On a Parallel Multilevel Preconditioned Maxwell Eigensolver

Report

Author(s):
Arbenz, Peter; Bečka, Martin; Geus, Roman; Hetmaniuk, Ulrich; Mengotti, Tiziano

Publication date:
2004

Permanent link:
https://doi.org/10.3929/ethz-a-006775556

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
Technical report 465
On a Parallel Multilevel Preconditioned Maxwell Eigensolver

Peter Arbenz\textsuperscript{a,\dagger}, Martin Bečka\textsuperscript{a,1}, Roman Geus\textsuperscript{b},
Ulrich Hetmaniuk\textsuperscript{c,1}, Tiziano Mengotti\textsuperscript{a,1}

\textsuperscript{a}Institute of Computational Science, Swiss Federal Institute of Technology, CH-8092 Zurich, Switzerland
\textsuperscript{b}Paul Scherrer Institut, CH-5232 Villigen PSI, Switzerland
\textsuperscript{c}Sandia National Laboratories, Albuquerque, NM 87185-1110, U.S.A.\textsuperscript{2}

Abstract

We report on a parallel implementation of the Jacobi–Davidson algorithm to compute a few eigenvalues and corresponding eigenvectors of a large real symmetric generalized matrix eigenvalue problem

\[ Ax = \lambda Mx, \quad C^T x = 0. \]

The eigenvalue problem stems from the design of cavities of particle accelerators. It is obtained by the finite element discretization of the time-harmonic Maxwell equation in weak form by a combination of Nédélec (edge) and Lagrange (node) elements.

We found the Jacobi–Davidson (JD) method to be a very effective solver provided that a good preconditioner is available for the correction equations that have to be solved in each step of the JD iteration. The preconditioner of our choice is a combination of a hierarchical basis preconditioner and the ML smoothed aggregation AMG preconditioner. It is close to optimal regarding iteration count.

The parallel code makes extensive use of the Trilinos software framework. In our examples from accelerator physics we observe satisfactory speedups and efficiencies.

Key words: Maxwell equation, finite element method, generalized eigenvalue problem, Jacobi–Davidson algorithm, smoothed aggregation AMG preconditioning

\dagger Corresponding author.

Email address: arbenz@inf.ethz.ch (Peter Arbenz).

1 The work of these authors has been supported by the ETH research grant TH-1/02-4 on “Large Scale Eigenvalue Problems in Opto-Electronic Semiconductor Lasers and Accelerator Cavities”.

2 Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed

Preprint submitted to Parallel Computing 16 December 2004
1 Introduction

Many applications in electromagnetics require the computation of some of the eigenpairs of the curl-curl operator,

\[
\text{curl } \mu_r^{-1} \text{curl } e(x) - k_0^2 \varepsilon_r e(x) = 0, \quad \text{div } e(x) = 0, \quad x \in \Omega,
\]

in a bounded simply-connected, three-dimensional domain \( \Omega \) with homogeneous boundary conditions \( e \times n = 0 \) posed on the connected boundary \( \partial \Omega \). \( \varepsilon_r \) and \( \mu_r \) are the relative permittivity and permeability. Equations (1.1) are obtained from the Maxwell equations after separation of the time and space variables and after elimination of the magnetic field intensity. While \( \varepsilon_r \) and \( \mu_r \) are complex numbers in problems from waveguide or laser design [8], in simulations of accelerator cavities the materials can be assumed to be loss-free, thus admitting real \( \varepsilon_r \) and \( \mu_r \), whence all eigenvalues are real. Here, we will assume \( \varepsilon_r = \mu_r = 1 \). Thus, the discretization of (1.1) by finite elements leads to a real symmetric generalized matrix eigenvalue problem

\[
Ax = \lambda Mx, \quad C^T x = 0,
\]

where \( A \) is positive semidefinite and \( M \) is positive definite. In order to avoid spurious modes we approximate the electric field \( e \) by Nédélec (or edge) elements [20]. The Lagrange multiplier (a function) introduced to treat properly the divergence free condition is approximated by Lagrange (or nodal) finite elements [3].

