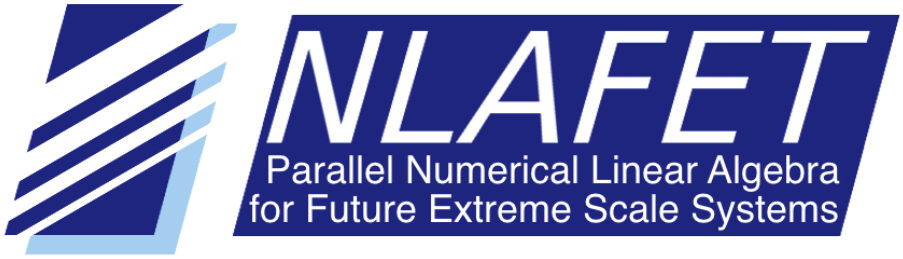


H2020–FETHPC–2014: GA 671633

D5.3

Validation and evaluation

April 2019

DOCUMENT INFORMATION

Scheduled delivery	2019-04-30
Actual delivery	2019-04-29
Version	2.0
Responsible partner	INRIA

DISSEMINATION LEVEL

PU — Public

REVISION HISTORY

Date	Editor	Status	Ver.	Changes
2019-04-09	INRIA, UNIMAN, RAL members	Draft	1.0	Initial version of document produced
2019-04-16	INRIA, UNIMAN, RAL members	Draft	1.1	Second version of document produced
2019-04-26	INRIA, UNIMAN, RAL members	Final	2.0	Revised for submission

AUTHOR(S)

Laura Grigori, Jan Papez, Olivier Tissot, INRIA
Jack Dongarra, David Stevens, Maksims Abalenkovs, UNIMAN
Iain Duff, Florent Lopez, Sebastien Cayrols, Stojce Nakov, RAL

INTERNAL REVIEWERS

Bo Kågström, UMU
Iain Duff, STFC

CONTRIBUTORS

Andrew Sunderland (STFC Daresbury)

ACKNOWLEDGEMENTS

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

List of Tables

1 Introduction

The *Description of Action* document states for Deliverable 5.3:

"*Evaluation of the NLAFET library in the context of the applications, leading to validation of the library and recommendations for future improvements.*"

This deliverable is in the context of Workpackage 5, and describes our efforts on validating and integrating the NLAFET library into several challenging applications.

1. Task-based Shared Memory Parallelism into 2DRMP software for modelling of electron scattering from H-like atoms and ions. Collaboration with Professor Stan Scott, Queen's University, Belfast. NLAFET contact Jack Dongarra, UNIMAN.

2. Load flow based calculations in large-scale power systems and PowerFactory code. Collaboration with Bernd Klöss, DIgSILENT GmbH, Germany. NLAFET Contact, Iain Duff, RAL.

3. Communication avoiding iterative methods and their efficiency for solving linear systems arising from several different applications, in particular linear elasticity problems. NLAFET contact Laura Grigori, Inria.

4. Data analysis in astrophysics and Midapack. Collaboration with University Paris 7, France. NLAFET contact is Laura Grigori, Inria, principal application contacts are Radek Stompor from APC/CNRS, France, and Carlo Baccigalupi of SISSA Italy.

The applications are discussed in detail in Sections 2 to 5 and we present our conclusions in Section 6.

The communication avoiding iterative methods tested in applications 3 and 4 above are currently available within preAlps library (Inria) via NLAFET github. PreAlps was developed by Inria in the context of Workpackage 4. The code has a reverse communication interface such that it can be easily integrated into other scientific application and is scalable up to 16,000 cores.

In the Description of Action document, we had planned to collaborate with Thomas Schulthess of ETH Zurich, Switzerland to test the dense solvers/eigensolvers developed in Workpackage 2 in materials science and chemistry. As already pointed out in Deliverable 5.2, this collaboration was not pursued due to a lack of time from ETH Zurich. We had also planned to validate our iterative methods on linear systems arising from Code Saturne in a collaboration with EDF, France. However this was not pursued given the feedback we received from the reviewers after our mid-term review.

2 Task-based Shared Memory Parallelism into 2DRMP software

This section describes improvements made to the 2DRMP software suite [36] for the modelling of electron scattering from H-like atoms and ions. Linear algebra routines developed within NLAFET have been integrated into an existing code, which was originally developed at Queen's University Belfast. A number of other improvements and modernisations have

also been made to the software, aimed at improving the usability and extensibility of the code.
Rather than focusing on explicit gains in computational performance on Intel-based architectures, the work undertaken has sought to improve the accessibility of the code, by allowing users access to non-proprietary software for the linear algebra components that dominate computational resources. In addition the work has sought to improve portability, allowing high-performance solutions to be obtained on non-Intel architectures, such as ARM or AMD machines. To achieve these goals, numerical linear algebra routines from the PLASMA library [1] have been introduced into the software; specifically solutions to the general matrix vector equation using LU factorisation (`xGETRS`), matrix-matrix multiplication (`xGEMM`), and symmetric eigenvalue and eigenvector computation (`xSYEVD`). The PLASMA library has been enhanced and expanded during the NLAFET project, and is available from the NLAFET software repository (`https://github.com/NLAFET/plasma`).
The structure of this section is as follows: Section 2.1 provides a high-level overview of the R-matrix method, with Section 2.2 introducing the concept of R-matrix propagation. Section 2.3 describes the structure of the 2DRMP code, and outlines the modifications made to it within this project. In Section 2.4 some numerical performance results are presented for the updated 2DRMP code and the constituent linear algebra kernels. Section 2.5 then discusses the status of ongoing work and plans for publication.

2.1 Background and Theory

R-matrix theory, a term derived from the study of resonance theory, is a well established technique for the analysis of nuclear interactions, having been first introduced in 1947 by Wigner and Eisenbud [40], and adapted for the study of atomic and molecular collisions in the early 1970s [10, 12]. In broad terms, R-matrix theory seeks to allow particle collisions to be analysed without considering the complexity of nuclear forces within the atomic nucleus. Instead, the unknown internal properties of the nucleus are treated as parameters and appear as elements within the R-matrix. Interest in R-matrix theory has remained strong since its introduction, e.g. [6, 8, 9, 32], as the R-matrix method has proven to be a remarkably stable, robust and efficient technique for solving the close-coupling equations that arise when describing electron and photon collisions with atoms, ions and molecules. An excellent overview of R-matrix theory is available within the MIT online lecture notes [29].
A number of codes for R-matrix-based computations have been developed over the years; see for example [5, 13, 41]. While packages such as these have been used extensively, they are primarily concerned with low-energy scattering, where the incident energy is insufficient to ionise the target. At intermediate energies, i.e. from close to the ionisation threshold to several times above it, modelling of the scattering processes becomes significantly more difficult, because account must be taken of the coupling amongst the infinite number of continuum states of the ionized target, and the infinite number of target bound states lying below the ionization threshold.
R-matrix approaches capable of representing this coupling with acceptable accuracy at intermediate energies fall into two categories; one is the R-matrix with Pseudostates Method (RMPS) [4], and the other is the Intermediate Energy R-Matrix Method (IERM) [11]. Broadly speaking, the RMPS approach offers relative simplicity of implementation, and results in R-matrices of generally manageable size. In contrast, the IERM approach results in a densely-

packed pseudostate basis, and therefore larger R-matrices that require a careful treatment in order to obtain numerical solutions at a reasonable computational cost. The improved resolution of the IERM method makes it more suitable for the study of scattering processes, such as electron impact ionisation close to the ionisation threshold [35]. The 2DRMP software suite [36], which we consider in this report, was developed to provide a framework for the study of two-particle collisions in which both particles are considered independently (i.e. a 2D approach), using IERM. Prior to the development of 2DRMP in 2009, computational codes using IERM were only available in a configuration where both particles are considered identically (i.e. a 1D approach).
The full detail of the 2D IERM approach can be found in [36]. The theory is complex, and requires an in-depth knowledge of quantum mechanical interactions in order to fully appreciate it. Given that such knowledge is beyond the scope of the NLAFET project, and has relatively little overlap with the developments made to the 2DRMP code in this project, we shall restrict our discussion to the broad practical steps involved in the 2D IERM approach, and the associated R-Matrix propagation procedure, as implemented in the 2DRMP software suite.

2.2 2D R-Matrix Propagation Procedure

The following provides a high-level overview of the steps involved in the 2D R-Matrix propagation procedure, with a particular focus on the steps where linear algebra routines are utilised. A comprehensive description of the procedure is available in [36].
R-matrix theory is based around dividing the configuration space describing the collision process into two regions by a sphere of radius a, where a is chosen such that the charge distribution of the target particle is contained within the sphere. This sphere is referred to as the *internal region*, in which the full quantum mechanical interaction of the particles is considered. In the *external region*, i.e. outside the sphere, the solution is approximated by potential theory, and acts as a boundary condition to the solution within the internal region. In the IERM method it can be shown that the number of basis functions required, and therefore the size of the dense matrices generated, is proportional to a^2 [34]. Therefore, as a increases to allow the study of excitation to higher-lying energy states, the size of the computation can rapidly become computationally intractable.
To mitigate this issue and allow the study of collisions at intermediate energies, the R-matrix propagation approach divides the (r_1, r_2) space of the internal region into a number of subregions. Within each subregion a local R-matrix is constructed, and these local R-matrices are used to propagate a global R-matrix throughout the internal region.
Within each subregion two matrices are formed; the local Hamiltonian matrix, which represents the energy states of the system, and the local R-matrix, which describes the particle interactions. In terms of numerical linear algebra, the local Hamiltonian matrix is diagonalised, producing the eigenvalues and eigenvectors of the system in this subregion. These eigenvalues and eigenvectors are used to build the local R-matrices, and to compute surface amplitudes, which are used as local boundary conditions to allow propagation of the global R-matrix throughout the internal region. The propagation of the local R-matrix to its neighbour requires the solution of a non-symmetric matrix equation $Ax = b$. The local Hamiltonian matrix and the corresponding R-matrix are generally not of equal size, with the

Hamiltonian being larger in most practical scenarios. However, the diagonalisation of the Hamiltonian occurs only once for each subregion, whereas the propagation of the local R-matrix occurs many times, dependent on the collision energy. For high energy collisions the propagation may occur tens or hundreds of times, and therefore more computational resources are typically expended on executing the the LU factor-solve routine than on matrix diagonalisation.

Figure 1 shows a schematic representation for the propagation of the local R-matrices

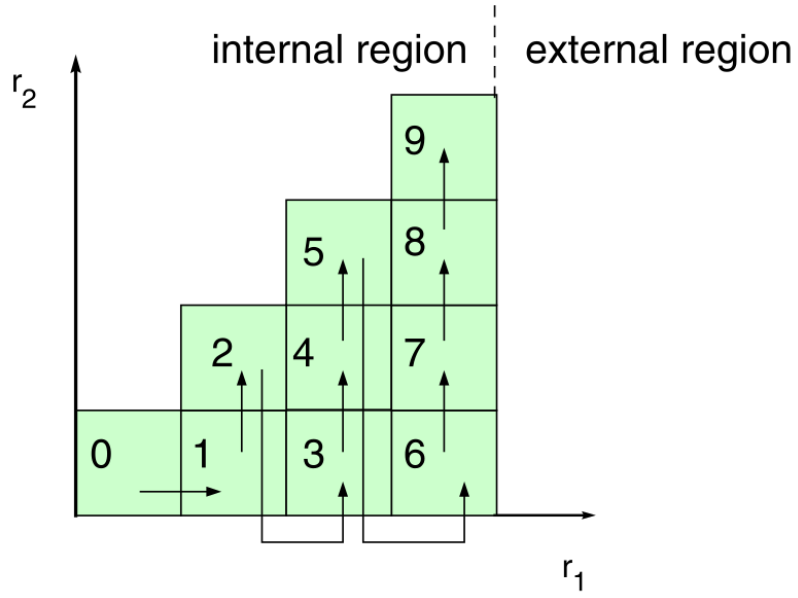


Figure 1: Propagation of the R-matrix across the internal region.

through the internal region. The (r_1, r_2) space is divided into subregions, with symmetry requiring that only the lower half of the global domain be analysed. Before computing the R-matrices, the eigenvectors and eigenvalues from the Hamiltonian matrices at all subregions are obtained, allowing computation of surface amplitudes at each of the domain boundaries. With this information the first local R-matrix may be computed, denoted as subregion 0 in Figure 1. The local R-matrix for subregion 0, along with the surface amplitudes corresponding to the interface with its neighbouring subregion, denoted as subregion 1, allow the local R-matrix at subregion 1 to be computed. In this way the global R-matrix is propagated across all subregions systematically, starting from the r_1 axis at the bottom of each strip, and working up the r_2 axis towards the diagonal. Once the propagation is complete and the global R-matrix is obtained, the basis functions that span outermost edges of the domain are projected onto the corresponding basis functions from the external region, and the global R-matrix is modified accordingly. Note that this propagation procedure produces a solution which is specific to a given scattering energy, and therefore must be performed many times in order to analyse the full range of energies required during the collision. The Hamiltonian matrices, however, need only be computed and diagonalised once.

