DELFT UNIVERSITY OF TECHNOLOGY

REPORT 12-12

A parallel linear solver exploiting the physical properties of the underlying mechanical problem

F.J. Lingen, P.G. Bonnier, R.B.J. Brinkgreve, M.B. van Gijzen, and C. Vuik

ISSN 1389-6520

Reports of the Delft Institute of Applied Mathematics

Delft 2012

Copyright © 2012 by Delft Institute of Applied Mathematics, Delft, The Netherlands.

No part of the Journal may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission from Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands.

A parallel linear solver exploiting the physical properties of the underlying mechanical problem

F.J. Lingen P.G. Bonnier R.B.J. Brinkgreve M.B. van Gijzen
C. Vuik *

September 13, 2012

Abstract

The iterative solution of large systems of equations may benefit from parallel processing. However, using a straight-forward domain decomposition in "layered" geomechanical finite element models with significantly different stiffnesses may lead to slow or non-converging solutions. Physics-based domain decomposition is the answer to such problems, as explained in this paper and demonstrated on the basis of a few examples. Together with a two-level preconditioner comprising an additive Schwarz preconditioner that operates on the sub-domain level, an algebraic coarse grid preconditioner that operates on the global level, and additional load balancing measures, the described solver provides an efficient and robust solution of large systems of equations. Although the solver has been developed primarily for geomechanical problems, the ideas are applicable to the solution of other physical problems involving large differences in material properties.

1 Introduction

Finite element models of 3-D geomechanical problems result in linear systems of equations of the form

$$\boldsymbol{K}\boldsymbol{a} = \boldsymbol{f} \tag{1}$$

in which $\boldsymbol{K}$ is a sparse matrix that is often symmetric positive definite. In realistic models the system of equations can involve more than one million degrees of freedom so that using a direct solver is not an option. An iterative solver, such as the Conjugate Gradient method, is the only feasible option, provided that it is combined with a good preconditioner [10]. The solver will not converge without such a preconditioner because the large variation

*Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft Institute of Applied Mathematics, P.O. Box 5031, 2600 GA Delft, The Netherlands, phone: +31 15 278 55 30, fax: +31 15 278 72 09 (c.vuik@tudelft.nl)

in material parameters typically results in a very ill-conditioned system. For instance, the stiffness of a loose soil layer can be four to five orders of magnitude smaller than the stiffness of a concrete structure within the same model. This results in a comparable difference in the magnitude of the (diagonal) coefficients of the matrix $\boldsymbol{K}$.

An effective preconditioner is based on an incomplete Cholesky decomposition of the matrix $\boldsymbol{K}$ combined with an adaptive drop tolerance [12, 9]. Indeed, this preconditioner typically limits the number of solver iterations to one hundred, and often much less. It has been used successfully in the commercial computer program PLAXIS [1] for solving a wide range of 3-D geomechanical problems. Unfortunately, this preconditioner can not be implemented in a way that efficiently exploits parallel computers, including multi-core processors [16, 3]. This is a significant drawback as the sequential processing speed more or less stagnates while the number of processing cores increases. Another drawback of the preconditioner is that it operates only at the matrix level; it does not directly use any information from the underlying physical model.

This paper presents an iterative solver that can be implemented efficiently on multi-core processors and parallel computers, and that exploits the physical properties of the problem to be solved. Although it has been developed primarily for solving problems in geomechanics – as a replacement of the solver in PLAXIS – it can also be used to solve other types of problems in solid mechanics involving large jumps in material properties. The solver makes use of domain decomposition to obtain parallelism and to separate regions with large differences in material properties. It features a two-level preconditioner comprising an additive Schwarz preconditioner that operates on the sub-domain level, and an algebraic coarse grid preconditioner that operates on the global level [15, 17, 14]. The former is based on the incomplete Cholesky decomposition – similar to the one used in the original PLAXIS solver – of the sub-domain matrices. The latter uses the rigid body modes of the sub-domains to construct the algebraic coarse grid. Its task is to ensure that the convergence rate of the solver does not deteriorate as the number of sub-domains increases.

Previous research [6] also involved a second-level preconditioner for mechanical problems. There are some important differences, however, between that research and the current research:

- In the previous research the sub-domains are formed before the sub-domain matrices are assembled. In the current research the domain is decomposed after the global matrix has been assembled.

- The target machines in the previous research are clusters of PCs, whereas the current research targets multi-core shared-memory machines.

- The primary preconditioner used in the previous research is based on the additive Schwarz method. The current research, on the other hand, uses the restricted additive Schwarz method.

- The implementation of the solver developed in the previous research is aimed at a

specific class of problems, whereas the implementation in this research is more of a black-box nature.

The current implementation of the solver targets multi-core shared-memory computers because this architecture represents almost all modern (desktop) computers. There is no other reason why this architecture has been selected; the algorithms on which the solver is based can be implemented equally well on parallel computers with a distributed memory architecture.

The performance of the solver has been examined on a standard eight-core workstation. The original solver in Plaxis has been used to obtain a reference point with which the performance of the new solver has been compared. The number of cores has been varied from one to eight to determine the parallel scalability of the solver.

The remainder of this paper is structured as follows. Section 2 starts with a description of the geomechanical finite element models that are targeted by the solver. This section introduces the terminology that is used throughout the subsequent sections. Section 3 continues with a (mathematical) description of the solver. After that, Section 4 explains how the solver has been implemented on top of a small thread-based message passing library. Next, Section 5 discusses the performance of the solver for various geomechanical problems. Finally, Section 6 presents the conclusions.

2 Description of the finite element models

The geomechanical problems of interest typically involve a (large) volume of soil (or rock) and various structural objects; see Figure 1. The soil is often composed of multiple (semi-) horizontal layers with different mechanical properties. The structural objects are embedded in the soil or are located on top of the soil. They may include steel or concrete walls, foundations, piles, anchors and tunnels.

Finite element models are used to compute the deformation of the soil and the structural objects. Such a model is composed of volume elements that model the soil and thick structural objects; shell elements that model plates and thin-walled structural objects; line elements that model piles and anchors; and interface elements that model the interaction between the soil and structural objects. The displacement field $\boldsymbol{u}$ in an element e is approximated by

$$\boldsymbol{u}(\boldsymbol{x}) \approx \boldsymbol{N}_e(\boldsymbol{x})\boldsymbol{a}_e \tag{2}$$

in which $\boldsymbol{N}_e$ is a matrix containing the element shape functions and $\boldsymbol{a}$ is a vector containing the *degrees of freedom* in the nodes of the element. The latter are the displacements along the three coordinate axes, and the rotations about those axes if the element is a shell element. Note that adjacent elements share the degrees of freedom in their common nodes.

Substitution of the approximate displacement field into the constitutive equations yields a non-linear system of equations that is solved with a (quasi) Newton method. Each

Figure 1: Example of a geomechanical problem.

Newton iteration requires the solution of a linear system of equations of the form

$$\boldsymbol{K}\boldsymbol{a} = \boldsymbol{f} \tag{3}$$

in which $\boldsymbol{K}$ is the tangent stiffness matrix (or an approximation thereof), $\boldsymbol{a}$ is a vector containing the incremental degrees of freedom in all the nodes, and $\boldsymbol{f}$ is a vector containing the unbalanced forces and moments in the nodes. The solution of this linear system can be approximated with moderate accuracy because it is only an intermediate of the non-linear system to be solved. This is an advantage when using an iterative solver.

3 Description of the solver

The solver is made up of the following building blocks:

- a physics-based domain decomposition method;
- a standard iterative solver;
- a two-level preconditioner.

Although these building blocks are not new, they have been adapted in various ways to obtain an efficient solution algorithm for geomechanical problems.

3.1 Physics-based domain decomposition

The domain decomposition method can be viewed as a variation of the age-old divide-and-conquer strategy. The idea is to divide the problem to be solved into multiple sub-problems which are more or less solved independently. As the sub-problems are not completely independent, information needs to be exchanged between the sub-problems to obtain the solution of the global problem.

In the context of this paper the domain decomposition method is applied to the degrees of freedom. The degrees of freedom are first partitioned into n_s non-overlapping groups, each one making up one sub-domain. The sub-domains are then extended by adding degrees of freedom from neighboring sub-domains to obtain n_s slightly overlapping sub-domains. The original degrees of freedom in a sub-domain are called the *internal* degrees of freedom, and the additional degrees of freedom are called the *overlap* degrees of freedom. Note that each degree of freedom is an internal degree of freedom in exactly one sub-domain. Also note that an overlap degree of freedom in one sub-domain is also an internal degree of freedom in exactly one other sub-domain.

The sub-domains are assigned to different threads of execution that are responsible for executing all computations related to the degrees of freedom in their sub-domain. The threads are also responsible for storing all data related to the degrees of freedom in their sub-domain. No data are shared between the threads, except for the data structures that are needed to exchange data between the threads.

By defining *restriction operators* for each sub-domain, the global stiffness matrix $\boldsymbol{K}$ can be expressed in terms of slightly overlapping sub-domain matrices $\boldsymbol{K}_i$ as follows:

$$\boldsymbol{K} = \sum_{i=1}^{n_s} \boldsymbol{L}_i^T \boldsymbol{K}_i \boldsymbol{R}_i$$

where $\boldsymbol{L}_i$ and $\boldsymbol{R}_i$ are the left and right restriction operators, respectively, associated with the i-th sub-domain. Both operators are non-square boolean matrices containing ones and zeroes; each row contains at most one non-zero entry. Note that the domain decomposition method typically involves only the right restriction operators. However, the sub-domain matrices in this context are not obtained by assembling the element matrices but by extracting them from the assembled stiffness matrix.

