Parallel sparse direct solvers for Poisson's equation in streamer discharges

Margreet Nool[1], Menno Genseberger[2] and Ute Ebert[1,3]

[1] *Centrum Wiskunde & Informatica (CWI),*
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands
[2] *Deltares, Delft, The Netherlands,*
[3] *Dept. Physics, Eindhoven Univ. Techn., The Netherlands.*

Abstract

The aim of this paper is to examine whether a hybrid approach of parallel computing, a combination of the message passing model (`MPI`) with the threads model (`OpenMP`) can deliver good performance in streamer discharge simulations. Since one of the bottlenecks of almost all streamer models is the solution of Poisson's equation, we focused on several direct solvers, which can solve large sparse systems in parallel. For this purpose, our basic thought was to concentrate on 'easy to get' performance improvements, or, without rewriting of the code.

We have investigated in `PARDISO`, a shared memory solver, and `CLUSTER_SPARSE_SOLVER` and `MUMPS`, which both can apply hybrid parallelism; the latter two solvers can be called from a single core and do not require minor awareness of `MPI`. We show their performance for solving two- and three-dimensional Poisson's equations on the Dutch national supercomputer, called Cartesius. A runtime study of a code developed for streamer propagation nearby a dielectric rod is included. We discuss various issues that appear to be critical in a mixed `MPI-OpenMP` environment.

1 Introduction

Simulation of streamer discharges is quite challenging, because of the multiscale aspect in time and space. The multiscale aspect requires very time-dependent simulations: in atmospheric air the smallest time scales are of order 10^{-13} and 10^{-11} seconds, see Teunissen [27]. Secondly, streamer discharges are spatially multiscale phenomena, see [7, 27, 18, 19]. The space charge layer at the head of streamer is very thin and curved, of the order of a few micrometers in atmospheric air.

Small time scales influence the cost of simulations, because the electric field in such simulations has to be recomputed at every time step. An often used approach is to compute the electric potential by solving the Poisson equation. A lot of solvers for the Poisson equation exist, but their suitability for streamer simulations largely depends on the speed of solving large problems.

There are three typical methods for solving these systems: 1) *iterative* methods such as multigrid solvers. For instance, geometric multigrid can solve the Poisson

equations very efficiently ($O(N)$ in time), 2) *direct* methods such as `FISHPACK`, `PARDISO`, `MUMPS`, and `CLUSTER_SPARSE_SOLVER`, 3) *hybrid* methods, that combine both direct and iterative techniques. In this paper we focus on direct methods.

For years `FISHPACK` played an important role in the simulation software of the Multidynamics (MD) group at CWI. `FISHPACK` is a collection of Fortran programs and subroutines that solves second- and fourth-order finite difference approximations to separable elliptic Partial Differential Equations (PDEs). The solver uses the cyclic reduction algorithm. Its speed compared to other solvers is very high. A disadvantage of `FISHPACK` is that it is not developed for modern computer architectures with many cores per node and lots of nodes, while these computers promise to be extremely suited for long simulations on streamer discharges. Another drawback is that `FISHPACK` can not deal with large grids, because of numerical instabilities due to round-off errors.

Initially, we were looking for fast sequential solvers for simple desktop computers, but later on also for parallel solvers for state-of-the art cluster computers. The goal of this work is to avoid any deep restructuring or rewriting of the code. The performance of two packages are examined, which can deal with `MPI` [10, 25] and `OpenMP` [9] and a mix of both.

All experiments have been executed on the Cartesius, the Dutch national super computer. After describing the Cartesius in section 2, we discuss in section 3 the Poisson solvers. All solvers belong to the category of direct methods, all consisting of three phases: analysis, factorization and solve. In section 4, we describe a special kind of two- and three dimensional Poisson's equations with known analytical solution, which enables to measure the convergence speed. Numerical experiments on this type of symmetric Poisson's equations are discussed in section 5. Besides the academic example in section 5, we focus on the performance of a code developed in our research group. This 2D code, written by Dubinova [7], called DIELightning, has been developed during the project *Creeping sparks* to gain insight into the physics of surface discharges by studying streamers near dielectrics. Section 6 provides timing results based on short runs with the multithreaded and parallel packages `MUMPS`, `PARDISO` and `CLUSTER_SPARSE_SOLVER`. Finally, in section 7 we summarize the results obtained at the Cartesius and will end up with some conclusions.

2 System overview of Cartesius

Cartesius is the Dutch national supercomputer. In November 2016, the Cartesius was ranked number 97 on the TOP500, the list shows the 500 most powerful commercially available computer systems. It is a general purpose capability system with many cores, large memory (130 TB memory), and a fast interconnection between nodes. In this report, we examine the role Cartesius can play in simulating streamer discharges,

- is there a good alternative to `FISHPACK` (see section 3.1.1)?
- is it easy to use more nodes?
- does the functionality of the Math Kernel Library `MKL` offer interesting solutions for solving Poisson's equation?

In this paper, we focus on these questions.

We begin with an overview of the composition of the Cartesius: Starting December 2016, Intel® Broadwell nodes became available on the Cartesius. Broadwell

Node Type	Number	Cores	CPU	Clock	Memory
thin (Broadwell)	177	32	E5-2697A v4	2.6 GHz	64 GB
thin (Haswell)	1080	24	E5-2690 v3	2.6 GHz	64 GB
thin	540	24	E5-2695 v2	2.4 GHz	64 GB
fat	32	32	E5-4650	2.7 GHz	256 GB
gpu	64	16	E5-2450 v2	2.5 GHz	96 GB

CPUs are similar to the Haswell CPUs in Cartesius, but they are more energy efficient and they have a higher memory bandwidth. The most important difference is that Broadwell nodes have 32 CPU cores, whereas all other thin nodes have 24 CPU cores. More CPU cores per node enables to run more MPI tasks, or to use more threads in multi-threaded programs. All experiments of section 5 are performed on Haswell nodes, and those of section 6 on Broadwell nodes.

In the context of a task being performed on a computer, *wall-clock* time is a measure of the real time that elapses from start to end, including time that passes due to programmed (artificial) delays or waiting for resources to become available. The wall-clock time is measured with function `omp_get_wtime`, returning an elapsed time on the calling processor. We measured a resolution time of 10^{-6} seconds for `omp_get_wtime`.

3 Software packages

In recent years, with the advent of multicore machines, interest of software developers has shifted towards multithreading existing software in order to maximize utilization of all the available cores. The packages designed for parallel computing and described in this paper take advantage of multithreading, some of them in combination with `MPI`.

The aim of this paper to test multithreaded and `MPI` packages in existing codes with the restriction to avoid any deep restructuring or rewriting of codes. Moreover, in our codes we use accelerating math processing routines of the Intel® `MKL` library [1], that increase application performance and reduce development time. Core mathematical functions of `MKL` include `BLAS`, `LAPACK`, `ScaLAPACK`, sparse solvers, fast Fourier transforms, and vector mathematical. The routines in `MKL` are optimized specifically for Intel® processors.

In this section we describe five packages that can solve Poisson's equation. We start with two sequential packages, `FISHPACK` and `LAPACK`, followed by packages designed for parallel computing on shared and distributed memory systems.

3.1 Poisson solvers

3.1.1 `FISHPACK`

To celebrate the 300th anniversary of the appointment of Johann Bernouilli as professor at the University of Groningen, in 1995, a one-day workshop entitled "Laplace Symphony" took place as one of the scientific activities. The comparison of solvers for a number of Laplace-like equations was the main activity of the workshop. The CPU time of several advanced solvers measured on the same computer are pre-

sented in [5]. For the 2D problem FISHPACK gives by far the best results, this package optimally exploits the symmetry and solves the systems of linear equation more than 10 times faster than the other well-known solvers, including UMFPACK/SuperLU, ILUT/Bi-CGSTAB(SPARSKIT), but also a ICCG solver with diagonal and hyperplane ordering, a MILU (modified ILU) solver, Nested Grids ILU (NGILU) and MGD9V, a multigrid method. Also for the three-dimensional problem, FISHPACK is more than a factor of 10 faster than the second best method.

Therefore our research group has used FISHPACK for time-consuming simulations to solve the Poisson equation. However, one of the major limitations of this solver is its inability to deal accurately with large grids ($> 1400 \times 1400$), due to numerical instabilities. Another drawback of FISHPACK is that it can only be used for matrices that can be represented by stencils with constant coefficients. In her thesis Montijn [19] illustrates the error for the Laplace equation in a radially symmetric coordinate system of the single respectively double precision computations. Also Li [18] discusses the inaccuracy of the FISHPACK solver; he tests a Laplace equation in a 3D system with this solver.

Teunissen mentioned in his thesis [27], that the Poisson equation for the electric field is solved in all models at each time step with the same fast electric solver [26], on a uniform grid of $256 \times 256 \times 512$. For a smooth system and double precision arithmetic his results up to $500 \times 500 \times 500$ are accurate (less than 10^{-6}), but for larger problems he runs into the limits of FISHPACK.

3.1.2 LAPACK

LAPACK (Linear Algebra Package) [3] is a standard software library for numerical linear algebra. It includes routines to implement the associated matrix factorizations such as LU, QR, Cholesky and Schur decomposition. DGBTRF [1] computes an LU factorization of a real m-by-n band matrix A using partial pivoting with row interchanges, whereas DGBTRS uses the LU factorization to compute the solution matrix/vector, depending on the number of right-hand sides. In LAPACK terms, A is a band matrix with KL subdiagonals and KU superdiagonals, when the distance between the main diagonal and the outer subdiagonal is KL and to the outer upper-diagonal is KU. This implies that also the zero diagonals between the outer lower and outer upper diagonal are part of the band matrix, resulting in a dense matrix. The 2D Poisson matrix DGBTRF counts $2n_x + 1$ diagonals, in case of a $n_x \times n_x$ domain. Actually, the 2D Poisson matrix counts only five nonzero diagonals, whereas the 3D Poisson matrix has nine nonzero diagonals; these matrices can be called sparse.

