Final Report of the Software - Project WS 06 / SS 07

Felix Heimann

Supervisors: Prof. Fred A. Hamprecht, Dr. Olaf Ippisch, Prof. Kurt Roth, Prof. Peter Bastian

The 'Richards Equation' is a nonlinear partial differential equation occuring in the simulation of multi-phase fluids in porous media. Numeric solvers, realizing acceptable solutions without severe constraints to boundary conditions or initial data, are available. During his research on coupled transport in natural porous media, Olaf Ippisch established a C++ library providing a serial node-centered finite-volume solver on a structured grid in an environment that permits the application on transport problems in spatial heterogenous porous media with time-dependent boundary conditions. During this project, the original sequential solver was extended by integrating parallel algorithms and data-structures, provided by the DUNE [2] 'Distributed and Unified Numerics Environment' toolbox, enabling its parallel applicability.

Notes about the code package: This report comes along with a tar-archive containing the sequential and parallel version of the solver. All pathnames appearing in this report are relative to the root of this archive.
The sequential solver can be installed by calling `make` in `/seq/sources` and afterwards in `/seq/example2d`. However, the *SuperLU* library[5] must be installed. A test run can be started by calling
`/seq/example2d/example2d example2d.dat`
The parallel solver is wrapped in a dune module and needs different treatment. For installation, unpack all archives in `/par` and execute the following commands:

```
/par/dune-common-1.0beta1/bin/dunecontrol --only=dune-common all
/par/dune-common-1.0beta1/bin/dunecontrol --only=dune-istl all
/par/dune-common-1.0beta1/bin/dunecontrol --only=dune-richards all
/par/dune-common-1.0beta1/bin/dunecontrol --only=example-duneapp all
```

The execution of dunecontrol for the dune-common and dune-istl package will most likely fail at some point to report an error concerning doxygen and some html files... - this can be ignored. A test run can be started by calling `/par/example-duneapp/example/mpirun -np 1 example2d example2d.dat` assuming that MPICH1 is installed. Other MPI packages might need calls to initialization commands.

Notes about the notation and symbols: Solving partial differential equations numerically entails the idea of interpolating continuous functions at a given set of points which are usually understood to form some kind of grid. In the following, the interpolation of a given continuous function $f(\vec{x})$ at the grid point with index i will be denoted as $(f)_i$. Sometimes it is useful to interpret the set of these interpolated values as a vector of size N. Here N is the number of grid points. The vector corresponding to the interpolated values $(f)_i$ of $f(\vec{x})$ will be denoted as $\overrightarrow{(f)}$. Consequently, a continuous vector valued mapping $\vec{f}(\vec{x})$ will correspond to the interpolated values$(\vec{f})_i$. For such a mapping, vector representation of interpolated values is not useful.
Superscripts used in this document denote the value of the superscripted entity for a discrete point in time or for a particular nonlinear iteration. The current meaning will be obvious within the context.

1 Definition of the Task

The task was to modify an existing solver of the nonlinear PDE called "Richards Equation", permitting a parallel execution according to the *single programm multiple data* paradigm. This should be achieved by using convenient data structures of the DUNE library. In the following, an exact listing of this project's tasks is given together with the time schedule.

18.12.06	Start of Project
18.12.06 - 23.12.06	Analysis of the existing data-structures and algorithms
23.12.06 - 15.01.07	Implementation of the DUNE Algebraic Multi Grid Solver
15.01.07 - 05.02.07	Implementation of the DUNE YASP Grid
05.02.07 - 18.02.07	Realizing the parallel applicability - Determination of scalability and speedup
18.02.07	End of Project

2 Introduction of the Task

2.1 The Richards Equation

The Richards Equation is a nonlinear partial differential equation, which describes soil water flow in the so called degenerate multphase regime. The latter refers to a situation in which the water content is small and the air phase may be considered as arbitrarily mobile. Hence, the water flux may be described by the Buckingham-Darcy law:

$$\vec{j_w}(\vec{x}) = -\mathbf{K}_w(\theta_w(\vec{x}))[\nabla\psi_m(\vec{x}) - \rho_w(\vec{x})\vec{g}] \tag{1}$$