The left restriction operators determine how the rows in the sub-domain matrices are mapped to the rows in the global matrix, while the right restriction operators determine how the columns in the sub-domain matrices are mapped to the columns in the global matrix.

The left restriction operators have the property that:

$$\boldsymbol{L}_i \boldsymbol{L}_j^T = \boldsymbol{0} \quad \text{if } i \neq j$$

That is, each left restriction operator $\boldsymbol{L}_i$ has a unique set of non-zero columns corresponding with the internal degrees of freedom in the i-th sub-domain. The right restriction operator is a superset of the left restriction operator; the matrix $(\boldsymbol{R}_i - \boldsymbol{L}_i)$ is a boolean matrix in which the non-zero columns correspond with the overlap degrees of freedom in the i-th sub-domain. The right restriction operator can be used to extract a sub-domain matrix from the global matrix as follows:

$$\boldsymbol{K}_i = \boldsymbol{R}_i \boldsymbol{K} \boldsymbol{R}_i^T$$

The right restriction operator can also be used to extract a sub-domain vector from a global vector as follows:

$$\boldsymbol{x}_i = \boldsymbol{R}_i \boldsymbol{x}$$

in which $\boldsymbol{x}_i$ is a sub-domain vector containing the elements from the global vector $\boldsymbol{x}$ that are associated with the i-th sub-domain. Note that the sub-domain vectors can be assembled into a global vector using the left restriction operators:

$$\boldsymbol{x} = \sum_{i=1}^{n_s} \boldsymbol{L}_i^T \boldsymbol{x}_i$$

The sub-domains are created by a partitioning algorithm – *partitioner* in short – that has three objectives: minimize the number of overlap degrees of freedom; minimize the variation in sub-domain sizes; and separate degrees of freedom associated with different material properties. The first objective is based on the relation between the number of

overlap degrees of freedom and the amount of data to be exchanged between the sub-domains. A smaller number of overlap degrees of freedom results in a smaller amount of data to be exchanged, which results in a lower communication and synchronization overhead, which results in a better parallel performance. The second objective is based on the relation between the size of a sub-domain (the number of degrees of freedom in that sub-domain) and the amount of work associated with that sub-domain. A smaller variation in sub-domain sizes results in a better distribution of the total work load over the sub-domains, which results in a higher parallel efficiency. The third objective is based on the observation that the preconditioner becomes more effective when the large jumps in material properties coincide with the sub-domain boundaries. That is, when degrees of freedom associated with different materials in the underlying finite element model are assigned to different sub-domains.

In general, all three objectives can not be achieved at the same time. The last objective, in particular, can restrict the way in which the sub-domains are created. The partitioner therefore applies a number of heuristics that have been tuned by solving a range of problems, involving different geometries and material compositions, with a varying number of sub-domains.

The partitioner does not actually operate on the level of degrees of freedom but on the level of nodes. That is, it first divides the nodes into overlapping groups and then collects all degrees of freedom that are associated with the nodes in those groups to obtain the sub-domains. This has the advantage that all degrees of freedom associated with the same node are assigned to the same sub-domain. Another advantage is that the number of nodes is less than the number of degrees of freedom so that the partitioner requires less processing time and memory. As there is a one-to-one mapping between the groups of nodes and the sub-domains, the latter term will also be used for the groups of nodes.

The partitioner creates the sub-domain in four stages: form the initial sub-domains by identifying groups of connected nodes associated with the same material type; merge small sub-domains into larger ones; split large sub-domains into smaller ones by applying a conventional graph partitioning algorithm; and extend each sub-domain with one or more levels of adjacent nodes to obtain the final sub-domains. The number of sub-domains, denoted by n_s, varies during the execution of the first three stages. After the third stage it is equal to the desired number of sub-domains which equals the number of threads n_t. The number of sub-domains remains equal to n_t during the execution of the fourth stage.

The structure of the global stiffness matrix determines which nodes are adjacent and which nodes are connected to each other. To be precise, let $\mathcal{D}_i$ denote the set of degrees of freedom associated with node i. Two nodes i and j are said to be *adjacent* if there is at least one non-zero element (k, l) or (l, k) in the stiffness matrix with $k \in \mathcal{D}_i$ and $l \in \mathcal{D}_j$. Two nodes i and j are said to be connected if there exists a path from i to j in the node *adjacency graph* that is obtained by defining edges between adjacent nodes. The adjacency graph can be written as:

$$\mathcal{G} = \{\mathcal{A}_1, \ldots, \mathcal{A}_{n_n}\} \tag{4}$$

in which a set $\mathcal{A}_i$ contains the nodes adjacent to node i.

In the first stage, the partitioner creates the initial sub-domains by executing Algorithm 1. The input of this algorithm consists of the node adjacency graph and the material type vector $\boldsymbol{m}$ that specifies the material type number (an integer) for each node. Its output is the partition vector $\boldsymbol{p}$ that specifies the sub-domain number for each node. The algorithm first initializes the free list $\mathcal{F}$ and the current sub-domain number s (lines 1 – 2). Next, it picks an arbitrary node from the free list, removes the node from the list and maps the node to the current sub-domain (lines 4 – 7). After that it executes a level-set traversal procedure to find all connected nodes that have been assigned the same material number (lines 8 – 20). Each node is removed from the free list and mapped to the current sub-domain (lines 14 – 15). Finally, it increments the sub-domain number (line 21) and continues with another node in the free list (line 4). The algorithm terminates when the free list is empty and all nodes have been mapped to a sub-domain.

Algorithm 1 Create the initial sub-domains. This algorithm takes the node adjacency graph and the material type vector $\boldsymbol{m}$ and returns the initial partition vector $\boldsymbol{p}$. Note that $\mathcal{A}_i$ denotes the set of nodes adjacent to node i.

```
 1: F = {1, ..., n_n}
 2: s = 1
 3: while F ≠ ∅ do
 4:    Pick any i ∈ F
 5:    L_1 = {i}
 6:    F = F\{i}
 7:    p[i] = s
 8:    while L_1 ≠ ∅ do
 9:       L_2 = ∅
10:       for all j ∈ L_1 do
11:          for all k ∈ A_j do
12:             if k ∈ F and m[k] = m[i] then
13:                L_2 = L_2 ∪ {k}
14:                F = F\{k}
15:                p[k] = s
16:             end if
17:          end for
18:       end for
19:       L_1 = L_2
20:    end while
21:    s = s + 1
22: end while
```

In the second stage, the partitioner merges sub-domains by executing Algorithm 2. The

input of this algorithm consists of the material type vector $\boldsymbol{m}$, the partition vector $\boldsymbol{p}$ and the number of desired sub-domains n_t. Its output is an updated partition vector. The algorithm starts with the initialization of the following variables (lines 1 – 10):

n_s the original number of sub-domains.

n the current number of sub-domains.

$\boldsymbol{r}$ the renumber vector. This is an integer vector that indicates how the sub-domains are renumbered.

$\boldsymbol{s}$ the active sub-domain vector. This is an integer vector containing the numbers of the remaining sub-domains. Only the first n entries of this vector are used.

$\boldsymbol{t}$ the sub-domain material vector. This integer vector stores the material type numbers associated with the sub-domains. Its length is equal to n_s.

$\boldsymbol{w}$ the sub-domain weight vector. This integer vector stores the number of nodes in each sub-domain. Its length is equal to n_s.

δw the sub-domain weight increment. This integer controls the execution speed of the algorithm and the quality of the resulting sub-domains. That is, a small value results in a longer execution time and – typically – in more equally-sized sub-domains. It is set to the ideal sub-domain weight (n_n/n_s) divided by 16, a value that seems to strike a good balance between the execution time and the quality of the results.

w_{lo} the minimum sub-domain weight. This integer determines which sub-domains are to be merged in each iteration of the algorithm. Its value is raised with δw at the end of each iteration.

w_{hi} the maximum sub-domain weight. This integer is set to the ideal sub-domain weight times $\frac{3}{2}$.

The core of the algorithm is formed by a while-loop (lines 11 – 33) that repeatedly merges small sub-domains. It first tries to merge small sub-domains with the same material type (line 16). If that is not possible, it merges all sub-domains with a weight smaller than w_{lo} (lines 17 – 19). After that, it raises the minimum weight with the weight increment δw (line 20) and continues with the next iteration, unless the convergence condition has been met. This is the case if the number of sub-domains is less than or equal to the desired number of sub-domains; and the weight of the smallest sub-domains is larger than or equal to δw; and the combined weight of large sub-domains is small enough after they have been partitioned to obtain the desired number of sub-domains. The exact conditions are described in Algorithm 5. After the while-loop has been terminated, the algorithm renumbers the sub-domains and updates the partition vector (lines 22 – 33). Note that

Algorithm 2 Merge the initial sub-domains. This algorithm takes the material type vector $\boldsymbol{m}$, the partition vector $\boldsymbol{p}$ and the desired number of sub-domains n_t. It returns an updated partition vector. The convergence condition is evaluated by Algorithm 5.