In this paper we consider a parallel eigensolver for computing a few of the smallest eigenvalues and corresponding eigenvectors of (1.2) as efficiently as possible with regard to execution time and memory cost. In earlier studies [3] we found the Jacobi–Davidson algorithm [21, 12] a very effective solver for this task. We have parallelized this solver in the framework of the Trilinos parallel solver environment [13].

In section 2 we review the symmetric Jacobi-Davidson eigensolver and the preconditioner that is needed for its efficient application. In section 3 we discuss data distribution and issues involving the use of Trilinos.

In section 4 we report on experiments that we conducted by means of problems originating in the design of the RF cavity of the 590 MeV ring cyclotron installed at the Paul Scherrer Institute (PSI) in Villigen, Switzerland. These experiments indicate that the implemented solution procedure is almost optimal in that the number of iteration steps until convergence only slightly depends on the problem size.

2 The eigensolver

In this paper we focus on the symmetric Jacobi–Davidson algorithm (JDSYM) for solving (1.2). This algorithm is well-suited since it does not require the factorization of the matrices \( A \) or \( M \). In [2, 3, 4] we found JDSYM to be the method of choice for this problem.

2.1 The symmetric Jacobi–Davidson algorithm

The Jacobi–Davidson algorithm has been introduced by Sleijpen and van der Vorst [21]. There are variants for all types of eigenvalue problems [5]. Here we use a variant adapted to the generalized symmetric eigenvalue problem (1.2) as described in detail in [2, 12].

Here we just sketch the algorithm. Let us assume that we have already computed \( q \) eigenvalues of (1.2) and have the corresponding eigenvectors available in the \( n \times q \) matrix \( Q \). Of course, \( C^TQ = 0 \). Let us further assume that we have available a search space \( \mathcal{R}(V_k) \) where \( V_k = [v_1, \ldots, v_k] \) with \( C^TV_k = 0 \) and \( Q^TMV_k = 0 \). JDSYM proceeds in three steps to expand the search space by one dimension.

1. **Extraction.** In the extraction phase, a Ritz pair of (1.2) restricted to \( \mathcal{R}(V_k) \) is computed. This amounts to computing the spectral decomposition of \( V_k^TAV_k \) and selecting a particular Ritz pair \((\tilde{\rho}, \tilde{q})\) in \( \mathcal{R}(V_k) \) that best approximates the searched eigenpair. Here \( \tilde{\rho} = \rho(\tilde{q}) \) denotes the Rayleigh quotient of \( \tilde{q} \).

2. **Correction.** In order to improve the actual best approximation \((\tilde{\rho}, \tilde{q})\) a correction \( t \) is determined that satisfies the correction equation

\[
(I - M\tilde{Q}\tilde{Q}^T)(A - \tilde{\rho}M)(I - \tilde{Q}\tilde{Q}^T M)t = -r,
\]

\[
\tilde{Q}^Mt = 0, \quad C^Tt = 0, \quad \tilde{Q} = [Q, \tilde{q}]
\]

(2.1)

where \( r = A\tilde{q} - \tilde{\rho}M\tilde{q} \) is called the residual at \( \tilde{q} \). \( t \) can be interpreted as a Newton correction at \( \tilde{q} \) for solving \( Ax - \rho(x)Mx = 0 \). For efficiency reasons, the correction equation is solved only approximately by a Krylov subspace method [23].

3. **Extension.** The solution \( t \) of (2.1) is made \( M \)-orthogonal to \( V_k \) and orthogonal to \( C \),

\[
\hat{t} = (I - V_kV_k^T M)(I - YH^{-1}C^T)t.
\]

(2.2)

After \( M \)-normalization, \( \hat{t} \) is appended to \( V_k \) to yield \( V_{k+1} \). Note that \( Y = M^{-1}C \) is a (very sparse) basis of the null space of \( A \) and that
\( H = Y^TC \) is the discretization of the Laplace operator in the nodal element space [3].

In order to limit the memory costs the dimension of \( V_k \) is limited. If \( \dim(V_k) = j_{\text{max}} \) then the iteration is restarted meaning that the vectors \( v_1, \ldots, v_{j_{\text{max}}} \) are replaced by the \( j_{\text{min}} \) best Ritz vectors in \( V_k \).

