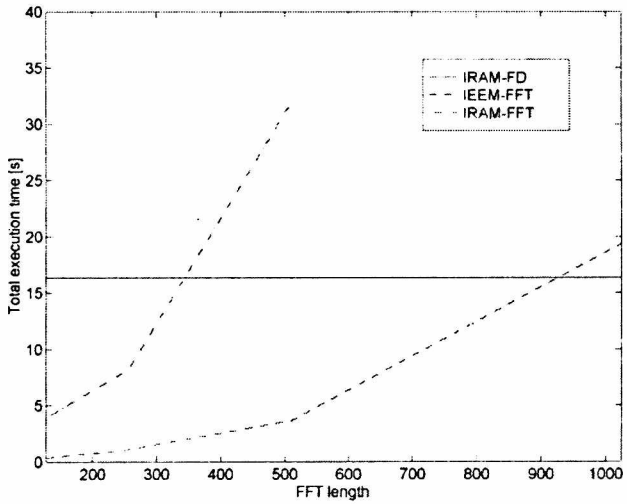

Figure 47. *Comparison of single-processor execution times for the IRAM-FD, IRAM-FFT and IEEM-FFT methods. The number of eigenvalues to be found equalled 1 for all the methods. The times for the FFT-based algorithms are given as a function of the FFT length. The tests have been performed in the Cray T3E system.*

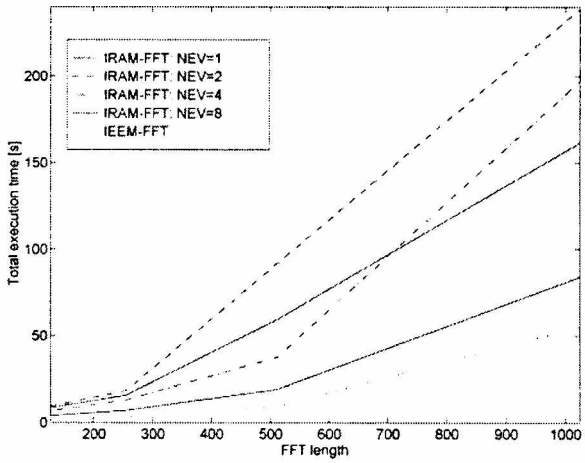

Figure 48. *Comparison of single-processor execution times for the IRAM-FFT and IEEM-FFT methods. The number of expansion terms used to represent functions equalled 40 in each spatial direction. The convergence criterion equalled 1.2e – 16. In the case of IRAM-FFT algorithm the number of eigenvalues to be found was 1, 2, 4 and 8, while for the IEEM-FFT method only a single eigenvalue was found. The tests have been performed in the Cray T3E system.*

based algorithms. Still, due to smaller size of the problem solved in the IRAM-FFT method (as compared to the IRAM-FD method) the growth of the number of iterations of the IRAM process necessary to obtain convergence is not so dynamic as in the IRAM-FD method and compensates the higher complexity of the IRAM-FFT algorithm. Table 25 shows a comparison of the number of update iterations and OP*x operations for the IRAM-FFT and IRAM-FD algorithms if one and four eigenvalues are to be computed. One may note that while for the IRAM-FFT algorithm the number of update iterations increases by less than 20 % (for NEV = 4), it doubles for the IRAM-FD algorithm. The increment in the number of matrix-vector products (inner products) to be computed also changes more rapidly for the IRAM-FD algorithm. Consequently the growth in execution time is also faster for the IRAM-FD as compared to IRAM-FFT solver. Lastly, it should be noted that the time spent on calculating the matrix-vector product does not exceed 10 % of the total execution time of the IRAM-FD solver, while for the IRAM-FFT solver this percentage may range from about 20 % to more than 90 %. This fact is a consequence of transferring the complexity of the solver from the IRAM iterative process to the operation of computing inner products (using 2D FFTs).