$n_s = \max(\boldsymbol{p})$
$n = n_s$
$\boldsymbol{r} = [1, \ldots, n_s]^T$
$\boldsymbol{s} = \boldsymbol{r}$
$\boldsymbol{t} = \boldsymbol{0}$
$\boldsymbol{w} = \boldsymbol{0}$
for $i = 0$ **to** n_n **do**
  $t[p[i]] = m[i]$
  $w[p[i]] = w[p[i]] + 1$
end for
$\delta w = n_n/(16n_s)$
$w_{\text{lo}} = \delta w$
$w_{\text{hi}} = 3n_n/(2n_s)$
while not converged **do**
  $n_0 = n$
  Merge small sub-domains with the same material type; see Algorithm 3.
  if $n = n_0$ **then**
    Merge small sub-domains; see Algorithm 4.
  end if
  $w_{\text{lo}} = w_{\text{lo}} + \delta w$
end while
$n = 0$
for $i = 1$ **to** n_s **do**
  if $r[i] = i$ **then**
    $n = n + 1$
    $r[i] = n$
  else
    $r[i] = r[r[i]]$
  end if
end for
for $i = 1$ **to** n_n **do**
  $p[i] = r[p[i]]$
end for

Algorithm 3 Merge small sub-domains with the same material type. This algorithm is part of Algorithm 2.

1: $\mathcal{F} = \{1, \ldots, n\}$
2: **repeat**
3: $j = 0$
4: $m = 0$
5: **for** $i = 1$ **to** n **do**
6: $k = s[i]$
7: $m = m + 1$
8: $s[m] = k$
9: **if** $k \in \mathcal{F}$ **and** $w[k] < w_{\text{lo}}$ **then**
10: **if** $j = 0$ **then**
11: $j = k$
12: $\mathcal{F} = \mathcal{F} \backslash j$
13: **else if** $t[k] = t[j]$ **and** $w[j] + w[k] \leq w_{\text{hi}}$ **then**
14: $r[k] = j$
15: $w[j] = w[j] + w[k]$
16: $m = m - 1$
17: **end if**
18: **end if**
19: **end for**
20: $n = m$
21: **until** $j = 0$

Algorithm 4 Merge all small sub-domains. This algorithm is part of Algorithm 2.

1: $j = 0$
2: $m = 0$
3: **for** $i = 1$ **to** n **do**
4: $k = s[i]$
5: $m = m + 1$
6: $s[m] = k$
7: **if** $w[k] < w_{\text{lo}}$ **then**
8: **if** $j = 0$ **then**
9: $j = k$
10: **else**
11: $r[k] = j$
12: $w[j] = w[j] + w[k]$
13: $m = m - 1$
14: **end if**
15: **end if**
16: **end for**
17: $n = m$

Algorithm 5 Determine when to stop merging sub-domains. This algorithm is part of Algorithm 2. It returns **true** if no more sub-domains need to be merged, and **false** otherwise. The symbol n_t denotes the desired number of sub-domains.

if $n > n_t$ **then**
  return false
end if
$w_{\min} = n_n$
$w_{\text{over}} = 0$
for $i = 1$ **to** n **do**
  $k = s[i]$
  if $w[k] > w_{\text{hi}}$ **then**
    $w_{\text{over}} = w_{\text{over}} + w[k]$
  else if $w[k] < w_{\min}$ **then**
    $w_{\min}$
  end if
end for
if $w_{\text{over}}/(1 + n_t - n) \leq w_{\text{hi}}$ **and** $w_{\min} \geq \delta w$ **then**
  return true
else
  return false
end if

the renumbering procedure only works because the relative order of the sub-domains is preserved.

The third stage is executed only if the second stage has produced less sub-domains than desired. In this stage the partitioner splits large sub-domains into smaller ones by executing Algorithm 6. The input of this algorithm consists of the node adjacency graph, the partition vector and the desired number of sub-domains. Its output is an updated partition vector. The algorithm involves the following variables:

n_s	the initial number of sub-domains.
n	the current number of sub-domains.
$\boldsymbol{k}_s$	the split vector. This integer vector stores the number of sub-domains into which each sub-domain is to be split.
$\mathcal{S}_i$	the set of nodes associated with the i-th sub-domain.
$\mathcal{G}_i$	the node adjacency graph associated with the i-th sub-domain.
$\boldsymbol{p}_i$	the partition vector associated with the i-th sub-domain. This vector indicates how that sub-domain is to be split.

After the first four of these variables have been initialized (lines 1 – 8), the algorithm determines a temporary partition vector $\boldsymbol{p}_i$ for each sub-domain. If the sub-domain does not need to be split, then the vector is simply filled with ones (line 11). Otherwise, the algorithm creates a sub-graph $\mathcal{G}_i$ from the node adjacency graph, and passes that graph to the METIS graph partitioning library [7] to calculate the temporary partition vector (lines 13 – 14). After that, the algorithms updates the global partition vector $\boldsymbol{p}$ and increments the number of new sub-domains n (lines 16 – 19).

To increase the quality of the split sub-domains, the edges in each sub-graph $\mathcal{G}_i$ are assigned weights in such a way that METIS will try to cut the graph along the boundaries between different materials. To be precise, an edge is assigned a weight of X if the two attached nodes have been assigned the same material number, and Y if not. Because METIS tries to minimize the total weight of the cut edges, it will prefer cutting edges between nodes that have been assigned different material numbers. The edge weights have been chosen on the basis of experience with different finite element models.

In the fourth stage, the partitioner creates the final, overlapping sub-domains by executing Algorithm 8. The input of this algorithm consists of the node adjacency graph, the partition vector and the amount of overlap n_v. Its output is a series of node lists that specify which nodes are associated with which sub-domain. This information can not be stored in a partition vector because a node may be associated with more than one sub-domain after this stage. The algorithm involves the following variables:

Algorithm 6 Split the merged sub-domains. This algorithm takes the node adjacency graph $\mathcal{G}$, the partition vector $\boldsymbol{p}$ and the desired number of sub-domains n_t. It returns an updated partition vector.

$n_s = \max(\boldsymbol{p})$
$n = 0$
Calculate the split vector $\boldsymbol{k}_s$; see Algorithm 7.
$\mathcal{S}_i = \emptyset \quad \forall \quad 1 \leq i \leq n_s$
for $i = 1$ **to** n_n **do**
  $j = p[i]$
  $\mathcal{S}_j = \mathcal{S}_j \cup i$
end for
for $i = 1$ **to** n_s **do**
  if $k_s[i] = 1$ **then**
    Set a dummy partition vector: $\boldsymbol{p}_i = \mathbf{1}$.
  else
    Create the sub-graph $\mathcal{G}_i \subset \mathcal{G}$ containing the nodes in $\mathcal{S}_i$.
    Partition $\mathcal{G}_i$ into $k_s[i]$ parts; store the partition vector in $\boldsymbol{p}_i$.
  end if
  for all $j \in \mathcal{S}_i$ **do**
    $p[j] = n + p_i[j]$
  end for
  $n = n + k_s[i]$
end for

Algorithm 7 Calculate the split vector $\boldsymbol{k}_s$. This algorithm is part of Algorithm 6. The symbol n_t denotes the desired number of sub-domains.

```
1: m = n_t − n_s
2: w = 0
3: for i = 1 to n_n do
4:    k_s[i] = 1
5:    w[p[i]] = w[p[i]] + 1
6: end for
7: while m > 0 do
8:    j = 0
9:    w_max = 0
10:   for i = 1 to n_s do
11:      if w[i]/k_s[i] > w_max then
12:         j = i
13:         w_max = w[i]/k_s[i]
14:      end if
15:   end for
16:   k_s[j] = k_s[j] + 1
17:   m = m − 1
18: end while
```

$\boldsymbol{a}$	an integer vector of length n_n (the number of nodes) that is used to make sure that each node set does not contain duplicate node numbers.
n_s	the current number of sub-domains.
$\mathcal{S}_i$	the node list containing the nodes associated with the i-th sub-domain.

After it has initialized these variables (lines 1 – 7), the algorithm extends each sub-domain with n_v levels of adjacent nodes by executing a level-set traversal procedure (lines 8 – 24). Note that the original nodes in the sub-domains are the internal nodes and that the added nodes are the overlap nodes.

Algorithm 8 Create the final, overlapping sub-domains. This algorithm takes the node adjacency graph, the partition vector and the amount of overlap n_v. It returns the node sets $\mathcal{S}_i$ containing the nodes associated with the sub-domains.

```
 1: n_s = max(p)
 2: a = 0
 3: S_i = ∅   ∀   1 ≤ i ≤ n_s
 4: for  i = 1 to n_n  do
 5:    j = p[i]
 6:    S_j = S_j ∪ i
 7: end for
 8: for  i = 1 to n_s  do
 9:    a[j] = i   ∀   j ∈ S_i
10:    L_1 = S_i
11:    for  l = 1 to n_v  do
12:       L_2 = ∅
13:       for all  j ∈ L_1  do
14:          for all  k ∈ A_j  do
15:             if  a[k] ≠ i  then
16:                a[k] = i
17:                S_i = S_i ∪ {k}
18:                L_2 = L_2 ∪ {k}
19:             end if
20:          end for
21:       end for
22:       L_1 = L_2
23:    end for
24: end for
```

The final sub-domains define maps from their local degrees of freedom to the corresponding global degrees of freedom. Such a map can be used to construct a right restriction operator as follows: for each row that is associated with a local degree of freedom, store the value one

in the column corresponding to its global degree of freedom. The left restriction operator is obtained by storing zeroes in the rows that correspond to the overlap degrees of freedom. Note that in the actual implementation of the solver the left and right restriction operators are stored as integer vectors and not as boolean matrices.

