Numerical tools and optimizations for EM geophysical methods

Tue Boesen[1, *]

[1] *Aarhus University*

(Dated: September 21, 2017)

* tue@geo.au.dk

CONTENTS

INTRODUCTION

Numerical modelling is becoming an increasingly important tool for any scientist, and the continued exponential growth in computational power enables the possibility of ever more realistic and powerful simulations.

Before multicore CPU's became common, around 2005, optimizing numerical code largely meant to make sure that the algorithm was mathematical optimal, while tricks that required in-depth hardware knowledge like: the number of clock cycles various mathematical operations take, or how to avoid cache misses, rarely meant more than a speedup factor of ∼10x. Moreover, before 2005 CPU advancements largely happened through faster clock frequencies, and thus old code could more or less run optimally on a new CPU without any modifications. Today this is no longer so. Since the introduction of the multicore CPU, processor advancements have largely happened through parallelization and vectorization. To the point where old serial code that does not use either of these advancements can easily loose out a computational speedup of ∼50x or even upwards of ∼500x on a modern CPU designed for heavy computational use (Sodani et al. 2016). In this context, it is important to note that this speedup is on top of the ∼10x speedup previously mentioned.

While vectorization is often partially utilized implicitly by compilers, a concious effort requiring intrinsic knowledge is generally required in order to design algorithms that can run optimally on modern hardware. Most scientist are experts in their own field, but lack the in-depth hardware knowledge required to design optimal algorithms. Because of this, it is imperative that general heavily optimized numerical algorithms are designed, and made available for the advancement of scientific modelling. BLAS and LAPACK, (Anderson et al. 1999), are perhaps the most well-known examples of such tools. In the following, I will present the various parallelization methods commonly used, explain why we chose OpenMP parallelization. Following this I will discuss the various issues this lead to in our code, and how we solved these by developing a general parallelization framework, designed for portability and a wide range of users. Building on top of this framework I present our highly optimized iterative sparse linear solver, which is able to achieve super-linear scaling on the linear systems we arrive at in geophysical EM inversions.

PARALLELISM

Parallelizing generally exist at three different levels. The highest level of parallelization is between different systems in a distributed memory environment, where the environment can either be a cluster or a grid. In both cases the system consist of separate systems, which communicate through a network, with message passing interface (MPI) (Gropp et al. 1999). If a problem needs a lot of computational power, then clusters with MPI are generally the only way to achieve it, since any number of systems can be combined to make an almost arbitrarily powerful cluster/grid. However, most programs do not require a supercomputer to run on, and since a single modern computer can reach more than 3 TFLOPS of performance, this is sufficient for solving many problems, provided the compute power can be used efficiently. The next level of parallelization is of the processing units on an individual system. CPU parallelization commonly occurs through the use of OpenMP directives (Dagum and Menon 1998), while GPU parallelization can be done with CUDA directives (Nickolls et al. 2008). The final level of parallelization happens on individual processing units, and is commonly referred to as vectorization. Vectorization is largely done automatically by the compiler, though there can still be a significant performance boost if done manually, since automatic compiler vectorization has to be very safe and conservative. It is important to note that the different kinds of parallelization are not mutually exclusive, though combining all levels of parallelization can be a difficult task.

Our code does currently not use MPI programming, for which there are several reasons: One reason being that MPI programming requires a significant restructuring of the already existing code. Furthermore MPI programming has a very large communication overhead and low communication bandwidth, which means that acceptable scaling can often be hard to achieve. But the main reason why MPI parallelization has not been implemented is that using a single computer system have thus far proven to be sufficient for the modelling we do. Instead our code relies on CPU parallelization through OpenMP, and vectorization.

OpenMP parallelization was chosen because OpenMP directives are designed for gradual parallelization of existing sequential code, while GPU parallelization was disregarded since it is only suited for very narrow and extremely parallel jobs, and not for the multi-purpose modelling program we have. During the last 5 years, most of the vital and processor-heavy algorithms have been parallelized in AarhusInv. However this parallelization did not come without issues. In the following several issues will be highlighted that led to the creation of our general parallelization

framework.