Figure 47 shows a comparison of single-processor execution times for three solvers: 1) IRAM-FD, 2) IRAM-FFT, 3) IEEM-FFT if only one eigenvalue is to be found. The execution time for the IRAM-FD algorithm has been drawn as a horizontal line, since the algorithm does not depend on the FFT length. The times for the FFT-based method have been given for different FFT lengths applied. From the graph one may note that the IRAM-FFT algorithm is faster than IRAM-FD

method for FFT length that equals 128 or 256. If the FFT length is smaller than 1024 then the execution time for the IEEM-FFT is significantly shorter than for two other algorithms. It is also interesting to note that the growth of execution time is much faster for the IRAM-FFT method than for the IEEM-FFT method, although both method are based on the same representation of the operators and the same method of calculating inner products. This effect is due to a different number of iterations needed to obtain convergence, which equals about 700 for the IRAM-FFT algorithm and about 100 for the IEEM-FFT algorithm. (The numbers of iterations stay approximately the same for different FFT lengths.) Another comparison of IRAM-FFT and IEEM-FFT solvers is shown in Figure 48. This Figure presents execution times while solving a different eigenproblem than in the previous tests. In this case, if only a single eigenvalue is to be found, the number of iterations equals 620-640 for the IRAM-FFT and 382-453 for the IEEM-FFT method. Consequently the execution times for IRAM-FFT and IEEM-FFT are comparable also for larger FFT lengths. If a larger number of eigenvalues is to be found, the number of iterations increases up to 1800 iterations for the IRAM-FFT solver which results in an appropriate growth in the total execution time.

8. Conclusions

In the conclusion we would like to make a general comparison of various features of the numerical algorithms proposed for solving operator eigenproblems in distributed memory parallel systems. We will focus on the following issues:

— The numerical complexity, memory storage requirements and the size of messages exchanged between the processors for the proposed parallel eigensolvers.

— Performance of the solvers in parallel distributed memory systems.

— Assessment of chosen general properties and functional parameters of the solvers,

Table 26. *The assessment of numerical complexity, memory storage requirements and the size of messages exchanged among the processors for the discussed parallel eigensolvers.*

Algorithm	*Computational complexity*	*Memory complexity*	*Size of the messages*
IRAM-FD	$O(k^2N/P)+$ $O(k\cdot nnz/P)$	$O(k^2N/P)+$ $O(2nnz/P+N/P)$	$O(k(P-1)b$
IRAM-FFT	$O(k^2N/P)+$ $O(k/P\cdot K\log K)$	$O(2K/P+2N/P)+$ $O(6\sqrt{K}+k^2N/P)$	$O(k\frac{P-1}{P}\sqrt{KN})$
IRAM-FFT-NI	$O(k^2N/P)+$ $O(k/P\cdot K\log K)+$ $O(kN\sqrt{K}/P)$	$O(2K/P+2N/P)+$ $O(6\sqrt{K}+k^2N/P)$	$O(k\frac{P-1}{P}\sqrt{KN})$
IEEM-FFT	$O(K/P\log K)+$ $O(12K/P)$	$O(2K/P+2N/P)+$ $O(6\sqrt{K})$	$O(k\frac{P-1}{P}\sqrt{KN})$

including e.g. the scope of eigenproblems which may be solved by a certain algorithm or the portability of the implementation.

— Applicability of the discussed algorithms to solving eigenproblems arising in electromagnetics.

The comparison of the complexities of the parallel solvers has been shown in Table 26. This Table presents numerical cost estimations for the following algorithms: 1) The IRAM-FD: The solver based on the IRAM iterative algorithm and applying the FD finite-dimensional mapping technique in order to obtain discrete operator representation; 2) The IRAM-FFT: The solver based on the IRAM iterative algorithm and applying implicit finite-dimensional projection of the input operator; 3) The IRAM-FFT-NI: The modification of the previous solver which applies a hybrid algorithm of calculating the matrix-vector product using FFT and Numerical Integration (NI); 4) The IEEM-FFT: The solver implementing the Iterative Eigenfunction Expansion Method using implicit projection of the input operator with FFT-based calculation of inner products in parallel distributed memory environment.

The following symbols have been applied in Table 26: P — the number of processors, N — global algebraic size of the problem, K — product of the lengths of Discrete Fourier Transforms in the x- and y- spatial dimensions, determining the grid size in the DFT (FFT) domain (usually $N << K$, although it may happen that $N = K$), nnz — the number of non-zero elements in the input operator matrix, k — number of eigenvalues to be found and b — bandwidth of the operator matrix.