$\vec{j_w}$	water volume flux	$[ms^{-1}]$
$\mathbf{K}_w$	tensor of hydraulic conductivity	$[J^{-1}m^5s^{-1}]$
θ_w	water volume fraction	
ψ_m	matric potential	$[Jm^{-3} or Pa]$
ρ_m	mass density of water	$[kgm^{-3}]$

It postulates a linear relation between the flux and the pressure gradients of the liquid water phase. The latter has a component induced by gravity and a component containing the matric potential, which describes the pressure difference between the liquid water and air phase. The tensor of conductivity $\mathbf{K}_w$ depends highly nonlinear on the water volume fraction θ_w and material properties. Combining (1) with water volume conservation yields the Richards Equation with an additional term q_w, representing possible water sources or sinks in the porous media:

$$\partial_t\theta_w(\vec{x}) - \nabla\cdot[\mathbf{K_w}(\theta_w(\vec{x}))[\nabla\psi_m(\vec{x}) - \rho_w(\vec{x})\vec{g}]] = q_w(\vec{x}) \tag{2}$$

There are multiple parameterizations describing the relation between ψ_m and θ_m (e.g. Brooks-Corey, Mualem Van Genuchten, ...), each of them with characteristic restrictions concerning their applicability. However, they allow the evaluation of all parts of the PDE for a given ψ_m. The integrated form which represents the demand of volume conservation without severe assumptions regarding the smoothness of ψ_m is the weak formulation of the PDE problem:

$$\partial_t M - A - Q = 0 \tag{3}$$

$$M = \int_\Omega \theta_w(\psi_w(\vec{x}))dx^3 \tag{4}$$

$$A = \int_\Gamma [\mathbf{K}_w(\psi_w(\vec{x}))(\nabla\psi_w(\vec{x}) - \rho_w(\vec{x})g)]\cdot d\vec{n} \tag{5}$$

$$Q = \int_\Omega q_w(\vec{x})dx^3 \tag{6}$$

Here, Ω may be a arbitrary but constant subvolume with surface Γ of the considered domain.

2.2 Discretization

For M, A and Q we may choose Ω to be a cuboid of constant side length with baryzone at $\vec{x}$ ([1]. We may thus interpret them as functions of space justifying the notation $M(\vec{x})$, $A(\vec{x})$ and $Q(\vec{x})$. Choosing Ω to be equal to the uniform grid element leads to the finite volume discretization scheme for a rectangular stuctured grid with N elements. The integration is done seperately for each grid element by application of the mid-point rule using the value of $\psi_m(\vec{x})$ at the element's baryzone for the volume integrals. We thus compute an approximate solution $\vec{\psi_m} \in \mathbb{R}^N$ The surface integrals are handled similarly by interpolating at the face's baryzone. We thus achieve a spatial discretization: ([2]

$$(M)_i = (\theta_w)_i V_i \tag{7}$$

$$(A)_i = \sum_{j}^{Faces} [(\mathbf{K}_w)_{ij}((\nabla\psi_w)_{ij} - (\rho_w)_{ij} g)] A_{ij} \tag{8}$$

$$(Q)_i = (q_w)_i V_i \tag{9}$$

$$\vec{M}, \vec{A}, \vec{Q} \in \mathbb{R}^N \tag{10}$$

Notice, that the former arbitrary subvolume used in (3) was now chosen as each element's volume. To finally set up the N equations of the discrete problem, we furthermore need a time discretization, which is realized by a finite difference approach. Therefore the simulation time intervall $(0, T)$ is divided into discrete timesteps $t^0, ..., t^n, ..., t^{Steps-1}$. We may now estimate the time derivate using a finite difference scheme:

$$(\partial_t M)_i^n \approx \frac{(M)_i^{n+1} - (M)_i^n}{\Delta t^n} \tag{11}$$

Finally, we are able to formulate the discrete problem:

$$(d)_i := (M)_i^{n+1} + (M)_i^n - \Delta t^n \cdot ((A)_i^{n+1} + (Q)_i^{n+1}) = 0 \qquad [m^3] \tag{12}$$