3.2 The iterative solver

The parallel solver presented here uses a Krylov method to compute the solution of the linear system of equations. It implements both the Conjugate Gradient and the Generalized Minimum Residual (GMRES) method [11]. Although the latter requires more memory and more work per iteration than CG, it can deal with non-symmetric matrices and preconditioners. This will be shown to be a significant advantage.

A preconditioner is applied (either from the left or from the right) to increase the convergence rate of the Krylov method. The preconditioner is the most important part of the solver as it largely determines the overall performance. It is composed of an additive Schwarz preconditioner that operates on the sub-domain level and an algebraic coarse grid preconditioner that operates on the global level.

The solver executes four types of operations: a vector addition, a scalar vector product, a sparse matrix-vector product, and a preconditioner-vector product. The first operation is trivial to execute in parallel: each thread simply adds its own sub-domain vectors; no data exchange is needed between the threads. The other two types of operations are relatively straightforward to execute and are described below. The last type of operation is explained in the next two sub-sections.

A scalar vector product is computed in parallel as follows:

$$s = \boldsymbol{x}^T \boldsymbol{y} = \sum_{i=1}^{n_s} \boldsymbol{x}_i^T \boldsymbol{L}_i \boldsymbol{L}_i^T \boldsymbol{y}_i$$

in which the matrix $\boldsymbol{L}_i \boldsymbol{L}_i^T$ is a boolean diagonal matrix containing non-zero values for the unique set of degrees of freedom in the i-th sub-domain. The above equation is equivalent with a series of local scalar vector products followed by a global sum that requires data exchanges between threads. The time required to execute these data exchanges is at least proportional to the logarithm of the number of threads, which means that a scalar vector product lowers the parallel efficiency of the solver. The solver therefore tries to compute multiple scalar products at the same time with the aim of lowering the communication latency that typically makes up a large fraction of the communication overhead.

A matrix-vector product can be written as:

$$\boldsymbol{y} = \boldsymbol{K}\boldsymbol{x} = \sum_{i=1}^{n_s} \boldsymbol{L}_i^T \boldsymbol{K}_i \boldsymbol{R}_i \boldsymbol{x} = \sum_{i=1}^{n_s} \boldsymbol{L}_i^T \boldsymbol{K}_i \boldsymbol{x}_i$$

The resulting vector associated with the i-th sub-domain is therefore given by:

$$\boldsymbol{y}_i = \boldsymbol{R}_i \boldsymbol{y} = \sum_{j=1}^{n_s} \boldsymbol{R}_i \boldsymbol{L}_j^T \boldsymbol{K}_j \boldsymbol{x}_j$$

As the matrix product $\boldsymbol{R}_i \boldsymbol{L}_j^T$ is non-zero only if the sub-domains i and j share common degrees of freedom, the matrix-vector product can be computed in two steps. First, each thread independently computes its local matrix-vector product $\boldsymbol{K}_i \boldsymbol{x}_i$. After that, it adds the contributions from the sub-domains that share degrees of freedom with its own sub-domain. This requires a data exchange between threads owning adjacent sub-domains. As these data exchanges can be executed in parallel, the communication overhead does not increase with the number of threads, provided that the available communication bandwidth scales with the number of processing cores.

3.3 The two-level preconditioner

The classic additive Schwarz (AS) preconditioner is constructed as follows [15]:

$$\boldsymbol{P}^{-1} = \sum_{i=1}^{n} \boldsymbol{R}_i^T \boldsymbol{P}_i^{-1} \boldsymbol{R}_i$$

in which $\boldsymbol{P}_i^{-1}$ is a sub-domain preconditioner that is obtained by means of an incomplete Cholesky decomposition in a similar way as the existing serial solver computes the preconditioner. Because the additive Schwarz preconditioner has the same structure as the global stiffness matrix $\boldsymbol{K}$, a product of the form

$$\boldsymbol{y} = \boldsymbol{P}^{-1} \boldsymbol{x}$$

can be computed in the same way as a matrix-vector product. In fact, the same data exchange procedures can be used.

The AS preconditioner can be formulated in an alternative way by using both the left and right location operators:

$$\boldsymbol{P}^{-1} = \sum_{i=1}^{n} \boldsymbol{L}_i^T \boldsymbol{P}_i^{-1} \boldsymbol{R}_i$$

This formulation, also known as the restricted additive Schwarz (RAS) preconditioner, is actually (much) more effective than the classic AS preconditioner [2]. It generally results in a higher convergence rate of the solver and it requires less data to be exchanged between the threads. A drawback, however, is that the RAS preconditioner is not symmetric, even when the sub-domain preconditioners are symmetric. This means that the RAS preconditioner must be combined with a non-symmetric Krylov method like GMRES which is more expensive than CG.

The coarse preconditioner $\boldsymbol{P}_0^{-1}$ is obtained by projecting the stiffness matrix onto a vector space that forms the algebraic coarse grid. If this vector space is spanned by the columns of the matrix $\boldsymbol{V}$, then

$$\boldsymbol{P}_0^{-1} = \boldsymbol{V}\boldsymbol{A}^{-1}\boldsymbol{V}^T = \boldsymbol{V}\left(\boldsymbol{V}^T\boldsymbol{K}\boldsymbol{V}\right)^{-1}\boldsymbol{V}^T$$

with $\boldsymbol{A}$ the projected stiffness matrix, also called the coarse matrix. The effectiveness of the coarse preconditioner is determined by the construction of the matrix $\boldsymbol{V}$. A good choice has proven to be:

$$\boldsymbol{V} = \begin{bmatrix}\boldsymbol{R}_1^T\boldsymbol{N}_1 & \cdots & \boldsymbol{R}_n^T\boldsymbol{N}_n\end{bmatrix}$$

That is, the columns of $\boldsymbol{V}$ are formed by the columns of the sub-domain matrices $\boldsymbol{N}_i$, of which the exact definition will be given later. By introducing the coarse restriction operators $\boldsymbol{C}_i$, the matrix $\boldsymbol{V}$ can also be written as:

$$\boldsymbol{V} = \sum_{i=1}^{n} \boldsymbol{R}_i^T\boldsymbol{N}_i\boldsymbol{C}_i$$

Each coarse restriction operator has a unique set of non-zero columns so that

$$\boldsymbol{C}_i\boldsymbol{C}_j^T = \begin{cases}\boldsymbol{0} & \text{if } i \neq j,\\ \boldsymbol{I} & \text{if } i = j\end{cases}$$

The coarse restriction operators can be used to extract a coarse sub-domain vector from a global coarse vector:

$$\boldsymbol{x}_i = \boldsymbol{C}_i\boldsymbol{x}$$

Note that the length of the coarse sub-domain vector $\boldsymbol{x}_i$ equals the number of columns in the matrix $\boldsymbol{N}_i$, and that the length of the global coarse vector $\boldsymbol{x}$ equals the number of columns in the matrix $\boldsymbol{V}$.

The coarse matrix can be constructed column wise through a series of parallel matrix-vector products and data exchanges. If $\boldsymbol{A}^j$ denotes the j-th column of the coarse matrix, then:

$$\begin{aligned}\boldsymbol{A}^j &= \boldsymbol{A}\boldsymbol{e}^j = \boldsymbol{V}^T\boldsymbol{K}\boldsymbol{V}\boldsymbol{e}^j\\ &= \boldsymbol{V}^T\boldsymbol{K}\boldsymbol{a}^j = \boldsymbol{V}^T\boldsymbol{b}^j\\ &= \sum_{i=1}^{n}\boldsymbol{C}_i^T\boldsymbol{N}_i^T\boldsymbol{R}_i\boldsymbol{b}^j\end{aligned}$$

with

$$\boldsymbol{a}^j = \boldsymbol{V}\boldsymbol{e}^j = \boldsymbol{R}_i^T\boldsymbol{N}_i\boldsymbol{C}_i\boldsymbol{e}^j$$

$$\boldsymbol{b}^j = \boldsymbol{K}\boldsymbol{a}^j = \sum_{i=1}^{n}\boldsymbol{L}_i^T\boldsymbol{K}_i\boldsymbol{R}_i\boldsymbol{a}^j$$

and with $\boldsymbol{e}^j$ a unit vector of which all elements are zero, except for the j-th element. Thus:

$$\begin{aligned}\boldsymbol{e}^1 &= \begin{bmatrix}1 & 0 & 0 & \cdots & 0\end{bmatrix}^T \\ \boldsymbol{e}^2 &= \begin{bmatrix}0 & 1 & 0 & \cdots & 0\end{bmatrix}^T \\ &\vdots\end{aligned}$$

The columns of the coarse matrix are not stored as single entities; each thread i stores only the partial columns $\boldsymbol{A}_i^j$ that are obtained by applying the restriction operator $\boldsymbol{C}_i$ to the columns of the coarse matrix. These partial columns can be computed in parallel by executing the following algorithm.

1. Form the coarse sub-domain vector $\boldsymbol{e}_i^j = \boldsymbol{C}_i \boldsymbol{e}^j$. Note that this vector is non-zero for only one thread.
2. Compute $\tilde{\boldsymbol{a}}_i^j = \boldsymbol{N}_i \boldsymbol{e}_i^j$.
3. Exchange data to obtain $\boldsymbol{a}_i^j = \sum_{k=1}^n \boldsymbol{R}_i \boldsymbol{R}_k^T \tilde{\boldsymbol{a}}_k^j$.
4. Compute $\tilde{\boldsymbol{b}}_i^j = \boldsymbol{K}_i \boldsymbol{a}_i^j$.
5. Exchange data to obtain $\boldsymbol{b}_i^j = \sum_{k=1}^n \boldsymbol{R}_i \boldsymbol{L}_k^T \tilde{\boldsymbol{b}}_k^j$.
6. Compute $\boldsymbol{A}_i^j = \boldsymbol{C}_i \boldsymbol{A}^j = \boldsymbol{N}_i^T \boldsymbol{b}_i^j$.