Portability

Our code is intended to be used by many people, with various computer systems. Ideally it should run optimally on all those systems, whether they are: NUMA or UMA systems (Panourgias 2011), contains 4 or 64 cores, has hyperthreading enabled, and whether it is a dedicated or shared machine.

Broad userbase

Our users range from novices to experts, with various desires for controlling how the code runs on their systems. As such we ideally want a program which can run decent autonomously, but where the expert user still has full thread and affinity control, if desired.

Complex code

The code contains various different modelling schemes developed by a lot of different people. Parts of the code use simple loop parallelization, others use more advanced nested parallelization, furthermore some parts can be memory bandwidth limited, and in such parts, it is advantageous to use fewer parallelization threads, in order to prevent bottleneck regions in the code. The result of this, was that we had a wide variety of parallelization schemes, which all made sense in their individual cases, but collectively were hard to maintain. Furthermore autonomous code becomes increasingly complex with the number of exceptions and special cases, that needs to be accounted for.

Global directives

OpenMp calls are global directives, and setting them in one subroutine can affect code outside that subroutines scope. This led to unintended cross-modular behaviour, especially when combined with the fact that developers often reuse - as they should, general subroutines from other parts of the code. Finally, it often resulted in people copying existing subroutines and changing the parallelization to fit their particular parallelization schemes, which makes maintenance of the code a nightmare.

Goals for our parallelization framework

Based on these issues, a set of goals were created, that the framework should fulfil:

- Portability - make the code run optimally on various different systems and architectures (NUMA/UMA/Hyperthreading).
- Make the code run well on both shared and dedicated servers.
- Enable autonomous parallelization, as well full control for the expert user.
- Create a simple, yet general parallelization structure, that can contain all the previously used parallelization schemes.
- Create a single parallelization module, that contains all parallelization information, and does all the global OpenMp directives needed, such that the general developer should never set these global directives manually.
- Create a dynamic parallelization structure, which removes the need for having multiple copies of the same routines for various degrees of parallelization.

With these goals in mind the following parallelization framework was created.

```
NCPUs                !Total number of threads
NCPUsLow             !Total number of threads in a memory bandwidth limited region
AffinityMode         !Dedicated or shared server?
AffinityPattern      !Scattered or compact affinity?
StartThread          !ThreadID of first affinity binding
UseNested            !Use nested parallelization?
    NCPUOuter        !Only used if UseNested=1, Number of CPUs in first parallel region
```

TABLE I. The input parameters for the parallelization framework. Any of these parameters can be specified by the user to dictate how the prallelization should occur, or set to -99 to be automatically defined

A general parallelization framework

The result of all this, is the following OpenMP parallelization framework created in Fortran, which fulfils all the goals previously listed. The parallelization framework can be found on: ??. The download contains the parallelization framework, as well as a small example showing how to setup and use it.

In short the module contains an initialization routine, where the user can specify the parameters shown in Table I, however these parameters can also be set to -99, in which case the initialization routine will get hardware information from the system and set the parameters autonomously.

Once the initialization is done any parallel region should be called with the StartOpenMP function, as demonstrated in the example. StartOpenMP will determine whether parallelization should occur at this point in the code, and how it should occur.

Limitations and caveats

This parallelization framework has been used for roughly a year in our code, so while it has been tested quite thoroughly, it is likely that some bugs still remain. The framework was created using OpenMP 4, which have limited directives for nested parallelization. In the future it is possible that smarter or more optimal ways of building this framework will occur. For instance, current OpenMP versions does not support different affinity-settings for different levels of parallelization. Furthermore, the framework only support two layers of nested parallelization, since our code has never needed more than two layers at any point. Though extending the code to more layers is fairly simple.

Finally it should be noted that the framework only works for windows machines, and while most functions can be used directly on other systems, the automatic hardware determination will likely need to be completely rewritten.

SPARSE LINEAR SOLVERS

Let A be a $n \times n$ matrix, and b a $n-$vector, a commonly encountered problem in numerical modelling, is then to find the $n-$vector x that solves the linear system:

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

Linear systems are generally easy to solve, and many efficient tools exists for this singular purpose. However, if the linear systems become sparse and large, then solving them efficiently often becomes a non-trivial issue.