3.1.3 PARDISO

Shared memory PARDISO has been optimized by applying multithreading. In [15], it is shown that the execution time reduces with the increase of the number of threads. It is also reported that super-linear acceleration can take place when increasing the number of OpenMP threads. The solver [23, 22, 24, 21, 17] was developed by Olaf Schenk, and Intel® MKL has incorporated it into the Intel® MKL library [1]. PARDISO calculates the solution of a set of sparse linear equations with multiple right-hand sides, $AX = B$, using an OpenMP, or multithreaded LU, LDL^T or LL^T factorization, where A and X, B are n by n and n by $nrhs$ matrices, respectively.

[1] GB denotes General Band matrix

In our experiments, $nrhs$ is always one. PARDISO supports a wide range of sparse matrix types on shared-memory multiprocessing architectures. It computes the solution of real or complex, symmetric, structurally symmetric or non-symmetric, positive definite, indefinite or Hermitian sparse linear systems of equations. For our implementation of Poisson's equation we have to do with real symmetric indefinite and real unsymmetric matrices.

The PARDISO solver [24] first computes a symmetric fill-in reducing permutation P based on either the minimum degree algorithm or the nested dissection algorithm from the METIS package [16], followed by the parallel left-right looking numerical Cholesky factorization $PAP^T = LL^T$ or $PAP^T = LDL^T$ for symmetric, indefinite matrices, or $PAP^T = LU$ for unsymmetric matrices. The solver uses diagonal pivoting or 1×1 and 2×2 Bunch-Kaufman pivoting for symmetric indefinite matrices and an approximation of X is found by forward and backward substitution and iterative refinement.

We realize the Intel® MKL version of PARDISO is based on an older version of PARDISO; the recent version is PARDISO 5.0.0 Solver Project, see http://www.pardiso-project.org. Although, a lot of new features and improvements of PARDISO are not available in the Intel® MKL library, we use the MKL implementation, because it is easy to use and clearly described.

3.1.4 CLUSTER_SPARSE_SOLVER

The CLUSTER_SPARSE_SOLVER implementation [15, 13, 14] distributes a tree node between different computational nodes and handles elements of a factorized matrix by different MPI processes using OpenMP on each process. Experiments demonstrate that it is recommended to exploit the computational threads on the nodes to the maximum and not to have several computational processes per node.

Quite recently, a hybrid implementation of PARDISO, has been implemented and added to the MKL library, called CLUSTER_SPARSE_SOLVER. It combines MPI technology for data exchange between parallel tasks running on different nodes, and OpenMP technology for parallelism inside each node. In 2014, when we decided to use the MKL version of PARDISO, it was not yet possible to use the hybrid implementation with parallelism over the nodes. Instead we use the MKL routine CLUSTER_SPARSE_SOLVER also based on work by the team of Olaf Schenk, which performance is described by Kalinkin [15].

3.1.5 MUMPS

MUMPS (MUltifrontal Massively Parallel sparse direct Solver) [8], originally designed by Duff and Reid in 1983, is a software application for the solution of large sparse systems of linear algebraic equations. The initial matrix will be represented as "elimination-tree" or "assembly tree" [8, 15], with the number of leafs equal to the number of processes. It is a free implementation of the multifrontal method, which is a version of Gaussian elimination for large sparse systems of equations. MUMPS is still maintained and supported by a group of CERFACS, IRIT-ENSEEIHT, and INRIA and it is still extended and improved. The latest version, release (July 2016) : 5.0.2 is suitable for hybrid shared-distributed memory architectures, with parallelism by multithreading and by MPI. It is written in Fortran 90 and it uses BLAS and ScaLAPACK [4] kernels for dense matrix computations. Many recent MUMPS

related publications can be found on the MUMPS website [2].

3.2 Decomposition and solution phases

To solve a positive indefinite symmetric system

$$Ax = b, \tag{1}$$

we factorize A into

$$A = LDL^T. \tag{2}$$

For the unsymmetric case A will be factorized as

$$A = LU. \tag{3}$$

As the solution process for the unsymmetric matrix passes the same phases, we concentrate on the positive indefinite symmetric matrix. The implementations of the direct solvers mentioned before, except for FISHPACK, consist of several stages:

- *Matrix reordering and symbolic factorization:* The initial matrix A will be reordered to reduce the fill-in in factor L, and a dependency tree representation of matrix A will be created.
- *Numeric factorization:* In this phase the factorization takes place where the total number of nonzero elements is computed in LL^T. We note that the process of numeric factorization is the most time-consuming part of the solution of (1).
- *Forward and backward substitution* This phase consists of three steps: 1) the forward step to solve $Ly = b$, 2) the diagonal step to solve $Dz = y$ and 3) finally the backward step $L^T x = z$, where x is the solution vector.

An advantage of the phasing out of solving (1) is that for constant matrices A, but different right-hand side vectors x, the first two phases, the matrix reordering and numerical factorization, have to be performed only once. This leaves the last phase of for- and backward substitution as the most important one. We remark that FISHPACK does not distinguish such separate stages, reuse of previous (cyclic reduction) steps is not possible.

4 Poisson's equations with known analytical solution

In this section, we concentrate on the two dimensional Poisson equation

$$\Delta U(x, y) = q(x, y) \tag{4}$$

on a rectangular domain $[0, 1] \times [0, 1]$, and the three-dimensional Poisson equation

$$\Delta U(x, y, z) = q(x, y, z) \tag{5}$$

on a cubic domain $[0, 1] \times [0, 1] \times [0, 1]$. We apply a 4- and 6-point centered, second order difference scheme, respectively. We use the method of manufactured solutions:

from an analytic solution the right-hand side and boundary conditions are computed. For the two-dimensional problem we choose the analytical solution

$$U(x,y) \quad = \quad e^{-C((x-x_0)^2+(y-y_0)^2)} + 1.0 \tag{6}$$

on a uniform grid defined on $x \in [0,1], y \in [0,1]$, obtaining

$$\begin{array}{rcl} \Delta U(x,y) & = & \left[-4C + 4C^2((x-x_0)^2 + (y-y_0)^2)\right] \times \\ & & e^{-C((x-x_0)^2+(y-y_0)^2)}. \end{array} \tag{7}$$

Analogously, for the three-dimensional case, we select

$$U(x,y,z) \quad = \quad e^{-C((x-x_0)^2+(y-y_0)^2+(z-z_0)^2)} + 1.0 \tag{8}$$

on a uniform grid defined on $x \in [0,1], y \in [0,1], z \in [0,1]$ obtaining

$$\begin{array}{rcl} \Delta U(x,y,z) & = & \left[-4C + 4C^2((x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2\right] \times \\ & & e^{-C((x-x_0)^2+(y-y_0)^2+(z-z_0)^2)}. \end{array} \tag{9}$$

For the value C in equations (6) and (8) we choose $C \in \{10^0, 10^2, 10^4, 10^6\}$, to vary from a smooth into a narrow peaked solution, see figure 1. The Gaussian curve can be shifted too, to increase the complexity of the problem. Unless otherwise specified, we use $x_0 = y_0 = 0.5$ for the 2D case, and, $x_0 = y_0 = z_0 = 0.5$ for the 3D case.

4.1 Implementation of Poisson's equation

Let us consider a one-dimensional *cell centered uniform grid* with nodes

$$x_i = (i - \frac{1}{2})h; i = 1, \cdots, M; h = 1/M. \tag{10}$$

Applying Dirichlet boundary conditions we need in $x_0 = -\frac{1}{2}h$ and in $x_{M+1} = 1 + \frac{1}{2}h$, the *virtual* values u_0 and u_{M+1}, such that

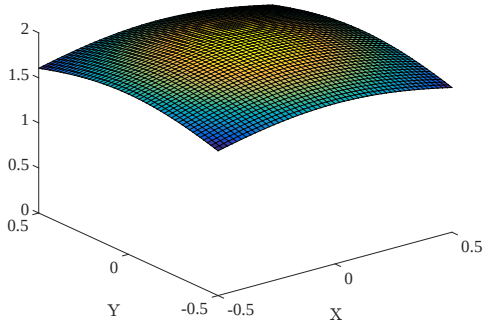
$$\frac{1}{2}(u_0 + u_1) = \gamma_0 \quad \text{and} \quad \frac{1}{2}(u_M + u_{M+1}) = \gamma_M. \tag{11}$$

Following [11] we obtain the following semi-discrete system

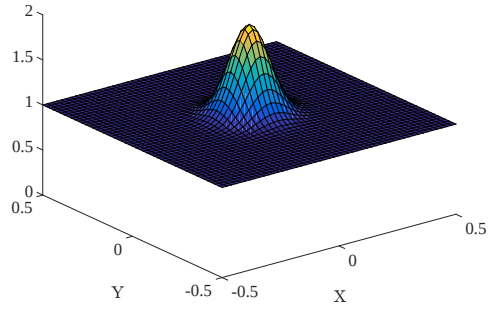
$$\begin{array}{rcll} u_1{}' & = & \frac{1}{h^2}(-3u_1 + u_2) & + \frac{2}{h^2}\gamma_0, \\ u_i{}' & = & \frac{1}{h^2}(u_{i-1} - 2u_i + u_{i+1}), & 2 \le i \le M-1, \\ u_M{}' & = & \frac{1}{h^2}(u_{M-1} - 3u_M) & + \frac{2}{h^2}\gamma_M, \end{array} \tag{12}$$