2.3 Code structure and Developments

The following section outlines the structure of the 2DRMP code. In its original form the code was essentially a suite of seven quasi-independent programs, each of which communicates input and output information via file writes; including the subregion matrices themselves where needed. Our work during the NLAFET project has condensed the code into four subprograms that communicate a minimal amount of information, drastically reducing time spent on file-writes. These changes, along with the addition of the PLASMA library, are described in Section 2.3.2.

2.3.1 2DRMP Code Structure

The 2DRMP code can be considered as a collection of seven fundamental operations, executed in sequence. The function of these constituent programs can be described as follows:

1. bp
This routine constructs the atomic basis functions, as required within the internal region, and the coefficients of the long-range potentials that are required in the external region. The execution of this program is not computationally intensive, and does not contain any linear algebra routines.

2. rint2
Here the radial integrals are computed, as required for the construction of the Hamiltonian matrices in each subregion. This routine is also comparatively inexpensive.

3. newrd
At this stage the Hamiltonian matrices are computed within each subregion. The method of computation varies slightly depending on whether the subregion is located on a diagonal or off-diagonal block in the discretised (r_1, r_2) plane. In the original version of the code these computations are performed in serial within each subregion.

4. diag
The Hamiltonian subregion matrices are now diagonalised, with their eigenvalues and eigenvectors saved as output. This routine is computationally intensive, and makes use of the `xSYEVD` routine.

5. amps
Here the subregion surface amplitudes are computed, i.e. the wavefunction values at the boundaries between subregions, based on the eigenvectors and eigenvalues of the Hamiltonian submatrices.

6. prop
At this stage the R-matrix propagation procedure begins. Local R-matrices are formed and propagated systematically throughout the domain, as described in Section 2.1. The propagation procedure requires a general matrix equation solve (`xGETRS`), as well as a matrix-matrix multiplication (`xGEMM`), for each subregion. This propagation procedure must be performed

for each scattering energy required in the simulation, which will typically number in the tens or hundreds. Therefore this routine typically dominates computational runtime.

7. FARM

The final stage of the procedure computes the solution in the external region, subject to the R-matrix boundary conditions on the outer surface of the internal region. This operation must also be performed for each scattering energy, however owing to the relative simplicity of the equations involved, the computational cost is typically not significant.

2.3.2 Developments and Improvements

The original version of the code executed a program for each of the seven fundamental operations described above, with the programs executed sequentially, and all required input and output data passed via file-writes. In particular, the Hamiltonian subregion matrices were passed between the **newrd** and **diag** stages, and the set of corresponding eigenvectors and eigenvalues were communicated between **diag** and **amps**. This highly segmented approach was useful for the independent development of constituent programs; however, the frequent reading and writing of large files was identified as a weakness, as it left the computational runtime strongly dependent on the efficiency of the storage system.

In order to address this issue, the code has been reformulated to eliminate the need to pass large datasets between programs. By formulating a unified nomenclature for variables, and utilising the more advanced datastructres available within FORTRAN 95+, the code has been reduced to four separate components. The first component combines the functionality of **bp** and **rint2**, thereby acting as a general pre-calculation stage. The second component, now named **blkb**, combines all computations associated with the Hamiltonian submatrices, i.e. **newrd**, **diag** and **amps**. In this way file writes associated with the Hamiltonian matrices and eigenvectors/eigenvalues are eliminated. The final two stages, **prop** and **FARM**, proceed functionally as before, though utilising the improved data-structures. This new approach removes the need for matrix-level objects to be passed via files. The remaining files transferred between programs are typically of a maximum size in the single-digit megabytes, even for large problems.

In the original version of the code the linear algebra components were executed using the Intel Math Kernel Library (MKL), via LAPACK routines. While this approach does generally lead to optimal performance on Intel architectures, it requires the user to have access to proprietary software. While access to MKL is commonplace in the HPC community, it cannot be taken for granted within other fields. Moreover, it restricts use of the code to Intel architectures. To address this, routines from the PLASMA library have been integrated into the 2DRMP code; specifically the PLASMA versions of `xSYEVD`, `xGETRS` and `xGEMM`. These routines exist alongside the original MKL-LAPACK routines, allowing the user to select at compile-time which branch will be utilised. By including PLASMA routines in this way we eliminate the need for proprietary software, and allow access to non-Intel architectures, without sacrificing the highly-tuned performance that these MKL routines can provide on Intel machines when the software is available.

In addition to these two major developments, other modifications have been made to the code. The setup and compilation procedure has been streamlined. A global parameter has

also been introduced to control arithmetic precision, allowing single- or double-precision floating point variables to be used throughout. As the code is written in FORTRAN, quad-precision variables are also available, although a need for this degree of precision is not anticipated. Finally, improvements have been made to the computation of the Hamiltonian subregion matrices. In the previous version of the code parallelism was available only at the subregion level, with the computation of each subregion matrix taking place on a single thread. This has now been extended to allow parallelisation of the subregion matrix across many cores within shared memory, via OpenMP.

2.4 Results and Testing

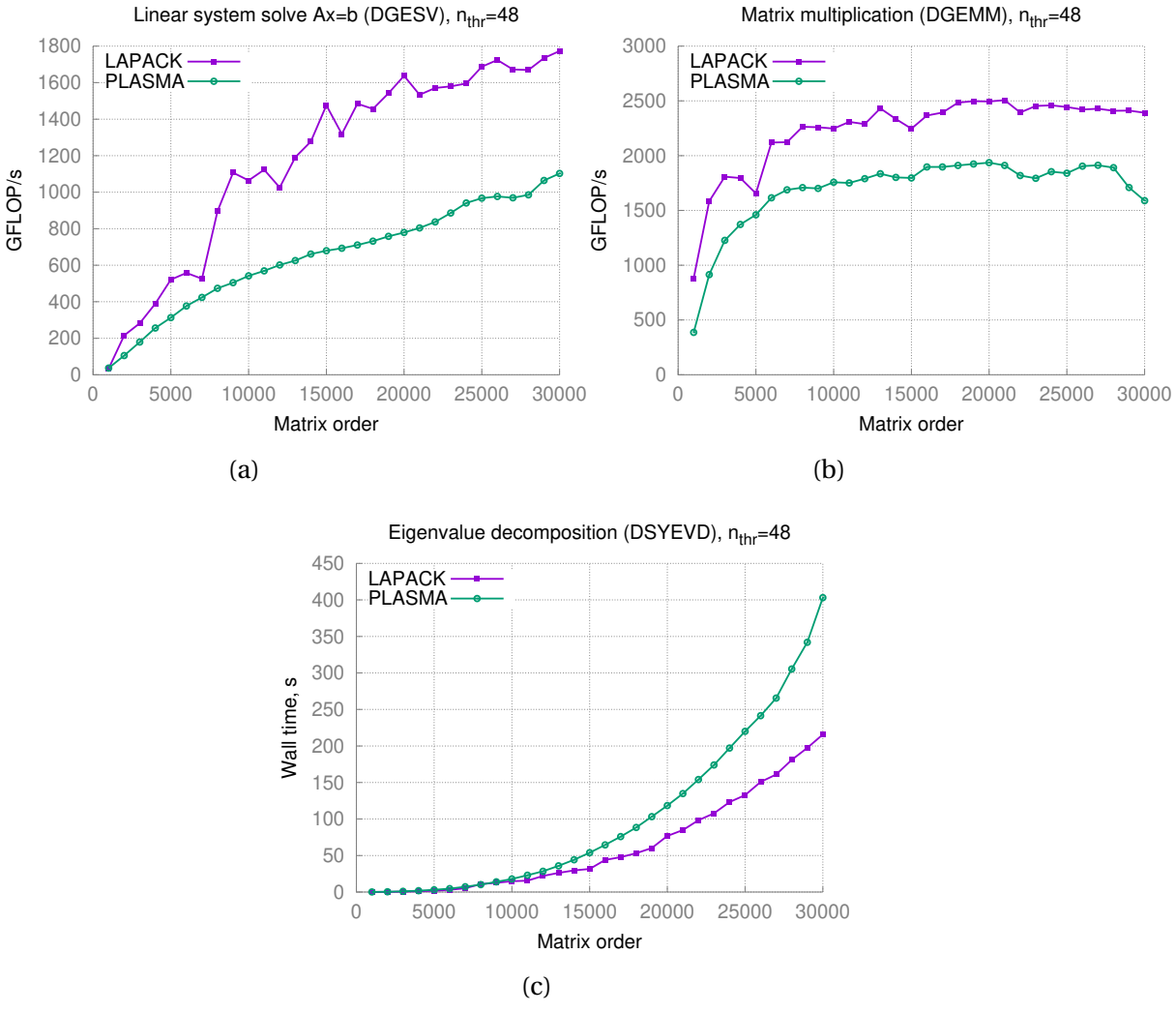
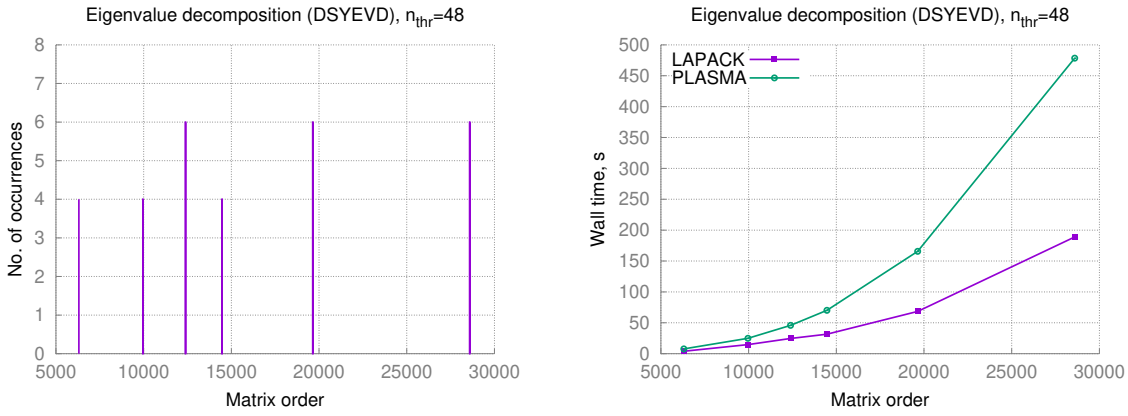


Figure 2: Performance tests for linear algebra kernels on Intel Skylake architecture (2.1Ghz, 2x24 core)

To demonstrate the capabilities of the new 2DRMP code we present results on the Intel Skylake architecture. We first examine the performance of the constituent linear algebra kernels, before presenting a particle collision simulation from the full 2DRMP code, whereby

physical parameters are adjusted that result in varying matrix sizes.
Figure 2 shows the performance of PLASMA in comparison to Intel MKL for the three constituent linear algebra kernels used within 2DRMP at double precision; i.e., `DSYEVD`, `DGETRS` and `DGEMM`. Tests have been run on the MareNostrum computing cluster in Barcelona, on a single node comprising two 24-core Intel Skylake processors (Intel Xeon Platinum 8160) clocked at 2.1Ghz. PLASMA 18 is compared to MKL version 2019.2.057 for a range of matrix sizes up to 30,000. We utilise one thread per core throughout, for 48 threads total. As expected PLASMA is able to produce performance that runs close to, but not matching, that of MKL 2019. For the eigenvalue decomposition routine `DSYEVD`, the precise FLOP count is unknown due to the nature of the algorithm, therefore wall-clock time is presented instead.
To demonstrate the linear algebra workload involved in the 2DRMP code we construct a

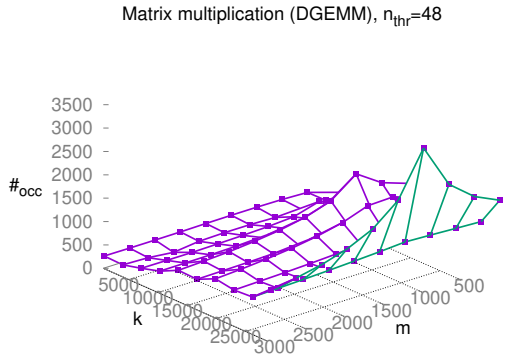


(a) Matrix size and frequency of occurrence

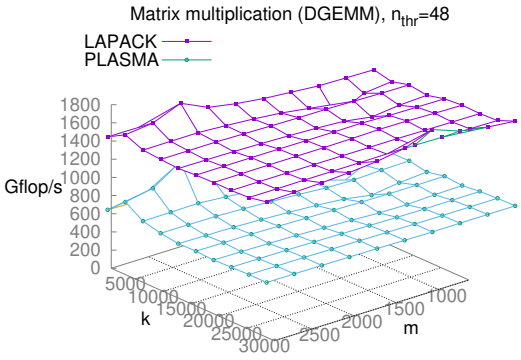
(b) Mean wall-clock time; MKL vs PLASMA

Figure 3: Occurrence and performance of Hamiltonian subregion matrix diagonalisation

two-particle collision simulation, and run it in three slightly different configurations using double precision arithmetic. For each collision we set the following parameters: Inner region radius of 60 a.u. (atomic units), a 4-strip subdivision of the inner region (resulting in a 4×4 grid of subregions), maximum orbital angular momentum = 12, maximum principle quantum number of target states = 70, and 1001 integration points in each subregion. We then run three simulations, adjusting the value of *nmax*, the maximum principal quantum number of the basis orbitals, to 25, 30 and 35 successively, in order to produce simulations with differing Hamiltonian matrix sizes. These three simulations produce a wide range of different matrices, and are intended to highlight the typical computational loads that can be expected when running simulations using the 2DRMP code.
Figure 3 shows the occurrence of subregion Hamiltonian matrix sizes throughout the three simulations (Figure 3a), along with the average wall-clock time spent on obtaining the eigenvalues and eigenvectors for each matrix size (Figure 3b). Since the Hamiltonian matrix must be computed only once per simulation, the number of matrices generated is small and predictable. The subdivision of the internal region into a 4×4 grid of subregions results in four matrices on the diagonal of the subregion grid, and six matrices in the off-diagonal (see Figure 1). In this case, each off-diagonal subregion results in a matrix double the size of those from the diagonal block. Matrix sizes are comparatively large, varying between around 6000 and 14000 for the diagonal blocks, and around 12000 and 28000 for off-diagonal blocks.