The methods for solving sparse linear systems are generally split in two categories: direct solvers, and iterative solvers. While favour seems to have shifted towards direct solvers within the last decade in the EM geophysics community, both methods have their strengths and weaknesses, as will be discussed in the following.

Direct solvers

Direct solvers do a full factorization of the system matrix, which can take quite a while, and use a lot of memory. However, once such a factorization is done, the system can efficiently be solved for any number of right-hand sides. For

direct solvers there exist a number of ready made libraries. This makes using direct solvers relatively straightforward. Amestoy et al. (2001) introduced the direct solver named MUMPS, which is fully parallelized using the MPI-interface, and thus making it capable of operating in parallel on a cluster. Schenk et al. (2000) introduced the direct solver used in the Pardiso library, which is parallelized using the OpenMP interface, and thus making it capable of running in parallel on a single computer. MUMPS and Pardiso are currently two state of the art direct solvers, and which should be employed, depends on the level of parallelization desired. A slightly outdated comparison of direct solvers can be found in Gould et al. (2007). The biggest problem with direct solvers is the fact that they are neither capable, nor efficient, at handling very large linear systems. This is because, even though the system matrix might be extremely sparse, there is no guarantee that the factorization matrix will remain sparse. A memory consumption of 10-40 times that of the original matrix is not unheard of (Rücker et al. 2006).

Iterative solvers

Iterative solvers use either no factorization, or at most an incomplete factorization, followed by multiple forward- and backward-substitutions, in order to iteratively converge towards a solution to the linear system. Unlike direct solvers, there exists no efficient, and ready to use library, for parallel iterative solvers, that I am currently aware of. The main reason for this is probably that, there exists a lot of different algorithms, and which one is superior is heavily system dependent (Ern et al. 1994, Nachtigal et al. 1992). Oldenburg et al. (2012) compares the use of direct and iterative solvers for the case of 3D EM forward modelling, and conclude that direct solvers are favourable for these kinds of problems. While this might indeed be the case, my personal experience with iterative solvers, suggests that the computational times shown in Oldenburg et al. (2012) for iterative solvers can be improved significantly by applying more optimal preconditioners, and thus the question of which method is superior remains.

Building an iterative solver is a matter of choosing an appropriate propagation method as well as good preconditioners, in the following I will go through our iterative linear solver.

Propagation methods

The first thing needed to iteratively solve a linear system is a propagation method. The number of propagation methods spread throughout literature can seem overwhelming, though Saad (2003) contains the most common algorithms used for iteratively solving sparse linear systems. In our solver we use conjugate-gradient (CG) (Hestenes and Stiefel 1952), which is known for its stability and widely used in our field of science (Brodie and Sambridge 2006, Kirkegaard and Auken 2015, Wu 2003). CG is valid for real positive definite systems, and is known for being very fast and stable.

Preconditioners

A preconditioner is essentially any transform to the linear system, before or during the propagation, such that the solution space becomes easier to probe iteratively. Experience within the iterative linear solving community have shown that proper preconditioning is of vital importance, since this to a large extent determines: memory requirements, solution time, and stability of the solver (Saad and van der Vorst 2000). Preconditioning generally comes in two forms when solving sparse linear systems. It can either be an initial preconditioner, which is applied before using any iterative solver. In that case, the job of the preconditioner is to be an inexpensive operation that either approximate the inverse of A or in some other way transforms A, such that the linear system becomes easier to iteratively solve.

The other kind of preconditioner is applied iteratively, while using the propagation method. Mathematically the job of such a preconditioner is to transform the eigenvalue distribution of the linear system, since the smaller the ratio between largest and smallest eigenvalue is, the faster the system will in general converge.

In our linear solver we currently use both kinds of preconditioners.