The matrix formulation for the 1D Poisson equation reads

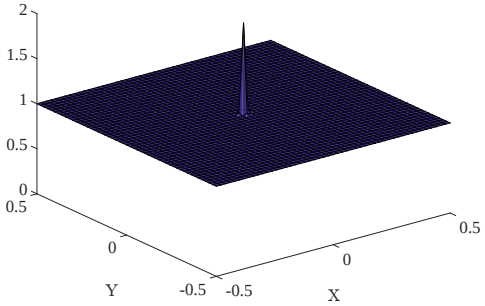
$$A = \frac{1}{h^2}\begin{bmatrix} \mathbf{-3} & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & \ddots & & \\ & & 1 & -2 & 1 \\ & & & 1 & \mathbf{-3} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 + \mathbf{\frac{2}{h^2}\gamma_0} \\ b_2 \\ b_3 \\ \ddots \\ b_{M-1} \\ b_M + \mathbf{\frac{2}{h^2}\gamma_M} \end{bmatrix}, \tag{13}$$



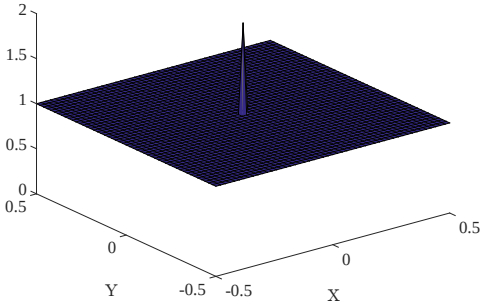
(a) $C = 10^0$



(b) $C = 10^2$



(c) $C = 10^4$



(d) $C = 10^6$

Figure 1: The analytical solution (6) for different values of C for $x_0 = y_0 = 0$ on domain $[-0.5, 0.5] \times [-0.5, 0.5]$

where matrix A of size $M \times M$ is constant for a fixed grid size, and the right-hand side vector b depends on the boundary values. We note that the matrix A is **symmetric positive indefinite**. Analogously, we calculate the matrix formulation for the 2D Poisson equation A of size $M^2 \times M^2$. The matrix A can be partitioned into block matrices. We distinguish two different main diagonal block matrices

$$R = \begin{bmatrix} \mathbf{-6} & 1 & & & \\ 1 & \mathbf{-5} & 1 & & \\ & 1 & \mathbf{-5} & 1 & \\ & & \ddots & & \\ & & 1 & \mathbf{-5} & 1 \\ & & & 1 & \mathbf{-6} \end{bmatrix}, \quad S = \begin{bmatrix} \mathbf{-5} & 1 & & & \\ 1 & \mathbf{-4} & 1 & & \\ & 1 & \mathbf{-4} & 1 & \\ & & \ddots & & \\ & & 1 & \mathbf{-4} & 1 \\ & & & 1 & \mathbf{-5} \end{bmatrix}. \quad (14)$$

Then the matrix A can be written as

$$A = \frac{1}{h^2} \begin{bmatrix} R & I & & & \\ I & S & I & & \\ & I & S & I & \\ & & \ddots & & \\ & & 1 & S & 1 \\ & & & 1 & R \end{bmatrix}. \tag{15}$$

where I denotes the identity matrix.

For the 3D case we distinguish three diagonal blocks, analogously to matrix A in equation (15)

- $[-9\ -8\ -8\ \cdots -8\ -8\ -9]$ for cells on edges
- $[-8\ -7\ -7\ \cdots -7\ -7\ -8]$ for cells on surfaces
- $[-7\ -6\ -6\ \cdots -6\ -6\ -7]$ for inner cells

supplemented with two sub diagonal identity blocks on distance n_x and n_x^2 and two super diagonal blocks on the same distance.

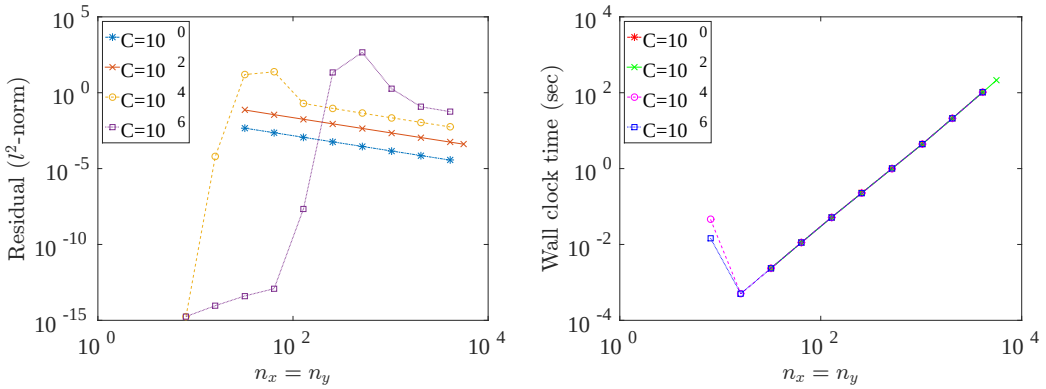
The formulation of the boundary conditions outside of the matrix A, but inside the right-hand side vector b, has the advantage that the factorization of matrix A has to be calculated only once. As we show in section 5, this formulation costs an enormous amount of computing time. This applies to all solution methods discussed in section 3, except for `FISHPACK`, which solves the system inseparable.

5 Numerical experiments on 2D and 3D Poisson's equations

The experiments with the different solvers in this section were carried out on the Haswell nodes of Cartesius with 24 cores. The Intel Fortran compiler 15.0 was used. We start giving the results with the two-dimensional Poisson equation (4) and analytic solution (6) with the results for different C values. The higher the value of C, the more difficult the problem is to solve. In figure 2.a we show the Euclidean norm of the computed error for $C \in \{10^0, 10^2, 10^4, 10^6\}$ on several grid sizes. For $C = 10^4$ and $C = 10^6$, it seems that the solution is computed exactly for "larger" grid size, $h = 1/8$ and $h = 1/16$. However, we presume that the grid size is not small enough to recognize the peak in the solution. For smaller grid sizes the residual values are following parallel lines. The smallest residual values are obtained for $C = 10^0$, as might be expected. Let $x = |x_c - x_{exact}|$ be the difference vector between the computed x_c and exact solution x_{exact}, then we distinguish two residuals

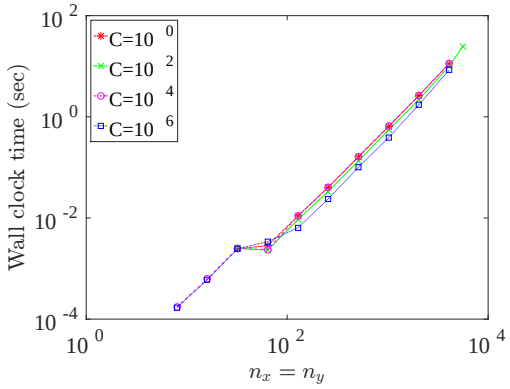
$$R_{max} = \|x\|_\infty \quad \text{and} \quad R_2 = \|x\|_2 . \tag{16}$$

Each convergence figure indicates which residual is displayed. Figure 3 shows the difference between both residuals for `MUMPS` results of the 2D Poisson problem. It appears that the C-values do not influence the wall-clock time for the reordering, factorization, and solution, as shown in figure 2 for the `PARDISO` experiments. The same is true for the other solution packages.

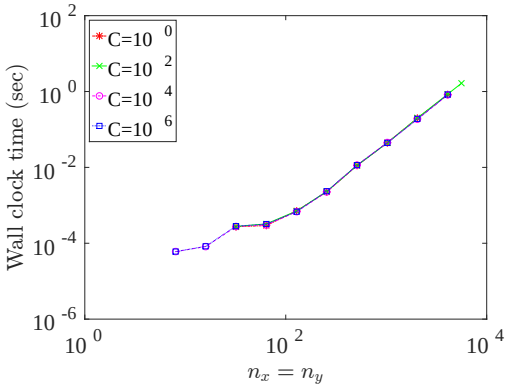


(a) Convergence

(b) Reordering `PARDISO`



(c) Factorization `PARDISO`



(d) Solution `PARDISO`

Figure 2: Figure (a) demonstrates the convergence behavior of `PARDISO` for different values of C, see (4). Figures (b),(c) and (d) show the timing results on a single Haswell node, `OMP_NUM_THREADS` $= 24$, for the reordering, factorization, and solution phases, respectively.

