Library of parallel PCG solvers

Version 1.0 Draft

Radim Blaheta, Ondřej Jakl, Jiří Starý

Department of Applied Mathematics and Computer Science
Institute of Geonics AS CR, Ostrava-Poruba, Czech Republic

Chapter 1

Introduction

1.1 About this document

This technical report provides a more detailed information about the software, namely the library of parallel PCG solvers, partly developed in framework of the project MSTeP [10]. It has two parts: Part I covers the mathematical background, methods and algorithms employed, Part II describes the library itself and includes also information about its performance.

1.2 The GEM32 package

The work is tightly connected with the research programme of the *Division of Applied Mathematics* of the Institute of Geonics Ostrava, which are the authors members of. Since its very beginning in the late seventies, an important direction in the research activities of the Division has been the development of software for mathematical modelling both for experimental purposes and practical modelling, primarily with applications in mining geomechanics. The latest generation of this software is called *GEM32.* It addresses 3-D finite element (FE) analysis of elasticity and plasticity problems and its (traditional) characteristics include

- *regular structured grids*,
- iterative solvers based on the *preconditioned conjugate gradient (PCG)* method,
- *displacement decomposition (DiD)* applied in the preconditioning.

As every package of this kind, GEM32 in its coarsest segmentation consists of three modules: The *preprocessor* attends to the generation of the computer model, the *solver* performs the major computations and *postprocessor* assists with the processing of the results. The main deal of the computational work is carried out by the solver when it solves the systems of linear equations arising form the FE discretization — the solution of systems of millions degrees of freedom can take many hours to complete. That is why the solvers attract the research interests directed towards the development and implementation of more efficient numerical methods, which would shorten the computation time and allow to manage larger models.

1.3 Parallel library

One of the possibilities in this respect is the *parallelization*, a widely accepted method to improve performance of demanding applications. It boasts an increasing popularity due to the general availability of computer systems with more processors and a better support for the application programmers. In the GEM32 case, the development of the parallel PCG solvers was further encouraged by the requests to compute large models coming from the geomechanical practice. However, parallelization is not a straightforward procedure — many approaches, techniques, alternatives and trade-offs can be considered. In our case, the development proceeded in several directions and resulted in a collection of computer codes which are now presented in a form of a programme library. That library is the subject of in this document.

Part I

Methods and Algorithms

Chapter 2

Problem formulation

Let us consider a 3D domain Ω and the solution u (the displacement) to the following elasticity problem

$$\sum_j \frac{\partial}{\partial x_j}\tau_{ij} = f_i, \quad \tau_{ij} = \sum_{kl} c_{ijkl} e_{kl}, \quad e_{kl} = \frac{1}{2}\left(\frac{\partial u_l}{\partial x_k} + \frac{\partial u_k}{\partial x_l}\right) \quad \text{in } \Omega$$

with appropriate boundary conditions. This problem can be expressed in the variational form: find the displacement $u\colon\ \Omega \to R^3$ that

$$u - u_0 \in V, \qquad V = \{v \in [H^1(\Omega)]^3 :\ v = 0 \text{ on } \Gamma_0\},$$

$$\int_\Omega \sum_{ijkl} c_{ijkl} \frac{\partial u_i}{\partial x_j}\frac{\partial v_k}{\partial x_l}\, d\Omega = \int_\Omega f v \, d\Omega \ + \int_{\Gamma_1} T v \, dS \qquad \forall v \in V.$$

Details about elasticity problems and different ways of their formulations are described, for example, in [7].

Numerical solution of the elasticity problem is based on the discretization of the studied domain Ω by a regular grid and on the application of the finite element method arising from the variational form and leading to the solution of a large linear system

$$Au = b, \quad u, b \in R^n,$$

with symmetric $(n \times n)$ positive definite stiffness matrix A. For this step, we use the conjugate gradient method improved by some preconditioning technique.

Chapter 3

Data storage

At first, we shall focus our attention to the discretization and data preparation. In the case of data processing by sequential programs, the whole domain Ω is divided into hexahedra by a regular grid and these hexahedra are further divided into six tetrahedral finite elements, see figure 3.1. Hence, the stiffness matrix A has all the nonzero entries within a 27-node regular stencil (each grid node has 81 degrees of freedom)[1].

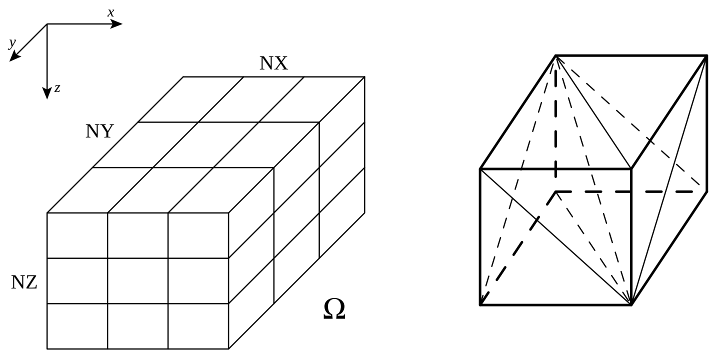

Figure 3.1: The discretization of the studied domain Ω by a regular grid (left). The splitting of a grid element to hexahedra (right).

The grid parameters important for the regular stencil of the stiffness matrix are the numbers of grid nodes in directions $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$: NX, NY, NZ. The total number of grid nodes is NN = NX*NY*NZ and the size of the resultant linear system is ND = 3*NN directions.

Parallel versions of the conjugate gradient method are derived from the data decomposition into certain blocks associated with individual parallel tasks during the solution. In the following sections, we will describe three different ways of data splitting.

[1]Implementation note: Due to the symmetry of the matrix A, we can store only upper triangular part of the matrix, row-by-row by using regular 42 element stencil for storage of the nonzero matrix entries.

3.1 Displacement decomposition

We assume the displacement decomposition of grid nodes corresponding to displacement directions. Figure 3.2 shows the scheme of rearranging and splitting the stiffness matrix A to separated blocks A_{ij}, $i, j = 1, 2, 3$. Similarly, every vector is splitted to three parts.

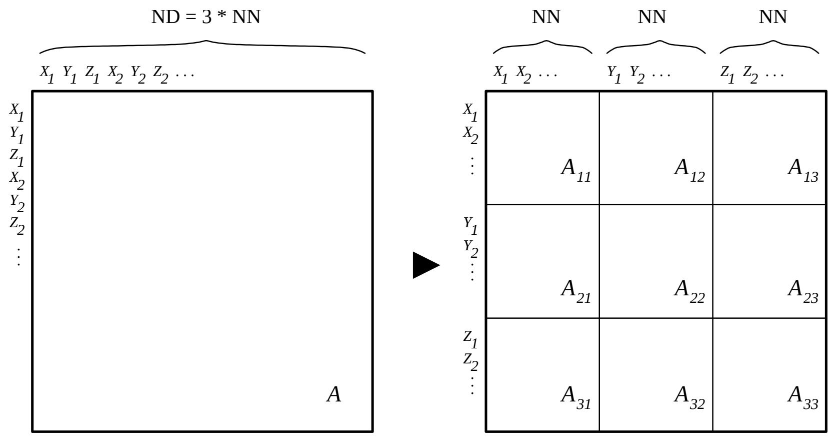

Figure 3.2: The displacement decomposition of the stiffness matrix A.

Matrices A_{ii}, $i = 1, 2, 3$ are symmetric and have a similar structure like the original matrix A. These matrices have all 14 nonzero entries on each row of their upper triangular parts within a regular stencil. Nondiagonal matrices A_{ij}, $i, j = 1, 2, 3$, $i \neq j$ have regularly at most 27 nonzero entries on every raw. Note, that $A_{ij} = A_{ji}^T$.

Individual parallel tasks correspond always to one of three displacement directions. Every task reads necessary data only. For the splitted stiffness matrix A holds: the first task reads blocks A_{11}, A_{21}, A_{31} (plus an appropriate part of the right hand side), the second task A_{12}, A_{22}, A_{32} and the third task A_{13}, A_{23}, A_{33}.

The parallelization of the conjugate gradient method's algorithm based on the displacement decomposition is static because the number of parallel tasks is constant. Therefore to implement this algorithm on systems having higher number of processors is disadvantageous.

3.2 Domain decomposition

At the domain decomposition, we consider a special one-dimensional partitioning of the domain Ω in the $\mathcal{Z}$ direction into m disjunct non-overlapping subdomains $\hat{\Omega}_1, \ldots, \hat{\Omega}_m$. Then we extend the subdomain $\hat{\Omega}_i (i = 1, \ldots, m - 1)$ by adding some "layers" of grid nodes from the adjacent subdomain $\hat{\Omega}_{i+1}$. The number of shared layers will be called the overlapping factor. Let us note that the overlapping factor influences the convergence speed of applied iterative method. Figure 3.3 illustrates three subdomains with overlapping factor 1.