Although this algorithm is relatively straightforward to implement, it does not exploit the special structure of the unit vectors $\boldsymbol{e}^j$. To be precise, it will perform a number of matrix-vector multiplications involving zero vectors. It is possible to implement a more efficient algorithm, but the gain in performance would only be significant for a large number of sub-domains.

The coarse preconditioner is formed by computing the QR-decomposition [5] of the coarse matrix:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{U}$$

in which $\boldsymbol{Q}$ is an orthonormal matrix and $\boldsymbol{U}$ is an upper triangular matrix. They can be obtained by applying a Gram-Schmidt orthonormalization procedure to the columns of $\boldsymbol{A}$. To execute this procedure in parallel, each thread computes the partial decomposition

$$\boldsymbol{A}_i = \boldsymbol{C}_i \boldsymbol{A} = \boldsymbol{C}_i \boldsymbol{Q} \boldsymbol{U} = \boldsymbol{Q}_i \boldsymbol{U}$$

This means that each thread stores the sub-matrix $\boldsymbol{Q}_i$ and a full copy of the matrix $\boldsymbol{U}$. A drawback of this procedure is that a backward substitution of the form

$$\boldsymbol{y} = \boldsymbol{U}^{-1} \boldsymbol{x}$$

takes a constant time, regardless of the number of threads. A more scalable approach is to explicitly compute the inverse of the upper triangular matrix $\boldsymbol{U}$ so that a backward

substitution becomes a matrix-vector product. Such a product can be executed in parallel by dividing the inverse matrix $\boldsymbol{U}^{-1}$ row-wise between the threads in the same way as the matrix $\boldsymbol{Q}$. That is, each thread stores the sub-matrix

$$\left(\boldsymbol{U}^{-1}\right)_i = \boldsymbol{C}_i \boldsymbol{U}^{-1}$$

The inverse matrix $\boldsymbol{U}^{-1}$ can be computed by executing a parallel algorithm that is similar to the GCR algorithm [8, 4].

Application of the coarse preconditioner involves a series of parallel matrix-vector products, inner products and data exchanges. This follows from:

$$\begin{aligned}
\boldsymbol{y} &= \boldsymbol{P}_0^{-1}\boldsymbol{x} = \boldsymbol{V}\boldsymbol{A}^{-1}\boldsymbol{V}^T\boldsymbol{x} = \boldsymbol{V}\boldsymbol{U}^{-1}\boldsymbol{Q}^T\boldsymbol{V}^T\boldsymbol{x} \\
&= \boldsymbol{V}\boldsymbol{U}^{-1}\boldsymbol{Q}^T\boldsymbol{a} = \boldsymbol{V}\boldsymbol{U}^{-1}\boldsymbol{b} = \boldsymbol{V}\boldsymbol{c} \\
&= \sum_{i=1}^{n} \boldsymbol{R}_i^T \boldsymbol{N}_i \boldsymbol{C}_i \boldsymbol{c}
\end{aligned}$$

with

$$\begin{aligned}
\boldsymbol{a} &= \boldsymbol{V}^T\boldsymbol{x} = \sum_{i=1}^{n} \boldsymbol{C}_i^T \boldsymbol{N}_i^T \boldsymbol{R}_i \boldsymbol{x} \\
\boldsymbol{b} &= \boldsymbol{Q}^T\boldsymbol{a} = \sum_{i=1}^{n} \boldsymbol{Q}_i^T \boldsymbol{C}_i \boldsymbol{a} \\
\boldsymbol{c} &= \boldsymbol{U}^{-1}\boldsymbol{b} = \sum_{i=1}^{n} \boldsymbol{C}_i^T \left(\boldsymbol{U}^{-1}\right)_i \boldsymbol{b}
\end{aligned}$$

To evaluate these equations all threads execute the following procedure in parallel.

1. Compute $\boldsymbol{a}_i = \boldsymbol{N}_i^T \boldsymbol{x}_i$ with $\boldsymbol{x}_i = \boldsymbol{R}_i \boldsymbol{x}$.
2. Compute $\boldsymbol{b}_i = \boldsymbol{Q}_i^T \boldsymbol{a}_i$.
3. Compute $\boldsymbol{b} = \sum_{i=1}^{n} \boldsymbol{b}_i$. This requires a global sum over all threads. Each thread stores a copy of $\boldsymbol{b}$.
4. Compute $\boldsymbol{c}_i = \left(\boldsymbol{U}^{-1}\right)_i \boldsymbol{b}$.
5. Compute $\tilde{\boldsymbol{y}}_i = \boldsymbol{N}_i \boldsymbol{c}_i$.
6. Exchange data to obtain $\boldsymbol{y}_i = \sum_{k=1}^{n} \boldsymbol{R}_i \boldsymbol{R}_k^T \tilde{\boldsymbol{y}}_k$.

The coarse preconditioner is most effective when it removes problematic eigenvalues from the stiffness matrix that can not be removed by the RAS preconditioner. In geomechanical applications these eigenvalues are typically related to the rigid body modes of the

sub-domains. That is, each matrix $\boldsymbol{N}_i$ contains six columns that represent three rigid translations and three rigid (linearized) rotations of the i-th sub-domain. The six column vectors can be viewed as a series of sextets, one for each node in the sub-domain, representing the three displacements along the global coordinate axes and the rotations about those axes. The triplets associated with the rigid body modes are given by:

$$\begin{array}{lcl} \text{translation along the x-axis} & : & [1,0,0,0,0,0] \\ \text{translation along the y-axis} & : & [0,1,0,0,0,0] \\ \text{translation along the z-axis} & : & [0,0,1,0,0,0] \\ \text{rotation about the x-axis} & : & [0,-z,y,1,0,0] \\ \text{rotation about the y-axis} & : & [z,0,-x,0,1,0] \\ \text{rotation about the z-axis} & : & [-y,x,0,0,0,1] \end{array}$$

where x, y and z are the coordinates of the node with which the sextet is associated. Note that the rotational degrees of freedom are only present when a shell element is attached to a node. Otherwise, only the translational degrees of freedom will be present.

The rigid body modes do not need to be computed for each sub-domain on an individual basis. An alternative approach is to setup a global matrix $\boldsymbol{N}$ of which the six columns represent the six rigid body modes of the complete finite element model without constraints. The rigid body modes for each sub-domain can then simply be obtained by applying the right restriction operator:

$$\boldsymbol{N}_i = \boldsymbol{R}_i \boldsymbol{N}$$

This approach simplifies the programming interface of the solver as the user only has to provide one matrix containing the global rigid body modes.

In the case that a sub-domain contains degrees of freedom that are associated with different materials, the effectiveness of the coarse preconditioner can be increased by using more than six rigid body modes. To be precise, a set of rigid body modes is used for each group of connected degrees of freedom that are associated with the same material. Although this results in a larger coarse grid, the extra computational effort can be more than offset by the increased effectiveness of the preconditioner. The current implementation of the solver created at most four groups per sub-domain. If the number of groups is larger, than adjacent groups are merged until four groups remain.

The coarse preconditioner can be combined with the AS or RAS preconditioner in an additive or a multiplicative way. In the former approach the coarse preconditioner is simply added to the AS or RAS preconditioner. In the latter approach the two preconditioners are applied as follows:

$$\begin{aligned} \tilde{\boldsymbol{y}} &= \boldsymbol{P}^{-1} \boldsymbol{x} \\ \tilde{\boldsymbol{r}} &= \boldsymbol{x} - \boldsymbol{K} \tilde{\boldsymbol{y}} \\ \boldsymbol{y} &= \boldsymbol{P}_0^{-1} \tilde{\boldsymbol{r}} \end{aligned}$$

Table 1: The essential functions provided by the solver library.

`new_context`	Creates a new solver context with an associated group of threads.
`new_matrix`	Defines a sparse matrix associated with a solver context.
`update_matrix`	Updates the values stored in a sparse matrix.
`new_precon`	Creates a preconditioner associated with a matrix.
`update_precon`	Updates a preconditioner.
`new_solver`	Creates a solver associated with a matrix and a preconditioner.
`run_solver`	Executes a solver and solves a system of equations.
`set_option`	Sets an integer property of an object.
`set_param`	Sets a floating point property of an object.
`get_info`	Retrieves information about an object.
`set_feedback`	Sets a feedback function for monitoring events.

This scheme has proven to be much more efficient, even though it requires one extra matrix-vector product per solver iteration. Note that the combined preconditioner is not symmetric, even when using a symmetric AS preconditioner. A symmetric preconditioner can be obtained by applying the AS preconditioner a second time in a multiplicative way, but this raises the computational cost of the total preconditioner considerably.

4 Implementation of the solver

The solver has been implemented in C as a stand-alone, portable library with programming interfaces for Fortran 77 and Fortran 90. The library is more or less a black box that can be linked with a serial program. It provides a small set of functions for defining a sparse matrix; for creating a preconditioner associated with that matrix; for solving a system of equations; for setting runtime parameters and options; and for retrieving information about the state and progress of the solver. Table 1 lists the essential functions making up the programming interface of the library.