Comparing the results shown in Table 26 the following conclusions may be drawn:

— Assuming that $k = 1$, it may be found that the IRAM-FD algorithm has the lowest, linear numerical complexity. The algorithms involving implicit (DFT-based) operator projection methods have at least a linear-logarithmic complexity. Potentially the highest numerical cost occurs for the IRAM-FFI-NI algorithm ($O(N^{3/2})$) which may result in a deterioration of performance for larger problem sizes.

— Although, as mentioned in the previous item, the IRAM-FFT and the IEEM-FFT solvers have generally higher computational complexities than the IRAM-FD solver, in many applications the resulting problem size is much smaller for the former methods than for the latter one. (In a typical situation one has $K = N_{FD}$ and $N_{FFT} = N_{FD}/25$.) Consequently in these cases the FFT-based methods are faster, while offering an equivalent quality of solutions.

— Referring to the memory cost it has to be noted that generally the storage requirements of the FFT-based methods are lower than the requirements of the FD-based solvers. For instance, typically $nnz = 5N$. In this case the IRAM-FD solver uses almost three times more memory than the IEEM-FFT algorithm. This comparison may even be more favourable for the FFT-based solvers if one keeps in mind that the problem size is usually significantly larger for the IRAM-FD method.

— Another important feature of parallel solvers is the size of messages exchanged by the processors during execution of the program in a parallel environment. This size is the lowest for the IRAM-FD solver if the bandwidth of the input operator matrix is considerably smaller than its size. On the other hand if the matrix is not banded the message size may increase very significantly. It may also be noted that in the case of the IRAM-FD algorithm the size of communicated messages is a function of the number of processors, while in the case of DFT-based methods this size is virtually independent of P. Still, the problem with the DFT-based algorithms is that due to relatively large size of communicated messages a parallel bottleneck is expected to occur for larger problem sizes.

Turning to the issues concerning serial and parallel performance of the solvers the following concluding remarks can be made:

— Due to relatively low numerical complexities of all the discussed solvers (estimated for a single iteration or a single p-step update of the algorithm) the execution time does not "blow up" with the increasing problem size (for a considerably large range of problem sizes).

— Still, it was found that in the case of the algorithms based on IRAM the number of updates may increase significantly for the increased problem size or the number of eigenvalues to be found and / or filtered-out during the execution of the algorithm.

— Referring to parallel performance, it has to be concluded that all the developed parallel solvers offer high efficiency and speed-up in scalable distributed memory systems.

— Although for large problem sizes the speed-up is higher for FFT-based solvers than for the FD-based algorithms, these methods require generally larger problem sizes to achieve high efficiency in a parallel environment.

The following general functional features of the presented parallel solvers may be outlined:

— The solvers based on the Implicitly Restarted Arnoldi Method (IRAM) enable one to find several eigenvalues from the desired part of the operator spectrum. This is a substantial advantage over the IEEM-FFT method which, in its basic version, allows one to find only a single eigenvalue.

— Investigating the properties of the IRAM-FD algorithm it may be concluded that the algorithm is best suited for banded, sparse matrix operators obtained from discretization of differential operators. In this case the solver may offer extremely high performance.

— Referring to the solvers using implicit discrete operator representation (IRAM-FFT, IEEM-FFT) it may be stated that their efficiency is directly related to the reduction of the emerging problem size. Consequently, they may be most

efficiently applied if the resulting low-cost representation offers an acceptable approximation of the considered input operator.

— Problems with the DFT-based solvers may arise if the input operator acting on the elements from its domain produces discontinuous, highly varying functions or distributions. This problem may be solved at the cost of increasing the numerical complexity of the algorithm (cf. the IRAM-FFT-NI method).

— A crucial functional feature of all the presented solvers is their portability which allows one to use efficiently the developed parallel methods in a variety of parallel distributed memory systems, including supercomputer facilities and network environments supporting message-passing programming model.

— The further advantage of the discussed eigensolvers which greatly extends their applicability to solving large-scale eigenvalue problems is relatively low memory complexity as compared to many classical methods, as well as balanced storage requirements across the processors.

Lastly, let us mention some characteristics of the discussed algorithms being of particular importance in the electromagnetic applications discussed in this study:

— The IRAM-FD algorithm has yielded an efficient tool while dealing with eigenproblems of non-symmetric differential operators arising in electromagnetics. In the current version of the solver it may be used to solve eigenproblems for waveguiding structures with discontinuous, rectangular permittivity profiles.