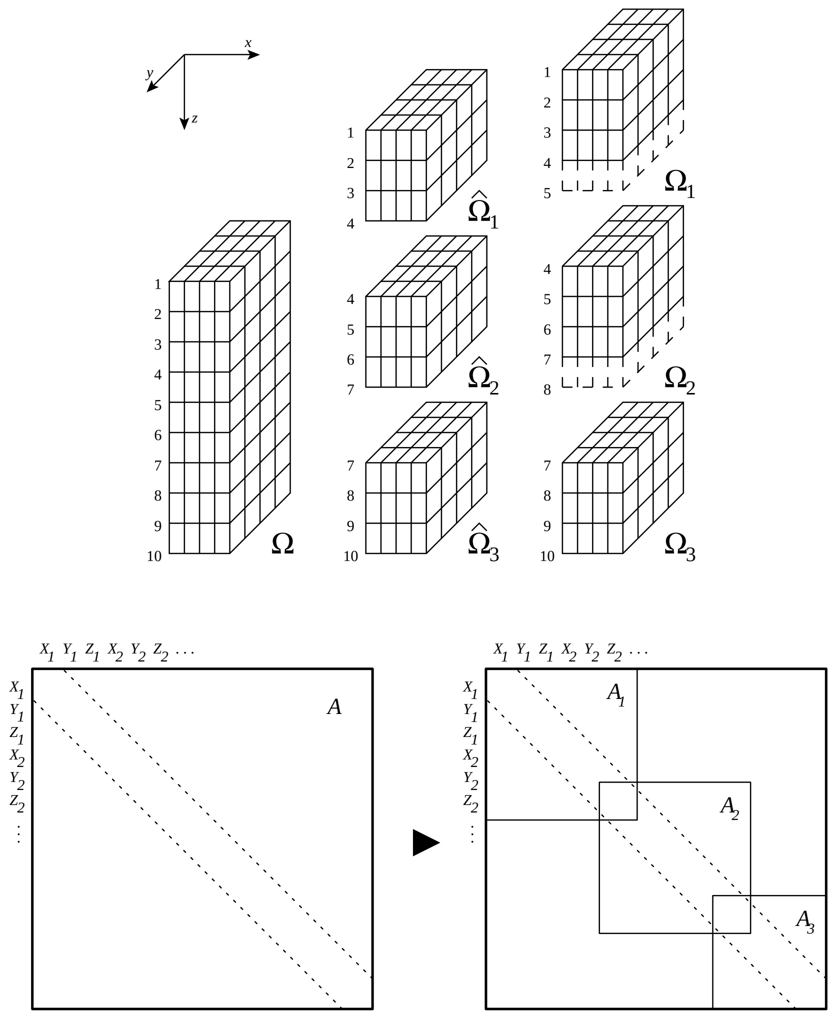

Figure 3.3: The domain decomposition into overlapping subdomains (above). The matrix representation of the decomposed domain (bellow).

Each of m parallel tasks reads data corresponding to the subdomain Ω_i, i. e. an appropriate matrix A_i and a loading vector b_i. Note, the matrices A_i have the same structures and properties as the original stiffness matrix A.

The parallelization of conjugate gradients based on domain decomposition is not static and can use large (in full generality arbitrary) number of processors being available on a parallel computer. But in practice, the huge parallelization can be greatly decelerating the convergence speed of the whole method, for example.

3.3 Combined decomposition

The higher level of parallelization can be achieved by using a combination both the domain decomposition and the displacement decomposition described above. After splitting the domain Ω, we consider a set of overlapping subdomains Ω_i, which are further decomposed to blocks according to displacement directions, see fig. 3.4.

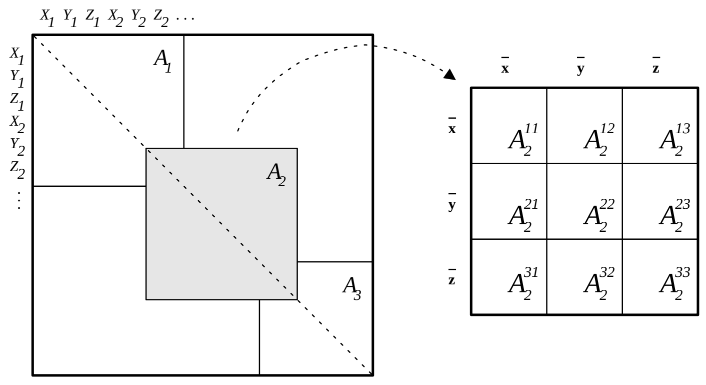
x_1 y_1 z_1 x_2 y_2 z_2 ...
A_1
A_2
A_3
$\bar{x}$ $\bar{y}$ $\bar{z}$
A_2^{11} A_2^{12} A_2^{13}
A_2^{21} A_2^{22} A_2^{23}
A_2^{31} A_2^{32} A_2^{33}

Figure 3.4: The combined decomposition into overlapping subdomains with displacement decomposition.

In the case of m overlapping subdomains, we will prepare data for $3 * m$ parallel tasks in this way. Therefore at the same number of processors, the convergence properties of the solver based on the combined decomposition could be better than the same one based only on the domain decomposition.

3.4 Coarse grid

For the solution of elasticity problems by the conjugate gradients' method, we can use the solution of the same problem on the coarser grid. After discretization of the studied domain Ω by the coarse grid, the finite element method is applied as in the original problem (including boundary conditions), see figure 3.5. Let us denote A_c the stiffness matrix of the coarse problem. The matrix A_c has the same structure as the original matrix A.

It is natural to implement the coarse grid computation as a stand-alone parallel task, which can run completely in parallel with the other tasks intended

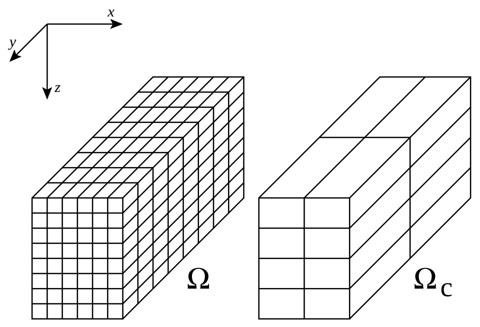

Figure 3.5: Explicit coarse grid.

for computations on subdomains.

It is impossible to make the "explicit" coarse grid in many cases of modelling the real geomechanical problems, but we can use the aggregate coarse grid. We shall aggregate grid nodes in original discretization grid in directions $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$ to bigger nodes of the aggregate coarse grid. Figure 3.6 shows the fragment of the aggregation with the aggregation factor 2 in each direction.

In practice, the aggregation process means, that we will make the new aggregate stiffness matrix A_{agc} by summation of appropriate rows and columns in the original stiffness matrix A. Hereat, the grid nodes cumulate separately on the boundary and inside the domain Ω.

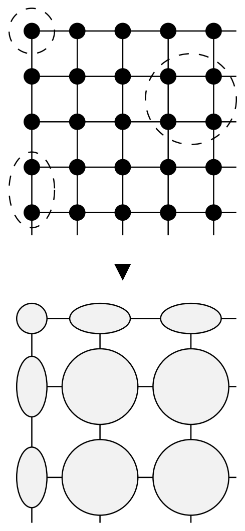

Figure 3.6: The fragment of the top side of an aggregate coarse grid.

Chapter 4

Numerical methods

We consider the numerical solution of elasticity problems and appropriate data storage described before. For the solution of a large linear system

$$Au = b$$

with the symmetric positive definite stiffness matrix A, we chose the preconditioned conjugate gradients' method. In the following sections, we will describe several variants of the method.

4.1 Sequential algorithm

Figure 4.1 shows the sequential version of the algorithm. In this scheme, u^* means the initial approximation of the solution, r the residual, v, w auxiliary vectors and α, β, s_0, s_1 scalars. The operation P is the preconditioning.

The initialization and iteration phases of the algorithm are separated by the termination criterion TC having the form $||r|| \leq \varepsilon ||b||$, where ε is the required relative accuracy (usually 10^{-3} – 10^{-5}).

4.1.1 Preconditioning

The preconditioning is intended for improving the spectral properties of the stiffness matrix A and is given by the inexpensive preconditioning operation $Cw = r$.

We shall consider the form of the preconditioning matrix C called the incomplete factorization:

$$C = (X + L)X^{-1}(X + L)^T,$$

where L is the strictly lower triangular part of the matrix A, X is a diagonal matrix determined by the condition of equal rowsums of the matrices C and $A + D$ for an appropriate diagonal perturbation matrix D.