4.1 The message passing programming model

The solver library is based on a multi-threaded message passing programming model to execute the solution algorithms in parallel. When a program defines a sparse matrix, the solver library spawns a number of threads, creates the sub-domains, and assigns one sub-domain to each thread. A custom-made and light-weight message passing module is used to exchange data between adjacent sub-domains and to perform global reduction operations such as a global sum. Because the threads share a common memory space, messages sent by one thread to another can be copied directly from the source memory area to the destination memory area without intervention of the operating system, and without using

intermediate memory buffers. To synchronize the execution of the threads, the message passing module uses POSIX mutexes and condition variables, or Windows serial sections and event objects. It tries to reduce system-call overhead by spinning for a short while when one thread needs to wait for another. Note that spinning is turned off when the number of threads exceeds the number of processor cores.

A significant part of the implementation is dedicated to error handling. If a system error or a numerical error occurs, the library will try very hard to return to a well-defined state and report the error to the calling program. A complicated situation occurs when a thread needs to abort an operation because of an error that is local to that thread. If this happens, the library will signal all other threads so that they can abort their operation too. In this way a thread will not be waiting forever on an event that never occurs because another thread has aborted an operation.

By using threads and an integrated message passing module, the solver library can be incorporated easily into an existing serial program; there is no need to install external libraries and set up a special runtime environment. A drawback of a thread-based implementation is that the solver can not make use of distributed memory systems. However, it would not be much work to replace the integrated message passing module by an external message passing library that supports distributed memory systems (one based on MPI, for instance).

4.2 Some implementation details

Both the GMRES algorithm and the coarse grid preconditioner make use of the Gram-Schmidt orthogonalization procedure. The solver implements a combination of the classic and the modified Gram-Schmidt procedure to find a balance between numerical accuracy and low communication overhead. The hybrid algorithm applies the modified Gram-Schmidt procedure to bundles of four (or six in case of the coarse preconditioner) vectors at a time, as shown in Algorithm 9. Because the four scalar vector products in the first inner for-loop are independent, only one global reduction operation is required to sum the partial products computed by the threads.

Algorithm 9 Create a vector $\boldsymbol{u}$ that is orthonormal to a set of n vectors $\boldsymbol{v}_i$. This algorithm combines the classic and modified Gram-Schmidt procedures.

```
1: for i = 1 to n step 4 do
2:    for j = i to i + 3 do
3:       a_j = u^T v_j
4:    end for
5:    for j = i to i + 3 do
6:       u = u − a_j v_j
7:    end for
8: end for
```

To improve the performance of the GMRES algorithm, the solver bundles each con-

secutive set of four Krylov vectors into one dense matrix of which the elements are stored row wise in memory. This makes it possible to replace the four scalar vector products in the first inner loop of hybrid Gram-Schmidt procedure by a dense matrix-vector product. This, in turn, increases the utilization of the memory cache, and therefore reduces the execution time of the solver. Note that if the number of Krylov vectors is not a multiple of four, then up to three zero vectors are added to fill a bundle of four Krylov vectors.

The solver speeds up the construction of the coarse matrix by computing bundles of four columns at the same time. To see how this helps, consider the main operation in the computation of one coarse matrix column: a matrix-vector product involving the global stiffness matrix. This operation requires that all matrix elements are transported from the (slow) main memory to the processor. The cache memory is typically not effective because the amount of data stored in the stiffness matrix tends to be much larger than the size of the cache. The execution time of one matrix-vector product is therefore dictated primarily by the memory bandwidth and much less by the processor speed. The only way to overcome this bottleneck is to re-use each matrix element once it has been fetched from the main memory. This is exactly what happens when multiple, independent, matrix-vector products are executed in one pass.

The maximum number of Krylov vectors is an important parameter in the GMRES algorithm. If this parameter is too large, then the solver may run out of memory or the time per GMRES iteration may become excessively large. If this parameter is too small, on the other hand, then GMRES may be forced to make many restarts with a low convergence rate as a result. As the average user can probably not make an informed choice, the solver sets the maximum number of Krylov vectors so that the space needed for storing the Krylov vectors is at most equal to the space required for storing the stiffness matrix and the preconditioner. This automatic approach seems to strike a good balance between performance and memory usage.

4.3 Hardware considerations

To achieve good parallel performance, the solver requires sufficient memory bandwidth for all processor cores combined. In each solver iteration, all threads need to fetch their sub-domain matrix and preconditioner from the main memory as the cache memory is typically much too small to store all the data. Unfortunately it is not possible to reduce the memory traffic by using (parts of) a sub-domain matrix or preconditioner multiple times within one iteration.

The available memory bandwidth must be sufficient to provide each thread full-speed access to the main memory. Otherwise, one thread will have to wait for another while it is fetching data from memory. This used to be a real problem in shared memory machines as the processing cores had to share one memory bus. Fortunately, both AMD and Intel have adopted a memory architecture that can scale with the number of processor cores. That is, each CPU (containing multiple processing cores) has its own memory connection; adding more CPUs will increase the number of memory connections. Note that some computer vendors will try to reduce costs by routing multiple memory connections through one CPU.

This will reduce the available memory bandwidth and may reduce the parallel performance of the solver. One simple way to maximize the memory bandwidth is to fill all available memory slots so that all memory connections are used.

Here is a simple procedure to determine whether the memory bandwidth is sufficient. First run one single-threaded computation and measure the execution time. Then run multiple, independent instances of the same computation concurrently and measure the average execution time. A significant difference between the two execution times is a clear sign that the memory bandwidth is not sufficient.

5 Performance of the solver

The performance of the solver has been measured by solving various systems of equations on a workstation with two quad-core Intel Xeon E5620 processors, 24GB memory and running Windows 7. Both processors can access the main memory through independent interconnects so that the maximum possible memory bandwidth can be obtained. The systems of equations have been obtained from various geomechanical problems using the commercial finite element program Plaxis.

The next sub-section uses an academic problem to analyze the various components making up the solver. The goal here is to gain insight in the way that those components effect the overall performance of the solver. The second sub-section shows what kind of performance can be obtained for two representative geomechanical problems. The third and last sub-section shows the performance of the solver for a difficult problem. The aim of this sub-section is to highlight the current limitations of the solver.

5.1 Analysis of the performance

Figure 2 shows the geomechanical model that has been used to analyze the performance of the various solver components. The model comprises about 1,400 15-node wedge elements and 10,000 degrees of freedom, and consists of a rectangular block of soil that is composed of five layers with large variations in stiffness. A uniform load (indicated by the blue arrows) is applied to an embedded block of concrete. A simple linear elastic material model is used to describe the deformations of the soil and the concrete. The relevant material parameter in this model is the stiffness modulus that differs five orders of magnitude between the different parts of the model; see Table 2.

Figure 3 shows the performance results that have been obtained for the four solver runs listed in Table 3. The first run makes use of the original solver in Plaxis that is based on the Conjugate Gradient algorithm in combination with an incomplete Cholesky decomposition as preconditioner. The second run uses the new solver but without the physics-based domain decomposition method; it simply uses the Metis partitioning library to create the sub-domains without taking advantage of the underlying model. The third run uses

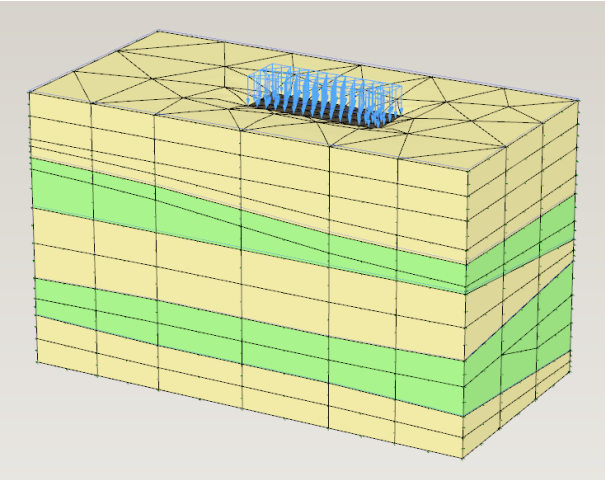


Figure 2: The geomechanical model that has been used to analyze the performance of the solver.

Table 2: The stiffness moduli associated with the different parts of the model.

Part	Stiffness modulus [MPa]
Yellow-colored soil	1.0
Green-colored soil	$1.0 \cdot 10^3$
Concrete block	$1.0 \cdot 10^5$

Table 3: Description of the four solver runs.

Run	Description
1	The original solver in PLAXIS.
2	The new solver, without the physics-based domain decomposition method and without the coarse grid preconditioner.
3	The new solver, with the physics-based domain decomposition method and without the coarse grid preconditioner.
4	The new solver, with the physics-based domain decomposition method and with the coarse grid preconditioner.

the new solver with the physics-based domain decomposition method, but without the algebraic coarse grid preconditioner. This preconditioner is added in the fourth run.

Note that the original solver uses only one thread; the results obtained with the original solver have been plotted as horizontal lines in the three graphs. Also note that the runs with the new solver are based on the GMRES algorithm in combination with the RAS preconditioner. Finally note that the maximum number of threads (sixteen) exceeds the number of processor cores (eight). The reason for this is that a larger number of threads – and therefore sub-domains – provides some insight into the numerical scalability of the solver.

One can draw the following four conclusions from Figure 3. First, the original solver is less efficient than the new solver when using a single thread or sub-domain. This is because the original solver implements a slightly different, and less efficient, incomplete Cholesky preconditioner than the new solver. Note, however, that this is not always the case; for some other problems the situation is reversed and the original solver is more efficient.