Here, $\overrightarrow{(d)}$ is called the defect. This equation is of fundamental importance. Hence, some comments are due at this point:

- $\overrightarrow{(d)}$, $\overrightarrow{(M)}$ and $\overrightarrow{(A)}$ depend on the values of $\overrightarrow{(\psi_m)}$. We may thus interpret them as mappings of type $\mathbb{R}^N \to \mathbb{R}^N$ and write their dependencies like $\overrightarrow{(d)}(\overrightarrow{(\psi_m)})$, $\overrightarrow{(M)}(\overrightarrow{(\psi_m)})$ and $\overrightarrow{(A)}(\overrightarrow{(\psi_m)})$.
- Although $(M)_i$ depends on $(\psi_m)_i$ only, the calculation of $(A)_i$ requires the additional knowledge of the components of $\overrightarrow{(\psi_m)}$ with the indices of the adjacent elements of element i.
- In general, the calculation of hydraulic parameters at the element faces by averaging the corresponding values at the adjacent element baryzones may lead to unexpected (or at least unappreciated) behaviour. However, as these problems are of numeric and physical nature only, they will not receive any further attention.
- The given discretization of $\overrightarrow{(M)}$, $\overrightarrow{(A)}$ and $\overrightarrow{(Q)}$ are correct for a structured rectangular grid only. The chosen approach (interpolating the solution function at the element's baryzones and calculating the flux at the element faces) is called "'cell-centered"'. For more sophisticated grids, other approaches would become convenient as exhaustively discussed in [1].

[1] Notice, that this is a severely weaker claim than (3).

[2] Some notes on symbols and notation may be found at the beginning of this document.

2.3 The Sequential Solver

The sequential solver of the Richards Equation, written by Olaf Ippisch, consists of the following components:

- The **Backward Difference Timestepping Scheme** which controls time steps and is thus the fundament of the solver.(Ressources: `BDFClass` in files `seq/source/bdf.h, seq/source/bdf.cc`)
- The **Assembler**, responsible for realizing the finite volume discretization scheme by setting up the defect and assembling the jacobian matrix of $\overrightarrow{(d)}(\overrightarrow{(\psi_m)})$. (Ressources: `RichardsClass` in files `seq/source/richards.h, seq/source/richards.cc`)
- The **Interface for the calculation of the hydraulic parameters**, providing methods that may be flexibly replaced for each individual application of the solver. (Ressources: `RichardsParamClass` in files `seq/source/richards_param.h, seq/source/richards_param.cc`)
- The **Nonlinear Solver**, achieving the solution of $\overrightarrow{(d)}(\overrightarrow{(\psi_m)}) = 0$ by inexact Newton iterations computed by the **Linear Solver**. The latter is a stabilized conjugate gradient method with an algebraic multigrid preconditioner. These are quite sophisticated numerical methods and will not be discussed in the context of this report. (Ressources: `NewtonClass` in files `seq/source/newton.h, seq/source/newton.cc`)
- The **Grid** which provides geometric information needed by the assembler. (Ressources: `GridClass` in files `seq/source/grid.h, seq/source/grid.cc`)