— On the other hand, the IRAM-FFT and IEEM-FFT solvers are to be particularly useful while dealing with waveguides with arbitrary continuous permittivity profiles.

— The scope of application of the FFT-based algorithms can be extended to structures with discontinuous permittivity profiles at the cost of increasing computational complexity of the algorithm.

Acknowledgments

First of all I wish to thank Professor Michał Mrozowski for his support in my first steps in the fields of scientific computing and computational electromagnetics, numerous valuable discussions and constant encouragement which enabled me to obtain the results presented in this study.

I am grateful to Jacek Mielewski for providing sequential code of the Finite Difference solver which has served as a basis for parallel implementation of the IRAM-FD method.

I also acknowledge the support of the Academic Computer Centre TASK in Gdansk and the Interdisciplinary Centre for Mathematical and Computational Modelling of the University of Warsaw in facilitating the access to supercomputer systems which served as platforms for all the numerical tests presented in this work.

The research was supported by the Polish State Committee for Scientific Research under contract 8 T11D 01911.

Appendix A

Matrix formulation of the IEEM

This section presents a new approach towards the Iterative Eigenfunction Expansion Method, suitable for solving eigenproblems of finite-dimensional linear operators. In this case the matrix formulation may be used. If $\underline{\underline{T}}$ $\left(\underline{\underline{T}} \in M(C)_{n\times n}\right)$ denotes the matrix of the finite-dimensional linear operator **T** then, analogously as in Section 2.1, the following decomposition can be made:

$$\underline{\underline{T}} = \underline{\underline{L}} - \underline{\underline{F}} \tag{73}$$

The eigenvalues of the matrix $\underline{\underline{L}}$ are assumed to be known and will be denoted as $\{\Lambda_i\}_1^n$. We do not assume here that there are exactly n different eigenvalues. For some i and j there may be $\Lambda_i = \Lambda_j$. In order to be able to appropriately represent every vector from the space C^n the eigenvectors of the matrix $\underline{\underline{L}}$: $\{\underline{\tilde{h}}_i\}_1^n$ have to be linearly independent. Consequently, the condition for the matrix $\underline{\underline{L}}$ is that it should be similar to some diagonal matrix or, in other words, to have a simple structure. If this is the case, the set of its eigenvectors $\{\underline{\tilde{h}}_i\}_1^n$ may be orthonormalized and in this way an orthonormal basis $\{\underline{h}_i\}_1^n$ in the C^n space is obtained. Initially the representation of the linear operator **F** is the matrix $\underline{\underline{F}}$, describing the linear transformation for the standard canonical basis in the C^n space. This matrix may also have a representation in the basis of the orthonormal eigenvectors $\{\underline{h}_i\}_1^n$ of the matrix $\underline{\underline{L}}$. By applying the similarity transformation:

$$\underline{\underline{\tilde{F}}} = \underline{\underline{H}}^{-1}\,\underline{\underline{F}}\,\underline{\underline{H}} \tag{74}$$

where the matrix $\underline{\underline{H}}$ is an orthonormal (unitary) matrix, whose columns are the eigenvectors $\underline{h}_i$, one obtains the matrix of the operator **F** in the basis $\{\underline{h}_i\}_1^n$. It is also obvious that:

$$\underline{\underline{D}} = diag\{\Lambda_1,\ldots,\Lambda_n\} = \underline{\underline{H}}^{-1}\,\underline{\underline{L}}\,\underline{\underline{H}} \tag{75}$$

If one denotes as $\underline{\tilde{v}}^{(k)}$ the k-th approximation of the eigenvector of the matrix $\underline{\underline{T}}$, represented in the standard canonical basis and as $\underline{v}^{(k)} = \underline{\underline{H}}^{-1}\underline{\tilde{v}}^{(k)}$ the same

vector represented in the $\{\underline{h}_i\}_1^n$ basis, then the steps of a single iteration of the IEEM method in the matrix formulation may be described as follows:

ALGORITHM 7: *IEEM-matrix.*

STEP 1: Compute the matrix-vector product $\underline{\underline{\tilde{F}}}\underline{v}^{(k)}$.