The second conclusion is that the physics-based domain decomposition method is absolutely essential to get decent performance with the new solver; without it, the number of solver iterations varies wildly with the number of sub-domains. Indeed, the maximum number of iterations is a about a factor five larger than the minimum number of iterations.

The third conclusion is that the coarse grid preconditioner raises the efficiency of the solver by a significant measure. It significantly lowers both the number of solver iterations and the solution time, without adding much to the time that is required to set up the preconditioner.

The final conclusion is that the new solver scales very well with the number of threads. Indeed, both the time required to set up the preconditioner and the time required to solve the system of equations are more than eight times smaller when using eight threads instead of one thread. This is partly caused by a numerical speedup: the sub-domain wise incomplete Cholesky preconditioner requires less retries to find the appropriate drop tolerance and the solver requires less iterations to satisfy the convergence criterion. The remainder of the speedup is the result of using more than one processor core.

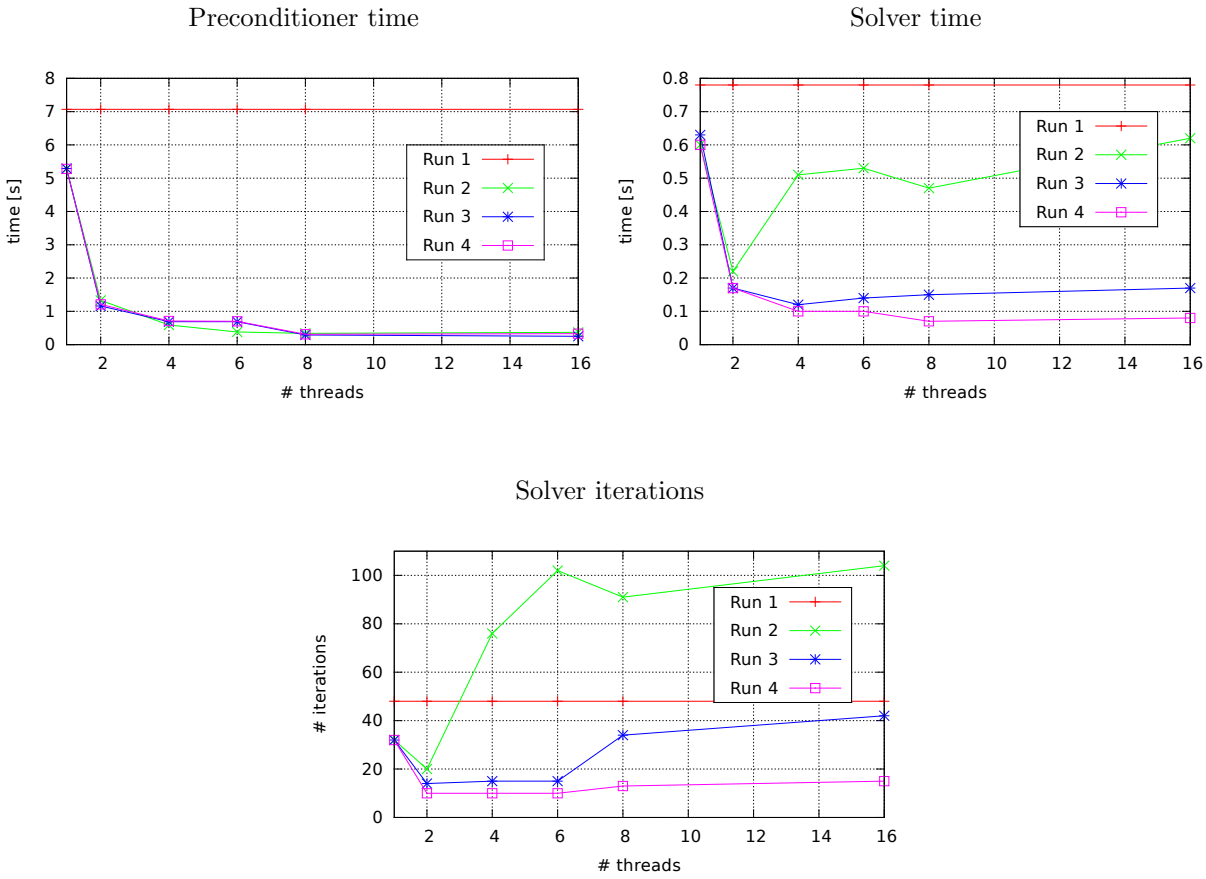


Figure 3: The performance results that have been obtained for the four solver runs listed in Table 3. The upper left graph shows the time required to create the preconditioner as function of the number of threads. The upper right graph shows the time required to solve one system of equations as function of the number of threads. The lower graph shows the number of solver iterations as function of the number of threads. Note that the number of threads equals the number of sub-domains.

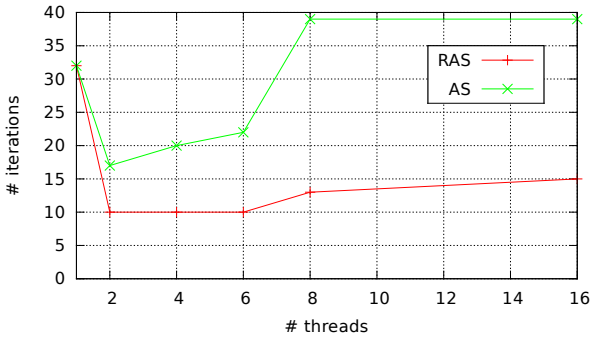
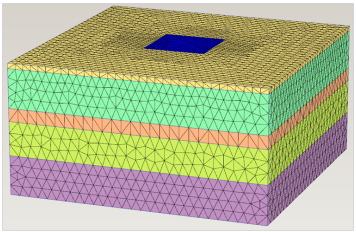


Figure 4: The number of solver iterations with the AS and RAS preconditioners versus the number of threads.

Model 1



Model 2

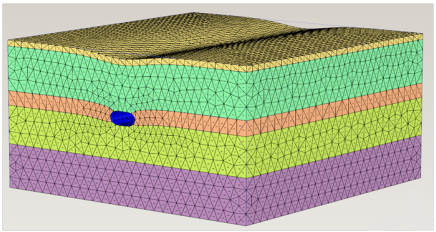


Figure 5: The two representative models that have been used to test the real-world performance of the solver.

Figure 4 shows the number of solver iterations when the RAS preconditioner is replaced by the AS preconditioner. It is clear that the RAS preconditioner is much more efficient in this case than the AS preconditioner. Indeed, the effectiveness of the RAS preconditioner easily justifies the use of the GMRES algorithm instead of the CG algorithm.

5.2 Results for two representative models

The new solver has been the standard iterative solver in Plaxis since the end of 2010 and it has been used to solve a wide range of geomechanical problems. Two representative models (see Figure 5) have been selected to show what kind of performance can be obtained in practice. Model 1 involves a concrete foundation (modelled by shell elements) on top of several soil layers with a varying stiffness. Model 2 is similar to Model 1 except that the foundation has been replaced by a tunnel of which the walls have been modelled by shell elements. Table 4 lists the relevant properties of the two models.

Table 5 shows the performance results that have been obtained for the two models. It also shows results obtained with the original solver and with Pardiso [13], a fast parallel direct solver that is available on a commercial basis. Note that the original solver uses a single

Table 4: The relevant properties of the two models shown in Figure 5. The second and third column list the minimum and maximum stiffness moduli, respectively.

Model	$E_{\min}$ [MPa]	$E_{\max}$ [MPa]	# DOFs
1	1.5	$3.0 \cdot 10^4$	680,000
2	1.5	$3.0 \cdot 10^4$	414,000

processing core and PARDISO uses all eight processing cores. Figure 6 shows the parallel speedup that has been obtained with the new solver. The speedup is defined as the ratio between the execution time with n threads over the execution time with one thread.

Four observations can be made when looking at these results. First, the performance of the new solver varies considerably between the two models, especially when looking at the number of solver iterations versus the number of threads. One reason is that the solver is not able to separate the different materials well enough when the number of threads (and sub-domains) is relatively small. Another reason is that the solver needs to make a trade off between a well-balanced work load and sub-domains that do not cut through material boundaries, resulting in a sub-optimal configuration of sub-domains. Because of this, the solver is allowed to use twice as many threads as processor cores, unless the number of threads has been explicitly set by the user.

The second observation is that the new solver is faster than the original one when using a single thread. The reason is that the new solver implements a somewhat different and more efficient incomplete Cholesky decomposition algorithm. This algorithm does not only lower the time to set up the preconditioner, but also to solve the system of equations, even when the new solver requires more iterations.

The third observation is that the construction of the preconditioner scales well with the number of threads. This is not surprising as the construction of the preconditioner requires relatively little communication between the threads; a large fraction of the computations are local to the threads. When solving the system of equations, on the other hand, the communication between the threads makes up a larger fraction of the execution time. In addition, there is less scope for re-using data that resides in the local caches of the processor cores so that the memory bandwidth becomes a more limiting factor.

The final observation is that the new solver is faster than PARDISO when 8 threads are used, both in preconditioning time and solve time. The performance gain in the given examples is of the order 3 to 5.

5.3 Seeking the limits of the solver

The new solver still has some limitations that can become apparent when solving difficult problems. One of those problems involves a non-linear model for simulating the construction of a tunnel at Toulon in France; see Figure 7. The model has about 750,000 degrees of freedomand consists of solid elements, shell elements and beam elements with a stiffness

Table 5: Performance results obtained for the two models. The first column specifies the solver for which the results have been obtained. The second column lists the number of threads or sub-domains. The remaining three columns list the time required to set up the preconditioner and to solve the system of equations, and the number of solver iterations. Note that in the case of PARDISO the third column specifies the time required to factor the matrix.