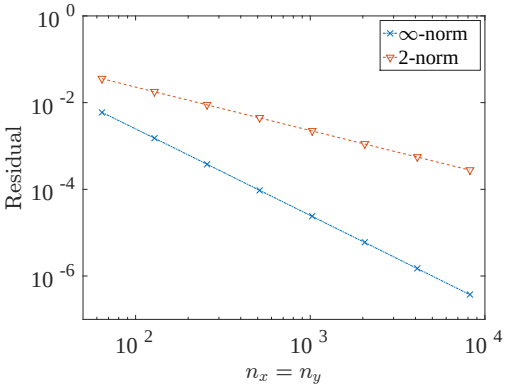
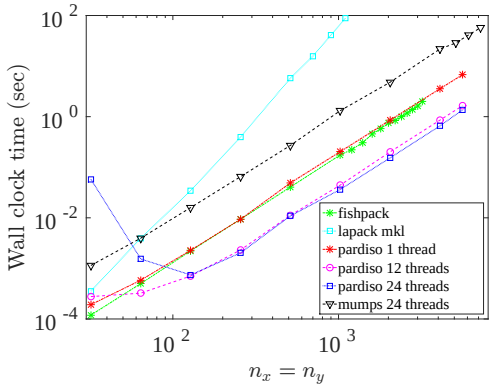
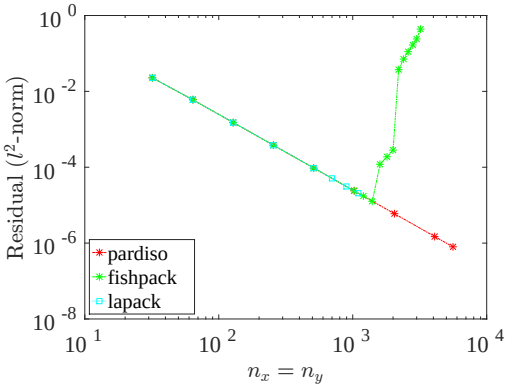


Figure 3: Difference between infinity and 2-norm for 2D `MUMPS` accuracy



(a) 2D Poisson problem solution time



(b) 2D Poisson problem accuracy

Figure 4: The wall-clock timings for the 2D Poisson solution (without time for factorization/reordering) on a Haswell node and the convergence results for the 2D Poisson problem for four solvers: `LAPACK`, `FISHPACK`, `PARDISO` and `MUMPS`

5.1 Results on 2D Poisson's equation

In figure 4a we benchmark four direct methods that can be used to solve the Poisson equation of the form $Ax = b$ on a single node. In case of a two dimensional problem, matrix A is a band matrix with five diagonals, where the outer diagonals are on distance n_x of the main diagonal. For $n_x = n_y$ being the size of the square domain, the main diagonal of matrix A has n_x^2 elements.

- `LAPACK` [3] : the general band solvers `DGBTRF` (for the factorization) and `DGBTRS` (for the solution) have been called. Due to the fill in, the maximum problem size is reached for $n_x = 1300$. Obviously, the wall-clock time to solve the problem with `LAPACK`'s band solver is much larger than for the `FISHPACK` and `PARDISO`. We may conclude that this general band solver is not a good choice to solve Poisson's equation.
- `FISHPACK`: is one of the fastest methods to solve Poisson's equation, however the solution obtained by `FISHPACK` is only accurate for small matrices, i.e., with $n_x < 1400$, see figure 4b. Unfortunately, no warning is given about the accuracy for larger values. We remark that `FISHPACK` does not distinguish several stages for reordering, factorization, and solution, this yields that the timing results includes the three stages.
- `PARDISO` (see section 3.1.3) : its wall-clock time is comparable with that of `FISHPACK` when running on a single thread. In case 12 or 24 threads are being used, `PARDISO` is much faster (nearly 5 times) than `FISHPACK`. We note that the maximum problem size (on a single node 5600×5600) is not limited by the accuracy, as for `FISHPACK`, but by the available memory.
- `MUMPS` (see section 3.1.5) is less efficient than `PARDISO`, even when all cores are used for `OpenMP` parallelism. We note that the maximum problem size on a single node is 7200×7200. In the next section we will see that `MUMPS` hardly benefits of `OpenMP`, but of `MPI`, when solving Poisson's equations.

5.2 Results on 3D Poisson's equation

The 3D experiments are performed on cubed domains for
$n_x = n_y = n_z \in \{64, 80, 96, 112, 128, 160, 192, 224, 256\}$. On a single node the maximum problem size is $160 \times 160 \times 160$. The figures in this section show the results for the maximum realizable problem size.

In figure 5 we focus on the timing results of the 3D Poisson equation performed by the `CLUSTER_SPARSE_SOLVER` (*left*) and `MUMPS` (*right*) implementation. The upper figures show the convergence behavior. Apparently, this behavior is similar, but `CLUSTER_SPARSE_SOLVER` can tackle larger problems for equal number of nodes. As might be expected, the residual does not depend on the number of nodes involved. The timing results of the different stages, c.f., section 3.2, are shown:

1. *the reordering phase:* `CLUSTER_SPARSE_SOLVER`: For small number of nodes (less or equal to 8) the profit of using more nodes translates into a lower wall-clock time. For larger number of nodes this gain is less significant, probably due to more communications between the nodes. The smallest grid size for the three-dimensional Poisson problem obtained is $1/256$ for $p_N = 32$ and $p_N = 64$, where p_N denotes the number of nodes. For `MUMPS` it appears that this phase is completely sequential, as described in [20]; the main limitation to run larger problems comes from the memory needed during this phase. The largest possible problem size for `MUMPS` on 64 nodes amounts $224 \times 224 \times 224$.
2. *the factorizing phase:* This phase is by far the most time consuming part of the solution process. From figure 5e and 5f we observe that an increase of the number of nodes decreases the wall-clock time. Fig. 7a shows the speedup achieved for the `MUMPS` factorization, that of `CLUSTER_SPARSE_SOLVER` is comparable. Speedup is the ratio of performance between two runs of the same code, with different number of nodes

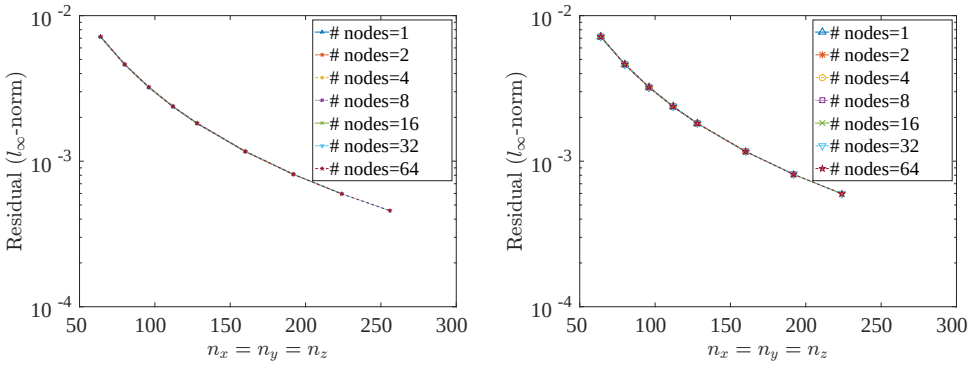
$$S_p = \frac{T_1}{T_p} \tag{17}$$

   where T_p is the time needed on p nodes.
3. *the solution phase:* For the kind of problems we want to solve, this phase is the most important one, because we need this part of the solution process countless times, compared to the reordering and factorizing part. Since in figures 5g and 5h the x- and y-axis are identical, we may conclude that the solution process of `MUMPS` is faster than that of `CLUSTER_SPARSE_SOLVER` when using 24 cores.

Figure 6 displays the difference using `CLUSTER_SPARSE_SOLVER` for the solution phase between using 12 and 24 threads per node. The results does not much differ, often the results obtained when only 12 cores used are faster and less error-prone. Using 16 nodes with 12 cores each, the maximum problem size is $256 \times 256 \times 256$, whereas using 24 cores per node the maximum allowed problem size is $224 \times 224 \times 224$. But it also happens that involving more cores results into a larger problem size.

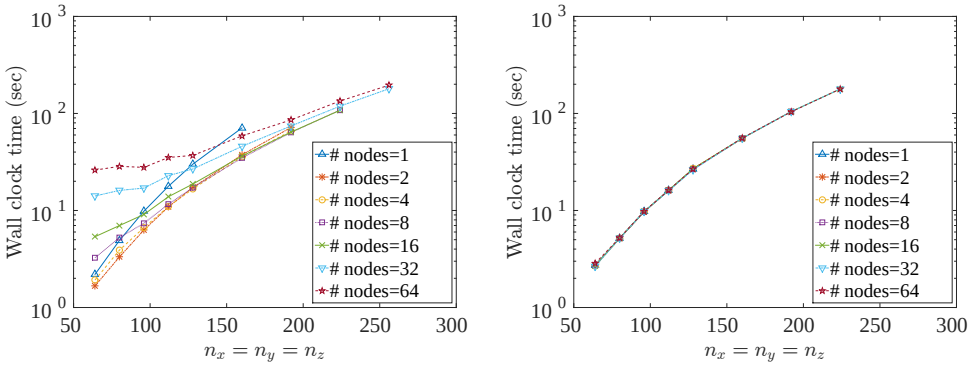
5.3 Conclusions

Summarizing, for large systems both `FISHPACK` and `LAPACK` are not apt for solving Poisson's equation, because the results are incorrect (`FISHPACK`), or, their memory



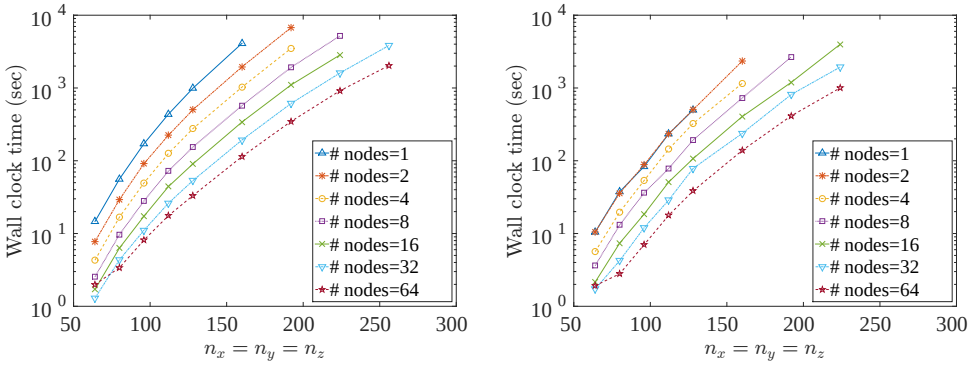
(a) `CLUSTER_SPARSE_SOLVER`: Convergence