STEP 2: Determine the (k+1)-th approximation of the eigenvector $\underline{v}^{(k+1)} = [v_1^{(k+1)},\ldots,v_n^{(k+1)}]$:

$$\underline{v}_i^{(k+1)} = \frac{\left(\underline{\underline{\tilde{F}}}\underline{v}^{(k)}\right)_i}{\Lambda_i - \lambda^{(k)}} \tag{76}$$

where $\lambda^{(k)}$ is the k-th approximation of an eigenvalue of the matrix $\underline{\underline{T}}$.

STEP 3: Normalize vector $\underline{v}^{(k+1)}$:

$$\underline{w}^{(k+1)} = \frac{\underline{v}^{k+1}}{\underline{v}^{k+1}} \tag{77}$$

STEP 4: Determine the (k+1)-th approximation of the eigenvalue:

$$\lambda^{(k+1)} = \sum_{i=1}^{n} \Lambda_i \left(w_i^{(k+1)}\right)^2 - \left(\underline{\underline{F}}\underline{v}^{(k)}\right)^H \underline{w}^{(k+1)} \tag{78}$$

The question that appears is under what conditions the above method converges to the solution, i.e. the eigenvalue and the eigenvector of the matrix $\underline{\underline{T}}$. For the operator version of the algorithm (described in Section 2.4) Jabłoński proved that in a Hilbert space being the domain of the operator $\mathbf{T} = \mathbf{L} - \mathbf{F}$, the iterative process converges provided the operator $\mathbf{L}$ is relatively compact with the operator $\mathbf{F}$ (cf. [16]). If the finite-dimensional Hilbert space l_n^2 (the linear space of n-dimensional vectors with the Euclidean norm and a standard inner product) is considered then for any two matrix operators $\underline{\underline{L}}$ and $\underline{\underline{F}}$ these two operators are relatively compact, as according to one of the definitions of the operator compactness, any finite-dimensional operator is compact. Consequently, the matrix operator $\underline{\underline{F}}\left(\lambda \underline{\underline{I}} - \underline{\underline{L}}\right)^{-1}$ (for $\lambda \in (C^n - \sigma_P(\underline{\underline{L}}))$) is also compact., it may be inferred that for any decomposition $\underline{\underline{T}} = \underline{\underline{L}} - \underline{\underline{F}}$, such that matrix $\underline{\underline{L}}$ has a simple structure, the iterative method converges to the solution.

Another question which immediately emerges is which solution (i.e. which eigenvalue from the matrix spectrum) is being found in the iterative process. In the

simplest case the matrix $\underline{\underline{L}}$ is similar to a diagonal matrix which has a single n-fold eigenvalue Λ. In this case for all i one has $\Lambda_i = \Lambda$ in the equation (76) and it may be clearly seen from the same equation that the vector $\underline{v}^{(k+1)}$ approaches the direction of an eigenvector corresponding to the dominant eigenvalue of the matrix $\underline{\underline{F}}$. (In this case the IEEM reduces to the Power Method and the subsequent vectors v_k are constructed in a simple power iteration for the matrix $\underline{\underline{F}}$.) If $\tilde{\lambda}$ denotes the dominant eigenvalue (the eigenvalue with the largest modulus) of the matrix $\underline{\underline{F}}$ then the method converges to $\Lambda - \tilde{\lambda}$. Unfortunately no results have been obtained so far for different choices of the matrix $\underline{\underline{L}}$, although the relations between the Iterative Eigenfunction Expansion Method and the Power Method seem to be apparent.

Symbol conventions and abbreviations

General symbols

$\mathbf{A}$	—	linear operator
$\mathbf{A}^*$	—	adjoint operator associated with $\mathbf{A}$
$\underline{\underline{A}}$	—	matrix
$\underline{a}$	—	vector
$\mathbf{B}(X,X)$	—	space of linear operators $\{\mathbf{A} \mid \mathbf{A} : X \to X\}$
C	—	the set of complex numbers
C^1	—	the class of functions with continuous derivatives
C^2	—	the class of functions from C^1 class with continuous second derivatives
$\delta(x)$	—	the Dirac delta distribution
$h(x)$	—	the Heaviside function
$L_2(\Omega)$	—	space of square integrable functions defined over the region Ω
$M(C)_{n\times n}$	—	the set of $n \times n$ matrices with complex elements
R	—	the set of real numbers
$\sigma_p(\mathbf{A})$	—	point spectrum of the operator $\mathbf{A}$
v, w	—	functions
X	—	complete, linear (Banach) space
$(\cdot, \cdot)$	—	inner product in a Hilbert space
$\|\cdot\|$	—	norm in a Hilbert space induced by the inner product