Initial preconditioners

The first preconditioner, which we always use is a reverse Cuthill-Mckee (RCM) reordering algorithm (Cuthill and McKee 1969), which is used before applying the iterative propagation method. In our particular case the reordering is not done explicitly, but rather implicitly when the matrix for the linear system is first created, as detailed in Kirkegaard and Auken (2015). This is highly beneficial for several reasons. First, reordering algorithms are generally rather expensive, however this way the reordering is done only once, and only on the soundings which the linear systems is based on, and since the soundings constitute a much smaller system, than the actual linear system used during a geophysical inversion, the compute time for the reordering becomes negligible in our case. Secondly because the reordering is done on the soundings, it preserves the sounding integrity, which is crucial for the block splitting done during the paralllelization of our iterative linear solver, as detailed later.

A reordering of the linear system is essential for two reasons. The first reason is that incomplete Cholesky (IC) factorization, which is one of our other preconditioners, only works well if the number of relevant elements in the factorization matrix is relatively evenly distributed among the different rows. Without reordering this is not so, and thus IC factorization would perform very poorly as a preconditioner (Kirkegaard and Auken 2015). The second reason why a reordering algorithm is crucial is due to the block parallelism, which we impose as a way of parallelizing the linear solver. In order for the block parallelism to be successful, it requires that no vital information is located far from the diagonal, which the RCM reordering algorithm helps ensure.

For completeness sake, the iterative solver includes an RCM reordering algorithm, however since the algorithm is not employed in our normal code it is neither parallelized nor optimized. A more effective RCM algorithm could likely be built based on the algorithm suggested in Karantasis et al. (2014).

Another initial preconditioning that we do is to normalize the diagonals of the matrix representing the linear system. This is done by letting D be the square root of the diagonal elements of A. The linear system is then transformed by multiplying equation 1 with D^{-1} from the left, which gives:

$$D^{-1}AD^{-1}Dx = D^{-1}b, \tag{2}$$

which can be rewritten as:

$$\tilde{A}\tilde{x} = \tilde{b}, \tag{3}$$

where $\tilde{A} = D^{-1}AD^{-1}$, $\tilde{x} = Dx$, and $\tilde{b} = D^{-1}b$. Thus all the diagonal elements become unity, which speeds up the iterative convergence significantly.

Iterative preconditioners

The other kind of preconditioners, are employed iteratively during the propagation. In our case we have two iterative preconditioners, where we use either one or the other, depending on the difficulty of the linear system. The difficulty of a linear system is not something that is very well understood, however the diagonal dominance, d, is often used:

$$d = \min_{i=1}^{n} \frac{|A_{ii}|}{\sum_{j\neq i}^{m_i} |A_{ij}|} \tag{4}$$

where n is the number of rows in A, and m_i is the number of elements in the i'th row of A. The diagonal dominance gives an estimate of the size of the diagonal in comparison to the off-diagonal elements, which is related to the eigenvalue distribution through Gershgorin's theorem (Saad 2003, p. 117).

In our iterative solver we use the average diagonal dominance, $\bar{d}$, to determine which preconditioner to employ for a given linear system.

$$\bar{d} = \frac{1}{n}\sum_{i=1}^{n} \frac{|A_{ii}|}{\sum_{j\neq i}^{m_i} |A_{ij}|}, \tag{5}$$

For simple systems, a symmetric Gauss-Seidel preconditioner, M_{SGS} is utilized:

$$M_{SGS} = (D - E)D^{-1}(D - F), \tag{6}$$

where $A = D - E - F$, with D being the diagonal elements of A, E the strict lower triangular half, and F the strict upper triangular part. The good thing about using Gauss-Siedel methods is that no initial factorization is needed, instead the algorithm can be directly applied during the forward- and backward-substitution during the iterative propagation. This makes propagation using symmetric Gauss-Seidel preconditioning extremely fast, when only a few iterations are needed in order to reach convergence. Gauss-Seidel can be proven to converge, if the system is diagonally dominant (Bagnara 1995), but though convergence is assured Gauss-Seidel preconditioning converges poorly in many cases. Based on experience we only use Gauss-Seidel preconditioning, when the average diagonally dominance is 10 or greater.