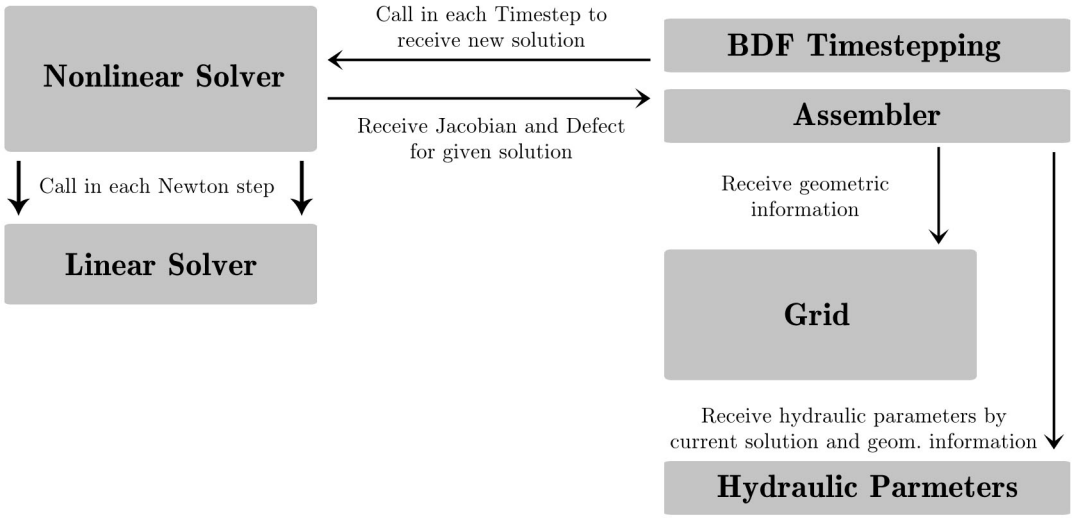

Figure 1: Components of the Sequential Solver

The following pattern gives a more precise idea about the algorithm's progression: (It is however, not an algorithm and differs slightly from the real implementation where it would obscure the ideas)

1. The **BDF Timestepping** object calls the **Nonlinear Solver** to progress the timestep for a given or initial solution.

2. The **Nonlinear Solver** calls the **Assembler** to calculate the jacobian matrix $\mathbf{J}(\overrightarrow{(\psi_m)}^{old})$ of $\vec{d}(\overrightarrow{(\psi_m)}^{old})$ and the defect itself.

3. The **Assembler** calculates the defect according to (12). Therefore it retrieves geometric information from the **Grid** and physical parameters by the **Hydraulic Parameter Interface**. Afterwards, it sets up the jacobian by numerical differentiation of $\overrightarrow{(d)}(\overrightarrow{(\psi_m)})$

4. The **Nonlinear Solver** calls the **Linear Solver** to solve $\mathbf{J}(\overrightarrow{(\psi_m)}^{old})[\overrightarrow{(\psi_m)}^{old} - \overrightarrow{(\psi_m)}^{new}] = \overrightarrow{(d)}(\overrightarrow{(\psi_m)}^{old})$ and thus calculate a correction for $\overrightarrow{(\psi_m)}^{old}$

5. A line search is done to optimize the correction (see [4]). If desired convergence is not yet reached, it sets $\overrightarrow{(\psi_m)}^{old} := \overrightarrow{(\psi_m)}^{new}$ and proceeds with 2.

6. If desired convergence is reached, the **Nonlinear Solver** passes the solution to the **BDF Timestepping** object.

The solver is implemented as a static C++ library. Its usage in a C++ program is realized by instanciating an `BDFClass` object and propagating the timestep by calling the member functions `TimeInit`, `TimePreProcess` and `TimeStep` as demonstrated in the example application in the file `seq/example2d/example2d.cc`. An appropriate use of the interface realized by the `RichardsParamClass` is demonstrated in the files `seq/example2d/example2d_param.cc` and `seq/example2d/example2d_param.h`.

2.4 DUNE

"The Distributed and Unified Numerics Environment is a modular toolbox for solving partial differential equations with grid-based methods. Using C++ techniques DUNE allows to use very different implementations of the same concept (i.e. grids, solvers, ...) using a common interface with a very low overhead." (cited from [2]).
DUNE provides the four packages *dune-common*, *dune-grid*, *dune-disc* and *dune-istl* of which the latter three depend on *dune-common* only:

- *dune-common*: Definition of general data-types (vectors, matrices, ...), the DUNE environment (exception handling, ...) and general communication.
- *dune-disc*: Tools for realizing discretization schemes as well as complete assemblers for common PDEs.
- *dune-grid*: A number of highly flexible sequential and parallel grids and reference elements.
- *dune-istl*: Tools for setting up and solving linear equations systems either sequential or parallel.