Physical quantities

β	—	propagation constant
ε	—	relative permittivity of medium
ε_0	—	permittivity of the free space
$\vec{E}_t$	—	transverse electric field intensity

f — frequency

$\vec{H}_t$ — transverse magnetic field intensity

k_0 — wavenumber in the free space

μ_0 — permeability of the free space

Z — normalized propagation constant

Selected abbreviations

DFT — Discrete Fourier Transform
FD — Finite Difference discretization method
FEM — Finite Element Method
FFT — Fast Fourier Transform
GM — Galerkin Method
IEEM — Iterative Eigenfunction Expansion Method
IRAM — Implicitly Restarted Arnoldi Method
MGS — Modified Gram-Schmidt orthonormalization algorithm ARPACK
P_ARPACK — Parallel ARnoldi PACKage
TRM — Transverse Resonance Method

References

[1] Mrozowski M., *Guided Electromagnetic Waves, Properties and Analysis*, Research Studies Press, Taunton, Somerset, England, 1997

[2] Shestopalov V. P., Shestopalov Y. V., *Spectral theory and excitation of open structures*, IEE, London, UK, 1996

[3] Dautray R., Lions J.-L., *Mathematical Analysis and Numerical Methods for Science and Technology - Functional and Variational Methods*, vol. 2, Springer-Verlag, Berlin, 1990

[4] Dębicki P., Jędrzejewski P., Kręczkowski A., Mielewski J., Mrozowski M., Nyka K., Przybyszewski P., Rewieński M., Rutkowski T., *Coping with numerical complexity in computational electromagnetics*, Int. Microwave Symp. MIKON-98, Cracow, 1998

[5] Przybyszewski P., Mielewski J., Mrozowski M., *Efficient Eigenfunction Expansion Algorithms for Analysis of Waveguides*, Technical Report No. 88/96, Dept. of Electronics, Telecommunications and Computer Science, Technical University of Gdansk, Gdansk, 1996

[6] Golub G. H., van Loan C. F., *Matrix Computations*, The John Hopkins University Press, Baltimore, 1996

[7] van der Vorst H. A., Golub G. H., *150 Years and still alive: eigenproblems*, Technical Report SCCM96-11, Dept. of Scientific Computing and Computational Mathematics, Stanford University, USA, 1996

[8] Davidson E. R., *The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices*, J. Comp. Phys., 17:87-94, 1975

[9] Saad Y., *Numerical methods for large eigenvalue problems*, Manchester University Press, Manchester, UK, 1992

[10] Arnoldi W. E., *The principle of minimized iterations in the solution of the matrix eigenvalue problem*, Quart. Appl. Math. 9, 17-29, 1951

[11] Sorensen D. C., *Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations*, Proceedings of an ICASE/LaRC, May 23-25 1994, Hampton, VA, D. E. Keyes, A. Sameh and V., eds., Kluwer, 1995

[12] Hebermehl G., Schlundt R., Zscheile H., Heinrich W., *Eigen mode solver for microwave transmission lines*, Weierstrass Institute for Applied Analysis and Stochastics, Preprint No. 308, Berlin, 1997

[13] Dębicki M. P., Jędrzejewski P., Mielewski J., Przybyszewski P., Mrozowski M., *Application of the Arnoldi Method to the Solution of Electromagnetic Eigenproblems on the Multiprocessor Power Challenge Architecture*, Technical Report No. 19/95, Dept. of Electronics, Technical Univ. of Gdansk, Gdansk, 1995

[14] Sorensen D. C., *Implicit application of polynomial filters in a k-step Arnoldi method*, Technical Report TR90-27, Rice University, Dept. of Math. Sci., Houston, TX, 1990

[15] Lehoucq R. B., Sorensen D. C., Yang C., *ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Method*, available from: ftp.caam.rice.edu

[16] Jabłoński T., *Iterative Eigenfunction Expansion Method for Cylindrical Fibers*, IFTR Reports, 3/1986, Warsaw, 1986. (in Polish)