(a) Matrix size and frequency of occurrence

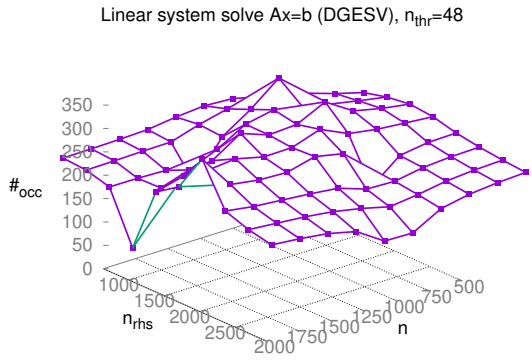


(b) Mean performance; MKL vs PLASMA

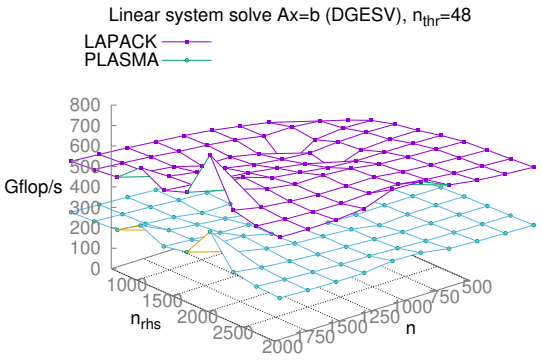
Figure 4: Occurrence and performance of subregion matrix multiplication

Computational performance of the routines closely mirrors that observed from the individual kernel tests.

Figure 4 shows the occurrence of matrix-matrix multiplication (`DGEMM`) operations throug-



(a) Matrix size and frequency of occurrence



(b) Mean performance; MKL vs PLASMA

Figure 5: Occurrence and performance of subregion R-Matrix LU factor-solve

hout the three simulations. In the 2DRMP code matrix multiplications are performed between rectangular matrices, resulting in a square matrix output; i.e. multiplication of an $(m \times k)$ matrix on the left with a $(k \times m)$ matrix on the right, resulting in a square matrix of size $(m \times m)$, where $k > m$. Figure 4a shows the frequency of such matrices, across the range of m and k encountered in the three simulations. While these matrices are typically smaller than the Hamiltonian subregion matrices, their frequency of occurrence is significantly higher, owing to the need to perform the R-matrix propagation stage of the simulation several times to account for the range of scattering energies encountered. The total number of matrix-matrix multiplications therefore ranges into the tens of thousands. Performance of the (`dgemm`) routine is shown in Figure 4b, and is relatively consistent across all encountered matrix sizes.

Figure 5 shows the occurrence of linear system solves (DGESV) encountered throughout the three simulations, along with the associated mean performance. Here matrices are square, of size represented by n, and are solved along with a varying number of right-hand-side vectors, n_{rhs}. The range of matrix sizes encountered varies between around 500 and 2000, whereas the number of RHS vectors associated with each matrix varies between around 1000 and 2000. As with the dgemm results, these linear system solves are associated with the R-matrix propagation stage, resulting in a large number of operations within each subregion. The total number of linear systems to be solved is around 25,000, each with over 1000 solution vectors, and so represents a significant computational load despite the relatively modest size of the individual matrices.

2.5 Ongoing Work

Work on the 2DRMP code is ongoing, with a paper planned for submission in the summer to Computer Physics Communications. The paper will detail the modifications made to the 2DRMP code, and demonstrate its capabilities on a large-scale test case. Towards this end, further work is planned. This work will include: a more comprehensive benchmarking of the PLASMA 2DRMP code on ARM and, potentially, AMD machines; inclusion of a tuning program (rightsizer) to optimise linear algebra parameters within PLASMA, such as optimal tile sizes and per-tile parallelism during LU factorisation; and creation of a user guide to allow potential end-users to access the software quickly and efficiently. In addition we plan to include results from a single large-scale test case, in order to demonstrate the capability of the software to scale to large problems that may be otherwise impossible to analyse. We are working with Andrew Sunderland at STFC Daresbury and Stan Scott at Queen's University Belfast in order to design a test case that would be of interest to the physics community, and also showcase the capabilities of the 2DRMP software from a numerical linear algebra perspective. In addition to the planned paper, a poster will be presented at the *Advances in Numerical Linear Algebra Conference and James Hardy Wilkinson Centenary* at the University of Manchester, May 29-30$^{\text{th}}$ this year.

3 Load flow in large scale power systems

The main NLAFET contact is Iain Duff and the main applications contact is Bernd Klöss of DigSILENT GmbH, Germany.

In this section we present performance results on test cases for the Power Systems application provided by Bernd Klöss using both the ParSHUM library and the BC library. We discussed earlier work on the smaller matrices from these test problems in Deliverable 5.1.

The ParSHUM library is a parallel multithreaded library for the factorization of highly unsymmetric matrices. For a given sparse matrix A, our solver decomposes the matrix into the form

$$PAQ = LU,$$

where P and Q are row and column permutation matrices respectively, and L and U are sparse lower and upper triangular matrices, respectively. For further information about this library, please refer to Deliverables 3.4 and 3.5. The major enhancement to ParSHUM since these earlier deliverables has been to first partition the matrix to singly bordered block

diagonal (SBBD) form using Zoltan [7]. We show this form in Figure 6. We assign one MPI

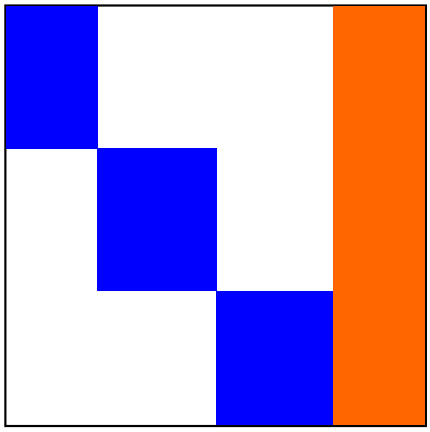
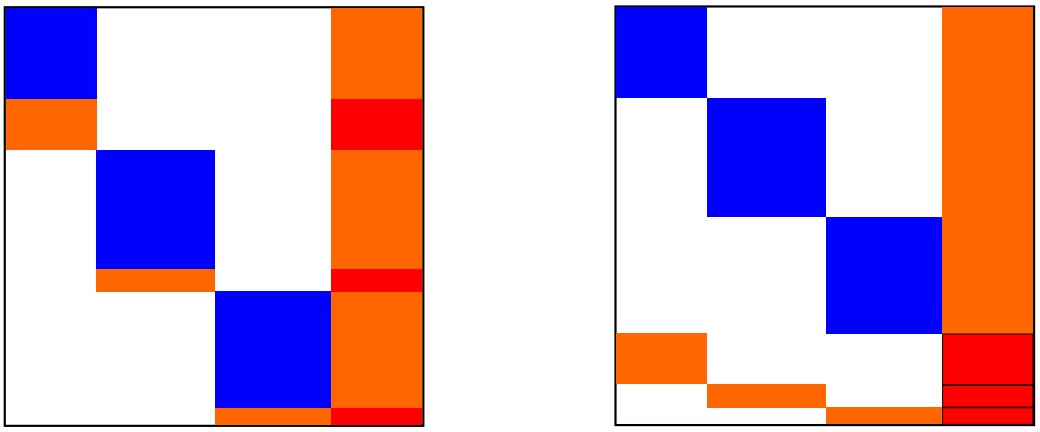


Figure 6: A matrix in singly bordered block diagonal form.

process to each diagonal block and then factorize these blocks simultaneously using the core multithreaded ParSHUM code. Each diagonal block is factorized and each forms a local Schur complement (shown in red in Figure 7a). Then the global Schur complement is formed on the process 0 by combining all the local Schur complements (see Figure 7b). This global Schur is then factorized using the dense solver from PLASMA [14]. We thus exploit two levels of parallelism: on a distributed level by performing the factorization of each diagonal block concurrently, and in shared memory through the multithreaded ParSHUM solver applied to each block.



(a) (b)

Figure 7: Factorization of SBBD form.

The BC code is described in Deliverable 3.7 and again exploits parallelism at both distributed and shared memory levels. For the experiments in this deliverable we use the

row partitioning version of the code. The system $Ax = b$ is partitioned by blocks of rows as:

$$\begin{pmatrix} A_1 \\ A_2 \\ . \\ . \\ . \\ A_p \end{pmatrix} x = \begin{pmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_p \end{pmatrix}.$$

The block Cimmino method, on which our algorithm is based, then computes a solution iteratively from an initial estimate $x^{(0)}$ according to:

$$u_i = A_i^+ \left(b_i - A_i x^{(k)} \right) \quad i = 1, ...p \tag{3.1}$$

$$x^{(k+1)} = x^{(k)} + \omega \sum_{i=1}^{p} u_i, \tag{3.2}$$

where we note the independence of the set of p equations. Since we assume that A is full rank, so are the A_i and $A_i^+ = A_i^T (A_i A_i^T)^{-1}$. Elfving [19] proves that this method converges if ω is sufficiently small. The algorithm can thus be slow to converge. We can accelerate this in two ways. One is to use a numerically aware partitioning [38] that makes the manifolds corresponding to the A_i close to orthogonal to increase the value of ω necessary for convergence and the other is to note that the iteration equations can be written as:

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + \omega \sum_{i=1}^{p} A_i^+ \left(b_i - A_i x^{(k)} \right) \\ &= \left(I - \omega \sum_{i=1}^{p} A_i^+ A_i \right) x^{(k)} + \omega \sum_{i=1}^{p} A_i^+ b_i \\ &= Q x^{(k)} + \omega \sum_{i=1}^{p} A_i^+ b_i, \end{aligned}$$

and we can write the fixed point iteration as

$$Hx = \xi, \tag{3.3}$$

where $H = I - Q$. Since we assume that the matrices A_i have full row rank, the matrix $H = \omega \sum_{i=1}^{p} A_i^+ A_i$ is a sum of projection matrices and is thus positive definite. Therefore the system (3.3) can be solved by the conjugate gradient method. We note that, since $\xi = \omega \sum_{i=1}^{p} A_i^+ b_i$, the parameter ω appears on both sides of equation (3.3) so we can arbitrarily set it to one.

The resulting algorithm is a classic hybrid method coupling iterative and direct solvers since we solve the systems (3.1) using our direct solver SpLDLT from the S*y*LVER library (see Deliverable 3.3) on the augmented systems

$$\begin{pmatrix} I & A_i^T \\ A_i & 0 \end{pmatrix} \begin{pmatrix} u_i \\ v_i \end{pmatrix} = \begin{pmatrix} 0 \\ r_i \end{pmatrix} \tag{3.4}$$

formed by each block row A_i, where $r_i = b_i - A_i x^{(k)}$. In our implementation we use the independence of these p equations to exploit distributed memory while our direct solver is designed to exploit many core systems.