4.2 Parallel algorithm

A natural step towards the parallelization of the preconditioned conjugate gradients' method is the data decomposition into (in general m) equally sized blocks.

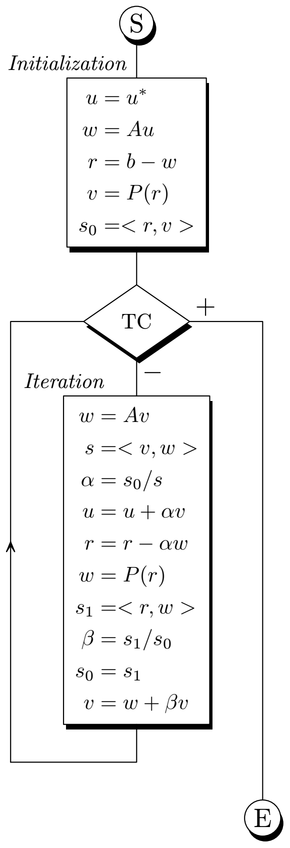

Figure 4.1: Sequential preconditioned conjugate gradients.

Following the common SPMD (single program - multiple data) parallelization model, the solution of the linear system is assigned blockwise to m parallel tasks, which perform the PCG algorithm concurrently (with appropriate data exchange) on their portion of data.

For synchronization and supervision purposes, it is practical to accompany these worker tasks by a master task, which combines partial results to global values, evaluates the termination criterion, etc. The scheme of such a parallel algorithm is presented in figure 4.2.

There are two particular points, where this straightforward parallelization procedure may cause difficulties. The first one denoted by $\mathcal{M}$ in figure 4.2, is the matrix-vector product, which requires extensive intertask communication $\mathcal{C}_{\mathcal{M}}$. The second point denoted by $\mathcal{P}$ is the preconditioning operation. The amount of intertask communication $\mathcal{C}_{\mathcal{P}}$ depends on the type of the preconditioning.

4.2.1 Algorithm DIS

Let us assume the displacement decomposition of data (see section 3.1) and the parallel algorithm mentioned above. At the matrix-vector product $\mathcal{M}$, the i-th worker computes the products $w_{ij} = A_{ji}v_i$, $j = 1, 2, 3$, and transfers all but the w_{ii} to other workers (w_{ij} to the j-th worker). On the contrary, it must obtain their w_{ji}, $j \neq i$, to be able to calculate the new value of $w_i = \sum_j w_{ji}$.

The i-th worker realizes the preconditioning $\mathcal{P}$ by the computation only on the block A_{ii}. In the first case, the incomplete factorization $C_i w_i = r_i$ is applied, where

$$C_i = (X_i + L_i)X_i^{-1}(X_i + L_i)^T,$$

see section 4.1.1. In the second case, the variable preconditioning defines the operation P as a low accuracy solution of the system $A_{ii}w_i = r_i$ by another, inner conjugate gradients' iterations with the preconditioning using the incomplete factorization. While being of limited benefit for the sequential solution, it turned out to be very useful in the parallel case, because it migrates the computational work to communication-free inner iterations. Both described types of the preconditioning are not general, because no task interactions $\mathcal{C}_{\mathcal{P}}$ is necessary.

4.2.2 Algorithm DOM

The algorithm shown on the figure 4.2 can be applied to the domain decomposed data too, see the section 3.2. Having this data partitioning, a standard matrix-vector product $\mathcal{M}$ is performed by using the multiplication $\hat{w}_i = \hat{A}_i\hat{v}_i$ (only on the non-overlapping part of A_i). To keep the vectors w_i updated for all rows of A_i and consistent with the global operation $w = Av$, pair of neighbouring tasks have to exchange calculated components related to the shared rows at the communication $\mathcal{C}_{\mathcal{M}}$. The amount of this data transfers depends on the bandwidth of A and the density of the discretization in the $\mathcal{Z}$ and $\mathcal{Y}$ directions.

For the operation $\mathcal{P}$, we can use the additive Schwarz preconditioner corresponding to the defined domain decomposition:

$$w = P(r) = \sum_{i=1}^{m} R_i^T A_i^{-1} R_i r,$$

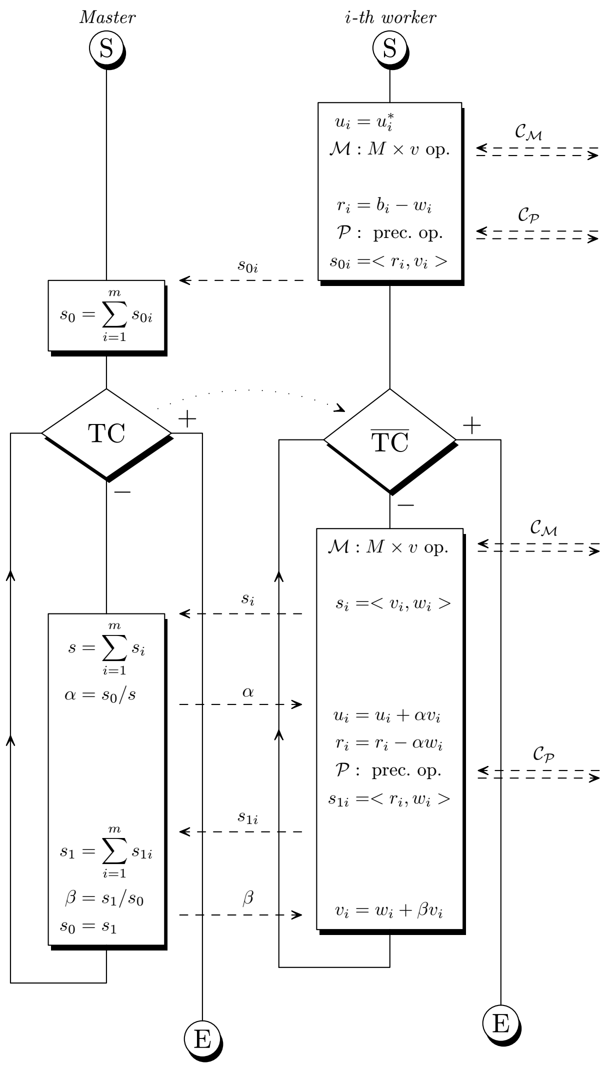

Figure 4.2: An universal scheme of parallel conjugate gradients.

where R_i are restrictions $R_i : u \to u_i$, u_i is the vector of those components of u that correspond to the nodes from the closure of the subdomain Ω_i. Note that the operation A_i^{-1} can be performed by replacing the matrix A_i by its incomplete factorization, for example. The amount of necessary data transfers $\mathcal{C}_{\mathcal{P}}$ depends on the overlapping factor.

4.2.3 Algorithm DIDO

Let us consider the combined decomposition of data (see the section 3.3) and an application of the algorithm depicted in figure 4.2. For the matrix-vector product $\mathcal{M}$ we can compute $w_i = A_i v_i$ on each subdomain Ω_i by means of the multiplications $w_i^{kl} = A_i^{kl} v_i^{kl}$, $i = 1, \dots, m$, $k, l = 1, 2, 3$. Data transfers among three workers are necessary for each subdomain, as in the case of the displacement decomposition (the first part of communication $\mathcal{C}_{\mathcal{M}}$). After this step, to keep the (decomposed) vectors w_i updated for all rows of A_i and consistent with the global operation $w = Av$, the first workers for each subdomain have to exchange calculated components related to the shared rows at the second part of communication $\mathcal{C}_{\mathcal{M}}$.

For preconditioning $\mathcal{P}$, workers use the additive Schwarz preconditioner analogous to the algorithm DOM. Therefore the amount of data transfers $\mathcal{C}_{\mathcal{P}}$ depends on the overlapping factor.

4.2.4 Application of coarse grid

The Schwarz preconditioners can be more efficient if they can take advantage of a "global" information represented by a solution of the same problem, but discretized by a coarser grid, see section 3.4. Let R_c be the matrix of linear interpolation transforming the values from the fine to the coarse grid. Then we can define the two-level additive Schwarz preconditioner

$$w = (P_c + \sum_i R_i^T A_i^{-1} R_i) r$$

or the two-level hybrid non-symmetric Schwarz preconditioner

$$w = P_c r + \sum_i R_i^T A_i^{-1} R_i (r - A P_c r),$$

where P_c is the coarse part of preconditioner

$$P_c = R_c^T A_c^{-1} R_c.$$

Note, that the preconditioner can be symmetrized, but our tests show that the behavior of the CG iterations with the symmetric preconditioner is nearly the same as with the cheaper non-symmetric one.