When solving a given PDE, there is usually a need for all of these packages. However, for this project it appeared most efficient to replace only the grid and linear algebra data structures of the given sequential solver and modify its other components to prepare them for their parallel application.
The *dune-common* package offers methods for general communication which are based on the MPI standard. Hence, MPI was chosen to constitute the communication platform for the parallel solver. As the discretization scheme depends on a structured and rectangular grid, the YASP Grid (Yet Another Structured Parallel Grid) of the *dune-grid* package appeared to be a convenient choice for substitution.
The *dune-istl* package contains implementations of the stabilized conjugate gradient solver and

algebraic multigrid preconditioner which may be combined with a corresponding communication object implementing an overlapping schwarz algorithm for the parallel solution of the global linear equation system. The methods provided by the dune-istl package depend on information about the parallel distribution of data which is automatically established by the YASP Grid. The design of DUNE naturally allows an easy transfer of this parallelization information from the grid to the solver. Hence, it was planned that the final parallel solver would, in the end, satisfy the following diagram:

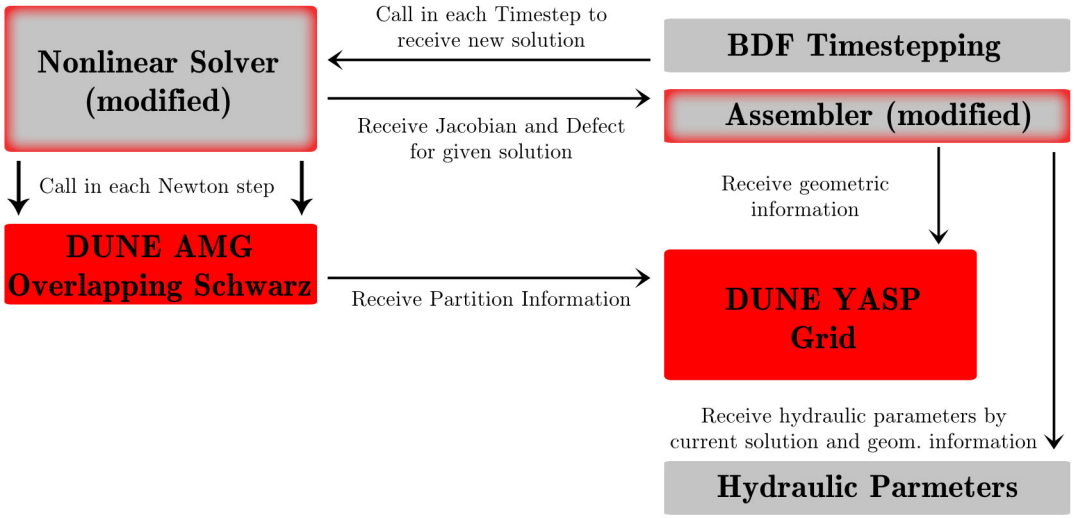

Figure 2: Components of the Parallel Solver (as planned)

2.5 Ideas for the Parallelization

The general idea for a parallel computation of the approximate solution function may be described as follows:

1. A partition of the global grid with overlapping local domains has to be set up. Therefore three different partition types are used: Interior, Copy and Foreign. An example partition for a 5x5 is shown in Figure 3.

2. Each processor assembles the rows of the jacobian matrix of $\overrightarrow{(d)}(\overrightarrow{(\psi_m)})$ that correspond to the indices of his local domain and their respective components of $\overrightarrow{(d)}(\overrightarrow{(\psi_m)})$. However, the matrix and defect are stored according to the local element indices resulting in a linear equation system which may be solved locally. For a grid with 3x3 elements as shown in Figure 7, the sequential solver would set up a 9x9 jacobian matrix:[3]