3.1 The test matrices and test environment

We follow [21] in defining the symmetry index si by the expression

$$si(A) = \frac{number_{i \neq j}\{a_{ij} * a_{ji} \neq 0\}}{nnz\{A\}},$$

where $nnz\{A\}$ is the number of off-diagonal entries in the matrix A. A symmetric matrix will thus have a symmetry index of 1.0. We define 0/0 to have the value 1.0 so that a diagonal matrix will be symmetric. A triangular matrix will have symmetry index zero. Our experiments suggest that matrices with symmetry indices of less than 0.9 can be considered highly unsymmetric. Additionally, we define the *fill-in factor* as the number of entries in the L and U factors divided by the number of entries in A, viz.

$$\frac{nnz\{L\} + nnz\{U\}}{nnz\{A\}}.$$

The main characteristics for the matrices used in this study are presented in Table 1. The first six matrices were used in Deliverable 5.1. For the experiments for this deliverable we asked Bernd Klöss for a much larger matrix to try and stress our codes. This matrix, which we call Newton_detailed, is of dimension 7 355 436 and comes from the first loop of an unsymmetric load flow calculation when solving a nonlinear system. It is particularly big for a power system matrix, because detailed substations are modelled in the nonlinear system, rather than post-processing the solution for equivalent busbars.

Matrix		n (10^3)	nnz (10^3)	si
InnerLoop1	Balanced load flow	197	745	0.44
InnerLoop2	Balanced load flow	197	806	0.46
InnerLoop3	Balanced load flow	197	806	0.46
InnerLoop4	Balanced load flow	197	806	0.46
Jacobian_unbalancedLdf	Unbalanced load flow	203	2410	0.80
Newton_Iteration1	Balanced optimal power flow	427	2380	0.14
Newton_detailed	Unbalanced load flow	7355	23741	0.29

Table 1: Statistics for the matrices that are used in this study. The size (n), number of nonzero entries (nnz) and the symmetry index (si) are given for each matrix.

Our tests were all performed on the Kebnekaise system located at the High Performance Computing Center North (HPC2N), Umeå University[1]. Each compute node contains 28 Intel Xeon E5-2690v4 cores organised into 2 NUMA islands with 14 cores in each. The nodes are connected with a FDR Infiniband Network. Each CPU core has 32 KB L1 data cache, 32 KB L1 instruction cache and 256 KB L2 cache. Moreover, for every NUMA island there is 35 MB of shared L3 cache. The total amount of RAM per compute node is 128 GB.

[1] See https://www.hpc2n.umu.se/resources/hardware/kebnekaise.

3.2 Performance results using ParSHUM

In Deliverable 5.2, we compared the performance of ParSHUM with two state-of-the-art solvers for unsymmetric matrices: MA48 [18] from the HSL library and UMFPACK [15] from the SuiteSparse library. We showed that our code in general outperformed these codes. Here, we first study the performance of our solver in distributed memory and then compare it to two major state-of-the-art parallel solvers for unsymmetric problems for distributed memory, MUMPS [2] and SuperLU [17].

The times presented in the tables in this subsection correspond to the time spent in the numerical factorization step only, once the data is partitioned and distributed to the corresponding nodes. Each MPI process is mapped on a NUMA node and runs on fourteen cores.

We first investigate the parallel behaviour of ParSHUM. In Table 2, the times when varying the number of partitions are presented. If this method has an Achilles heel, it is that the size of the Schur complement can become quite large when the number of partitions increases. We show these sizes for our test problems in Table 3 where we note that, although this does grow with the number of partitions, it is still very low compared to the size of the original problem. At the moment, we use a dense solver to factorize the Schur complement although it is still a sparse matrix. Our future work will look at ways to exploit this.

For the largest matrix, Newton_detailed, a speed-up of 6.7 is obtained when eight NUMA nodes are used. For the smaller matrices, the performance is limited by the granularity, resulting in a speed-up of only 2.6 for the InnerLoop1 matrix. Furthermore, the Newton_Iteration1 matrix, in addition to its small size, suffers from load imbalance among the partitions and so achieves a speed-up of only 2.

	#MPI processes			
Matrix	1	2	4	8
InnerLoop1	0.41	0.32	0.24	0.16
InnerLoop2	0.38	0.32	0.27	0.17
InnerLoop3	0.41	0.28	0.24	0.17
InnerLoop4	0.41	0.30	0.27	0.16
Jacobian_unbalancedLdf	2.29	1.50	1.19	0.68
Newton_Iteration1	1.03	0.87	0.61	0.51
Newton_detailed	10.9	4.90	2.75	1.63

Table 2: The execution time in seconds for ParSHUM on the matrices presented in Table 1 partitioned in SBBD form.

In Table 4 the execution times and the fill-in factors for ParSHUM, MUMPS, and SuperLU are presented. We obtain the lowest execution times with ParSHUM for all the matrices except the Jacobian_unbalancedLdf, and Newton_Iteration1 matrices for which the MUMPS solver yields the lowest execution time. There is only one case in which we require more memory than the other solvers showing that our algorithms better accommodate high asymmetry. In summary, our code is at least competitive with other state-of-the art codes on matrices from this application, and it is particularly gratifying that we do so well on the largest test problem.

	#MPI processes		
Matrix	2	4	8
InnerLoop1	96	232	459
InnerLoop2	89	189	486
InnerLoop3	114	255	440
InnerLoop4	114	255	440
Jacobian_unbalancedLdf	662	1208	1954
Newton_Iteration1	192	426	978
Newton_detailed	240	918	1996

Table 3: The order of the global Schur matrices for the matrices presented in Table 1 in SBBD form.

	ParSHUM		MUMPS		SuperLU	
Matrix	time	fill-in	time	fill-in	time	fill-in
InnerLoop1	**0.16**	3.23	0.61	3.90	0.91	6.02
InnerLoop2	**0.17**	2.94	0.34	3.72	0.92	5.57
InnerLoop3	**0.17**	2.96	0.33	3.72	0.88	5.60
InnerLoop4	**0.16**	2.50	0.39	3.73	0.85	5.56
Jacobian_unbalancedLdf	0.68	7.03	**0.35**	3.45	0.86	3.59
Newton_Iteration1	0.51	3.57	**0.43**	6.02	2.00	5.60
Newton_detailed	**1.63**	5.32	11.6	6.09	28.0	5.39

Table 4: The execution time and the fill-in factor for ParSHUM, MUMPS, and SuperLU solvers on four nodes (that is with 8 processes). The best results are highlighted in bold.

3.3 Performance results using BC

We ran tests on Kebnekaise with the same number of partitions as MPI processes. The four smallest matrices were too small to benefit from this method so we show, in Table 5, some results from the matrices Jacobian_unbalancedLdf and Newton_Iteration1 on from 2 to 16 processes. Although we do see a reduction in time as we increase the number of processes, it is at best linear and we might have expected better performance. We stopped the iterations when our backward error was less than 10^{-6} and we show the number of these iterations in the column niter. The time in the column Facto is the elapsed time for all the factorizations to complete. As the size of the matrices involved decreases as we increase the number of partitions we would have expected these values to decrease as we increase the number of partitions. Clearly that is not happening, certainly not to the extent that we would expect. It is this factor that affects the total time so we investigate this further on the largest matrix in our test set.

Problem	npart	niter	Facto	SumProject ($\times 10^{-1}$)	BCG ($\times 10^{-1}$)	Total Time	Backward ($\times 10^{-7}$)
jacobian_unbalanced	2	14	26.9	9.19	10.3	30.8	9.29
	4	11	23.7	4.19	5.19	26.3	9.79
	8	31	7.01	6.23	9.65	9.65	9.69
	16	27	12.1	4.03	7.60	14.7	9.69
newton_iteration1	2	8	4.78	8.26	10.6	9.75	9.05
	4	16	3.38	8.42	12.8	6.99	9.65
	8	13	3.43	4.43	9.08	6.13	8.80
	16	11	2.72	2.68	7.01	5.25	9.83

Table 5: Execution times in seconds and backward errors for matrices Jacobian_unbalancedLdf and Newton_Iteration1 for differing numbers of partitions.

We thus examine in more detail runs on the matrix Newton_detailed shown in Tables 6 to 9 where we increase the number of processes from 2 to 64. This problem was considered large for these applications although the power of our approach would be more evident on even larger problems when the power of our multicore sparse direct solver, SpLDLT, on the augmented indefinite systems on each partition would be more evident.

npart	n($AugA_i$)		nnz($AugA_i$)	
	mean($\times 10^5$)	max/min	mean($\times 10^5$)	max/min
2	73.6	1.11	156	1.16
8	18.4	1.06	38.9	1.29
16	9.20	1.02	19.4	1.23
32	4.60	1.11	9.72	1.27
64	2.30	1.07	4.86	1.35

Table 6: Statistics on the size of the augmented systems for the Newton_detailed matrix for differing numbers of partitions.

When we study Table 6 all is as expected. As we increase the number of partitions, the average row order and number of entries for each partition decreases linearly and the partitions are quite well balanced for these two measures.

npart	nnz(L)		Flops	
	mean($\times 10^6$)	max / min	mean($\times 10^7$)	max/min
2	74.8	1.19	371	1.40
8	18.5	1.53	85.2	2.17
16	9.25	1.57	41.5	4.51
32	4.52	1.76	18.3	8.88
64	2.22	1.96	8.10	13.7

Table 7: Size of the factors and number of flops to factor the augmented systems for the Newton_detailed matrix for differing numbers of partitions.

When we study the data in Table 7, we notice that this linearity is quite well conserved in that the mean values for both entries in the factors and the number of floating point operations for the factorizations are roughly halved when the number of partitions increases

by two. However, the load balance is significantly worse, particularly for the number of floating-point operations. We note that the figures in this table are obtained from the analysis phase of the direct solver and so do not take into account additional numerical pivoting needed for a stable factorization of the indefinite systems.

npart	Time to factor	
	mean	max/min
2	19.1	2.66
8	3.02	3.28
16	1.37	5.07
32	0.671	3.38
64	0.728	30.0

Table 8: Time to factor the augmented systems for the Newton_detailed matrix for differing numbers of partitions.

We thus show, in Table 8, the mean times for the actual factorizations and the ratio of the maximum and minimum of these times over the partitions. It is quite clear from these results that the imbalance becomes a real problem and severely degrades our elapsed time for computing all the factorizations in parallel and we see the effect of this in the times in Table 9.

npart	niter	Facto	SumProject	BCG	BC Time	Backward (10^{-9})
2	102	27.7	165	271	358	9.73
8	105	6.10	49.4	163	185	9.89
16	107	3.66	23.5	141	153	9.65
32	131	1.51	17.9	158	166	9.93
64	107	5.28	9.62	125	138	9.93

Table 9: Execution times and backward errors for the BC method for the Newton_detailed matrix for differing numbers of partitions.

We investigated this further and found that our imbalance was caused by the pattern of the matrices coming from our power system applications. In every case there was one block of rows that was denser than the rest of the matrix. They would naturally be held in one partition since the partitioning algorithms try to reduce the interactions between blocks. This is true whether we use a symbolic or a numerically aware partitioning. When we examined the distribution of the factorization times more closely we found that, in every case, there was only a single partition requiring the maximum time and all other partitions required much less time. In fact, the maximum time was always associated with the block containing these denser rows. The issue concerns an intrinsic aspect of multifrontal methods. As the name suggests there are, during the factorization, several fronts that await assembly later in the factorization and the structure of an augmented system where the A_i have relatively dense blocks of rows means that there could be very many fronts awaiting assembly. We discuss a possible way of overcoming this problem in Section 6.

Although it is not evident in our earlier results, we encountered another major problem caused by matrices from the application that we were happily able to overcome. The problem was that our direct solver was reporting that the matrices it was factorizing were singular and so the computation was stopped. This was happening because the condition number of the matrices in Table 1 is very high (over 10^{10}) and simple scaling techniques do not reduce this. The direct solver is solving the systems (3.4). The condition number of the coefficient matrix is the square of that of A_i. Since the conditioning of these matrices will reflect that of the matrix A, the condition number of the augmented systems will often be greater than $1/\epsilon$ where ϵ is the double precision round off, and so the direct solver will find these matrices to be singular. In order to avoid this problem, we perform a scaling by multiplying the identity matrix in equation (3.4) by α which ideally should be around the smallest singular value of the matrix A.

4 Communication avoiding iterative methods for solving linear systems arising from a few applications, in particular linear elasticity problems

In this section we describe the efficiency of the enlarged CG method (ECG) on matrices arising from several different applications. We focus in particular on matrices arising from solving linear elasticity problems, since these matrices are known to be difficult to solve by iterative methods. Linear elasticity problems arise in structural analysis. The enlarged CG method was developed at Inria in the context of WP 4 and was described in detail in Deliverables 4.2 to 4.5. Several parallel performance results were already presented in these deliverables. However for completeness we recall some of them here, in addition to several novel performance results.

Enlarged CG is an iterative method that relies on adding t vectors to the Krylov subspace at each iteration of the iterative method. We refer to t as the enlarging factor. Two variants of enlarged CG were introduced in [25] and [26] that use different approaches for orthogonalizing the vectors of the Krylov subspace. They are referred to as Orthodir (Odir) and Orthomin (Omin). We have also shown the explicit link between the two methods and we have studied theoretically the convergence behaviour of ECG. We greatly improved the previous result in [22], and showed that ECG acts as if the smallest eigenvalues were somehow deflated. In practice, we observe that enlarging the Krylov subspaces can drastically reduce the number of iterations for linear elasticity problems. Indeed in the numerical experiments ECG is used with a block Jacobi preconditioner and acts as a second-level preconditioner that, in a way, deflates the smallest eigenvalues. This is in accordance with the theory and leads to significant speed-ups over the standard PCG for linear elasticity problems, for example.