It is natural to implement the coarse grid computation as a stand-alone worker process, which can run completely in parallel with the subdomain workers in the additive case. In the multiplicative (hybrid) case, the subdomain workers have to wait for the computation and for sending $P_c r$ from the coarse grid worker. However, in practice this clear drawback of the multiplicative approach disappears with the reduction of the number of iterations.

4.2.5 Aggregate coarse grid

In the case of the aggregate coarse grid (see the section 3.4), we can use the above described two-level additive Schwarz preconditioner too, but the coarse part of this preconditioner will be in the form

$$P_c = R_{agc}^T A_{agc}^{-1} R_{agc},$$

where R_{agc} is the restriction $R_{agc} : u \rightarrow u_{agc}$, u_{agc} is the coarse grid displacement given by aggregation (summation) of appropriate values from fine grid nodes. The coarse grid application is the same as in the case of the explicit coarse grid.

Part II

Software and Testing

Chapter 5

The ELPAR library

Simultaneously with the development of parallel PCG algorithms described in the previous part, we proceeded with their computer realization in form of a programme library, denoted as *ELPAR* throughout this document. At the time of its writing, all of those algorithms were coded mostly as β-versions and their testing was still going ahead. Of course, this may motivate some changes of the code in the future.

In this (second) part of the report we give information about the current state of the library, its organization and the way how it can be used, acting partially as a substitute for a manual. Some benchmark results are added to provide an idea about the performance of the solvers.

5.1 General characteristics

Before we proceed to the description of the individual solvers of the library, let us underscore some of their common features.

5.1.1 Message passing

As one could already notice in Part I, the parallel algorithms were designed having *message passing model* in mind. Although possibly more difficult to use, this model of parallel computations, assuming just communication channels between the processor nodes, is very general and can be implemented on any parallel architecture. Nowadays, the most important realizations of the message passing model are *Parallel Virtual Machine* (PVM) [5] and especially *Message Passing Interface* (MPI) [6].

Our parallel codes are prepared to make use of either of those message passing systems. In fact, they have the potential to be independent of a particular message passing library, since for interprocess interactions the solvers just call general routines from a *communication interface* defined for this purpose. This interface may be implemented in any message passing system (now in MPI and PVM) and the solver can switch to it just by relinking its code with a corresponding version of the interface.

5.1.2 Model of the parallel computation

The parallel solvers apply the *master–worker model*. Here, the *master* process is a computationally inexpensive task that controls the progress of the solution, distributes the parameters, summarizes partial results to global values, evaluates the termination criterion, etc. Usually it occupies one processing node for its own.[1] Most of the computational work is performed by the *worker* processes, usually one per node, which in general follow the PCG algorithm on their portion of data. Their number differs according to the partitioning used.

From the programming point of view the solvers are SPMD[2] programmes, what means that there is a single code realizing both the master's and the workers' roles.

5.1.3 Memory vs. disk

The solvers have to work with data of considerable size, e.g. the stiffness matrix, initially stored in files. In principle, the data might be read into the main memory once at the beginning of the computation, or read step by step from the disk, as soon as it is needed, sometimes repeatedly. There is a trade-off between those two approaches, the former promising faster execution paid for by greater memory requirements, the latter maybe slower, but more modest on resources.

The solvers in the ELPAR library employ the first approach, which also allows more transparent comparison and evaluation (the solvers print out both the time needed to read in the data and the time spent in the computational phase). On the other hand, insufficient memory may pose strict limit on the size of solvable problems on a given computer. In any case, the solvers can be quite readily reprogrammed to use the disk oriented approach.

The constants limiting the size of the largest task can be found in the source codes as Fortran parameters. If they are to small for the given problem, the programmes usually print out an error message informing about the required value(-s). After increasing those constants and recompilation, the codes can be prepared to manage larger problems. Keep in mind however that there are another limits imposed by the operating system, and with statically linked libraries, the size of the executables may be huge.

5.1.4 User interface

Since some parallel environments do not support it, the solvers avoid interactive mode of operation. Instead, they read all the run-time control parameters from files, exceptionally from the command line. The basic *parameter file* of a solver has an identical name, with an `.in` suffix. In this text file the user specifies the following run-time parameters pertinent to the PCG algorithm itself:

1. relative accuracy of the solution
2. preconditioning: incomplete factorization (value 1) or variable preconditioning (2)

[1]In fact, there is a small loss of performance only, if the master shares its processor node with one of the workers (must be supported by the parallel run-time environment).

[2]Single Programme Multiple Data

3. limit on the number of iterations
4. starting approximation: the right-hand side (0), user specified – read in from file (1), zeros (2)

Note: Some solvers may be restricted in allowable values.

Another set of parameters, more related to the partitioning and parallelization, is taken into account during the initialization of the parallel execution, performed by supporting programmes. See the next section 5.1.5.

The solvers (their masters) write an *execution log* to the standard output. It includes the code version, task information, parameters of the solution, convergence characteristics of the iteration sequence and timing information. The output is duplicated in a `.rep` file.

5.1.5 Auxiliary procedures

Before a parallel solver can start the solution, the data must be prepared for the parallel processing. Similarly, after the parallel execution the results have to be gathered and combined together. For this purpose, each solver is accompanied by a suite of supporting (sequential) programmes that (1) split the data files generated by the preprocessor to a new set of files appropriate in form and content to the solver, applied partitioning and specified parameters of the solution, (2) combine the results computed by the workers to a form acceptable for the given postprocessor and (3) depending on the solver, possibly carry out some other auxiliary work. Their execution may be also controlled through parameter files (`.in` extension). We shall denote the actions taken to prepare the run of a parallel solver (point (1) in the first place) as *initialization* of the parallel solution.

If the parallel facility does not provide disk space shared by its processor nodes or if its performance is poor, it is necessary to distribute the data files with workers' input to directories accessible by the workers (usually on nodes' local disks). Similarly, after the computation is over, it might be needed to gather the output files from the nodes.

5.2 Parallel solvers

Keeping those common features in mind, let us describe the individual solvers in the ELPAR library.

5.2.1 The displacement decomposition solver

A parallel solver based on the displacement decomposition (DiD) technique, described in section 4.2.1, was already implemented before the LB98273 project was launched, see e.g. [3]. Its code, has been revised before it was included to the ELPAR library. In particular, now it uses a unified user interface with the other solvers and the stiffness matrix is stored in the main memory during the computations. Its features include two preconditioning options and the ability to solve singular systems making use of projections.

Name of the programme: `itera`

Parallelization: The displacement decomposition partitions the domain to three subdomains, each of which corresponds to the displacements in one of the x, y, z directions. Thus, `itera` gives rise to exactly three worker processes, plus one master process, i.e. it employs at most four processors. Being not scalable, it is appropriate for small parallel environments.

Preconditioning: Based on a control parameter, `itera` can apply tho kinds of preconditioning: *incomplete factorization* and *variable preconditioning*. Because this option has an essential impact on the solution process, it is often convenient to speak about two different solvers. We shall reference them as follows:

- *DiD-IF* – `itera` with incomplete factorization preconditioning

- *DiD-VP* – `itera` with variable preconditioning

For the *DiD-VP* solution, there are two constants controlling the inner PCG iterations, after some tuning set as follows:[3] The relative accuracy to 10^{-1}, the iteration limit to 30.

Parameter file: `itera.in`; see 5.1.4 for details.

Supporting programmes:

- The initialization programme `dsplit` prepares data for the run of the `itera` solver. It separates blocks of the the rearranged stiffness matrix and the right-hand side that correspond to the displacement directions x, y, z and stores them in new files to be processed by the workers. The only parameter in its parameter file `dsplit.in` switches on projections for the solution of *singular* systems. In such a case a projection matrix is generated, too.

- The `smerge` utility just composes a single file of resulting displacements from partial results obtained by the workers.

5.2.2 The domain decomposition solver

This solver is a new development with the goal to achieve more scalability, i.e. to employ more than four processors, if available. It is based on the domain decomposition (DD) technique and follows the algorithm described in 4.2.2. The user can optionally provide a coarse grid to improve the efficiency of the solution or use an automatically generated aggregate coarse grid for the same purpose.

Name of the programme: `isol`

[3]They can be changed in the source code.

Parallelization: The current DD implementation considers a simple one-dimensional "geometric" partitioning of the domain in the z direction to a chosen number of subdomains. The adjacent subdomains can partially overlap: The size of the shared area is specified by the *overlapping factor*. Each subdomain is processed by one worker process and interactions are necessary only between "neighbours" in the 1-D decomposition. The computation is supervised by a master process. Thus, the `isol` solver can employ an arbitrary number of processor nodes and has the potential to take advantage of large parallel facilities.