Model 1

Solver	# threads	Precon [s]	Solve [s]	# iter
PARDISO	8	200	150	1
Original	1	320	680	140
New	1	140	550	134
	2	82	180	80
	4	43	150	111
	8	23	100	113

Model 2

Solver	# threads	Precon [s]	Solve [s]	# iter
PARDISO	8	71	72	1
Original	1	170	140	32
New	1	58	90	62
	2	22	84	74
	4	19	41	39
	8	14	29	45

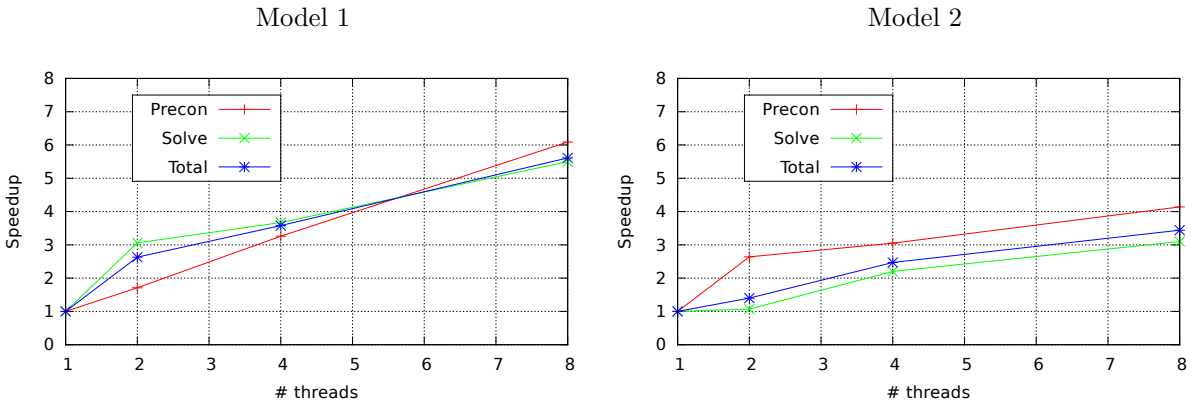


Figure 6: The parallel speedup obtained for the two models. The curve labelled "Precon" depicts the parallel speedup of the preconditioner. The curve labelled "Solve" depicts the parallel speedup of the solver, excluding the time required to set up the preconditioner. The curve labelled "Total" depicts the total speedup that is based on the time for setting up the preconditioner and for solving the system of equations.

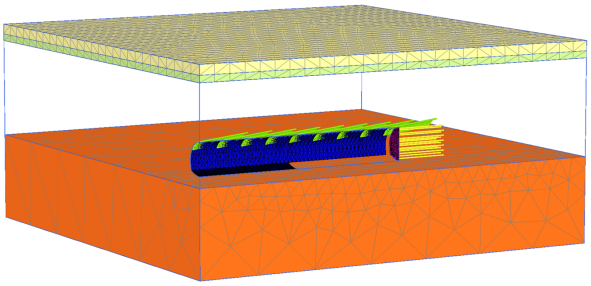


Figure 7: A model of the Toulon tunnel in France. Part of the soil is not shown so that the tunnel is visible.

modulus that ranges from 1.0 MPa to $2.1 \cdot 10^5$ MPa. A complete simulation of the construction process requires the solution of multiple linear systems of equations. Only the first system is considered here because the subsequent systems of equations are slightly different when using different solvers or different numbers of threads.

The problem is difficult to solve for at least three reasons. First there is the large variation in the stiffness modulus, but this is also the case for the two models in the previous sub-section. A second reason is that the simulation of the construction process involves cutting away part of the soil, resulting in very small elements. A third reason is that the beam elements only have very small rotational stiffness to prevent the system of equations from becoming singular. All these three factors lead to a stiffness matrix of which the diagonal elements differ fourteen orders of magnitude. This does not need to be a problem *per se*, but it does indicate that this is a non-standard problem.

Table 6 and Figure 8 show the performance results that have been obtained for the model of the Toulon tunnel. It is clear that the new solver does not perform very well because the number of solver iterations increases substantially when using more than one thread. The exact cause of this is not known at this moment. One possibility is that a node-based domain decomposition algorithm is not able to separate the different materials and element types into different sub-domains. In that case, switching to an element-based decomposition algorithm would help, but this would make it more difficult to use the solver as a black box.

Note that in this case PARDISO is much faster than both the new and the old solver.

6 Conclusions

The solver proposed in this paper can solve geomechanical problems efficiently on multi-core machines, even when they involve materials with very different properties, for instance a stiffness modulus that varies five orders of magnitude. As the solver is not tightly coupled to the geomechanical nature of the underlying problem, it is reasonable to expect that

Table 6: Performance results obtained with PARDISO, the original solver and the new solver for the model of the Toulon tunnel.

Solver	# threads	Precon [s]	Solve [s]	# iter
PARDISO	8	16.5	6.8	1
Original	1	290	17	33
New	1	110	19	59
	2	31	50	214
	4	54	38	150
	8	17	27	139

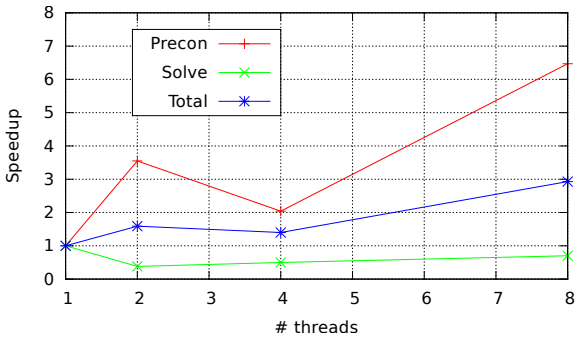


Figure 8: The parallel speedup obtained for the model of the Toulon tunnel.

the solver can also be used efficiently for solving other types of solid mechanics problems involving large variations in material properties.

The physics-based domain decomposition approach plays an important role in the efficiency of the solver. Examples with straight-forward domain decomposition of "layered" finite element models showed a low convergence rate, whereas the physics-based domain decomposition clearly improves the efficiency and robustness of the solver. This means that the solver can lower the runtime significantly even when using only one processor core.

The new solver has been the standard iterative solver in PLAXIS since 2010 and it has shown good performance for a wide range of problems in geomechanics. Indeed, the new solver is between seven and eight times faster than the original solver on an eight-core machine for two representative problems discussed in this paper.

The new solver has its limitations and in some cases it performs worse than the original solver. This has been illustrated with a difficult problem involving the simulation of the construction of a tunnel at Toulon in France. The reason for the sub-optimal performance is not yet completely clear; this is the subject of future research.

The solver has been implemented as a kind of black box that can easily be linked with a serial program without needing additional libraries or a special runtime environment. To achieve this the solver is currently limited to shared memory machines. However, the use of a message passing programming model makes it possible to port the solver to distributed memory systems without much effort.

References

[1] R.B.J. Brinkgreve, W.M. Swolfs, and E. Engin. *PLAXIS 3D 2011 User Manual*. Plaxis BV, Delft, The Netherlands, 2011.

[2] X.-C. Cai, M. Dryja, and M. Sarkis. Restricted additive schwarz preconditioners with harmonic overlap for symmetric positive definite linear systems. *SIAM J. Numer. Anal.*, 41:1209–1231, 2003.

[3] J.J. Dongarra, L.S. Duff, and D.C. Sorensen end H.A. van der Vorst. *Numerical linear algebra for high-performance computers*. SIAM, Philadelphia, 1998.

[4] S.C. Eisenstat, H.C. Elman, and M.H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM J. Num. Anal.*, 20:345–357, 1983.

[5] G.H. Golub and C.F. van Loan. *Matrix computations*. The John Hopkins University Press, 1996.

[6] T.B. Jönsthövel, M.B. van Gijzen, C.Vuik, C. Kasbergen, and A. Scarpas. Preconditioned conjugate gradient method enhanced by deflation of rigid body modes applied to composite materials. *Computer Modeling in Engineering and Sciences*, 47:97–118, 2009.

[7] G. Karypis and V. Kumar. Metis. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Version 4.0. Technical report, University of Minnesota, Department of Computer Science / Army HPC Research Center, Minneapolis, USA, Sep 1998.

[8] F.J. Lingen. *Design of an object oriented finite element package for parallel computers*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2000.

[9] J.A. Meijerink and H.A. Van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[10] Y. Saad. *Iterative methods for sparse linear systems, Second Edition*. SIAM, Philadelphia, 2003.

[11] Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[12] S. Saukh. Incomplete Cholesky factorization in fixed memory with flexible drop-tolerance strategy. In *Proceedings of the Second IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, Sep 2003.

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

[14] J.M. Tang, S.P. MacLachlan, R. Nabben, and C. Vuik. A comparison of two-level preconditioners based on multigrid and deflation. *SIAM. J. Matrix Anal. and Appl.*, 31:1715–1739, 2010.

[15] A. Toselli and W. Widlund. *Domain decomposition methods*. Computational Mathematics, Vol. 34. Springer, Berlin, 2005.

[16] H.A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Stat. Comput.*, 10:1174–1185, 1989.

[17] C. Vuik, A. Segal, and J.A. Meijerink. An efficient preconditioned CG method for the solution of a class of layered problems with extreme contrasts in the coefficients. *J. Comp. Phys.*, 152:385–403, 1999.