[17] Jabłoński T., Sowiński M., *Analysis of dielectric guiding structures by the iterative eigenfunction expansion method*, IEEE Trans. Microwave Theory Tech. vol. 37, pp.63-70, Jan. 1989

[18] Mrozowski M., *IEEM FFT — A Fast and Efficient Tool for Rigorous Computations of Propagation Constants and Field Distributions in Dielectric Guides with Arbitrary Cross-Section and Permittivity Profiles*, IEEE Trans. Microwave Theory Tech. vol. 39, pp. 323-329, Feb. 1991

[19] Byron F. W., Fuller R. W., *Mathematics of classical and quantum physics*, Addison-Wesley, Reading (Polish edition: PWN, Warsaw, 1975)

[20] Auslander L., Tsao A., *On a divide and conquer algorithm for the eigenproblem via complementary invariant subspace decomposition*, Supercomputing Research Center Technical Report SRC-89-003, Bowie, USA, 1989

[21] Auslander L., Tsao A., *On parallelizable eigensolvers*, Advances in Applied Mathematics 13, 253-261, 1992

[22] Wilkinson J. H., Reinsch C., editors, *Handbook for Automatic Computation, Vol. 2, Linear Algebra*, Springer Verlag, Heidelberg - Berlin - New York, 1971

[23] Fernandez F., Lu Y., *Microwave and Optical Waveguide Analysis by the Finite Element Method*, Research Studies Press, Taunton, Somerset, England, 1996

[24] Dautray R., Lions J.-L., *Mathematical Analysis and Numerical Methods for Science and Technology — Integral Equation and Numerical Methods*, vol. 4, Springer-Verlag, Berlin, 1990

[25] Jones D., *Methods in Electromagnetic Wave Propagation, vol. 1: Theory and Guided Waves*, Clarendon Press, Oxford, 1987

[26] Mrozowski M., *Eigenfunction expansion techniques in the numerical analysis of inhomogeneously loaded waveguides and resonators*, Zeszyty Naukowe Politechniki Gdanskiej, Elektronika, No. 81, 1994

[27] Briggs W., *Multigrid tutorial*, SIAM, Philadelphia, 1987

[28] Nyka K., Mrozowski M., *Combining function expansion and multigrid method for efficient analysis of MMICs*, Int. Microwave Symp. MIKON-96, pp. 203-207, Warsaw, 1996

[29] Foster I., *Designing and Building Parallel Programs*, Addison-Wesley, Reading, 1995

[30] Minty E., Davey R., Simpson A., Henty D., *Decomposing the potentially parallel*, Course Notes, Edinburgh Parallel Computing Centre, The University of Edinburgh, 1996, available: http://www.epcc.ed.ac.uk

[31] Saad Y., *SPARSKIT: a basic tool kit for sparse matrix computations. Version 2*, CSRD — University of Illinois and RIACS (NASA Army Research Center), 1994, available: ftp.cs.umn.edu/dept/sparse

[32] Cooley J. W., Tukey J. W., *An algorithm for the machine evaluation of complex Fourier series*, Math. Comp. 19, pp. 297-301, 1965

[33] Briggs W. L., Van Emden Henson, *The DFT. An Owner's Manual for the Discrete Fourier Transform*, SIAM, Philadelphia, 1995

[34] Winograd S., *On computing the discrete Fourier transform*, Math. Comp., 32, pp. 175-199, 1978

[35] Press W. H., Flannery B. P., Teukolsky S. A., Vetterling W. T., *Numerical recipes: The art of scientific computing*, Cambridge University Press, Cambridge, 1986

[36] Richardson H., *High Performance Fortran — History, Overview and Current Status*, Edinburgh Parallel Computing Centre, The University of Edinburgh, 1995, available: `http://www.epcc.ed.ac.uk/epcc-tec/documents.html`

[37] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Scientific Programming 2(1-2), pp. 1-170, 1993

[38] Gupta M., Midkiff S., Schonberg E., Seshadri V., Shields D., Ko-Yang Wang, Wai-Mee Ching., Ngo T., *An HPF Compiler for the IBM SP2*, High Performance Computing in Europe on IBM Platforms, Conference Proceedings, Academic Computer Centre CYFRONET, Cracow, 1996