Preconditioning: At present, `isol` is restricted to the incomplete factorization preconditioning.

Coarse grid: The DD algorithm is expected to be more efficient if it can take advantage of a "global" information, represented by a solution of the same problem, but discretized by a coarser grid. There are the following options in this respect:

- *Explicit* coarse grid: Before the solution, the user generates an appropriate coarse grid task, which is then linked (see the `dconv` utility below) with the original fine grid problem. During the solution, the coarse grid is handled by an additional worker. We shall refer to this option, i.e. `isol` using an explicit coarse grid, as *DD-EC*.

- *Aggregate* coarse grid: It may be intricate to design a coarse grid for a practical problem. For this reason, we experimented with a coarse grid generated automatically (by the `magr` utility below) by the aggregation of adjoining nodes of the original grid, see 4.2.2. Again, one more worker is employed for the aggregate problem. This alternative of the solution is called *DD-AC*.

- No coarse grid: This basic mode of `isol` is labelled *DD-0C*.

Parameter file: `isol.in`; see 5.1.4 for details.

Supporting programmes:

- For `isol` the initialization programme is called `dconv`. Its parameter file contains the following items:

  1. number of subdomains to partition the domain into
  2. their overlapping factor
  3. application of the coarse grid: no coarse grid (value 0), explicit coarse grid (1), aggregate coarse grid (2)

  If the explicit coarse grid parameter is specified, `dconv` expects to find the data files of the coarse grid problem (supplied by the user, specially named) in order to link it with the original task. In case of the aggregate coarse grid, another utility, `magr`, must be launched before `dconv`.

- The `magr` programme arranges for the preparation of the aggregate grid. Roughly speaking, it contracts every $3 \times 3 \times 3$ adjacent nodes[4] of the original (regular) grid into a single node of the aggregate grid, producing the corresponding data files for `dconv` and `isol`.
- The `srest` utility just composes a single file of resulting displacements from partial results obtained by the workers.

5.2.3 The combined solver

The two partitionings applied in the DD and DiD solvers are on different "levels" and it is possible to combine them. This was the motivating idea behind the latest development, the DDiD[5] solver, promising to take the best of both: the efficiency of the DiD solver and scalability of the DD solver. See 4.2.3 for further information. At present, this solver is in its final stage of debugging, and hence no benchmark results are available yet.

Name of the programme: `iscom`

Parallelization: The primary partitioning is realized by the domain decomposition, as described in the previous section 5.2.2. For the computation on each subdomain the displacement decomposition (5.2.1) is then applied, engaging a single master process for all the workers. Thus, the total number of parallel processes is $3n + 1$, where n is the number of subdomains. If a coarse grid is included in the DD computation, another worker is added. The `iscom` solver therefore targets multicomputers with large number of processor nodes.

Preconditioning: `iscom` also employs the incomplete factorization preconditioning only.

Coarse grid: `iscom` is able to take advantage of the coarse grid (both explicit and aggregate), exactly as the `isol` solver. Thus, an analogy to the corresponding paragraph in the previous section holds and we shall distinguish between

- *DDiD-EC* – `iscom` with an explicit coarse grid
- *DDiD-AC* – `iscom` with an aggregate grid
- *DDiD-0C* – `iscom` without a coarse grid

Parameter file: `iscom.in`; see 5.1.4 for details.

Supporting programmes: The `iscom` code is accompanied by an analogical set of utilities as `isol`. These are:

- `dtrans` for the initialization; its parameter file `dtrans.in` has the same contents as `dconv.in`
- `macg` for the preparation of the aggregate grid
- `sorfo` for the composition of the resulting displacement file

[4]This aggregation degree can be changed in the source code.

[5]DDiD means DD + DiD

5.2.4 The sequential solver

Although it is not a parallel code and therefore not a regular component of the ELPAR library, let us also give basic information about the sequential alternative to the parallel PCG solvers in this place. This solver has been derived from the original GEM32 solver [4], the code being made more portable and compatible with the other solvers. The main objective of the programme is to serve as a reference in terms of results and performance.

In fact, there are two sequential codes, which differ in the way how the stiffness matrix is dealt with. The first one (`sesol`) stores all its data in memory, the second one (`sesolo`) reads its rows directly from the disk at their processing time (c.f. the treatment in section 5.1.3). Thus, we shall distinguish the following sequential approaches:

- *SEQ-M* – sequential solver with data in memory `sesol`
- *SEQ-D* – sequential solver with data on disk `sesolo`

Name of the programme: `sesol/sesolo`

Preconditioning: incomplete factorization

Parameter file: `sesol.in/sesolo.in`; see 5.1.4 for details.

Supporting programmes: –

Chapter 6

PortaGEM

In principle, the methods and algorithms employed in the solvers of the previous chapter are general and could be applied for the solution of linear systems generated by an arbitrary software using FE method with linear finite elements. However, any particular implementation has to cooperate with some pre- and postprocessor modules, agreeing at least on the data structures. With respect to its origin it was natural that the ELPAR library in its initial realization was embedded in the GEM32 package (cf. section 1.2), which provides the necessary hinterland. To work with another FE software, just changes related to the specific layout of the stiffness matrix in GEM32 are necessary. Throughout this document, we consider GEM32 as the hosting environment for ELPAR.

So far, the GEM32 package has been developed and operated exclusively on the IBM RISC System/6000 workstations, making use of their Fortran 77 compiler (AIX XL Fortran), without giving mind to the portability issues. To be able to test ELPAR on various platforms and meet the natural requirements on its portability, the GEM32 package itself had to be revised in this respect. Thus, the work on ELPAR was simultaneously an impulse to develop a portable version of GEM32, which we call *portaGEM*. We expect that portaGEM will gradually become the “norm” for further development of GEM32. At present, it includes all the codes of GEM32 that are necessary to run the solvers in the ELPAR library and serves as the source for its distribution.

6.1 Structure of portaGEM

PortaGEM is in fact a directory tree that comprises all the files necessary to build the ELPAR library and run selected benchmarks on a given platform. On the highest level, it includes the following subdirectories important for ELPAR:

- `benchmrk`: Data files of various benchmarks. Scripts for testing.
- `common`: Files of universal utility, e.g. scripts for an automated build on a new platform.
- `gemin`: Source codes of the `gemin3d` preprocessor.
- `local`: Dedicated for platform-dependent files. E.g. executables or results of benchmarks.

- `parsol`: Subdirectory of the ELPAR library. The second-level subdirectories (`itera`, `isol`, `iscom`) contain the source codes of individual parallel solvers and their supporting programmes.
- `solver`: Similarly, its subdirectories hold source codes of the sequential solvers (`sesol`, `sesolo`).
- `stiffmat`: Source codes of the `stiffmat/smat` routine (generates the stiffness matrix from the preprocessor output).
- `tgrid`: Source codes of the `tgrid` preprocessor.
- `utils`: Subdirectories of various auxiliary programmes.

Some more details can be added to the description of the portaGEM structure: For example, there can be several versions of a programme in portaGEM. Most directory names have an extension indicating the author of the code. `README` files describe the contents of directories and summarize the history of the programmes. There is also a simple "stamping" mechanism helping keep the integrity of portaGEM when more programmers modify it.

6.2 Building the codes

For compilation and linking of the executables, the standard MAKE utility is taken advantage of. Each set of source files in a portaGEM directory making up a programme is accompanied by a *makefile*. These makefiles have a unified form independent of the target platform: Specific commands are isolated from the makefiles and included only during their interpretation based on an *architecture parameter*. The makefiles also keep track of the history of compilations in individual directories.

An additional parameter is used in makefiles of the parallel codes (solvers), so as to be able to produce both the MPI and PVM codes.

In the `common` directory, there is a shell script that automates the generation of all the portaGEM executables. Because of their number (more than twenty), this is especially useful when installing portaGEM on a new computer.

Chapter 7

Testing

In this chapter we shall illustrate the performance of the solvers of the ELPAR library on same examples. Recall that both the development of the codes and their testing has not been finished yet, so the presented results should be regarded as preliminary.

7.1 Benchmark problems

In [1], the Division of Applied Mathematics has designated several problems as benchmarks suitable for testing finite element solvers. From those, we employed the following ones in our tests.

7.1.1 The Square Footing problem

In the majority of experiments we used the so called *Square Footing* (FOOT) problem, a 3-D task of soil mechanics, which deals with the influence of flexible footing represented by localized pressure (on the top side of the domain) to the stress development in a soil medium and is illustrated in fig. 7.1. Due to symmetry, only a quarter of the domain of interest is discretized. The standard discretization is accomplished by a rectangular structured grid of $40 \times 40 \times 40$ nodes, providing a linear system of 192 000 equations.