For more difficult linear systems an IC factorization with dual dropping strategy is used as an iterative preconditioner (Saad 1994). The IC factorization computes a lower triangular matrix L and an upper triangular matrix U such that the matrix A can be factorized as:

$$A = LU + E \tag{7}$$

where E is an error matrix. The dual dropping strategy ensures that only the n largest elements of each row is kept, where n is an input parameter, and only if those elements are above a certain threshold value. Using these dropping criteria gives an upper bound on both the memory consumption of the factorization, as well as the time required to compute it. As previously mentioned, it is imperative for a successful IC factorization, that the elements are somewhat evenly distributed among the rows in the matrix. However, if this is the case then the method can be very effective.

Block parallelism

Solving a linear system is inherently a sequential problem, and simple parallelization attempts generally scale poorly for dense linear systems. For sparse linear systems, it is however possible to get decent scaling, and in particular for the linear systems arising in EM geophysics, as will be shown.

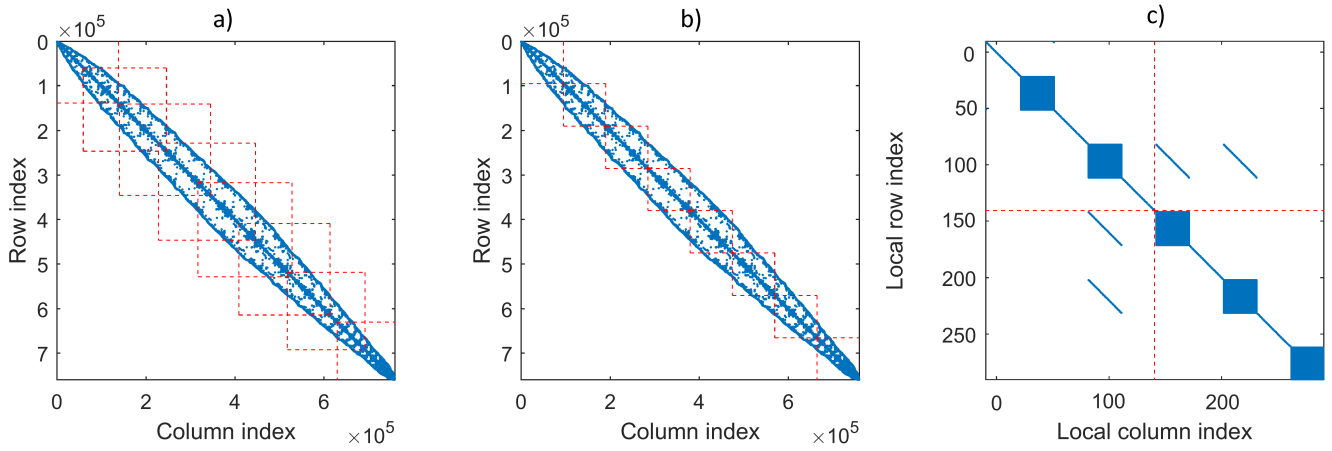

FIG. 1. Characteristic matrix sparsity pattern. Non-zero elements of the matrix are shown as blue dots, while the striped red lines indicate block-splitting boundaries. a) shows a block-splitting with overlap. b) shows a block-splitting without overlap, c) shows a zoom-in near a block boundary without overlap. The filled blue squares near the diagonal represent data from individual soundings, while the off-diagonal blue lines represent constraints between soundings.

Our parallelization method of the linear system consists of splitting the linear system into n or fewer blocks, where n is the number of parallel threads desired. In order for this to work, it is imperative that there are no significant elements in A far from the diagonal, which is one of the reasons why a RCM reordering was part of the preconditioning. Once the matrix has been reordered, the first version of our iterative solver used a blocksplitting algorithm that determined the max distance an element is placed from the diagonal and created overlapping blocks with this amount of overlap

```
Example.f90            !Contains a small example, that initialize the parallelization, loads a matrix and
    solves the linear system
Solver.f90             !Contains the iterative sparse solver
SparseMatrix.f90       !Contains various sparse matrix functions needed by the iterative solver
Paralllelization.f90   !Contains the parallelization framework
Misc.f90               !Contains various small helping routines needed by the other modules
Load.f90               !Contains a minimalistic structure for loading the matrix, and setting the various
    settings for the example
Matrix.zip             !Contains a matrix saved in compressed sparse row format
Readme.txt             !Contains information needed for running the example
```