### 2.2 Solving the correction equation

For the Krylov subspace method to be efficient, a preconditioner is a prerequisite. Following Fokkema et al. [10] for solving (2.1) we use preconditioners of the form

\[
(I - M\tilde{Q}\tilde{Q}^T)K(I - \tilde{Q}\tilde{Q}^TM),
\]

where \( K \) is a symmetric preconditioner of \( A - \tilde{\rho}M \). For efficiency reasons, we compute \( K \) only once for a fixed shift \( \sigma \) such that \( K \approx A - \sigma M \). We experienced best results when \( \sigma \) is in the middle of the set of desired eigenvalues. However, in the experiments of this paper, we choose \( \sigma \) somewhere among the desired eigenvalues, usually a little above the smallest eigenvalue. This makes it possible to use the same \( K \) for all equations we are solving.

In each preconditioning step, an equation of the form

\[
(I - M\tilde{Q}\tilde{Q}^T)Kc = b \quad \text{and} \quad \tilde{Q}^TMc = 0
\]

has to be solved. The solution \( c \) is [12, p. 92]

\[
c = (I - K^{-1}M\tilde{Q}(\tilde{Q}^TMK^{-1}M\tilde{Q})^{-1}\tilde{Q}^TM)K^{-1}b.
\]

Briefly, the Krylov subspace method is invoked with the following arguments:

- **system matrix:** \((I - M\tilde{Q}\tilde{Q}^T)(A - \tilde{\rho}M)\)
- **preconditioner:** \((I - K^{-1}M\tilde{Q}(\tilde{Q}^TMK^{-1}M\tilde{Q})^{-1}\tilde{Q}^TM)K^{-1}\)
- **right hand side:** \(- (I - M\tilde{Q}\tilde{Q}^T)r\)
- **initial vector:** \(0\)

Both the system matrix and the preconditioner are symmetric. However, because of the dynamic shift \( \tilde{\rho} \), they can become indefinite. For this reason, the QMRS iterative solver [11] is suited particularly well.
2.3 The preconditioner

Our preconditioner \( K \), cf. (2.3), is a combination of a hierarchical basis preconditioner and an algebraic multigrid (AMG) preconditioner.

Since our finite element spaces consist of Nédélec and Lagrange finite elements of degree 2 and since we are using hierarchical bases, we employ the hierarchical basis preconditioner that we used successfully in [3]. Numbering the linear before the quadratic degrees of freedom, the matrices \( A \) and \( M \) in (1.2) get a 2-by-2 block structure,

\[
\begin{align*}
A &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \\
M &= \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}.
\end{align*}
\]

(2.5)

Here the \((1,1)\)-blocks, i.e. \( A_{11} \) and \( M_{11} \), correspond to the bilinear forms involving linear basis functions. The hierarchical basis preconditioners as discussed by Bank [6] are stationary iteration methods for solving

\[
\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \quad K_{ij} = A_{ij} - \sigma M_{ij}.
\]

that respect the 2-by-2 block structure of \( A \) and \( M \). If the underlying stationary method is the symmetric block Gauss-Seidel iteration then

\[
K = \begin{bmatrix} K_{11} \\ K_{21} & \tilde{K}_{22} \end{bmatrix} \begin{bmatrix} K_{11} \\ \tilde{K}_{22} \end{bmatrix}^{-1} \begin{bmatrix} K_{11} & K_{12} \\ \tilde{K}_{22} & \tilde{K}_{22} \end{bmatrix}.
\]

The approximation \( \tilde{K}_{22} \) of \( K_{22} \) again represents a stationary iteration method of which we execute a single iteration step. We implemented two iterations that access local information only. First, the Jacobi iteration

\[
\tilde{K}_{22} = \text{diag}(K_{22}),
\]

(2.6)

and, second, an iteration that executes a symmetric Gauss-Seidel step on the largest diagonal block owned by a processor. Both, approaches are quite efficient in a parallel environment and easy to implement. The latter is more powerful than the former. It deteriorates with increasing processor numbers, though.

For very large problems, the direct solve with \( K_{11} \) becomes inefficient and in particular consumes far too much memory due to fill-in. In order to reduce the memory requirements of the two-level hierarchical basis preconditioner, and, at the same time, not lose its optimality with respect to iteration count, we
replaced the direct solves by a single V-cycle of an AMG preconditioner. This makes our preconditioner a true multilevel preconditioner.