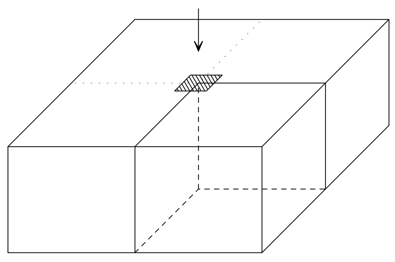

Figure 7.1: The Square Footing problem. A square area on the top side of the domain is loaded by a uniform load.

By changing the density of the grid, we derived a sequence of sixteen problems with the same geometry, but increasing in the number finite elements, which ranges from 5^3 nodes (375 linear equations) to 80^3 nodes (1 536 000 equations). We shell denote the problems of this collection according to their size by FOOT05, FOOT10, …, FOOT80.

7.1.2 The Dolní Rožínka problem

The second benchmark used in the tests is an example of a large scale computation. It has been derived from a practical problem related to mining in the uranium ore deposit at *Dolní Rožínka* (DR) in the Bohemian-Moravian Highlands. This model considers a domain of $1430 \times 550 \times 600$ meters, with the top side 700 m under the surface. Three uranium ore veins, where the mining process is concentrated, are located in this domain. For both the basic materials (uranium ore and surrounding rocks) and the goaf material filling the volume of the extracted ore, linear elastic behaviour is assumed. The discretization of the domain by a regular structured grid has $124 \times 137 \times 76$ “bricks”, see fig. 7.2. This leads to a finite element system of 3 873 264 degrees of freedom.

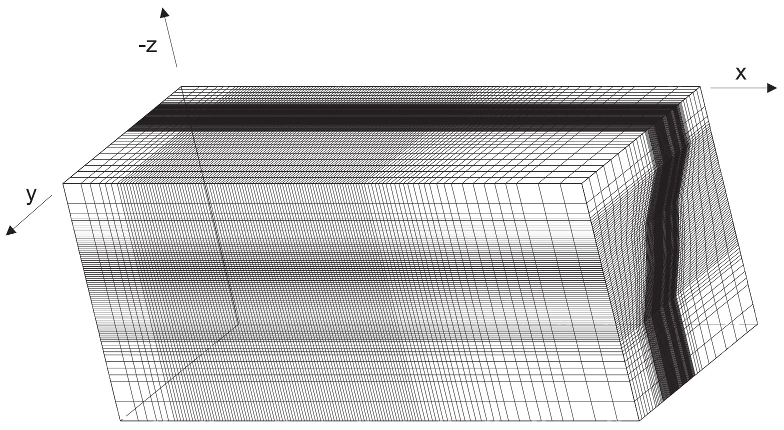

Figure 7.2: The mesh of the DR problem

Mathematical modelling aimed at the assessment of geomechanical effects of mining, e.g. at the comparison of different mining methods from the point of view of the stress changes and the possibility of dangerous rockbursts. The task was to simulate four selected stages of ore extraction, represented by a four-step sequence of problems with different material distribution. By the way, the modelling raised some mathematical issues, such as coping with singular, slightly inconsistent systems. See [2] for details.

As the DR benchmark, the fourth step of the sequence mentioned above was selected.

7.2 Computing environments

To verify a programme library, it is important to test it on as many computer architectures as possible. In case of parallel programmes and especially when plenty of processor nodes are needed, this may be not easy to procure. Thus, we appreciated very much the possibility of making use of the computing facilities at the Edinburgh Parallel Computing Centre (EPCC) at the very end of the solution of the LB98212, c.f. 8.2. However, the primary platform for the development and runs of the ELPAR library remained the IBM systems at VŠB – Technical University Ostrava and the Institute of Geonics.

As explained in chapter 6, to make the programmes of ELPAR run in the GEM32 environment, the whole portaGEM package has to be ported to the new platform. So far, this has been successfully accomplished for the following parallel platforms:

- *IBM* (VŠB – TUO): IBM RISC System/6000 SP massively parallel system, 8 processor nodes (not identical: differences in types and frequencies of the processors (POWER2/67 – POWER2SC/160), local memory size (128 – 512 MB), disk access, etc.), High Performance Switch, Ethernet and ATM interconnects; AIX 4.3 operating system, XL Fortran compiler, PVM 3.3 or IBM MPI (a proprietary MPI implementation within the IBM Parallel Environment (PE) package)

- *SUN* (EPCC): Sun Enterprise HPC 3500 and 6500 symmetric multiprocessor systems, 8 and 18 processor nodes (UltraSPARC-II/400, 7 and 18 GB of shared memory); Solaris 7 operating system, SUN Fortran 90 compiler, SUN MPI; a Sun Enterprise HPC 3000 as a front-end server

- *LINUX* (EPCC): a Beowulf cluster, 16 processor nodes (AMD Athlon/650 processors, 128 MB of local memory, diskless); 2x FastEthernet interconnect, Linux RedHat 6.0 operating system, Portland Group Fortran 90 compiler, MPICH MPI implementation; a front-end/NFS server

N.B.: EPCC provides also a large Cray T3E system (more than 300 processor nodes), which we had to omit in our limited time. The port to a new platform is usually not a straightforward procedure, since in spite of a progressing standardization, parallel architectures boast numerous proprietary features and there are many factors that exercise influence upon a parallel application. As a rule, one has to learn a lot of tools, experiment with compilers and their options, sometimes inducing changes in the source codes, manage the local parallel environment (a message passing system and often a batch queuing system), etc. A lot of additional effort is then needed to optimize the implementation.

7.3 Selected results

Among the platforms above, which portaGEM has been ported to, the best conditions to test and compare the solvers of the ELPAR library provided the SUN systems, especially the HPC 6500 machine. They surpassed the other ones in the number of processors, memory size and overall performance, and offered also an appropriate development and run-time environments. The Load Sharing Facility (LSF), their batch processing system, guarantees quite precise and repeatable timings, usually with little impact of other running jobs, because it assigns one user process per processor. That is why we focused on getting a complete set of results on the SUN platform.

7.3.1 Technical details

- In the tests below, the following codes were used:
  - `itera` version 4.00c, compiled by a
    `mpf90 -fast -xchip=ultra2 -xarch=v8plusa -lmpi ...`
    command line
  - `isol` version 1.20b, compiled also using
    `mpf90 -fast -xchip=ultra2 -xarch=v8plusa -lmpi ...`
  - `sesol` version 1.00a, compiled by
    `f90 -fast ...`

  (Note that the unfinished `iscom` solver did not take part in the testing.)
- At the compile time, all source codes were dimensioned to manage the DR problem.
- Because PVM was not available on the SUN platform, the tested codes employ MPI for message passing.

7.3.2 The FOOT results

The subsequent graphs show the behaviour of the `itera` and `isol` solvers on the FOOT40, FOOT60 and FOOT80 benchmarks as far as the execution time and number of iterations is concerned. The solver (see section 5.2 for the abbreviations) are distinguished as follows:

- DD-0C: dot (●)
- DD-AC: x-mark (×)
- DD-EC: star (∗)
- DiD-IF: plus (+)
- DiD-VP: circle (◯)

Since we are primarily interested in the parallel iterative solution itself, the values, given in seconds, represent the wall-clock times of the *solution phase* alone, measured by the solvers. It does not include the time spent reading in the data, which was about 4 s for FOOT40 and up to 30 s for FOOT80. The whole set of measured data is available in appendix A.

The accuracy of the solution prescribed for all benchmarks was 10^{-4}. In the DD-EC solution of all three benchmarks, FOOT10 as the coarse grid task was employed.

Our **observations** include:

1. The FOOT40 benchmark confirmed that the DD-0C solver (without a coarse grid) lags behind the other DD solvers both in the magnitude and evenness of the achieved times. To reduce the number of test runs a little, we omitted DD-0C in the FOOT60 and FOOT80 benchmarks.