(b) `MUMPS`: Convergence



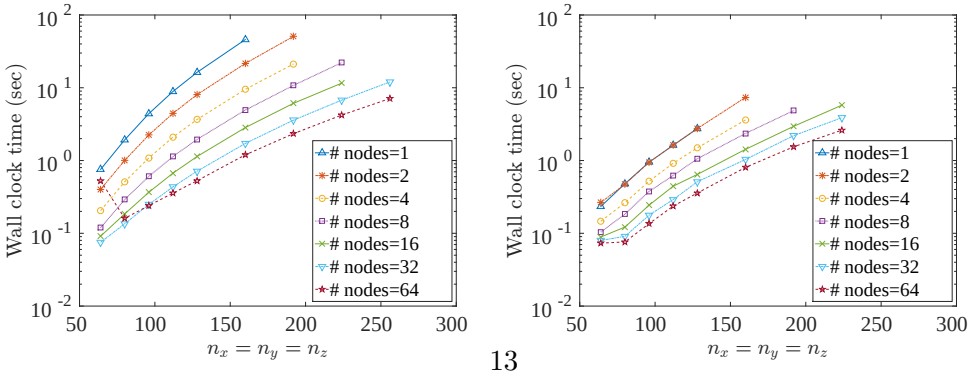
(c) `CLUSTER_SPARSE_SOLVER`: Reordering

(d) `MUMPS`: Reordering



(e) `CLUSTER_SPARSE_SOLVER`: Factorization

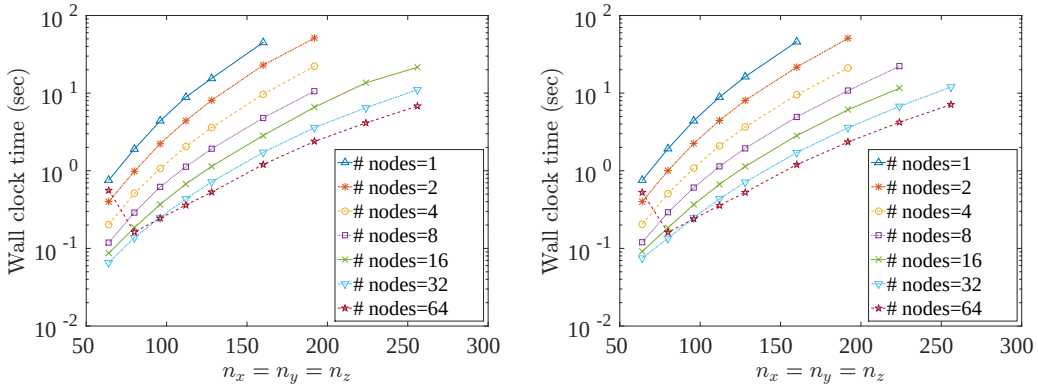
(f) `MUMPS`: Factorization



(g) `CLUSTER_SPARSE_SOLVER`: Solution

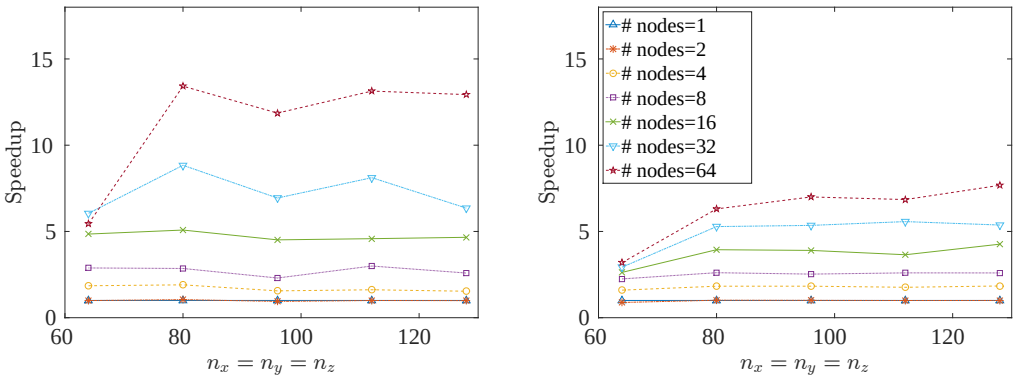
(h) `MUMPS`: Solution

Figure 5: Convergence behavior and wall-clock times for the three phases of `CLUSTER_SPARSE_SOLVER` and `MUMPS` varying the number of nodes. The experiments are performed on Haswell nodes with `OMP_NUM_THREADS` = 24. Along the y-axis the wall clock time is shown, which differs for each phase.



(a) `OMP_NUM_THREADS` = 12 (b) `OMP_NUM_THREADS` =24

Figure 6: The wall-clock time needed for the solution phase of the 3D Poisson equation by `CLUSTER_SPARSE_SOLVER` by 12 (*left*) and 24 (*right*) threads used per Haswell node



(a) Speedup factorization (b) Speedup solution

Figure 7: The speedup obtained for the factorization and solution process configurations belonging to figure 5f and 5h.

consumption is too high (LAPACK), cf. section 5.1. Moreover, both solvers are not designed for parallel computing. A parallelized alternative for LAPACK is ScaLAPACK [4], but this one is not considered in this paper, while also the general band solver of ScaLAPACK is not suited for this kind of sparse not narrow-banded systems.

More promising results are obtained by CLUSTER_SPARSE_SOLVER and MUMPS. For both solvers it holds that they do not require special knowledge of OpenMP and MPI parallelism to accelerate codes including large sparse systems of linear equations. It goes without saying, that restructuring the codes by applying MPI, and, to a lesser extent, OpenMP will result in better performance. For instance in the recent code, the systems to be solved are stored on a single node and distributed by CLUSTER_SPARSE_SOLVER or MUMPS, which restrict the maximum attainable problem size. Since the reordering process of MUMPS is a purely sequential, we wonder whether it can solve much larger systems. However, this is out of the scope of this paper, where we concentrate on 'easy to get' performance improvements.

6 Numerical experiments with DIELightning

In direct solvers, factorization time is the dominant cost whereas solve time is not, at least not in case of a small number of right-hand sides. But this cost becomes dominant in cases where the number of right-hand sides is large, or when an application iterates on successive solves without refactoring the matrix. In this section we focus on a two-dimensional problem from practice, a streamer discharge simulation code, developed in our research group, in order to profile the most time-consuming parts and to reduce computation time.

The DIELightning code [7], developed by Anna Dubinova during the project *Creeping sparks* is a program to simulate streamer propagation in an axisymmetrical plate-to-plate geometry. Figure 8, made by Dubinova, shows the electric field and electron density of a positive streamer. Also for this code it applies that the greatest challenge is to solve Poisson's equations. She has chosen to solve the systems using MUMPS, a good and reliable package. Her results have been achieved on local shared memory machines each with eight cores, specially purchased for this purpose, because a single run can take more than a week. On these machines MPI was not installed and IFORT compilers were not available. Instead GNU-compilers have been applied, and acceleration has been achieved by OpenMP.

MUMPS can combine MPI and multithreading and apparently, the solve process of 2D Poisson's equations has more advantage of MPI parallelism than of multithreading. The multifrontal approach of MUMPS for this kind of systems results in small matrices, too small to benefit from multithreaded BLAS.

In the DIELightning code different systems of Poisson's equations

$$A_i x_i(t_j) = b_i(t_j), \quad i = 1, \cdots, 4; \quad j = 0, \cdots, N \tag{18}$$

have to be solved, where N can be tens of thousands. The right-hand side vectors $b_i(t_j)$ depend on the simulation time, while the matrices A_i do not change. Suppose the computational domain is of size $M_r - by - M_z$, where M_r denotes the number of cells in the radial direction, and M_z denotes the number of cells in the z direction. Let $M = M_r \times M_z$ then A_1 is of size $M \times M$. System (18) with $i = 1, j = 0$ is solved to calculate the initial background electric field. This happens only once at the initialization phase.