We found ML [19] the AMG solver of choice as it can handle unstructured systems that originate from the Maxwell equation discretized by linear Nédélec finite elements. ML implements a smoothed aggregation AMG method [24] that extends the straightforward aggregation approach of Reitzinger and Schöberl [17]. ML is part of Trilinos which is discussed in the next section.

3 Parallelization issues

For very large problems, the data must be distributed over a series of processors. To make the solution of these large problems feasible, an efficient parallel implementation of the algorithm is necessary. Such a parallelization of the algorithm requires proper data structures and data layout, some parallel direct and iterative solvers, and some parallel preconditioners. For our project, we found the Trilinos Project [22] to be an efficient environment to develop such a complex parallel application.

3.1 Trilinos

The Trilinos Project is an ongoing effort to design, develop, and integrate parallel algorithms and libraries within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific applications [22, 13, 18]. Trilinos is a collection of compatible software packages. Their capabilities include parallel linear algebra computations, parallel algebraic preconditioners, the solution of linear and non-linear equations, the solution of eigenvalue problems, and related capabilities. Trilinos is primarily written in C++ and provides interfaces to essential Fortran and C libraries.

For our project, we use the following packages

- Epetra, the fundamental Trilinos package for basic parallel algebraic operations. It provides a common infrastructure to the higher level packages,
- Amesos, the Trilinos wrapper for linear direct solvers (SuperLU, UMFPACK, KLU, etc.),
- AztecOO, an object-oriented descendant of the Aztec library of parallel iterative solvers and preconditioners,
- ML, the multilevel preconditioner package, that implements a smoothed aggregation AMG preconditioner capable of handling Maxwell equations [7,
19].

For a detailed overview of Trilinos and its packages, we refer the reader to [13].

3.2 Data structures

Real valued double precision distributed vectors, multivectors (collections of one or more vectors) and (sparse) matrices are fundamental data structures, which are implemented in Epetra. The distribution of the data is done by specifying a communicator and a map, both Epetra objects.

The notion of a communicator is known from MPI [16]. A communicator defines a context of communication, a group of processes and their topology, and it provides the scope for all communication operations. Epetra implements communicators for serial and MPI use. Moreover, communicator classes provide methods similar to other MPI functions.

Vectors, multivectors and matrices are distributed row wise. The distribution is defined by means of a map. A map can be defined as the distribution of a set of integers across the processes, it relates the global and local row indices. To create a map object, a communicator, the global and local numbers of elements (rows), and the global numbering of all local elements have to be provided. So, a map completely describes the distribution of vector elements or matrix rows. Note that rows can be stored on several processors redundantly. To create a distributed vector object, in addition to a map, one must assign values to the vector elements. The Epetra vector class offers standard functions for doing this and other common vector manipulations.

Trilinos supports dense and sparse matrices. Sparse matrices are stored locally in the compressed row storage (CRS) format [5]. Construction of a matrix is row by row or element by element. Afterwards, a transformation of the matrix is required in order to perform matrix-(multi)vector product \( Y = A \times X \) efficiently, specifying maps of the vectors \( X \) and \( Y \).

Some algorithms require only the application of a linear operator, such that the underlying matrix need not be available as an object. Epetra handles this by means of a virtual operator class. Epetra also admits to work with block sparse matrices. Unfortunately, there is no particular support for symmetric matrices.

To redistribute data, one defines a new, so-called target map and creates an empty data object according to this new map as well as an Epetra’s import/export object from the original and the new map. The new data object can be filled with the values of the original data object using the import/export
object, which describes the communication plan.

3.3 Data distribution

A suitable data distribution can reduce communication costs and balance the computational load. The gain from such a redistribution can, in general, overcome the cost of this preprocessing step.

Zoltan [25, 9] is a library that contains tools for load balancing and parallel data management. It provides a common interface to graph partitioners like METIS and ParMetis [15, 14]. Zoltan is not a Trilinos package. But the Trilinos package EpetraExt provides an interface between Epetra and Zoltan.