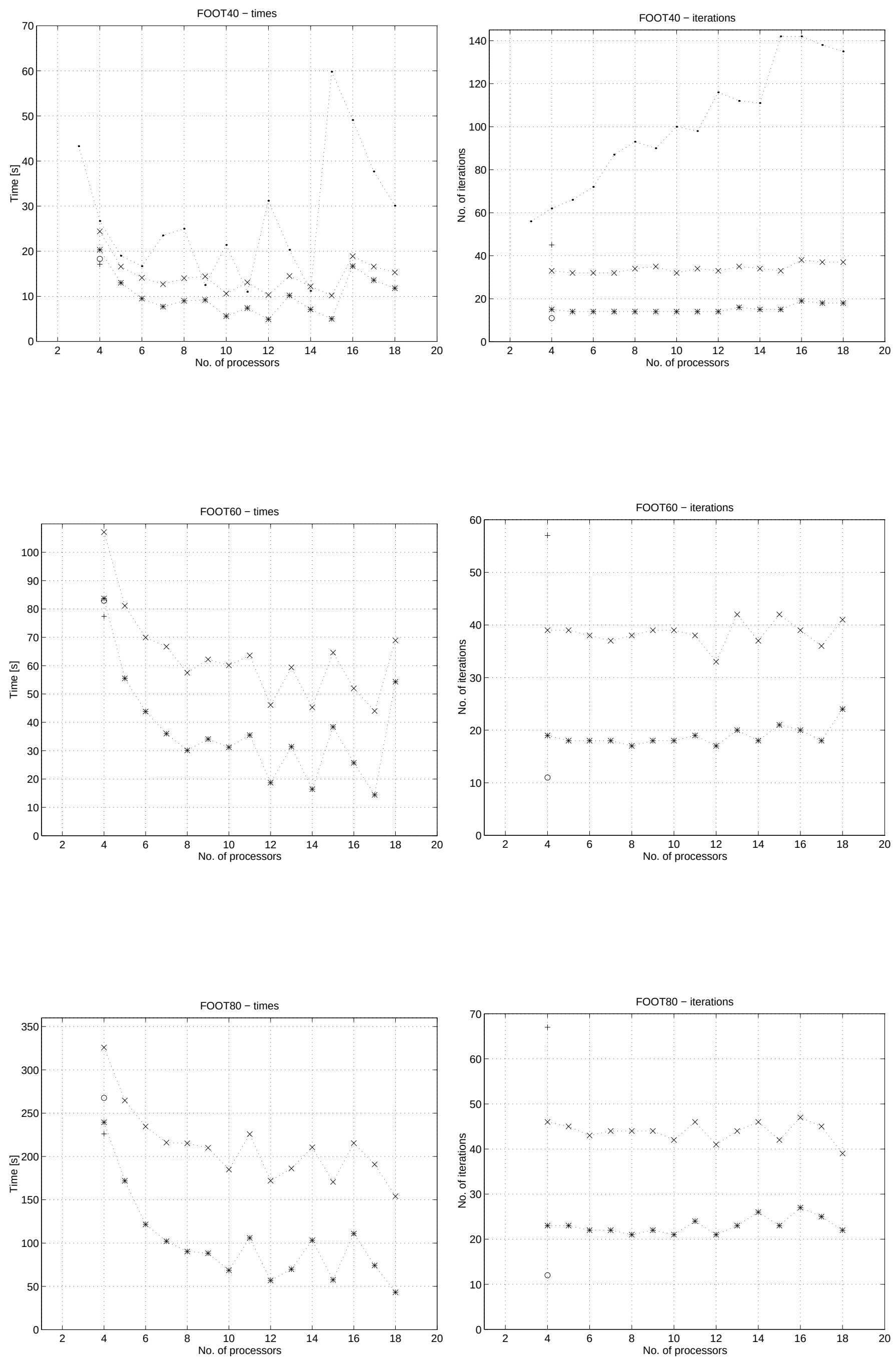

Figure 7.3: Interpolated solution times (*left*) and iteration counts (*right*) for the FOOT40, FOOT60, FOOT80 benchmarks (192 000, 648 000, 1 536 000 equations)

2. DD solvers outperform the DiD solvers when the shortest execution time is the priority. But their efficiency is lower than that of the DiD solvers and they do not scale well with the number of processors. In fact, "oscillations" in their solution time can be related to the oscillating number of iterations, but this seems not to be sufficient to explain some dramatic changes in the times of solution. This phenomena needs further investigation.

3. When comparing the DiD solvers between themselves, somewhat better performance of DiD-IF compared with DiD-VP could been expected because of the enormous speed of message passing reachable on shared memory machines such as SUN HPC.

4. No regular experiments have been made yet to investigate the influence of the size of the coarse grid on the DD-EC solution. At the moment, there is a problem with the performance of the `dconv` utility: Its execution time (needed for the "synchronization" of the coarse and fine grids) is much longer than the solution itself (more then 3 minutes in the case of FOOT60 and FOOT10 as its coarse grid).

Some additional remarks:

1. The SUN HPC platform boasts the best times ever achieved with the GEM32 solvers.

2. When using the same number of processors, the solvers run faster on the HPC 3500 machine than on the HPC 6500, mainly due to shorter data reading. This can be explained by the fact that the working directory was located on the HPC 3500 disk and only mounted to the HPC 6500.

7.3.3 The DR results

The conditions to run the DR benchmark were similar to the FOOT benchmarks above. That is, the accuracy prescribed for the DR benchmark was also 10^{-4} and we were interested in the solution phase, which does not include the data reading time (60 – 110 s, see appendix A for details).

Because of some problems with the installation of the DR benchmark and its longer execution time, we carried out the experiments with a limited combination of solvers and processors. In fact, only the most promising solvers were tested:[1]

- DD-AC: x-mark ($\times$)
- DiD-IF: plus ($+$)

Nevertheless, the results shown below seem to provide quite clear picture about the current situation:

The main conclusion is that for the DR benchmark, the DiD solver works better then the DD solver regardless of the number of processors DD employs.

[1]There was no appropriate coarse grid to use DD-EC.

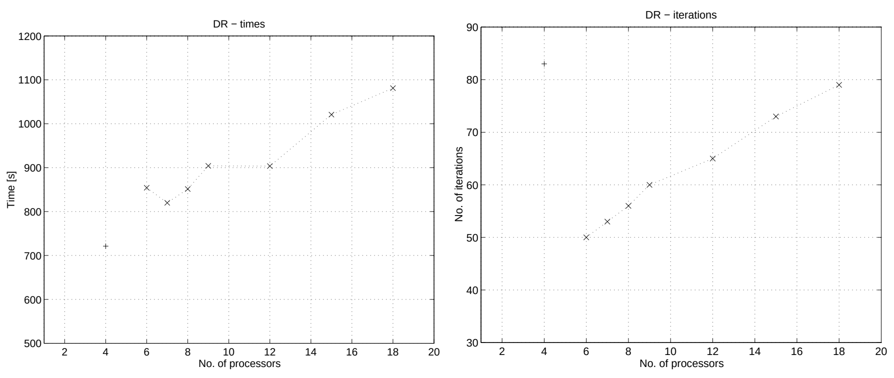

Figure 7.4: Solution times (*left*) and iteration counts (*right*) for the DR benchmark (3 873 264 equations)

Chapter 8

Conclusions

8.1 Future work

The previous text has left a lot of issues open — they need more work and investigation. As the important we consider the subsequent list, which makes our working programme for the near future.

- Finishing and benchmarking of the `iscom` (DDiD) solver; comparison with the other solvers.
- Explanation/elimination of the oscillations in the run times of the `isol` (DD) solver, possibly with some improvement of its efficiency.
- Port of ELPAR/portaGEM to the Cray T3E platform, to improve its portability and verify the behaviour of the codes in a parallel environment with tens of processor nodes.
- Solution of a refined DR problem ($6-10$ million degrees of freedom).
- Testing on the LINUX platform. An affordable Beowulf cluster like this is a parallel hardware with the greatest perspective to be acquired in our financially restricted conditions.
- Testing of the PVM-versions of the solvers, where possible. Comparison with the MPI-versions.