In the experiments, we use a block Jacobi preconditioner, associating with each block an MPI process. Before calling ECG, each MPI process factorizes the diagonal block of A corresponding to the local row panel that it owns. At each iteration of ECG, each MPI process performs a backward and forward solve locally in order to apply the preconditioner. Hence the application of the preconditioner does not need any communication. It is likely that there exist better preconditioners than block Jacobi for our test cases, however we are

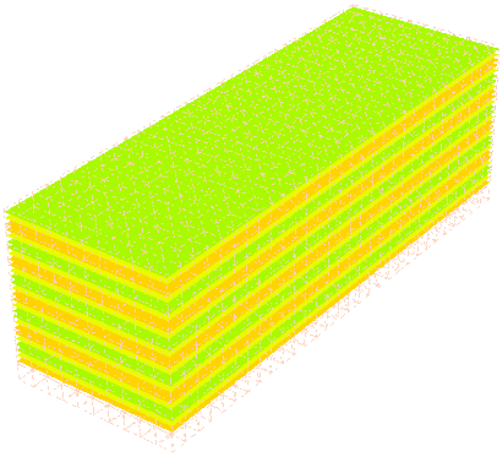


Figure 8: Heterogeneity pattern of the Young's modulus (E) and Poisson's ratio (ν) for elasticity matrices.

interested in using a highly parallel preconditioner. Although in theory it is possible to apply any preconditioner within this implementation, in practice it is essential that applying this preconditioner to several vectors at the same time is not too costly, *e.g.* a sublinear complexity with respect to the number of vectors.

The following experiments were performed on the Kebnekaise system located at the High Performance Computing Center North (HPC2N), Umeå University. It is a heterogeneous machine formed by a mix of Intel Xeon E5-2690v4 (Broadwell) with 2x14 cores (and E7-8860v4 for large memory computations), Nvidia K80 GPU and Intel Xeon Phi 7250 (Knight's Landing) with 68 cores. In our experiments, we use the so-called compute nodes, which are formed by Intel Xeon E5-2690v4 (Broadwell) with 2x14 cores. For a detailed description of the machine, we refer to the online documentation[2].

We compile the code (and its dependencies) using the Intel toolchain installed on the machine: `mpiicc` (based on `icc` version 18.0.1 20171018) and MKL [39] version 2018.1.163. We use PETSc [3] in order to compare the ECG implementation with the PETSc PCG implementation. In particular, PETSc is configured to use MKL-PARDISO as the exact solver for sparse matrices in the block Jacobi preconditioner. For partitioning the matrix we use the METIS library downloaded and installed by PETSc.

4.1 Test cases

The elasticity matrices, Ela_x, arise from the linear elasticity problem with Dirichlet and Neumann boundary conditions defined as follows

$$\operatorname{div}(\sigma(u)) + f = 0 \qquad \text{on } \Omega \tag{4.1}$$

$$u = 0 \qquad \text{on } \partial\Omega_D \tag{4.2}$$

$$\sigma(u) \cdot n = 0 \qquad \text{on } \partial\Omega_N \tag{4.3}$$

where Ω is some regular domain, e.g, a parallelepiped. $\partial\Omega_D$ is the Dirichlet boundary, $\partial\Omega_N$ is the Neumann boundary, f is some body force, and u is the unknown displacement field. We

[2] https://www.hpc2n.umu.se/resources/hardware/kebnekaise

Name	Size	Nonzeros	Problem
Hook_1498	1,498,023	59,374,451	Structural problem
Flan_1565	1,564,794	117,406,044	Structural problem
Bump_2911	2,911,419	130,378,257	Reservoir simulation
Queen_4147	4,147,110	316,548,962	Structural problem
Ela_1	615,168	21,373,272	Linear elasticity
Ela_2	1,210,800	42,611,160	Linear elasticity
Ela_3	2,383,125	84,726,039	Linear elasticity
Ela_4	4,615,683	165,388,197	Linear elasticity

Table 10: Test matrices.

denote by $\sigma(.)$ the Cauchy stress tensor given by Hooke's law: it can be expressed in terms of Young's Modulus E and Poisson's ratio ν. For a more detailed description of the problem see [23]. We consider a heterogeneous beam made of several layers of a hard material $(E_1, \nu_1) = (2 \times 10^{11}, 0.25)$ and a soft material $(E_2, \nu_2) = (10^7, 0.45)$, i.e., discontinuous E and ν (Figure 8). The matrices Ela_N in Table 10 correspond to this equation on a beam discretized with FreeFem++ [27] using a triangular mesh that is refined as N increases, and a P1 finite elements scheme. More precisely, the mesh used for generating the Ela_4 matrix contains $1600 \times 30 \times 30$ points on the corresponding vertices. This mesh is coarsened by dividing the number of vertices in each dimension by $2^{1/3}$ in order to construct the Ela_3 matrix, and so on and so forth for the Ela_2 and Ela_1 matrices. This test case is known to be difficult because the matrix is ill conditioned. In particular, the standard one-level preconditioners are not expected to be very effective.

As previously pointed out, ECG is an algebraic method that does not rely on any particular assumption on the matrix, except that it is symmetric positive definite. As an illustration, we also test the implementation on the four largest SPD matrices coming from the Sparse Matrix Collection of Tim Davis [16]. Numerical properties of the test matrices are summarized in Table 10.

In all the experiments the tolerance is set as the default tolerance of PETSc, i.e., 10^{-5} and the maximum number of iterations is set to 25,000. The right-hand side is chosen uniformly random and normalized and the initial guess is set to 0. We do not use any kind of threading and use 28 MPI processes per node. Unless otherwise stated, we use one OpenMP thread per MPI process — we also perform numerical experiments to observe the effect of threading in Section 4.4.

4.2 Strong scaling study

In this section we perform a strong scaling study on Hook_1498, Flan_1565, Queen_4147, and Ela_4, the results are presented in Figure 9.

For Bump_2911 and Queen_4147, we observe that the runtime is increasing when the number of vectors t added to the Krylov subspace at each iteration increases. In this case, the cause is inherent to the method: the number of iterations is not decreasing enough to

compensate the increase in runtime of one iteration when the value of t increases. This is particularly illustrated on Queen_4147, with Odir from $t = 2$ to $t = 28$, the number of iterations is decreased by only 20%. For the sake of brevity, we omit the results obtained with Bump_2911 because they are similar to those obtained with Queen_4147. There is an interplay between the choice of t and the number of MPI processes. Indeed, increasing the number of MPI processes deteriorates the quality of the preconditioner, and reduces its application cost. However, for the sake of simplicity, we decide to keep the value of t constant while increasing the number of MPI processes.

More precisely, we compare PETSc PCG (blue bars), ECG (orange bars), and a modified PETSc PCG (green bars) where the sparse matrix-vector is applied using `MatMatMult` routine from PETSc, i.e. the vector is regarded as a dense matrix with one column. For Hook_1498, we use Orthodir with a dynamic detection of useful vectors to be added to the Krylov subspace (referred to as D-Odir) and set $t = 10$, which corresponds to one of the best choice over the values of t we tested in our studies. We observe that D-Odir is faster than PETSc PCG when the number of MPI processes is relatively low (252 and 504), but when it is large (more than 1,000) PETSc PCG becomes almost twice as fast. For the other matrices, we also observe that the performance of ECG deteriorates significantly with respect to PETSc PCG when the number of MPI processes becomes large. This is because the routine `MatMatMult` is not optimized when the number of MPI processes is large and the number of columns in the right-hand side is very small. For example, for Hook_1498 and Flan_1565, when $P = 2,016$, the `MatMult` routine (sparse matrix-vector product) is around 10 times faster than the `MatMatMult` routine where the right-hand side is regarded as a dense matrix with one column. The total runtime per right-hand side of `MatMatMult` is indeed slightly lower than the runtime of `MatMult` when the number of right-hand sides is large enough; in this case, the total runtime of `MatMatMult` is significantly larger. However if the number of right-hand sides is not large enough — which is the case in our strong scaling study — then the runtime per right-hand side is larger than the runtime of the `MatMult` routine. Furthermore, the gap between the performance of `MatMatMult` and `MatMult` routines increases when the number of MPI processes increases.

Hence, we also compare ECG with a modified PETSc PCG (green bars) where the routine `MatMatMult` is used for computing the sparse matrix-vector product. We believe that this comparison is relevant because both ECG and this modified PETSc PCG rely on the exact same routine for computing the sparse matrix application to a (set of) vector(s). We indeed observe that this modified version of PETSc PCG is less scalable than the default one, for example for Hook_1498 the runtime increases from 1,008 to 2,016 MPI processes. Furthermore, we observe that this modified version of PETSc PCG is also less scalable than ECG; for Hook_1498, Flan_1565, and Ela_4 the speed-up is slightly increasing from 1,008 to 2,016 MPI processes.

In conclusion, we have shown that ECG's scaling is highly dependent of the routine that performs the sparse matrix-set of vectors product. This is of course not very surprising. What is more surprising, however, is the fact that the `MatMult` routine of PETSc is much more scalable than the `MatMatMult` when the number of right-hand sides is small. In practice, this explains the difference in terms of scalability of ECG compared to PETSc PCG. We believe that it should be possible to optimize the `MatMatMult` routine so that this difference would at least be reduced, or even be removed. Also, we want to emphasize that enlarging the

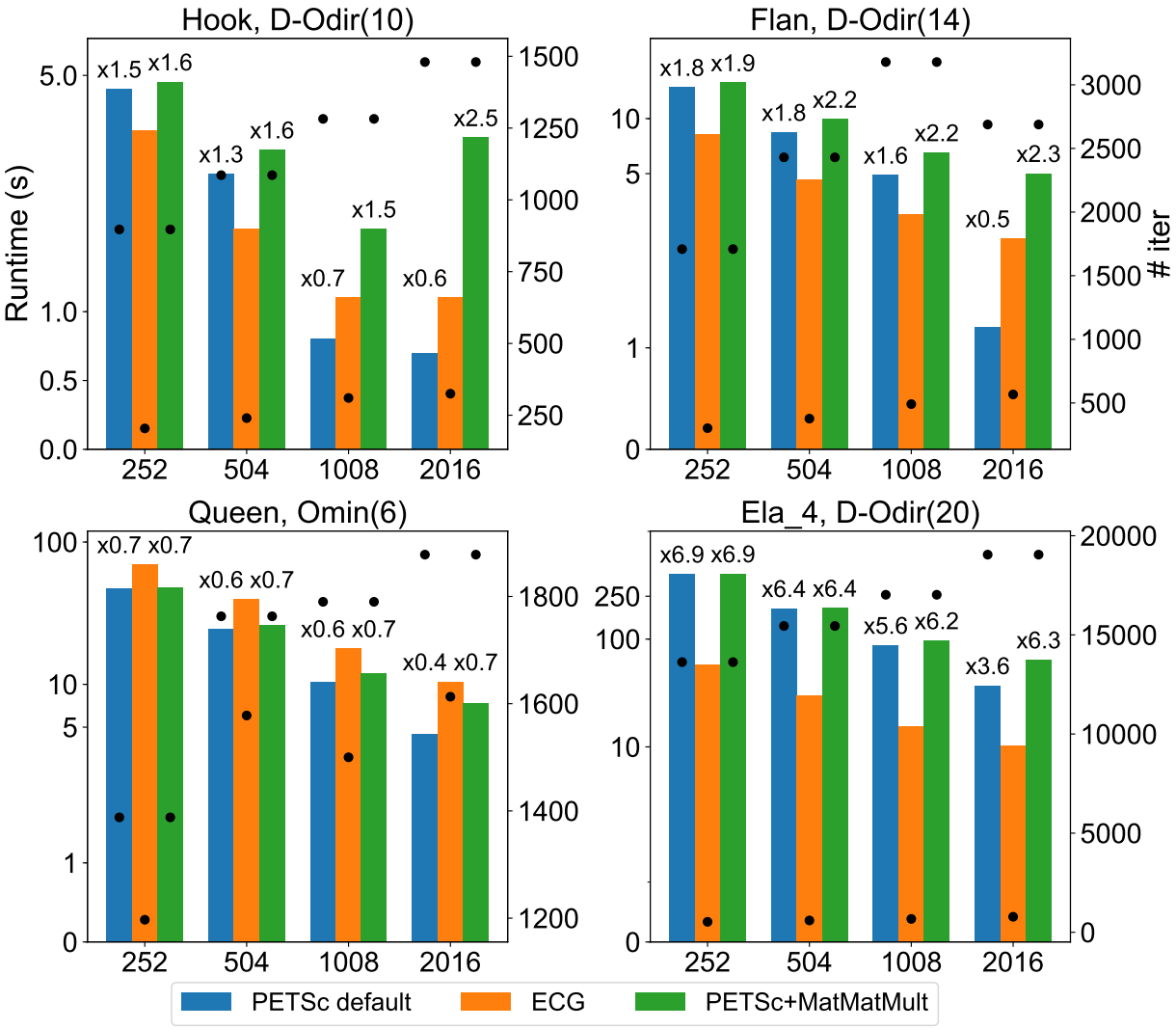


Figure 9: Strong scaling results for Hook_1498, Flan_1565, Queen_4147, and Ela_4 with P varying from 252 to 2016. We indicate both the runtimes, with bars (left scale), and the iteration counts, with black dots (right scale). The speed-up of ECG with respect to PETSc and PETSc using `MatMatMult` routine is indicated on top of the bars. The orthogonalization method used for each matrix, Omin(t) or D-Odir(t), is also displayed, where t represents the enlarging factor.