In our experiments, we use ParMetis to distribute the data. This partitioner tries to distribute a graph such that (1) the number of graph vertices per processor is balanced and (2) the number of edge cuts is minimized. The former balances the work load. The latter minimizes the communication overhead by concentrating elements in diagonal blocks and minimizing the number of non-zero off-diagonal blocks. In our experiments, we define a graph $G$, which contains connectivity informations for each node, edge, and face of the finite element mesh. $G$ is constructed from portions of the sparse matrices $M$, $H$, and $C$.

4 Numerical experiments

In this section, we discuss the numerical experiments used to assess the parallel implementation. Preliminary results have been presented in [1].

4.1 General comments

The experiments have been executed on a 32 dual-node PC cluster in dedicated mode. Each node has 2 AMD Athlon 1.4 GHz processors, 2 GB main memory, and 160 GB local disk. The nodes are connected by a Myrinet providing a communication bandwidth of 2000 Mbit/s. The system operates with Linux 2.4.20.

For these experiments, we use the developer version of Trilinos on top of MPICH 1.2.5. We compare the execution times for computing the 5 smallest positive eigenvalues and corresponding eigenvectors using JDSYM with the multilevel preconditioner defined in section 2.3. We set $j_{\text{min}} = 6$ and $j_{\text{max}} = 15$. 
An approximate eigenpair \((\rho, \mathbf{q})\) is considered converged when the norm of the residual \(\mathbf{r} = A\mathbf{q} - \rho M\mathbf{q}\) satisfies

\[ \|\mathbf{r}\|_2 \leq \varepsilon \|\mathbf{q}\|_M, \]

where \(\varepsilon\) is set to \(10^{-8}\).

The projector (2.2) is applied only once per outer iteration. For this projector, applying \(H^{-1}\) amounts to solving a Poisson equation [3]. In order to do so, we use the preconditioned conjugate gradient method (PCG), combined with a multilevel preconditioner. We require high accuracy from this iterative solver (residual norm reduction by a factor \(10^{10}\)), so that the solution vector \(\mathbf{x}\) satisfies the constraints \(C^T \mathbf{x} = 0\).

The accuracy of the results was satisfactory. The computed eigenvectors were \(M\)-orthogonal and orthogonal to \(C\) to machine precision. The 2-norm of the residuals of the computed eigenpairs were below \(10^{-9}\).

4.2 Academic problem

The first example is a rectangular box denoted \texttt{box170k}. The matrix \(A\) is of size \(1,030,518\). The shifted operator \(A - \sigma M\) has \(20,767,052\) non-zero entries. The eigenvalues to be computed are

\[ \lambda_1 \approx 1.27, \quad \lambda_2 \approx 2.37, \quad \lambda_3 \approx 3.99, \quad \lambda_4 \approx 4.19, \quad \lambda_5 \approx 5.09. \]

We set the shift \(\sigma = 1.5\). The discrete Laplacian \(H\) is of size \(209,741\) and has \(5,447,883\) non-zero entries.

Table 1

\begin{tabular}{cccccccc}
\hline
\textbf{p} & \textbf{t [sec]} & \textbf{E(p)} & \textbf{t\_prec [%]} & \textbf{t\_proj [%]} & \textbf{n\_outer} & \textbf{n\_avg} & \textbf{n\_inner} \\
\hline
4 & 3131 & 2895 & 1.00 & 1.00 & 49 & 39 & 20 & 22 & 52 & 54 & 21.29 & 23.26 \\
8 & 1793 & 1709 & 0.87 & 0.85 & 46 & 38 & 22 & 24 & 52 & 54 & 20.85 & 23.06 \\
12 & 1298 & 1201 & 0.80 & 0.80 & 46 & 38 & 22 & 25 & 53 & 54 & 20.40 & 22.07 \\
16 & 971 & 915 & 0.81 & 0.79 & 46 & 39 & 23 & 25 & 53 & 54 & 20.58 & 22.33 \\
\hline
\end{tabular}

In Table 1, we report the execution times \(t = t(p)\) for solving the eigenvalue problem with various numbers \(p\) of processors. These times do not include preparatory work, such as the assembly of matrices or the data redistribution. \(E(p)\) describes the parallel efficiency with respect to the simulation run with the smallest number of processors. \(t\_prec\) and \(t\_proj\) indicate the percentage of the
time the solver spent applying the preconditioner and the projector, respectively. \( n_{\text{avg}} \) is the average number of QMRS iterations per outer iteration. The total number of applications for the preconditioner \( K \) is approximately \( n_{\text{outer}} \cdot n_{\text{avg}}^{\text{inner}} \).