	PARDISO					
# threads	32	16	8	4	2	1
factor(4x)	**7.64**	7.61	**7.57**	7.60	7.57	7.51
solve(2x)	**0.12**	0.11	0.11	0.11	0.11	0.11
solve(248x)	**9.12**	9.02	9.05	8.96	**8.91**	8.95
PARDISO	**16.77**	16.64	16.62	16.56	16.49	**16.47**
left over	4.96	4.39	**4.39**	4.52	4.96	**5.63**
after 101 time steps	21.73	21.03	**21.01**	21.09	21.45	**22.10**
T_{10}	1.35	1.35	**1.29**	1.29	1.33	**1.40**
	CLUSTER_SPARSE_SOLVER					
# MPI processes	1	2	4	8	16	32
# threads	32	16	8	4	2	1
factor(4x)	**6.55**	3.09	**2.70**	3.07	4.11	6.07
solve(2x)	**0.03**	**0.18**	0.08	0.06	0.05	0.04
solve(248x)	**2.29**	**11.97**	4.66	3.49	5.17	3.35
CLUSTER_SPARSE_SOLVER	8.85	**15.07**	7.36	**6.56**	9.28	9.42
left over	4.85	**4.31**	4.61	5.33	7.74	**13.70**
after 101 time steps	13.69	19.38	11.97	**11.89**	17.02	**23.12**
T_{10}	**0.67**	1.56	0.88	0.84	1.24	**1.64**
	MUMPS					
# MPI processes	1	2	4	8	16	32
# threads	32	16	8	4	2	1
factor(4x)	10.69	**11.87**	10.09	8.69	8.48	**7.96**
solve(2x)	0.87	**1.71**	1.14	0.61	0.54	**0.31**
solve(248x)	71.02	**142.20**	110.44	50.90	40.38	**0.89**
MUMPS	81.72	**154.10**	120.53	59.60	48.87	**8.86**
left over	5.12	**4.65**	4.93	5.55	7.70	**13.40**
after 101 time steps	86.83	**158.75**	125.45	65.16	56.57	**22.26**
T_{10}	7.34	**14.15**	11.14	5.44	4.60	**1.34**

Table 1: Problem I. Time (seconds) spent in the factor and solve phases with mixed MPI and OpenMP on a single Broadwell node. CLUSTER_SPARSE_SOLVER and MUMPS are using 32 cores on a single node. The red values correspond with the longest time and the green ones with the shortest time.

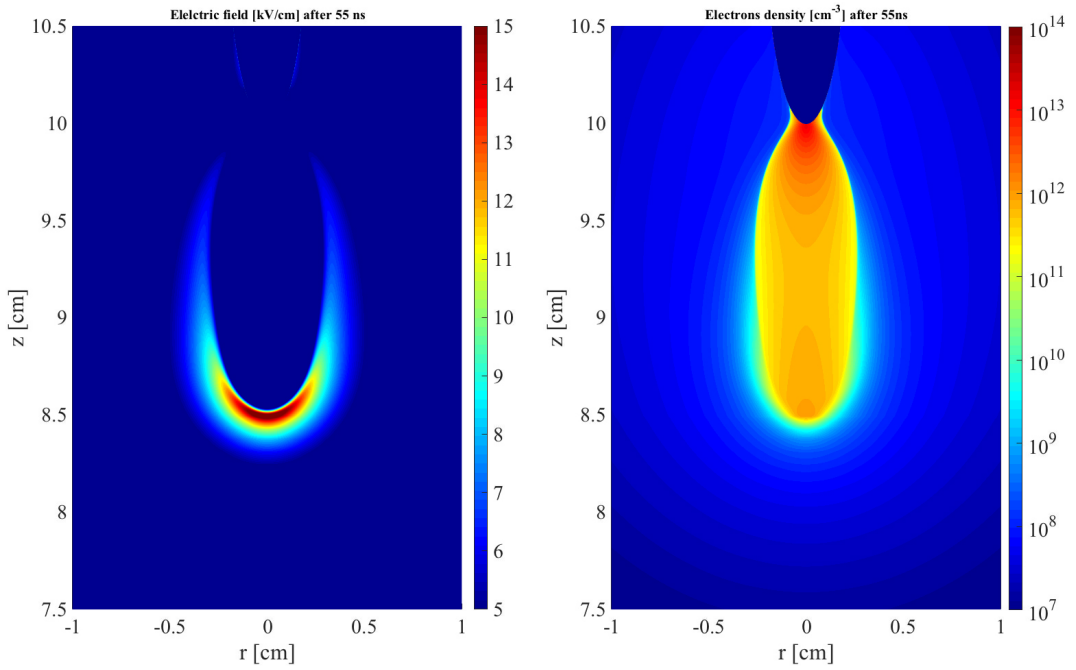


Figure 8: Electric field and electron density of a positive streamer in air at 150 mbar. The streamer starts at an electrode with a voltage of 10 kV. The rod is not present.

The matrices $A_i, i = 2, 3, 4$ are of size $M \times M/2$. Every time step $t_j, j = 0, \cdots, N$, system $A_2 x_2(t_j) = b_2(t_j)$ is solved to update the electric field in radial and z-direction. The matrices A_3 and A_4 match with the calculation of the photoionization; the systems are solved every 10th time step in our experiments. Since the matrices $A_i, i = 1, \cdots, 4$ are constant, the analysis, reordering and factorization of the matrices has to be calculated only once per simulation. Noteworthy, in case of applying AMR, all systems has to be refactorized for each new grid configuration. We remark that the largest system ($i = 1$) limits the size of the domain, but because this system is solved just once at initialization to calculate the initial background electric field, one may consider to solve this system out-of-core (disk is used as an extension to main memory). Both `MUMPS` and `CLUSTER_SPARSE_SOLVER` offer this slow but smart facility.

6.1 Description of Problems and T_{10}

In contrast with the experiments in section 5 the systems (18) are unsymmetric. In table 1, we distinguish two different sets of input parameters, one the default set as offered by `DIELightning`, with $M_r = 500$ and $M_z = 1000$, referred to as Problem I, and another input set with other parameters and with $M_r = 800$ and $M_z = 6000$, referred to as Problem II. Next, we increase the value of M_z to 24000, referred to as Problem III, causing a growth of the Poisson system size by a factor 4. Finally, the largest problem size in our experiments is $M_r = 3200$ and $M_z = 24000$ (Problem IV). Here, the radial grid size and grid size are divided by 4, making the system to

solve 16 times larger than of Problem II.

In streamer discharge modeling the calculation of photoionization is accompanied by calculations of the electrical potential with one of sparse solvers `PARDISO`, `CLUSTER_SPARSE_SOLVER` or `MUMPS`. The tables of this section display: The next tables display:

- **factor(4x)**: the reordering and factorization time of all systems
- **solve(2x)**: the solution time of all $M \times M$ systems
- **solve(248x)**: the solution time of all $M \times M/2$ systems performed during 101 time steps, where the photoionization update has been calculated every 10th step
- `SOLVER`: time spent by one of the Poisson's solvers: `PARDISO`, `CLUSTER_SPARSE_SOLVER` and `MUMPS`, respectively
- **left over**: the remaining time spent by the program
- **after 101 time steps**: time spent by initialization and performing the first 101 time steps

The simulations have not been executed till the end time T_{end} was reached, but are restricted to 101 time steps. The portion of time of the factorization influences the total time in the tables, whereas for a process with tens of thousand of time steps the contribution is nearly negligible. As a conclusion the total time listed in the tables does not indicate which solver, or combination of multithreading and number of `MPI` processes is to be preferred.

Therefore, we do not consider only the time spent after 101 time steps, but estimate the time spent by 10 time steps, T_{10}, on the basis of extended time measurements. Although the time of factorization is by far the most time consuming part, it does not count in the estimation time T_{10}, because the factorizations are executed only once per simulation. However, we would like to record the results, just in case another application requires a regular update of the factorization. All results showed in this section were obtained on Broadwell nodes of the Cartesius. The bold values in the tables in the next sections correspond to the most efficient combination of threads and `MPI` processes.

6.2 Timings results of problem I

Table 1 contains the timing results of problem I with the default set of input parameters as offered by `DIELightning`. The results on a single node are listed for `PARDISO`, `CLUSTER_SPARSE_SOLVER` and `MUMPS`.

- Since the `PARDISO` solver is a shared-memory multiprocessing parallel direct sparse solver, not using `MPI`, we show the results at a varying number of threads. It appears that the shortest wall-clock time is achieved using 8 threads. Also the best estimated time T_{10} is achieved for 8 threads, but the differences are minimal. This applies to almost all intermediate results, except the 'left over' time on a single thread is somewhat larger that using more threads. This means that the parallelization of loops by `OpenMP`, in parts of the program outside calls to `PARDISO`, is paying off.
- The results of `CLUSTER_SPARSE_SOLVER` produce a much more varied picture, even on a single node. The experiments are chosen such that the multiplication

of the number of MPI processes and the number of threads is always 32, being the number of cores on Broadwell nodes. The combination of 8 MPI processes and 4 threads means that each MPI process uses 4 threads. The factorization is optimal for 4 MPI processes, whereas the solution process (solve(2x) and solve(248x)) profit from the maximum number of threads. The same is true for 'left over', the less threads the less advantage can be taken of OpenMP in other parts of the program. The single thread case results in a 'left over' time much larger than the solve time. The best performance is obtained for applying only OpenMP. Obviously, the CLUSTER_SPARSE_SOLVER implementation is twice as fast as the PARDISO implementation, although no MPI is applied.

- It is clear that MUMPS gains from parallelism by MPI. The solve process with 32 MPI processes is significantly faster compared to the other packages, but by the expensive 'left over' time, the overall performance, the estimated time T_{10} is slower. Note that the time spent by MUMPS exhibits large differences. Later in the discussion on problem II we come back to this phenomenon.

For this relatively small problem, amazingly enough, it is preferable to use CLUSTER_SPARSE_SOLVER without applying MPI, rather than the PARDISO variant developed for shared memory applications.

6.3 Timings results of problem II

Problem II is similar to problem I, except that the set of input parameters differs, resulting amongst other things into a larger problem. DIELightning has been designed to calculate accurately positive streamer propagation along a dielectric rod. One of the input parameters determines whether such a dielectric rod is present (problem II) or not (problem I). The simulations of both problems correspond, except that the time step of problem II is smaller, resulting in more steps to be taken to reach the physical end time. The figures in table 2 have been obtained in a similar way as for problem I (see the previous section). Again we do not do any specific effort on the distribution of the systems, the data of all systems is present on just one node. Only the factor and solve parts by CLUSTER_SPARSE_SOLVER and MUMPS are taken advantage of MPI parallelism. OpenMP directives have been added to other parts of the code to gain speedup as well.