TABLE II. The files contained in the iterative solver, as well as an exmplanation of the various files.

```
[A,x,b] = RCM(A,x,b)           !RCM Reorder the linear system.
[A,x,b]=Normalize_diag(A,x,b)  !Normalize the diagonal elements of A and transform the system accordingly.
Blocksplitter(A)               !Split the linear system into blocks.
d=GetDiagDominance(A)          !Find the diagonal dominance of A, see eq 4
if d<Threshold then            !System not sufficiently diagonally dominant for Gauss-Seidel
  P=IC                         !Choose Incomplete Cholesky as preconditioner
  M=IC_Factorization(A)        !Compute Incomplete Cholesky factorization of each block
else
  P=SGS                         !Choose Symmetric Gauss-Seidel as preconditioner
end if
x=ApplyPropagation(A,x,b,P)    !Iteratively solve the full system using the appropriate block
    preconditioner.
```

TABLE III. Pseudo-code description of the iterative solver.

between each block, as shown in Figure 1 a). The overlap ensures that no information is lost during the parallel block propagation of the linear system. In practice, it turned out that this overlap was not needed, when solving the linear systems we encounter in EM, provided the soundings are RCM reordered. Thus a much faster linear solver without overlap was created and used in the solver, see Figure 1 b). The reason why block-splitting without overlap works for our problems, is because all the vital information lies very close to the diagonal, while only non-vital information, about constraints between soundings, lie far away, see Figure 1 c). Note that in order for the non-overlapping block solver to be successful, it is imperative that the block-splitting is done such that no sounding information is split, thus a splitting should only be done in-between soundings as done in Figure 1 c).

With the above block-splitting, most of the difficulties about parallelizing the linear solver have been removed, especially in the non-overlapping case, where the large linear system essentially have been transformed to a series of smaller independent linear systems. However parallelized versions of both: CG, symmetric Gauss-Seidel and IC factorization still had to be made.

Our iterative sparse linear solver

The above solver was created, and can be found here: ??. An overview of what is contained in the code can be seen in Table II. It should be noted that the solver operates on sparse matrices in compressed sparse row format.

Table III contains a holistic overview of how the iterative solver operates, written in pseudo-code.

Solver performance

Once the solver was completed, it was thoroughly validated and benchmarked with various linear systems relevant to us. Furthermore comparison with Pardiso was conducted, the results of which can be seen in Figure 2. Overall

the results show that our solver performs rather poorly in comparison to Pardiso, at low levels of parallelization, but because of the superior scaling our iterative solver has, it outperforms Pardiso at high levels of parallelization. However this comparison isn't completely fair since Pardiso does a zero order reordering before factorization, which does not scale well, while the matrices are already built in an optimal way that removes the need for reordering in the case of IC factorization. Nevertheless, the fact that the iterative solver is able to achieve super-linear scaling is noteworthy, but not unheard of in highly optimized code. The reason this is possible, is likely twofold. First, as the matrix is cut into smaller pieces, more of the non-essential constraints are removed from the preconditioner factorization, while this makes the factorization less accurate, it also makes it faster. The second contributing factor to the super-linear scaling is the more standard reason, called the cache effect Eggers and Katz (1989). In short, every CPU has a cache associated with it (modern CPU's have several), the cache can be thought of as the CPU's local memory, and reading information from the cache is much faster than reading it from memory. However, the cache is very small, so it can only contain very limited information. Because of the limited cache storage, modern CPU's use advanced prediction algorithms to anticipate which information the CPU will need next from the cache. Whenever the information needed by the CPU is in the cache, it is called a cache hit, and otherwise a cache miss. The cache effect comes from having multiple threads working on similar data, if those threads happen to share the same cache, then what might have been a cache miss if only one thread was employed can become a cache hit when multiple threads are employed, since the relevant data was transferred to the cache for another thread.