In Table 1, we use AMG preconditioners for the block \( K_{11} \) and for the whole \( H \). However, we distinguish, in the table, the results obtained by applying, to the block \( K_{22} \), one symmetric Gauss-Seidel (SGS) step with the diagonal block owned by a processor (left columns) or one Jacobi step (right columns), see (2.6). The block SGS reduces the number of QMRS iterations. However, each iteration is more expensive than one iteration with Jacobi step. For this test case, the overall computation time is faster with the Jacobi steps. In both cases, \( n_{\text{avg}}^{\text{inner}} \) is almost constant, which indicates that the preconditioner \( K \) does not deteriorate as \( p \) increases.

Note that the problem box170k was too large to be solved on 1 or 2 processors.

Table 2

| Box170k: Results using the combined 2-level/ML preconditioner for \( K \) and \( H \) |
|-----|-----|-----|-----|
| \( p \) | \( t \) [sec] | \( E(p) \) | \( n_{\text{outer}} \) | \( n_{\text{avg}}^{\text{inner}} \) |
|-----|-----|-----|-----|
| 4   | 2653 | 1.00 | 54  | 23.26 |
| 8   | 1575 | 0.84 | 55  | 23.22 |
| 12  | 1105 | 0.80 | 54  | 22.07 |
| 16  | 845  | 0.78 | 54  | 22.33 |

In Table 2, we use the AMG preconditioner for the block \( K_{11} \), Jacobi steps for \( K_{22} \), and a similar strategy for \( H \) (AMG preconditioner for \( H_{11} \) and Jacobi steps for \( H_{22} \)). Comparing with the corresponding columns in Table 1, we see that \( n_{\text{avg}}^{\text{inner}} \) and \( n_{\text{outer}} \) did not change. However, the execution times are reduced significantly.

4.3 Engineering problem

The next two problem originate in the design of the RF cavity of the 590 MeV ring cyclotron installed at the Paul Scherrer Institute (PSI) in Villigen, Switzerland. We deal with two problem sizes. They are labelled cop40k and cop300k. Their characteristics are given in Table 3, where we list the order \( n \)

Table 3

<table>
<thead>
<tr>
<th>Matrix characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_{A-M} )</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>cop40k</td>
</tr>
<tr>
<td>cop300k</td>
</tr>
</tbody>
</table>
and the number of non-zeros \( nnz \) for the shifted operator \( A - \sigma M \) and for the discrete Laplacian \( H \). Here the eigenvalues to be computed are

\[
\lambda_1 \approx 1.13, \quad \lambda_2 \approx 4.05, \quad \lambda_3 \approx 9.89, \quad \lambda_4 \approx 11.3, \quad \lambda_5 \approx 14.2.
\]

We again set \( \sigma = 1.5 \).

Table 4

cop40k: Comparison of the block Gauss-Seidel (left) and the Jacobi (right) preconditioners for \( K_{22} \)

<table>
<thead>
<tr>
<th>( p )</th>
<th>( t ) [sec]</th>
<th>( E(p) )</th>
<th>( t_{\text{prec}} ) [%]</th>
<th>( t_{\text{proj}} ) [%]</th>
<th>( n_{\text{outer}} )</th>
<th>( n_{\text{inner}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1806 2092</td>
<td>1.00 1.00</td>
<td>45 37</td>
<td>18 18</td>
<td>48 53</td>
<td>12.62 19.02</td>
</tr>
<tr>
<td>2</td>
<td>1142 1219</td>
<td>0.79 0.86</td>
<td>47 38</td>
<td>16 17</td>
<td>51 54</td>
<td>15.47 18.96</td>
</tr>
<tr>
<td>4</td>
<td>634 642</td>
<td>0.71 0.81</td>
<td>46 37</td>
<td>16 17</td>
<td>51 54</td>
<td>16.29 19.43</td>
</tr>
<tr>
<td>8</td>
<td>327 321</td>
<td>0.69 0.81</td>
<td>46 37</td>
<td>17 18</td>
<td>51 53</td>
<td>16.24 19.23</td>
</tr>
<tr>
<td>12</td>
<td>216 227</td>
<td>0.70 0.77</td>
<td>47 40</td>
<td>19 19</td>
<td>51 53</td>
<td>15.51 19.47</td>
</tr>
<tr>
<td>16</td>
<td>175 174</td>
<td>0.65 0.75</td>
<td>50 43</td>
<td>19 20</td>
<td>51 53</td>
<td>16.35 18.96</td>
</tr>
</tbody>
</table>