			D-Odir		PETSc CG		
# MPI	n	t	# iter	T_{tot}	# iter	T_{def}	T_{MMM}
252	6.15×10^5	20	360	3.4	8,803	13.0 (x3.8)	16.3 (x4.8)
504	1.21×10^6	20	463	4.9	11,333	17.6 (x3.6)	23.7 (x4.8)
1008	2.38×10^6	20	608	6.8	14,801	23.8 (x3.5)	36.8 (x5.4)
2016	4.61×10^6	20	784	10.2	19,047	36.1 (x3.5)	64.0 (x6.4)

Table 11: Weak scaling study. The dimension of the matrix is denoted n, and t denotes the enlarging factor. The ratio between PETSc runtime and ECG runtime is indicated in parentheses.

Krylov subspaces is not incompatible with other techniques currently developed in order to increase the performances of Krylov methods. For instance, we mention that we are currently performing 4 calls to `MPI_Allreduce` per iteration, but that could be reduced to 2, and even 1 with Odir (and D-Odir), by fusing them. This technique is described in detail in Deliverable 4.5, *Integration*.

4.3 Dependence on the mesh size — weak scaling study

Given the importance of the enlarging factor t for the efficiency of the method, we perform a study of convergence with respect to the mesh size for the elasticity test case. More precisely, we consider the Ela_N matrices ($N = 1, \ldots, 4$). Our focus is not only the weak scaling of ECG, but also the comparison between PETSc's CG and D-Odir in terms of runtime.

As for the strong scaling study, we use D-Odir(20). The results are summarized in Table 11. We observe that D-Odir(20) is always at least 3.5 times faster than PETSc PCG. However the gap tends to slightly decrease when the number of MPI processes increases. Indeed, when $P = 256$ D-Odir(20) is up to 3.8 times faster than PETSc PCG. As outlined in the strong scaling study, the `MatMatMult` routine scales poorly (compared to the `MatMult` routine) when the number of columns in the right-hand side is very small. Thus, we also indicate, in the column labelled T_{MMM}, the runtimes when replacing the sparse matrix-vector product within PETSc PCG by a call to `MatMatMult`. In this case, the performance gap between D-Odir(20) and the modified PETSc PCG increases when the number of MPI processes increases. For instance, when $P = 2016$, D-Odir(20) is around 6.5 times faster than the modified PETSc PCG, whereas it is 4.8 times faster for the smallest problem.

4.4 Impact of threads on performance

One motivation for enlarging the Krylov subspaces is to increase the arithmetic intensity of the resulting methods. This is of particular interest when taking advantage of the so-called manycore architecture as Nvidia GPUs, Intel Xeon Phi, or Sunway SW26010 used in the Sunway TaihuLight supercomputer. As the implementation relies on the MKL library which is multi-threaded [39], it is straightforward to assess its efficiency on the Xeon Phi processors.

In order to do so, we performed the following experiments on NERSC's supercomputer

	PETSc PCG		Odir(20)	
# threads	T_{tot}	speed-up	T_{tot}	speed-up
1	83.3	-	32.6	-
2	75.4	1.1	25.0	1.3
4	62.5	1.3	19.5	1.7
8	56.1	1.5	16.6	2.0

Table 12: Runtime results (in seconds) on the Ela_4 matrix when $P = 2,048$. We indicate the speed-up when increasing the number of threads for each method.

Cori. It consists of two partitions, one with Intel Haswell processors and another with the last generation of Intel Xeon Phi processors: Knights Landing (KNL). More precisely, the second partition consists of 9,688 single-socket Intel Xeon Phi 7250 (KNL) processors with 68 cores each. For a detailed description of the machine, we refer to the online documentation[3]. We compile the code (and its dependencies) using the default compilers and libraries installed on the machine: `icc` version 18.0.1, `cray-mpich` version 7.6.2, MKL version 2018.1.163 and METIS version 5.1.0. We have installed PETSc and, as for Kebnekaise, it has been linked with MKL so that it uses MKL-PARDISO in the block Jacobi preconditioner. We consider the Ela_4 test case and study the impact of the number of threads on the strong scaling of Odir(20). We do not use the dynamic reduction of the search directions in order to keep the cost of one iteration constant during the solve to better understand the effect of threading. We fix the number of MPI processes to 2048 and increase the number of threads from 1 to 8 — this means at most $2,048 \times 8 = 16,384$ threads, each one being bound to one physical core.

The results obtained are summarized in Table 12. We observe that using more than 2 threads, and up to 8, has always a significant effect on the speed-up, even when the number of MPI processes is high. For instance, as shown in Table 12, increasing the number of threads from 1 to 8 with a fixed number of $2,048$ MPI processes leads to a decrease in runtime of 2. Of course, we are not close to full efficiency when using multiple threads, but we are still taking advantage of the Level 3 BLAS routines. Indeed, the corresponding speed-up with PETSc PCG is only 1.5. In particular, we observe that the difference between ECG's speed-up and PETSc PCG's speed-up increases when the number of threads increases. Thus, ECG is more adapted to the current trend in hardware architecture for reaching exascale, namely manycore processors.

For instance for a 3D linear elasticity problem with heterogeneous coefficients with 4.5 millions of unknowns and 165 millions of nonzero entries, we observe that ECG is up to 5.7 times faster than the PETSc implementation of PCG, both using a block Jacobi preconditioner. This test case is known to be difficult because the standard one-level preconditioners are not expected to be very effective.

[3] http://www.nersc.gov/users/computational-systems/cori/configuration/

5 Data analysis in astrophysics and Midapack

Studies of the minute fluctuations of the intensity and polarization of primordial photons, called CMB for Cosmic Microwave Background, have been for nearly 25 years one of the main sources of invaluable information about our universe and the fundamental laws of physics. Astrophysicists produce and analyse multi-frequency images of the universe when it was 5% of its current age. The new generation of CMB experiments produces overwhelmingly large and complex data sets by observing the sky with thousands of detectors over long time periods (many years). For example, Planck – a keystone satellite mission which has been developed under the auspices of the European Space Agency (ESA) – has been surveying the sky since 2010. Planck produces Terabytes of data and requires 100 Petaflops to compute each image of the universe. The volumes of the data sets produced over the last two decades grow consistently at Moore's Law rate.

One of the main computational challenges in the CMB data analysis involves reconstructing a 2-dimensional map of the sky from enormous data sets. This problem, called a *map-making problem*, is difficult for a number of reasons, including data volumes, long-term noise correlations and presence of parasitic signals. With current algorithms, the reconstruction of the sky maps from the data available by early 2020 is expected to require 100 Exaflops. In the analysis of any CMB data, the map-making step may need to be performed hundreds if not thousands of times as, for instance, is the case whenever popular Monte Carlo sampling algorithms are used to characterize the uncertainties of the estimated sky maps.

Given the size of linear systems solved at each map-making step, the approximate solution is most readily obtained using iterative algorithms (in particular Krylov subspace methods), and is facilitated by exploiting efficient methods for multiplying the system matrix by a vector. For our experiments we consider the MIDAPACK library[4]. The volumes of the data and the requirement to process the entire data set in one step in order to deal with and properly account for the long-term noise correlations, require the use of the largest available supercomputers with their complex architectures and communication networks and appropriate numerical algorithms.

In the context of Workpackage 5, our goal was to adapt and validate communication-avoiding iterative methods for the CMB map-making problem and to make them available to the CMB community by integrating them with the MIDAPACK package. Previous experiments show that the communication is the bottleneck that prevents scaling the map making problem to a very large number of processors; see, e.g., [24, 37].

We search for novel preconditioning techniques, which are suitable for single runs but also for preconditioners appropriate for multiple sequential solves with different right-hand sides, where the information from the previous runs is accumulated and exploited in the forthcoming runs or a precomputation is used to construct a better and more efficient preconditioner. In either case, the methods we are seeking should lead to reducing the overall time-to-solution. Thus the required preconditioners need to be cheap to construct and apply in a massively parallel context appropriate for modern computer architectures.

Our experiments were performed on National Energy Research Scientific Computing

[4] www.apc.univ-paris7.fr/APC_CS/Recherche/Adamis/MIDAS09/software/midapack/ver1.1/index.html

Center (NERSC) Cori Machine and on High Performance Computing Center North (HPC2N) at Umeå University machine Kebnekaise.

5.1 Map-making problem

The observed data is modelled as

$$d = Pm + n, \tag{5.1}$$

where d stands for a vector of all measurements with dimension n_t up to 10^{15}, m is an unknown pixelized map of the sky signal of the order typically 10^6–10^8 and n is the (instrumental) noise. The *pointing matrix* P describes, which pixel of the map is observed at each time, it is very sparse, tall and skinny.

Then, the maximum-likelihood is computed as

$$(P^t N^{-1} P) m_{ML} = P^t N^{-1} d, \tag{5.2}$$

where the weight matrix N^{-1}, of size n_t and structured (typically block-Toeplitz), approximates the covariance matrix of the noise n.

5.2 Data for experiments

In contrast to other communities (cf. , e.g., the Suite Sparse Collection), there does not exist a set of test cases for problems in CMB data analysis. Typically, authors generate a new data set for each of their manuscripts in order to demonstrate the behaviour of the methods in a particular setting. This makes a comparison of various solvers and preconditioners quite difficult. Additionally, we were able to run only a limited number of experiments because of having only a few data sets provided to us by our partners from APC. While generating smaller test cases (e.g., based on the observation of the small fraction of the sky and/or not involving more complex features) can be done using several freely available packages, the data for large tests require careful and time-consuming preprocessing, e.g., in order to avoid singularity of the resulting algebraic system.

We run prototyping experiments using our sequential code developed in Python on several (relatively small) test cases. The parallel experiments on Enlarged Conjugate Gradient method were run using one larger dataset. The results for more complex and even larger datasets are a subject of our further work.

5.3 State-of-the art solvers

Traditionally the iterative method of choice is Preconditioned Conjugate Gradient method (PCG) with a simple preconditioner given by

$$P^T \mathrm{diag}(N^{-1}) P. \tag{5.3}$$

Thanks to the properties of the pointing matrix, this preconditioner is block-diagonal and easy to construct and apply. It was observed that this preconditioner successfully removes the effect of large eigenvalues of the system matrix $P^T N^{-1} P$; here we can see an analogy with block-diagonal preconditioners from domain decomposition methods despite their principally different construction.

The block-diagonal preconditioner was improved by using the deflation of a few vectors approximating the eigenvectors associated with the smallest eigenvalues of the system matrix, which often harm the convergence of the PCG solver; see e.g., [24,30,33,37]. However, the construction of the deflation space can be even more costly than solving a single system with the preconditioner (5.3).

We discuss two other techniques for solving problem (5.2) in the later subsections.

5.4 Messenger-field method for map-making

Recently, the application of the so-called messenger-field technique to the map-making problem was suggested in [28]. This technique was originally proposed in [20] for the Wiener Filter, another application in CMB data analysis. Because of the strong claims in the above mentioned papers and presented numerical results, we decided to analyse this technique and test it for our simulated data.

Our findings were published in [31]. We showed that the messenger-field techniques correspond to fixed point iterations of an appropriately preconditioned initial system (5.2). In the map-making application, this preconditioner is exactly the standard one given in (5.3). We then argue that a conjugate gradient solver applied to the same preconditioned system will in general ensure at least a comparable and typically better performance in terms of the number of iterations to convergence and time-to-solution. This is illustrated in Figure 10, where the PCG preconditioned by (5.3) is compared with the messenger-field method. The figure also gives the comparison of the methods when using so-called *cooling*, a technique proposed to improve the convergence of the messenger-field; see [31] for more details.