As mentioned, the splitting of the linear system results in a less accurate preconditioning matrix, which can also be seen in Figure 2 c), where the number of propagation needed to reach convergence is shown to rise steadily with the number of threads, due to the rising inaccuracy in the preconditioner. It should be noted that the full matrix is still being solved, it is merely the preconditioning doing a slightly worse job at it, and thus additional propagations steps are required.

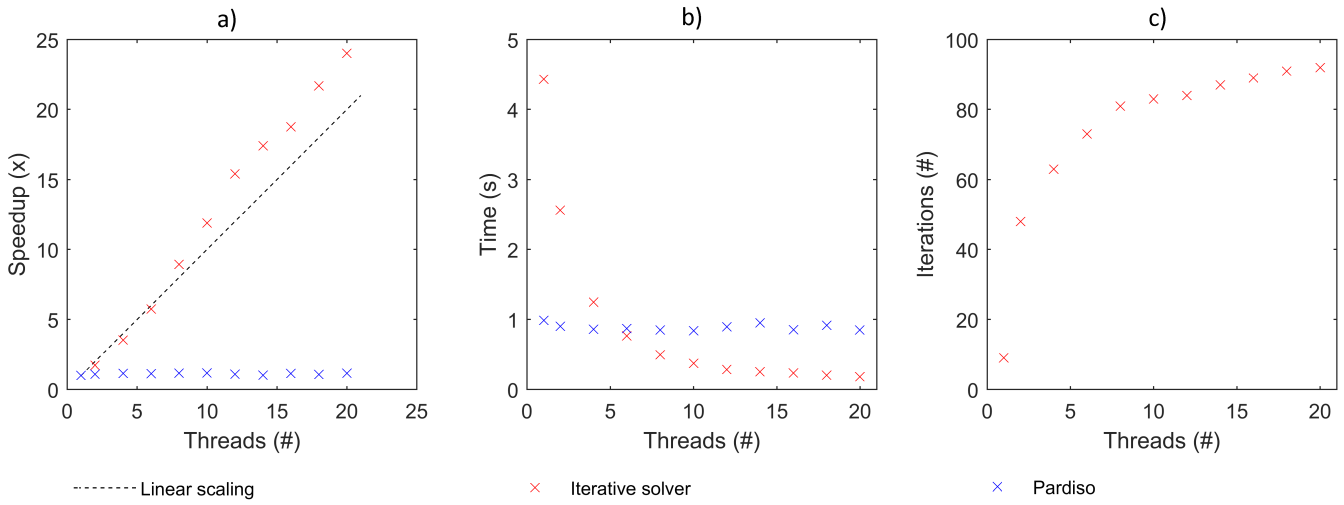

FIG. 2. Benchmarking the iterative solver, on a system with a matrix containing 42k rows and columns, with 500k elements. a) shows the speedup of the iterative solver and Pardiso. While the iterative solver reaches super-linear scaling, Pardiso scales very poorly. b) shows solving time as a function of threads. Pardiso is faster for low thread numbers, but because of its poor scaling the iterative solver outscales it around 6 threads. c) shows the number of propagation required for the iterative solver to reach convergence as a function of threads.

Both the solvers compute time and memory consumption has previously been shown to scale linearly with the matrix size, and thus the solver is capable of handling very large linear systems (Kirkegaard and Auken 2015). Successful tests have been performed on matrices as large as $\sim 6 \cdot 10^6$ rows/columns with $\sim 110 \cdot 10^6$ elements.

Limitations and caveats

The solver included here only covers positive definite matrix systems with real numbers. A complex and more general version which uses the BiCGStab routine was not included (van der Vorst 1992), since our code does not currently solve complex matrices, and thus the complex version has never been thoroughly tested.

The solver has primarily been tested on matrices created from 1D or 2D EM modelling, though a few tests have also been done on 3D modelling systems. While the dimensionality of the problem plays a role, the primary criteria for success is that the data is structured using an RCM reordering algorithm. For problems outside EM modelling it is likely that some of the parameters and thresholds need to be re-set in order to achieve optimal performance. All problems currently tested on the solver have been RCM reordered implicitly, and it is highly recommended that any matrix given to the solver remains so in the future, see Kirkegaard and Auken (2015) for the importance of using RCM reordering before solving.