- In comparison with the PARDISO results in table 1 the most efficient configuration is achieved for 16 threads and not for 8. The speedup for the estimation time T_{10} is about a factor of 1.5 between the fastest and slowest results.
- Outstanding is the empty column of CLUSTER_SPARSE_SOLVER for 32 MPI processes, the execution runs into an error during the factorization. The result for the (16,2) combination is very promising, much faster than the timing results of PARDISO, except for the 'left over' time. This means that we can not ignore the 'left over' contribution to the physical end time T_{end}.
- The results of MUMPS on a single node are simply disappointing, especially when compared to CLUSTER_SPARSE_SOLVER. It looks like MPI conflicts with OpenMP. In [6] it turned out that Goto BLAS in combination with MUMPS appeared to conflict with the other OpenMP regions, so that the net performance of MUMPS with high numbers of threads turned out to be better with other libraries. It may happen that something similar is the case with the MKL library, that is

	PARDISO					
# threads	32	16	8	4	2	1
factor(4x)	95.9	96.0	**101.9**	**94.5**	97.8	101.0
solve(2x)	1.3	1.2	**1.5**	**1.2**	1.2	1.5
solve(248x)	99.5	96.0	120.5	**95.5**	96.6	**124.0**
PARDISO	195.4	192.0	222.5	**190.0**	192.1	**225.0**
left over	42.5	**39.1**	45.8	41.8	46.9	**67.9**
after 101 time steps	238.0	**231.1**	268.3	231.8	239.0	**292.9**
T_{10}	13.63	**12.33**	16.73	13.07	13.78	**18.44**
	CLUSTER_SPARSE_SOLVER					
# MPI processes	1	2	4	8	16	32
# threads	32	16	8	4	2	1
factor(4x)	**71.6**	35.1	**30.1**	31.9	42.0	-.-
solve(2x)	0.2	**2.1**	0.7	**0.4**	**0.4**	-.-
solve(248x)	**19.7**	**167.9**	54.8	35.7	31.0	-.-
CLUSTER_SPARSE_SOLVER	91.3	**203.0**	84.9	67.6	**73.0**	-.-
left over	**41.1**	47.2	42.3	49.5	**70.4**	-.-
after 101 time steps	132.5	**250.3**	127.2	**117.0**	143.4	-.-
T_{10}	5.81	**20.66**	8.30	8.14	**4.49**	-.-
	MUMPS					
# MPI processes	1	2	4	8	16	32
# threads	32	16	8	4	2	1
factor(4x)	**120.6**	98.5	109.4	98.6	141.4	**91.9**
solve(2x)	8.8	**17.6**	10.0	7.6	5.1	**4.3**
solve(248x)	729.4	**1637.2**	943.1	498.4	585.1	**408.3**
MUMPS	850.0	**1765.3**	1052.6	597.0	726.6	**500.3**
left over	**42.0**	129.4	48.7	50.3	**147.4**	129.1
after 101 time steps	892.1	**1805.7**	1101.3	647.3	874.1	**629.4**
T_{10}	74.34	**161.89**	95.71	52.70	67.46	**51.80**

Table 2: Problem II. Time (seconds) spent in the factor and solve phases with mixed MPI and OpenMP on a single Broadwell node. CLUSTER_SPARSE_SOLVER and MUMPS are using 32 cores. The red values correspond with the longest time and the green ones with the shortest time.

	# nodes = # MPI processes					
# MPI processes	1	2	4	8	16	32
# nodes	1	2	4	8	16	32
# threads	32	16	8	4	2	1
factor(4x)	**120.6**	103.0	94.4	90.3	89.5	**88.3**
solve(2x)	**8.8**	6.3	3.6	2.6	**2.2**	3.0
solve(248x)	**729.4**	584.7	322.8	158.7	160.3	**119.7**
MUMPS	**850.0**	687.8	417.2	249.0	250.0	**208.2**
left over	42.0	**40.1**	40.9	44.7	65.9	**67.7**
after 101 time steps	**892.1**	727.9	458.1	293.7	315.8	**275.9**
T_{10}	**74.34**	60.17	34.83	19.25	19.94	**16.94**
	# threads = 32					
# MPI processes	1	2	4	8	16	32
# nodes	1	2	4	8	16	32
factor(4x)	**120.6**	104.7	95.4	90.6	89.7	**88.7**
solve(2x)	**8.8**	6.3	3.7	2.7	2.0	**1.8**
solve(248x)	**729.4**	585.3	319.0	170.9	160.3	**117.3**
MUMPS	**850.0**	690.1	414.4	266.2	260.6	**206.2**
left over	**42.0**	44.8	45.3	44.7	48.8	**58.9**
after 101 time steps	**892.1**	734.9	457.7	311.5	309.5	**265.1**
T_{10}	74.34	60.59	33.80	20.87	20.47	**15.68**

Table 3: Problem II. Time (seconds) spent in the factor and solve phases with mixed MPI and OpenMP using MUMPS. In the upper part of the table the number of nodes equals the number of MPI processes and the result of multiplying the number of nodes and number of threads is always 32. In the lower part of the table the number of threads equals 32. The red values correspond with the longest time and the green ones with the shortest time.

why we rerun the experiments but now with the number of nodes equal to the number of MPI processes. The results can be found in the upper part of table 3. The results looks much better now, it applies to almost the entire factor and solve phases, that an increase in the number of MPI processes accelerates the code. Only the 'left over' time does not profit of the added nodes. Finally, we have decided to use on each node all threads, in order to reduce the time of 'left over'. The overall gain in execution time is not spectacular, see the lower part of table 3. From the timings presented in section 5, we should expect that at least the solve time of MUMPS should be less than of CLUSTER_SPARSE_SOLVER. Unfortunately, this is not the case.

6.4 Timings results of problem III

Initially, we focused on the performance of problem II on a single node, but the Cartesius is mainly interesting as a supercomputer with many fast nodes. As for MUMPS has been shown in the previous section, that MPI and OpenMP conflict on a single node, we want to examine which combination of MPI and OpenMP parallelism across multiple nodes leads to the best performance for a larger problem, i.e., problem III. In table 4, we show its results, where the number of nodes equals the number of MPI processes. For the CLUSTER_SPARSE_SOLVER implementation it hardly pays to extend the number of nodes, a doubling from 16 to 32 delivers even a small decrease

	# nodes	# threads	# MPI proc.	T_{10}
CLUSTER_SPARSE_SOLVER	1	8	1	28.05
CLUSTER_SPARSE_SOLVER	2	8	2	66.07
CLUSTER_SPARSE_SOLVER	4	8	4	43.80
CLUSTER_SPARSE_SOLVER	8	8	8	42.28
CLUSTER_SPARSE_SOLVER	16	8	16	39.15
CLUSTER_SPARSE_SOLVER	32	8	32	39.96
CLUSTER_SPARSE_SOLVER	16	1	16	35.80
MUMPS	1	8	1	279.23
MUMPS	2	8	2	203.82
MUMPS	4	8	4	144.32
MUMPS	8	8	8	82.34
MUMPS	16	8	16	68.20
MUMPS	32	8	32	57.17
MUMPS	16	1	16	75.98

Table 4: Problem III. Estimate time T_{10} needed for 10 time steps of problem III with mixed MPI and OpenMP using CLUSTER_SPARSE_SOLVER and MUMPS

of T_{10}. From the CLUSTER_SPARSE_SOLVER output on a single node, we observed that not CLUSTER_SPARSE_SOLVER, but PARDISO, was called. resulting in the best performance compared for problem III. Some remarks as 'Cannot allocate memory' appeared in the output of CLUSTER_SPARSE_SOLVER on two nodes, although the results agree with runs on more nodes. Although MUMPS displays that doubling the number of MPI processes leads to a better performance, the speedup from 32 nodes over 1 node is 4.88, it appears that MUMPS for this application is too slow. A remark about the number of threads per node: on 16 nodes CLUSTER_SPARSE_SOLVER is faster when using only one thread per node than 8, while MUMPS benefits from more threads. Summarizing, we may conclude that the use of CLUSTER_SPARSE_SOLVER is preferable to MUMPS for this larger application.

6.5 Timings results of problem IV

Finally, we have benchmarked the largest problem, problem IV. The results of MUMPS are shown in figure 9. Surprisingly, the factor phase seems to be independent of the number of nodes involved. The speedup achieved for the solve phase amounts 5.4 for 32 nodes, while the speedup for the time spent by performing 101 steps (including initialization and factorization), is 2.8. In table 5 the T_{10} values are listed and also their speedup compared to T_{10} on a single node.

In section 5, we remarked that the maximum possible problem size can be achieved when calling the solver CLUSTER_SPARSE_SOLVER in stead of the solver MUMPS, c.f., figure 5. However, here CLUSTER_SPARSE_SOLVER fails to solve problem IV with error message '`not enough memory`', even in case of more than one node.

For problems from practice, like discussed in this section, other rules seem to apply than for the academic problems of section 5.