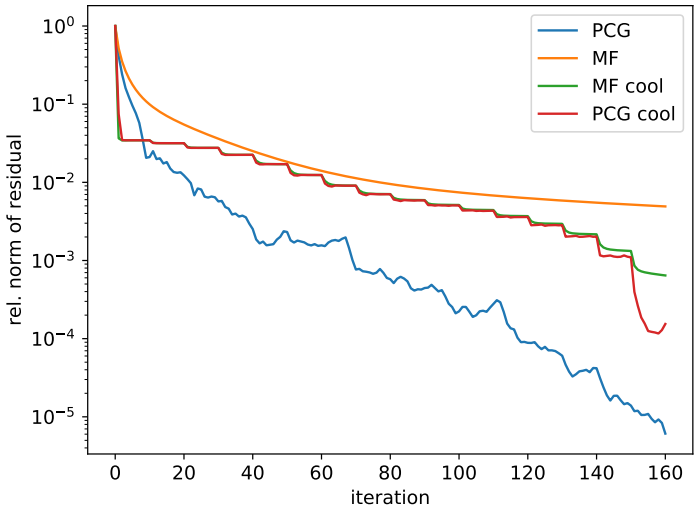


Figure 10: Convergence of PCG and the messenger-field with and without cooling in the map-making application.

Based on these results, we decided to concentrate on (P)CG solvers and their modifications as a more promising direction in which to continue our work.

5.5 Enlarged Conjugate Gradients (ECG) for map-making

As already mentioned in the introduction, the adaptation and the validation of communication-avoiding iterative methods in the context of the CMB map-making problem was one of our main goals within this project.

The numerical observations and recent theoretical proof (see, [26, Sect. 2.4]) show that ECG with the enlarging factor equal to t (asymptotically) converges as the PCG applied to the operator with the $t-1$ smallest eigenvalues deflated. Since the two-level preconditioner, i.e. the combination of the standard and deflation preconditioners, proved to be efficient in the map-making problem (see, e.g., [24, 37]), we consider ECG as a prospective method for highly parallel computations within this application.

We first run prototyping experiments on smaller datasets to identify possible suggestions for choosing a proper orthogonalization variant and/or the value of the enlarging factor. Despite the fact that those smaller datasets may not capture all the features and difficulties present in real large scale data, we expected that a possible speed-up w.r.t. the standard PCG might be already observed. The numerical results of the sequential implementation of ECG in Python are presented in the following subsection.

5.5.1 Prototyping results

The results of initial experiments confirmed our expectations about the similar behaviour of ECG and PCG with a two-level preconditioner based on the deflation of the eigenvectors corresponding to the smallest eigenvalues. These eigenvectors are computed in our experiments using the SciPy wrapper[5] for ARPACK. The comparison for two test cases is shown in Figure 11.

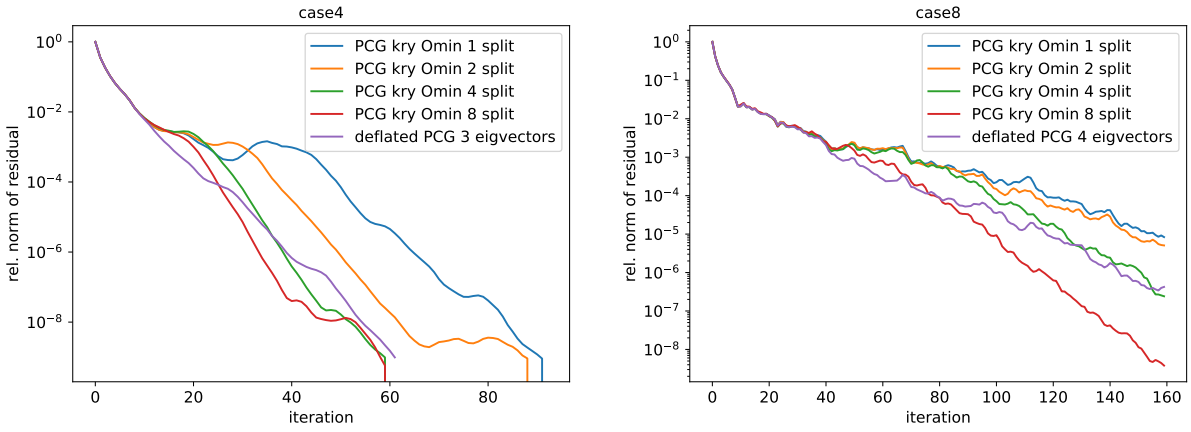


Figure 11: Comparison of ECG and PCG with deflation in two test cases.

In other experiments we concentrated on the comparison of two (orthogonalization) variants for computing the new search direction in ECG. Those variants are called Orthodir and Orthomin. They have different memory and computational requirements and they can differ in finite precision; see, e.g., [25]. Their convergence in some test cases was quite different and surprising. In particular for the Orthodir variant, we observed stagnation of

[5] docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.eigsh.html

the relative residual at some (relatively) high level, which mildly depends on the enlarging factor. In the same cases, Orthomin converges to much higher accuracy. One of these results is given in Figure 12. To our knowledge, such behaviour has not been observed before and poses important questions on the numerical stability of the method, which is a subject of our further research.

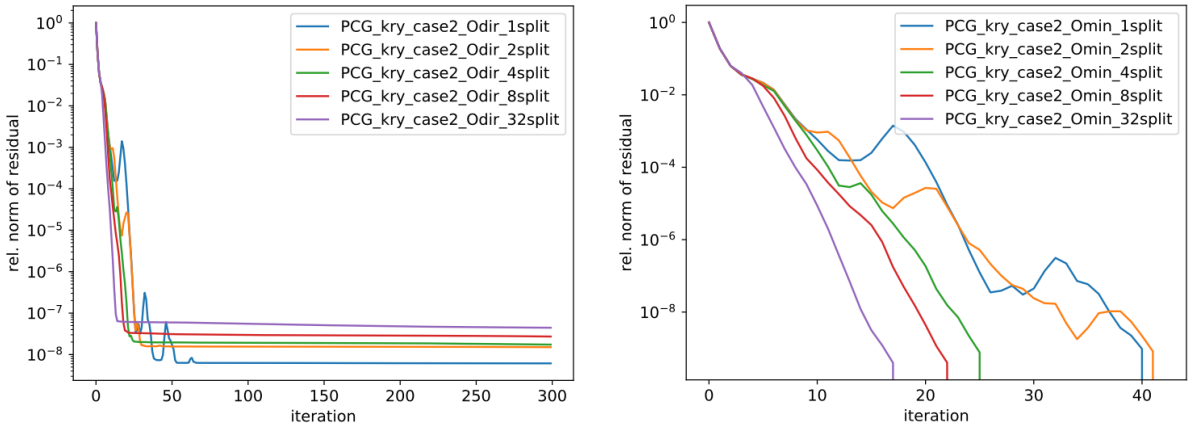


Figure 12: Comparison of two orthogonalization variants of ECG, Orthodir (left) and Orthomin (right). Here, t `split` indicates the enlarging factor t.

Since the negative effect of small eigenvalues on the PCG convergence in these test cases was not as harmful as, e.g., in the elasticity problems ECG was tested on, the decrease of iteration counts with the increasing enlarging factor seems not to compensate for the increasing cost and time of each iteration.

5.5.2 Results using the highly parallel map-making code MIDAPACK

In order to use ECG in the highly parallel map-making test case, we have interfaced ECG implemented in preAlps NLAFET library with MIDAPACK library used for applying the system matrix in (5.2). This required some modifications of the original ECG code to handle in a different way how the vectors are split in MIDAPACK.

In order to preserve the structure of the weighting matrix N^{-1} (which is necessary in order to be able to apply this matrix to a vector), MIDAPACK splits vectors not in the pixel but in the time domain. This means that each MPI process is assigned a part of the vector but those parts are overlapping and the (full) vector is given as a properly weighted sum of the local parts. For ECG, this requires a modification of the inner product to include the weighting and also a modification of the splitting of the initial residual – the entries of the residual vector are assigned to the same column in each MPI process that manipulates the particular row index.

Because the "overlapping ECG" presents a very specific generalization of the ECG code, we decided not to include the corresponding scripts in the preAlps library but to make this code available later as a part of the MIDAPACK library.

Finally, we point out that the current version of MIDAPACK does not allow us to apply a matrix to a set of vectors, which is one of the crucial operations in ECG that should bring a speed-up over simple PCG. Work on implementing this is in progress and should be available

soon. In the current experiments with ECG, we apply the matrix to vectors one by one. As expected, applying a matrix to t vectors then takes t times longer than a single matrix-vector product.

Given a single dataset to run the experiments on, we study the number of iterations and timings for various enlarging factors and timings for a number of MPI processes. We note that, in contrast to, e.g., domain decomposition methods, the block-diagonal preconditioner in map-making *does not* depend on the splitting of the vectors, and therefore the iteration counts for PCG and ECG are independent of the number of processes.

Similarly to the prototyping experiments, we observed a different convergence behaviour for Orthodir and Orthomin variants. For Orthodir with the enlarging factor bigger than one, the relative residual again stagnates at certain level. In the current case, this level is mostly below the commonly used tolerance 10^{-6} but those results pose important questions for the other applications of ECG with Orthodir, in particular in map-making. The convergence of ECG with various enlarging factors is shown in Figure 13. Here, we considered 128 processes but, as noted above, the convergence for other numbers of processes is nearly the same (the timing differs as we discuss below).

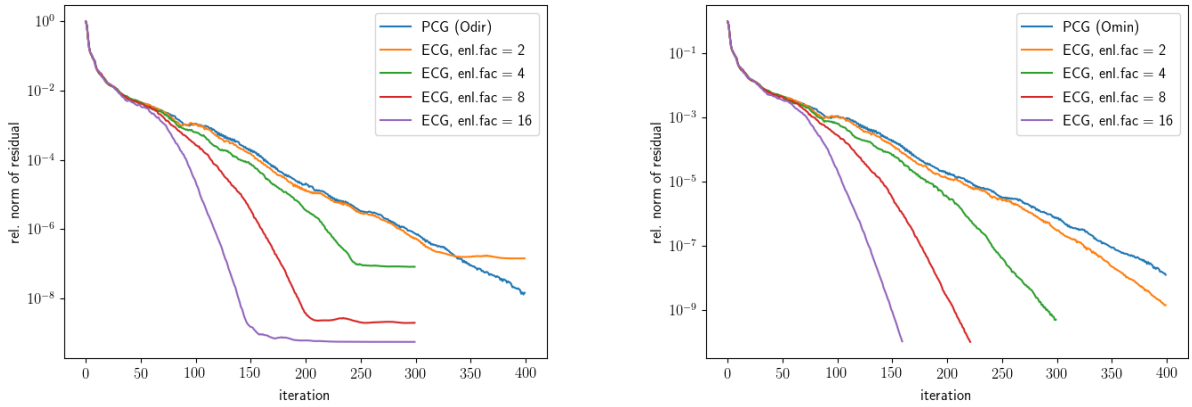


Figure 13: Convergence of ECG in the code using the MIDAPACK library; Orthodir (left) and Orthomin (right).

We run the experiments on the NERSC Cori machine with Haswell computational nodes. Following the suggestion of the MIDAPACK developers, we assign one core to each MPI process. The timings of ECG runs are given in Table 13 including times of overall ECG runs (denoted as "ECG total"), time for internal ECG operations, i.e. all the operations except the application of the system matrix and the preconditioner to a set of vectors (denoted as "ECG inner op."), and the time spent on applying the system matrix ("$P^T N^{-1} P \times V$"). We can see that the matrix-vector product significantly dominates the overall time, despite the fact that it scales satisfactorily when the increasing number of processes. Because the decrease of iteration counts with increasing the enlarging factor is quite mild, the overall ECG time significantly increases for higher enlarging factors. This rather unsatisfactory behaviour is also due to a particular test case, where the PCG convergence is (after few initial iterations) linear and the effect of small eigenvalues is not pronounced (see Figure 13).

We note that the present ECG implementation (yet intended for enlarging factors larger than 1) presents a more efficient and faster implementation of the (standard) PCG than the

current code from MIDAPACK. The total timing of a PCG run as implemented in MIDAPACK is 108.3 s, 55.7 s and 38.4 s for 128, 256 and 512 processes, respectively, cf. the timing for Odir($t = 1$) or Omin($t = 1$) in Table 13.

	Odir					Omin				
enl. factor t	1	2	4	8	16	1	2	4	8	16
	128 MPI processes (local problem size = 11121)									
# iter	293	286	218	161	117	291	279	216	160	117
ECG total	94.27	179.6	271.6	399.2	583.4	94.01	175.3	269.4	396.5	581.8
ECG inner op.	1.876	1.265	1.913	3.372	8.355	1.468	1.151	1.590	2.630	6.371
$P^t N^{-1} P \times V$	92.29	178.2	269.4	395.5	574.7	92.44	174.0	267.7	393.6	575.1
	256 MPI processes (local problem size = 5622)									
# iter	291	285	219	160	118	293	278	216	159	118
ECG total	49.31	92.00	138.1	200.4	294.2	49.80	89.90	136.5	197.8	293.7
ECG inner op.	1.344	0.904	0.894	1.651	3.715	1.360	0.710	0.925	1.162	2.567
$P^t N^{-1} P \times V$	47.93	91.04	137.1	198.6	290.3	48.40	89.13	135.5	196.5	291.0
	512 MPI processes (local problem size = 2865)									
# iter	292	>400	219	160	118	291	278	216	159	118
ECG total	29.12	—	76.68	109.1	159.5	29.85	50.01	73.74	105.7	155.4
ECG inner op.	1.838	—	1.231	1.286	2.058	2.534	1.141	0.823	0.880	1.503
$P^t N^{-1} P \times V$	27.26	—	75.41	107.7	157.3	27.29	28.84	72.88	104.8	153.8

Table 13: Timing of ECG runs, in seconds. Tolerance for the relative residual was set to 10^{-6}. Each MPI process is assigned to one core. Odir($t = 2$) for 512 processes did not converge in 400 iterations.