[39] Message Passing Interface Forum, MPI: *A Message-Passing Interface Standard*, International Journal of Supercomputer Applications and High Performance Computing, 8(3/4), 1994

[40] Gropp W. D., Lusk E., Skjellum A., *Using MPI — Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994

[41] Sunderam V. S., Geist G. A., Dongarra J. J., Manchek R., *The PVM Concurrent Computing System*, Parallel Computing 20(4), pp. 531-45, 1994

[42] Geist A., Beguelin A., Dongarra J. J., Jiang W., Manchek R., Sunderam V., *PVM 3. A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994

[43] Anderson E., Brooks J., Grassl C., Scott S., *Performance of the CRAY T3E Multiprocessor*, Technical Report, Cray Research, 1997, available from: `http://www.cray.com/products/systems/crayt3e`

[44] *IBM PVMe for AIX, Users Guide and Subroutine Reference, Version 2, Release 1*, document number GC23-3884-00, IBM Corporation, 1995

[45] Geist G. A., *Advanced Capabilities in PVM 3.4*, Lecture Notes in Computer Science 1332, pp. 107-115, 1997

[46] MPI-2 comittee, MPI: *A message-passing interface standard*, 1997, available from: `http://www.mcs.anl.gov/Projects/mP/standard.html`

[47] *A User's Guide to the Basic Linear Algebra Communication Subprograms (BLACS) v. 1.1*, available: `ftp.netlib.org`

[48] *Scientific Libraries Reference Manual*, Cray Research, SR-2081

[49] *IBM Parallel Engineering and Scientific Subroutine Library Release 2 Guide and Reference*, IBM Corporation, GC23-3836, 1996

[50] Krommer A., McDonald K., *The NAG Numerical PVM Library*, High Performance Computing in Europe on IBM Platforms, Conference Proceedings, Academic Computer Centre CYFRONET, Cracow, 1996

[51] Dias da Cunha R., Hopkins T., *PM 1.1 — the Parallel Iterative Methods Package for Systems of Linear Equations. Users's Guide*, Technical Report, Computing Laboratory, University of Kent, 1994

[52] Malard J., Richardson H., *An Introduction to Parallel Numerical Libraries*, Technology Watch Report, Edinburgh Parallel Computing Centre, Edinburgh University, 1996, available from: `http://www.epcc.ed.ac.uk/epcc-tec/documents.html`

[53] Maschhoff K. J., Sorensen D. C., *P_ARPACK: An Efficient Portable Large Scale Eigenvalue Package for Distributed Memory Parallel Architectures*, Rice University, 1996, available at: `ftp.caam.rice.edu`

[54] Allan R. J., Guest M. F. (Eds.), *Parallel Application Software on High Performance Computers, I. The IBM SP2 and Cray T3D*, technical report, the CCLRC HPCI Centre at Daresbury Laboratory, Daresbury, 1996

[55] Klepacki D., *Application Performance and Benchmark Experience on the IBM SP using MPI, HPF and Virtual Shared Memory*, High Performance Computing in Europe on IBM Platforms, Conference Proceedings, Academic Computer Centre CYFRONET, Cracow, 1996

[56] Katz D. S., Cwik T., Kwan B. H., Lou J. Z., Springer P. L., Sterling T. L., Wang P., *An Assessment of a Beowulf System for a Wide Class of Analysis and Design Software*, paper presented at 4th NASA Symposium on Large-Scale Analysis and Design on High-Performance Computers and Workstations, to appear in Advances in Engineering Software, April 1998

[57] Swarztrauber P. N., *FFTPACK, version 4, A package of fortran subprograms for the Fast Fourier Transform of periodic and other symmetric sequences.*, 1985, available from: `http://www.netlib.org`

[58] Rewieński M., *Implementation of the Parallel Arnoldi Method in the IBM SP2 Distributed Memory System*, Technical Report No. 89/96, Faculty of Electronics, Telecommunications and Informatics, Technical University of Gdansk, Gdansk, 1996

[59] Lehoucq R. B., Sorensen D. C., Yang C., *ARPACK USERS GUIDE: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*, 1996, available from: `ftp://ftp.caam.rice.edu`

[60] Allan R. J., Bush I. J., Henty D., Bush T., *Serial and Parallel FFT Routines*, technical report, the CCLRC HPCI Centre at Daresbury Laboratory, Daresbury, 1996