RESULTS AND CONCLUSION

An easy to use, yet highly optimized, parallelization framework has been built. The framework can run autonomously or with simple user input. It furthermore removes the necessity of setting global OpenMP variables throughout the code, which is known to be a dangerous coding practice from the early days of coding where this was common practice. The parallelization framework follows best codings practices, and promotes a safer and more dynamic parallelization environment. The parallelization framework is also incorporated in the second coding example provided, which is a sparse iterative linear solver.

A sparse iterative linear solver, with super-linear scaling has been created. The solver is optimized for geophysical EM problems which have been implicitly reordered using RCM, but should work well on any real positive definite sparse linear system. The solver is designed for modern multicore systems and relies on OpenMP parallelization. Because of the super-linear scaling the solver outperforms the direct solver Pardiso when used on a modern system, and is capable of solving much larger systems than Pardiso.

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

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition.

Bagnara, R. (1995). A unified proof for the convergence of jacobi and gauss–seidel methods. *SIAM review*, 37(1):93–97.

Brodie, R. and Sambridge, M. (2006). A holistic approach to inversion of frequency-domain airborne em data. *Geophysics*, 71(6):G301–G312.

Cuthill, E. and McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

Eggers, S. J. and Katz, R. H. (1989). *The effect of sharing on the cache and bus performance of parallel programs*, volume 17. ACM.

Ern, A., Giovangigli, V., Keyes, D. E., and Smooke, M. D. (1994). Towards polyalgorithmic linear system solvers for nonlinear elliptic problems. *SIAM Journal on Scientific Computing*, 15(3):681–703.

Gould, N. I., Scott, J. A., and Hu, Y. (2007). A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):10.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press.

Hestenes, M. R. and Stiefel, E. (1952). *Methods of conjugate gradients for solving linear systems*, volume 49. NBS.

Karantasis, K. I., Lenharth, A., Nguyen, D., Garzarán, M. J., and Pingali, K. (2014). Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932. IEEE Press.

Kirkegaard, C. and Auken, E. (2015). A parallel, scalable and memory efficient inversion code for very large-scale airborne electromagnetics surveys. *Geophysical Prospecting*, 63(2):495–507.

Nachtigal, N. M., Reddy, S. C., and Trefethen, L. N. (1992). How fast are nonsymmetric matrix iterations? *SIAM Journal on Matrix Analysis and Applications*, 13(3):778–795.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with cuda. *Queue*, 6(2):40–53.

Oldenburg, D. W., Haber, E., and Shekhtman, R. (2012). Three dimensional inversion of multisource time domain electromagnetic data. *Geophysics*, 78(1):E47–E57.

Panourgias, I. (2011). Numa effects on multicore, multi socket systems. *The University of Edinburgh*.

Rücker, C., Günther, T., and Spitzer, K. (2006). Three-dimensional modelling and inversion of dc resistivity data incorporating topographyi. modelling. *Geophysical Journal International*, 166(2):495–505.

Saad, Y. (1994). Ilut: A dual threshold incomplete lu factorization. *Numerical linear algebra with applications*, 1(4):387–402.

Saad, Y. (2003). *Iterative methods for sparse linear systems*. SIAM.

Saad, Y. and van der Vorst, H. A. (2000). Iterative solution of linear systems in the 20th century. *J. Comput. Appl. Math.*, 123(1-2):1–33.

Schenk, O., Gärtner, K., and Fichtner, W. (2000). Efficient sparse lu factorization with left-right looking strategy on shared memory multiprocessors. *BIT Numerical Mathematics*, 40(1):158–176.

Sodani, A., Gramunt, R., Corbal, J., Kim, H.-S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y.-C. (2016). Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46.

van der Vorst, H. A. (1992). Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644.

Wu, X. (2003). A 3-d finite-element algorithm for dc resistivity modelling using the shifted incomplete cholesky conjugate gradient method. *Geophysical Journal International*, 154(3):947–956.