6 Summary and recommendations for future improvements

Section 2 has described developments made to the 2DRMP software suite for the modelling of electron scattering from H-like atoms and ions. Two major modifications have been made to the code: Firstly, the introduction of the PLASMA library to provide an alternative to the use of proprietary MKL software, and to improve portability by allowing high-performance solutions to be obtained on non-Intel architectures. Secondly, the code structure has been modified to provide a more integrated approach, which does not rely on passing large files between constituent components. A number of other minor improvements have also been made to improve the user-friendliness of the code.

The modified code has been tested on the Intel Skylake platform. While, as expected, the PLASMA routines do not quite match the performance of MKL on this Intel platform, inclusion of the PLASMA library does allow the opportunity to achieve high performance on non-Intel architectures. In cases where users have access to the MKL library, the option to use it in preference to PLASMA is available. A number of other benchmarks and minor code developments are planned, including performance analysis on ARM and AMD platforms, which will result in submission of a paper later this year.

The matrices from the application that we discuss in Section 3 are unsymmetric and very ill-conditioned with condition numbers greater than 10^{10}. Simple scaling does not reduce this condition number. Although the matrices can be quite large, they are also very sparse and so we can factorize even the largest matrix of order over 7 million quite quickly. In fact, the power of our codes cannot be fully demonstrated since we would require much larger matrices to do so. In solving the power flow systems, we use most of the codes developed in Workpackage WP3. Both solvers that we use, from the ParSHUM library and the BC library combine distributed memory parallelism, handled using MPI, with shared memory parallelism using OpenMP or the runtime system StarPU. In both cases, we first partition the matrix and then run a sparse direct code on subsystems defined by this partition. In the case of ParSHUM, we partition the matrix to singly bordered block diagonal form using Zoltan and then use our multicore unsymmetric sparse direct factorization on the rectangular blocks on the diagonal of the SBBDF form. In the case of BC we partition the matrix into blocks of rows using a memory-aware partitioning that tries to keep these blocks mutually orthogonal. We then use an iterative method based on the block Cimmino approach to precondition block conjugate gradients.

We found the matrices from this application quite challenging and this has helped us to plan future enhancements for our codes that will continue to be maintained after the end of the NLAFET Project. The code that was particularly challenged by these matrices was BC. Here we plan to make the code able to detect and react to possible problems. For example, if the direct code detects a singularity we could automatically try to refactor the matrix with the α scaling discussed in Section 3.3, perhaps iterating until a good value for α is obtained. Another enhancement would be to preprocess the matrix to remove dense rows, and partition the matrix so the main factorization can be done on this block but the solution can be obtained by forming the small Schur complement from these dense rows. A more ambitious enhancement, which would be particularly for the use of SpLDLT in the context of BC, would be to monitor the analysis estimation of the number of floating-point operations for factorization and use this to assign a different number of cores to each factorization in order to better balance the load. Although it did not appear to be a major problem with this application, one enhancement to ParSHUM might be to look at alternatives to using the PLASMA dense codes in the factorization of the global Schur as mentioned in Section 3.2. ParSHUM could be recalled to factorize that, or the whole factorization routine with its prior SBBD partitioning could be used on this block. This would enable us to factorize even larger matrices where more partitions were needed and the order of the Schur was too large for the dense solver.

One issue when working with a commercial company such as DIgSILENT GmbH is that we do not have access to their proprietary code and have had to experiment on matrices supplied by them. It is only quite recently that we have given access to the codes to our collaborator, Bernd Klöss, who is planning to test the codes within the framework of their packages. The other issue is that we need very large matrices to stress our codes and to achieve very high levels of parallelism. We asked explicitly for a particularly large matrix for power systems but even that was factorized rather too quickly to see benefits from greater levels of parallelism. There is nothing in the SuiteSparse set of test matrices that comes close to the size of this problem. Now that we have made our codes available, our main future work

will be in continuing to tune and enhance these when we get feedback from the solution of even larger problems than we have so far had access to.

We have presented in Section 4 the parallel efficiency of the enlarged CG method developed at Inria on matrices arising from several different applications, in particular from solving linear elasticity problems. Enlarged CG increases the arithmetic intensity and reduces the communication between processors, and hence is well suited for modern and future architectures that exhibit massive parallelism. For the previous elasticity problem, we show that the method can scale up to 16,384 threads, each one being bound to one physical core, which means that each core owns nearly 280 unknowns.

Enlarged CG does not require any information from the underlying PDE and does not rely on any assumption, except that the matrix is symmetric positive definite. Hence it can be seen as a black-box solver and integrated very easily in any existing code. For instance for a 3D linear elasticity problem with heterogeneous coefficients with 4.5 million unknowns and 165 millions of nonzero entries, we observe that ECG is up to 5.7 times faster than the PETSc implementation of PCG, both using a block Jacobi preconditioner. This test case is known to be difficult because standard one-level preconditioners are not expected to be very effective.

In section 5 we have adapted our ECG code to be used in CMB data analysis, in particular the map-making problem, and tested it on a few datasets that were provided by our partners at APC. Despite obtaining quite pessimistic results, we argue that communication-avoiding methods have the potential of being efficient also in this application. This requires implementation of matrix-matrix products in MIDAPACK (which is currently in progress) and considering test problems where the convergence of PCG with the standard preconditioner is indeed harmed by small eigenvalues. Such cases were, from the very beginning, identified as appropriate for enlarged Krylov methods and this crucial feature is not present in the current test datasets. The ECG code with the modification done to the interface for the MIDAPACK library will be made available for the (astrophysical) community within the new version of the MIDAPACK.

Acknowledgments

This project is funded from the European Union's Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633. We thank the High Performance Computing Center North (HPC2N) at Umeå University, which is part of the Swedish National Infrastructure for Computing (SNIC), for providing computational resources and valuable support.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Analysis and Applications*, 23(1):15–41, 2001.

[3] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. Curfman McInnes, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.8, Argonne National Laboratory, 2017.

[4] K. Bartschat, E. T. Hudson, M. P. Scott, P. G. Burke, and V. M. Burke. Convergent *r* matrix with pseudostates calculation for e^--he collisions. *Phys. Rev. A*, 54:R998–R1001, Aug 1996.

[5] K. A. Berrington, W. B. Eissner, and P. H. Norrington. Rmatrx1: Belfast atomic r-matrix codes. *Computer Physics Communications*, 92(2):290 – 420, 1995.

[6] C. J. Bocchetta and J. Gerratt. The application of the Wigner R-matrix theory to molecular collisions. *The Journal of Chemical Physics*, 82(3):1351–1362, 1985.

[7] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco. *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 2007. Tech. Report SAND2007-4748W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.

[8] C. R. Brune, J. A. Caggiano, D. B. Sayre, A. D. Bacher, G. M. Hale, and M. W. Paris. *r*-matrix description of particle energy spectra produced by low-energy ^{3}H + ^{3}H reactions. *Phys. Rev. C*, 92:014003, Jul 2015.

[9] P. G. Burke and Keith A. Berrington. *Atomic and molecular processes: an R-matrix approach*. Institute of Physics Pub., 1993.

[10] P G Burke, A Hibbert, and W D Robb. Electron scattering by complex atoms. *Journal of Physics B: Atomic and Molecular Physics*, 4(2):153–161, feb 1971.

[11] P.G. Burke, C.J. Noble, and P. Scott. R-matrix theory of electron scattering at intermediate energies. In *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, volume 410, 1987.

[12] P.G. Burke and W.D. Robb. In D.R. Bates and Benjamin Bederson, editors, *The R-Matrix Theory of Atomic Processes*, volume 11 of *Advances in Atomic and Molecular Physics*, pages 143–214. Academic Press, 1976.

[13] V.M. Burke, P.G. Burke, and N.S. Scott. A new no-exchange r-matrix program. *Computer Physics Communications*, 69(1):76 – 98, 1992.

[14] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(6-8):38–53, 2009.

[15] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Analysis and Applications*, 18(1):140–158, 1997.

[16] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[17] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[18] I. S. Duff and J. K. Reid. The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Math. Softw.*, 22(2):187–226, 1996.

[19] T. Elfving. Block-iterative methods for consistent and inconsistent linear equations. *Numerische Mathematik*, 35(1):1–12, 1980.

[20] F. Elsner and B. D. Wandelt. Efficient Wiener filtering without preconditioning. *A&A*, 549:A111, 2013.

[21] A. M. Erisman, R. G. Grimes, J. G. Lewis, W. G. Poole Jr., and H. D. Simon. Evaluation of orderings for unsymmetric sparse matrices. *SIAM Journal on Scientific and Statistical Computing*, 7:600–624, 1987.

[22] L. Grigori, S. Moufawad, and F. Nataf. Enlarged Krylov Subspace Conjugate Gradient Methods for Reducing Communication. *SIAM Journal on Scientific Computing*, 37(2):744–773, 2016. Also as INRIA TR 8266.

[23] L. Grigori, F. Nataf, and Soleiman Y. Robust algebraic Schur complement preconditioners based on low rank corrections. Research Report RR-8557, Jul 2014.

[24] L. Grigori, R. Stompor, and M. Szydlarski. A parallel two-level preconditioner for cosmic microwave background map-making. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 91:1–91:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[25] L. Grigori and O. Tissot. Reducing the communication and computational costs of Enlarged Krylov subspaces Conjugate Gradient. Research Report RR-9023, Inria Paris, NLAFET WN-13, February 2017.

[26] L. Grigori and O. Tissot. Scalable Linear Solvers based on Enlarged Krylov subspaces with Dynamic Reduction of Search Directions. Research Report RR-9190, Inria Paris ; Laboratoire Jacques-Louis Lions, UPMC, Paris, July 2018.

[27] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[28] K. M. Huffenberger and S. K. Næss. Cosmic microwave background mapmaking with a messenger field. *The Astrophysical Journal*, 852(2):92, jan 2018.

[29] L.C. Leal. MIT online courses, Nuclear Engineering, Neutron interactions and applications, Lecture Notes: Brief Review of R-Matrix Theory. URL: https://ocw.mit.edu/courses/nuclear-engineering/22-106-neutron-interactions-and-applications-spring-2010/lecture-notes/MIT22_106S10_lec04b.pdf., 2018.

[30] Sigurd K. Næss and Thibaut Louis. A fast map-making preconditioner for regular scanning patterns. *Journal of Cosmology and Astroparticle Physics*, 2014(08):045–045, aug 2014.

[31] J. Papež, L. Grigori, and R. Stompor. Solving linear equations with messenger-field and conjugate gradient techniques: An application to CMB data analysis. *A&A*, 620:A59, 2018.

[32] M. Plummer, J. D. Gorfinkiel, and J. Tennyson. *Mathematical and computational methods in R-matrix theory*. Collaborative Computational Project on Continuum States of Atoms and Molecules, 2007.

[33] G. Puglisi, D. Poletti, G. Fabbian, C. Baccigalupi, L. Heltai, and R. Stompor. Iterative map-making with two-level preconditioning for polarized cosmic microwave background data sets - a worked example for ground-based experiments. *A&A*, 618:A62, 2018.

[34] T.T. Schultz. *Electron scattering by atomic hydrogen*. PhD thesis, Queen's University Belfast, 1990.

[35] M.P. Scott, P.G. Burke, N.S. Scott, and Timothy Stitt. Correlations, polarization, and ionization in atomic systems. *AIP Conference Proceedings*, 604:82–89, 01 2002.

[36] N. Stanley Scott, M. Penny Scott, Phil G. Burke, Timothy Stitt, V. Faro-Maza, Christophe Denis, and A. Maniopoulou. 2DRMP: A suite of two-dimensional R-matrix propagation codes. *Computer Physics Communications*, 180:2424–2449, 2009.

[37] M. Szydlarski, L. Grigori, and R. Stompor. Accelerating the cosmic microwave background map-making procedure through preconditioning. *A&A*, 572:A39, December 2014.

[38] F. S. Torun, M. Manguoglu, and C. Aykanat. A novel partitioning method for accelerating the block Cimmino algorithm. *SIAM J. Scientific Computing*, 40(6):C827–C850, 2018.

[39] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

[40] E. P. Wigner and L. Eisenbud. Higher angular momenta and long range interaction in resonance reactions. *Phys. Rev.*, 72:29–41, Jul 1947.

[41] O. Zatsarinny. BSR: B-spline atomic R-matrix codes. *Computer Physics Communications*, 174(4):273–356, 2006.