Table 5

cop40k: Comparison of results with (left) and without (right) redistribution

<table>
<thead>
<tr>
<th>( p )</th>
<th>( t ) [sec]</th>
<th>( E(p) )</th>
<th>( n_{\text{outer}} )</th>
<th>( n_{\text{inner}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1957 2005</td>
<td>1.00 1.00</td>
<td>53 53</td>
<td>19.02 19.02</td>
</tr>
<tr>
<td>2</td>
<td>1159 1297</td>
<td>0.84 0.77</td>
<td>54 53</td>
<td>19.06 19.66</td>
</tr>
<tr>
<td>4</td>
<td>622 845</td>
<td>0.79 0.59</td>
<td>54 55</td>
<td>19.43 19.18</td>
</tr>
<tr>
<td>8</td>
<td>318 549</td>
<td>0.77 0.45</td>
<td>53 54</td>
<td>19.23 19.67</td>
</tr>
<tr>
<td>12</td>
<td>231 451</td>
<td>0.71 0.37</td>
<td>53 54</td>
<td>20.47 19.78</td>
</tr>
<tr>
<td>16</td>
<td>184 366</td>
<td>0.66 0.34</td>
<td>53 54</td>
<td>19.00 19.04</td>
</tr>
</tbody>
</table>

In Table 5, we use the AMG preconditioner for the block \( K_{11} \), Jacobi steps for \( K_{22} \), and a similar strategy for \( H \) (AMG preconditioner for \( H_{11} \) and Jacobi steps for \( H_{22} \)). We investigate the effect of redistributing the matrices. Results in Table 5 show that the quality of data distribution is important.
For the largest number of processors \((p = 16)\), the execution time with the redistributed matrices is half the time obtained with the original matrices. These were straightforward block distributions of the matrices given in (2.5).

Table 6

cop300k: Results with the best parameters

<table>
<thead>
<tr>
<th>(p)</th>
<th>(t) [sec]</th>
<th>(E(p))</th>
<th>(n_{\text{outer}})</th>
<th>(n_{\text{avg}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4346</td>
<td>1.00</td>
<td>62</td>
<td>28.42</td>
</tr>
<tr>
<td>12</td>
<td>3160</td>
<td>0.91</td>
<td>62</td>
<td>28.23</td>
</tr>
<tr>
<td>16</td>
<td>2370</td>
<td>0.92</td>
<td>61</td>
<td>28.52</td>
</tr>
</tbody>
</table>

Finally, in Table 6, we report results for our largest problem size cop300k. We use the 2-level preconditioner for \(K\) and \(H\): an appropriate AMG preconditioner for the blocks \(K_{11}\) and \(H_{11}\) and one step of Jacobi for the blocks \(K_{22}\) and \(H_{22}\). Table 6 shows that, for these experiments, the iteration counts behave nicely and that efficiencies stay high.

5 Conclusions

In conclusion, the parallel algorithm shows a very satisfactory behavior. The efficiency of the parallelized code does not get below 65 percent for 16 processors. We usually have a big efficiency loss initially. Then efficiency decreases slowly as the number of processors increases. This is natural due to the growing communication-to-computation ratio.

The accuracy of the results are satisfactory. The computed eigenvectors were \(M\)-orthogonal and orthogonal to \(C\) to machine precision. The 2-norm of the residuals of the computed eigenpairs were below \(10^{-9}\).

Our next goals in this project are the reduction of the memory costs and the overhead of the redistribution of our matrices. The memory costs are still so high that we are not capable to solve the very large problems that really interest the engineers. The reason for this lies not primarily in the size of these problems, but in a lack of memory scalability of the code. The cost of the redistribution may be reduced by working with a smaller than our artificial graph \(G\). We may, e.g., just determine a good parallel distribution for the vertices and then adjust the edges and faces.
References