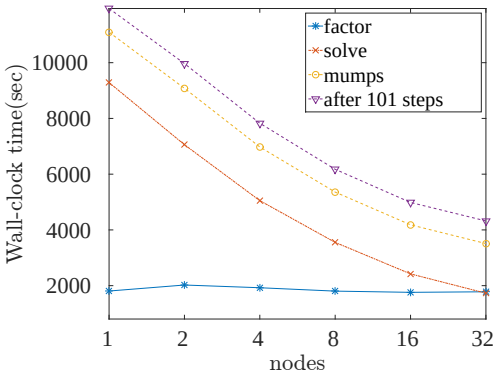


Figure 9: Problem IV. Wall-clock time of factor and solve phase and 'after 101 time steps' and spent by `MUMPS` versus number of nodes. The number of `OpenMP` threads equals 8.

	`MUMPS`					
# nodes	1	2	4	8	16	32
T_{10}	983.02	764.59	567.80	420.78	309.66	243.02
Speedup	1.00	1.28	1.73	2.33	3.18	4.05

Table 5: Problem IV. Estimate time (seconds) T_{10} for 10 time steps calling `MUMPS`, `OMP_NUM_THREADS` = 8.

7 Conclusions

The aim of our experiments was to examine which role the Dutch national supercomputer Cartesius can play for streamer simulations compared to a fast local desktop machine. Since one of the bottlenecks of almost all streamer models is the solution of Poisson's equation, we focused on several packages, which can solve large sparse systems in parallel. However, our basic thought was to concentrate on 'easy to get' performance improvements. We want to avoid any deep restructuring or rewriting of the codes.

The packages discussed in this paper are `PARDISO`, `CLUSTER_SPARSE_SOLVER` and `MUMPS`, all of them can be called from a single node. `PARDISO` is developed for shared memory parallelism and may profit from multithreading. `CLUSTER_SPARSE_SOLVER` and `MUMPS` are designed to get the best performance out of current modern hardware machines, like Cartesius, and apply a hybrid approach of parallel computing: a combination of the message passing model (`MPI`) with the `OpenMP` threads:

- threads perform computationally intensive kernels using local, on-node data.
- communications between processes on different nodes occur over the network using `MPI`.

Our experiments could be divided into two groups.

- solving 2- and 3-dimensional Poisson's equations with known analytic solution. A clear distinction has been made between the three different phases: analytic and reordering, factorization and solution phase.

- profiling of an existing streamer code. On the basis of an estimate of the wall-clock time is predicted which package delivers the best performance. In addition to the range of problem sizes, also many combinations of numbers of `OpenMP` threads and `MPI` processes are examined.

First of all we have shown that `FISHPACK` is no longer the fastest method to solve the Poisson equation. Furthermore, `PARDISO`, but also `CLUSTER_SPARSE_SOLVER` and `MUMPS` remain accurately also for large systems, whereas `FISHPACK` becomes inaccurate for systems larger than 1300×1300.

From section 5, it is clear, that neither parallelization nor the method affects the convergence speed. The maximum problem size is determined by the number of nodes and for a fixed problem size the computation time decreases when more nodes are used. Whether we are using 12 or 24 cores per node makes little difference. The `CLUSTER_SPARSE_SOLVER` package is more memory efficient in use than `MUMPS`, whereby it can solve larger systems on an equal number of nodes. On the other hand, the solve phase of `MUMPS` requires less processing time. Disappointing is the maximum problem size for 3D problems. An adaptive mesh refinement (AMR) approach, as e.g., AFIVO designed by Teunissen [12], may be more suitable for this specific type of 3D problems than a direct solver. We remark that the experiments in section 5, were performed on Cartesian meshes, but the results will be the same for nonuniform grids. However one exception, `FISHPACK` can handle only on uniform grids.

In section 6 we present the timing results of the streamer propagation code `DIELightning`, for three different problem sizes. At first we looked at the performance on a single node. Haswell or Broadwell nodes with many cores per node (more than 20) seem to be an option for this type of problems. Unquestionably, for all investigated problem sizes `CLUSTER_SPARSE_SOLVER` is favorite considering the T_{10} value. It is striking that in case of hybrid parallelism, a number of `MPI` processes in combination with a fixed number of threads per node, such that all cores are used, delivers inconsistent, disappointing timing results.

When we increase the number of nodes up to the number of `MPI` processes, we obtain the performance as we expected. We remark that compute nodes on Cartesius, with relatively many cores per node, are allocated exclusively to one job at a time. The usage of complete nodes, even if only part of the cores is used, will be charged. As a consequence, an increase of the number of nodes leaving the total number of cores constant, makes the performance *less efficient*, or, *more expensive* in terms of system billing units (SBU's). Therefore further optimization of the *black box* solvers for efficient use of all cores in a node is still needed.

Acknowledgments This work was sponsored by NWO Exacte Wetenschappen (Physical Sciences) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organization for Scientific Research, NWO).

The authors would like to thank John Donners and Valeriu Codreanu of SURFsara for their support to get optimal performance on the Cartesius.

Anna Dubinova was kind enough to make available her `DIELightning` code. Although it is very good piece of software, written in `FORTRAN 2003`, she was always willing to answer our questions. We want to thank her for the good collaboration. In addition, we would like to thank Willem Hundsdorfer for reading carefully this paper and for his comments.

References

[1] Intel® MKL . https://software.intel.com/en-us/intel-mkl.

[2] MUMPS. http://mumps.enseeiht.fr/.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM Pub., Philadelphia, third edition, 1999.

[4] S. Blackford, J. Choi, A. Cleary, E.D'Azevedo, J. Demmel, I. Dhillon, J.J. Dongarra, S. Hammerling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*. Software, Environments and Tools. 1997. http://dx.doi.org/10.1137/1.9780898719642.

[5] E.F.F. Botta, K. Dekker, Y. Notay, A. van der Ploeg, F.W. Vuik, C. Wubs, and P.M. de Zeeuw. How fast the Laplace was solved in 1995. Technical Report W-9701, 1997.

[6] Indranil Chowdhury and Jean-Yves LExcellent. Some Experiments and Issues to Exploit Multicore Parallelism in a Distributed-Memory Parallel Sparse Direct Solver. Technical Report RR-7411, 2010. inria-00524249.

[7] A.A. Dubinova. *Modeling of streamer discharges near dielectrics*. PhD thesis, Technische Universiteit Eindhoven, 2016. https://pure.tue.nl/ws/files/30935440/20160901_Dubinova.pdf.

[8] I.S. Duff and J.K. Reid. The Multifrontal Solution of Indefinite Sparse Sysmetric Linear Equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[9] OpenMP Forum. OpenMP application interface, version 2.5. [online], May 2005. http://www.openmp.org.

[10] W. Gropp, S. Huss-Ledermann, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference Vol. 2*. MIT Press, 1998.

[11] W. Hundsdorfer and J.G. Verwer. *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations*. Springer Series in Comput. Math. Springer, 2003.

[12] Teunissen Jannis and Ebert Ute. Afivo, a framework for quadtree/octree AMR with shared-memory parallelization and geometric multigrid methods. 2017.

[13] Kalinkin, Anders, Anders, Kuznetsov, Shustrov, and Pudov. Sparse Linear Algebra support in Intel Math Kernel Library. Coventry, England.

[14] Alexander Kalinkin, Anton Anders, and Roman Anders. Intel Direct Sparse Solver for Clusters, a research project for solving large sparse systems of linear algebraic equations on clusters, 2013. Differential Equations. Function Spaces. Approximation Theory conference.

[15] Alexander Kalinkin and Konstantin Arturov. Asynchronous Approach to Memory Management in Sparse Multifrontal methods on Multiprocessors. *Applied Mathematics*, 4, Dec 2013. http://www.scirp.org/journal/am.

[16] George Karypis. METIS. http://glaros.dtc.umn.edu/gkhome/views/metis.

[17] A. Kuzmin, M. Luisier, and O. Schenk. Fast methods for computing selected elements of the Greens function in massively parallel nanoelectronic device simulations. In F. Wolf, B. Mohr, and D. Mey, editors, *Euro-par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 533–544. Springer Berlin Heidelberg, 2013.

[18] Chao Li. *Joining particle and fluid aspects in streamer simulations*. PhD thesis, Technische Universiteit Eindhoven, 2009.

[19] Carolynne Montijn. *Evolution of negative streamers in nitrogen : a numerical investigation on adaptive grid*. PhD thesis, Technische Universiteit Eindhoven, 2005. `https://pure.tue.nl/ws/files/2999953/598717.pdf`.

[20] Amestoy Patrick, Guermouche Abdou, L'Excellent Jean-Yves, and Pralet Steṕhane. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32, November 2005.

[21] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004.

[22] O. Schenk, K. Gartner, and W. Fichtner. Efficient sparse lu factorization with left-right looking strategy on shared memory multiprocessors. 1(40):158–176, 2000.

[23] Olaf Schenk. *Scalable Parallel Sparse LU Factorization Methods on Shared Memory Multiprocessors*. PhD thesis, ETH Zurich, 2000. PhD thesis.

[24] Olaf Schenk and Klaus Gärtner. *PARDISO, User Guide Version 5.0.0*, Updated Febrary 07, 2014. `http://www.pardiso-project.org`.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference Vol. 1, The MPI Core*. MIT Press, 1998.

[26] P. Swarztrauber, J. Adams, and R. Sweet. Fishpack90. `https://www2.cisl.ucar.edu/resources/legacy/fishpack90`.

[27] Jannis Teunissen. *3D Simulations and Analysis of Pulsed Discharges*. PhD thesis, Technische Universiteit Eindhoven, 2015. `https://pure.tue.nl/ws/files/8728008/20151112_Teunissen.pdf`.