8.2 Acknowledgements

The authors would like to acknowledge the support of:

- the European Commission through grant No. HPRI-1999-CT-00026 (the TRACS Programme at EPCC)
- the Czech Ministry of Education through grant No. LB98212
- VŠB – TU Ostrava through project No. CEZ:J17/98:2724019
- the Grant Agency of the Czech Republic through grant No. 105/99/1229

Appendix A

Numerical form of benchmark results

In this appendix, we provide the complete set of data used to generate diagrams in chapter 7. For each of the solvers and tests, the following values are given:

`#p`	Number of processor employed (for both the master and workers)
`#it`	Number of iterations performed to achieve the prescribed accuracy
`Tr`	Time (in seconds) to read in data from files
`Tc`	Time (in seconds) spent in the solution (iterative) phase
`Tot`	Total wall-clock time (in seconds) of the solution

Let us add that whereas the `Tr` and `Tc` times provide the solvers themselves, the `Tot` time was measured by the `/usr/bin/time` system utility. Due to the system overhead involved in the starting and stopping of programmes, the following inequality holds:

$$\mathtt{Tr} + \mathtt{Tc} < \mathtt{Tot}$$

A.1 FOOT results

A.1.1 FOOT40

```
DD solvers:

    | DD-AC (aggreg.) |  DD-EC (explic.) |   DD-0C (no coarse)
  #p|#it| Tr| Tc | Tot  #it| Tr| Tc | Tot   #it| Tr | Tc | Tot
   3                                          56  3.2 43.3 47.7
   4  33 3.6 24.4 29.1   15 3.9 20.3 25.3    62  3.3 26.7 31.2
   5  32 3.6 16.6 21.4   14 3.3 13.0 17.5    66  3.2 19.0 23.3
   6  32 3.3 14.1 18.6   14 3.4  9.5 14.0    72  8.2 16.7 26.5
   7  32 3.5 12.7 17.6   14 3.4  7.7 12.2    87 10.5 23.5 35.5
   8  34 3.5 14.0 18.8   14 3.6  9.0 13.8    93 11.8 25.0 38.8
   9  35 3.6 14.4 19.3   14 3.7  9.2 14.2    90 13.9 12.5 28.4
  10  32 3.7 10.6 15.5   14 3.7  5.6 10.7   100 11.9 21.4 35.6
  11  34 3.8 13.1 18.2   14 3.8  7.4 12.6    98 10.4 11.0 23.2
  12  33 3.8 10.3 15.4   14 3.8  4.9 10.1   116  4.4 31.2 37.1
  13  35 4.0 14.5 20.0   16 4.1 10.2 15.7   112  3.8 20.3 25.7
  14  34 4.0 12.2 17.8   15 4.1  7.1 12.8   111  4.1 11.2 16.9
  15  33 4.1 10.2 15.9   15 4.3  5.0 10.9   142  4.2 59.8 65.6
  16  38 4.6 18.9 25.1   19 4.5 16.7 22.9   142  4.3 49.1 54.9
  17  37 4.5 16.6 22.7   18 4.5 13.6 19.9   138  4.3 37.7 43.9
  18  37 4.6 15.3 21.5   18 4.5 11.8 18.1   135  4.3 30.1 36.0


DiD solvers:

    | DiD-IF (inc.f.) |  DiD-VP (var.p.)
  #p|#it| Tr| Tc | Tot  #it| Tr| Tc | Tot
   4  45 4.6 17.1 22.9  11  3.7 18.3 23.2


Sequential SEQ-M solver:

  #p|#it| Tr| Tc | Tot
   1  45 4.1 77.3 82.2
```

A.1.2 FOOT60

```
DD solvers:

    |  DD-AC (aggreg.)   |   DD-EC (explic.)
  #p|#it| Tr | Tc   | Tot   #it| Tr | Tc | Tot
   4  39 11.0 107.1 119.7   19 10.7 83.7 95.9
   5  39 14.6  81.1  97.7   18 10.8 55.5 67.8
   6  38 15.5  69.9  88.1   18 10.9 43.8 56.3
   7  37 12.9  66.7  82.2   18 10.9 36.0 48.3
   8  38 11.3  57.5  70.8   17 11.4 30.1 43.1
   9  39 11.6  62.2  75.7   18 11.5 34.1 47.1
  10  39 11.6  60.1  73.8   18 11.6 31.2 44.4
  11  38 12.0  63.6  78.0   19 12.1 35.5 49.4
  12  33 12.2  46.1  60.8   17 11.8 18.7 32.4
  13  42 12.2  59.4  73.4   20 12.2 31.4 45.4
  14  37 12.4  45.3  60.3   18 12.3 16.4 30.5
  15  42 12.8  64.6  80.6   21 12.7 38.4 53.2
  16  39 13.0  52.0  67.9   20 12.5 25.7 40.3
  17  36 13.0  44.0  59.0   18 13.0 14.4 29.4
  18  41 13.8  68.9  86.1   24 13.6 54.3 70.5


DiD solvers:

    | DiD-IF (inc.f.)    |  DiD-VP (var.p.)
  #p|#it| Tr | Tc   | Tot   #it| Tr | Tc | Tot
   4  57 15.8  77.4  94.8   11 12.5 82.9 96.9


Sequential SEQ-M solver:

  #p|#it| Tr | Tc   | Tot
   1  57 14.1 343.7 360.3
```

A.1.3 FOOT80

DD solvers:

```
   |  DD-AC (aggreg.)   |   DD-EC (explic.)
 #p|#it| Tr | Tc  | Tot   #it| Tr | Tc  | Tot
  4  46 26.0 325.8 354.2   23 25.0 239.5 266.7
  5  45 25.9 264.6 292.8   23 25.4 171.9 199.5
  6  43 25.9 234.6 262.8   22 25.7 121.6 149.5
  7  44 26.3 216.0 244.5   22 25.7 102.1 130.1
  8  44 25.7 215.2 243.1   21 26.1  90.3 118.6
  9  44 27.6 210.0 239.8   22 27.9  88.3 118.4
 10  42 26.5 184.9 213.5   21 26.6  68.6  97.1
 11  46 28.1 225.8 256.0   24 27.5 106.0 135.6
 12  41 27.4 172.0 203.7   21 27.5  56.9  86.6
 13  44 27.7 186.2 219.1   23 27.7  69.8  99.8
 14  46 28.6 210.6 241.4   26 28.8 103.2 134.2
 15  42 27.8 170.6 204.5   23 28.3  57.5  88.0
 16  47 30.3 215.4 248.0   27 29.2 111.0 144.8
 17  45 29.1 190.9 223.6   25 28.7  74.2 105.5
 18  39 28.9 154.0 187.2   22 28.5  43.2  74.3
```

DiD solvers:

```
   | DiD-IF (inc.f.)    |  DiD-VP (var.p.)
 #p|#it| Tr | Tc  | Tot   #it| Tr | Tc  | Tot
  4  67 42.6 226.1 273.0   12 32.8 267.6 302.7
```

Sequential SEQ-M solver:

```
 #p|#it| Tr | Tc  | Tot
  1  67 33.3 961.2 998.2
```

A.2 DR results

```
DD solver:

    |   DD-AC (aggreg.)
  #p|#it| Tr  |  Tc  |  Tot
   6  50  65.4  854.3  923.2
   7  53  65.2  819.9  988.8
   8  56  67.7  851.4  922.5
   9  60  69.2  904.2  976.8
  12  65  71.8  903.7  980.8
  15  73  75.4 1020.8 1103.2
  18  79  77.8 1081.2 1162.7


DiD solver:

    |   DiD-IF (inc.f.)
  #p|#it| Tr  |  Tc  | Tot
   4  83 109.9  721.3  835.5
```

Bibliography

[1] Blaheta, R. et al.: *IGAS collection of benchmarks, solvers and test results.* Technical report COPERNICUS 940820 – DAM IG – 9705, Institute of Geonics Cz. Acad. Sci., Ostrava, 1997

[2] Blaheta, R., Jakl, O., Kohut, R., Kolcun, A.: *An application of large scale mathematical modelling in geomechanics.* In: Proc. Modern Mathematical Methods in Engineering. VŠB – Technical University, Ostrava, 1997.

[3] Blaheta, R., Jakl, O., Starý, J.: *PVM-implementation of the PCG method with Displacement Decomposition.* In: Bubak, M, Dongarra, J., Wasniewski, J. (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface. Lecture Notes in Computer Science, Vol. 1332, Springer-Verlag, Berlin, 1997, pp. 321–328

[4] Blaheta, R., Kohut, R.: *The PCG–1 iterative solver.* Technical report. Institute of Geonics Cz. Acad. Sci., Ostrava, 1995

[5] Geist, A. et al.: *PVM: Parallel Virtual Machine — Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, 1994

[6] *MPI: A Message-Passing Interface Standard.* University of Tennessee, 1995. See also `http://www.mpi-forum.org/docs/mpi-11.ps`

[7] Nečas, J., Hlaváček, I.: *Mathematical theory of elastic and elasto-plastic bodies: an introduction.* Elsevier Scientific Publishing Company, 1981

[8] Smith, B. F., Bjørstad, P. E., Gropp, W. D.: *Domain Decomposition. Parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, 1996

[9] Tichý, P. et al.: *Development of the Regional Metropolitan Supercomputing Centre in Ostrava — supporting research in the area of parallel algorithms.* Final report of the grant No. LB98273 (Cz. Ministry of the Education), VŠB – Technical University, Ostrava, 2001

[10] *Modelling and Simulation of complex Technical Problems*, project of the Program of the Information Society of the Thematic Program II of the National Research Program of the Czech Republic, project No. 1ET400300415, `http://www2.cs.cas.cz/mweb`