$$\mathbf{J}(\vec{\psi_m}) = \begin{pmatrix} j_{00} & j_{01} & 0 & j_{03} & 0 & 0 & 0 & 0 & 0 \\ j_{10} & j_{11} & j_{12} & 0 & j_{14} & 0 & 0 & 0 & 0 \\ 0 & j_{21} & j_{22} & 0 & 0 & j_{25} & 0 & 0 & 0 \\ j_{30} & 0 & 0 & j_{33} & j_{34} & 0 & j_{36} & 0 & 0 \\ 0 & j_{41} & 0 & j_{43} & j_{44} & j_{45} & 0 & j_{47} & 0 \\ 0 & 0 & j_{52} & 0 & j_{54} & j_{55} & 0 & 0 & j_{58} \\ 0 & 0 & 0 & j_{63} & 0 & 0 & j_{66} & j_{67} & 0 \\ 0 & 0 & 0 & 0 & j_{74} & 0 & j_{76} & j_{77} & j_{78} \\ 0 & 0 & 0 & 0 & 0 & j_{85} & 0 & j_{87} & j_{88} \end{pmatrix} \tag{13}$$

[3]Due to the numerical differentiation scheme, only components with index pairs of adjacent elements can be nonzero.

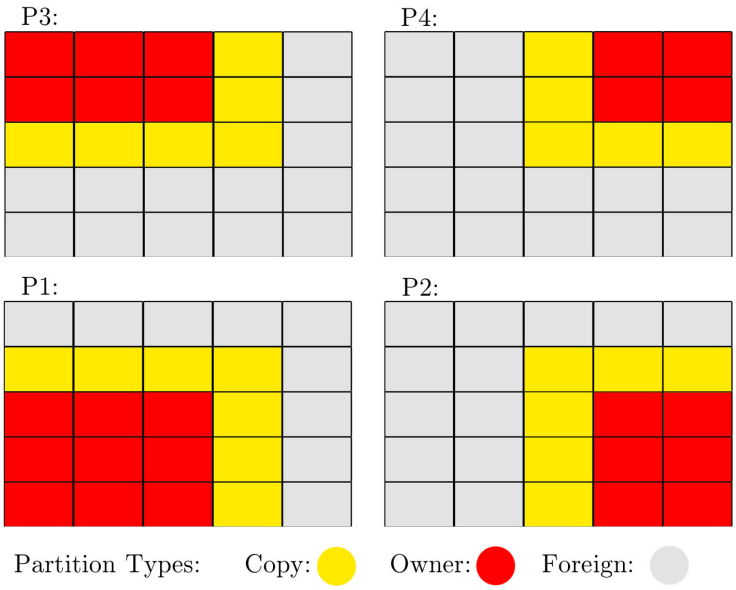

Figure 3: A possible partition for a 5x5 grid. Notice that every element has partition type Interior for one and only one processor.

6	7	8
3	4	5
0	1	2

Figure 4: A 3x3 grid with the respective global indices

With the partition shown in Figure 5, the local jacobian matrices (for P1 and P4) would appear as shown in (14)[4]. Notice, that rows corresponding to element indices with partition type Copy are nonzero only for the diagonal element which is set to one. This conserves the corresponding defect entry.

$$\mathbf{J_1}(\overrightarrow{(\psi_m)}) = \begin{pmatrix} j_{00} & j_{01} & 0 & j_{03} & 0 & 0 & 0 & 0 & 0 \\ j_{10} & j_{11} & j_{12} & 0 & j_{14} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ j_{30} & 0 & 0 & j_{33} & j_{34} & 0 & j_{36} & 0 & 0 \\ 0 & j_{41} & 0 & j_{43} & j_{44} & j_{45} & 0 & j_{47} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & j_{63} & 0 & 0 & j_{66} & j_{67} & 0 \\ 0 & 0 & 0 & 0 & j_{74} & 0 & j_{76} & j_{77} & j_{78} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{J_4}(\overrightarrow{(\psi_m)}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & j_{85} & j_{87} & j_{88} \end{pmatrix} \tag{14}$$

3. Each processor computes the solution of his local linear equation system occuring in each Newton iteration. The values of $\overrightarrow{(\psi_m)}$ at the overlapping elements are communicated such that the value of the processor which posesses an element with partition type Interior sends the value to all processors which possess the element with partition type Copy. This is called

[4]Since the local jacobian of processor 1 has the same dimensions as the global jacobian, no speedup could be gained for this example.

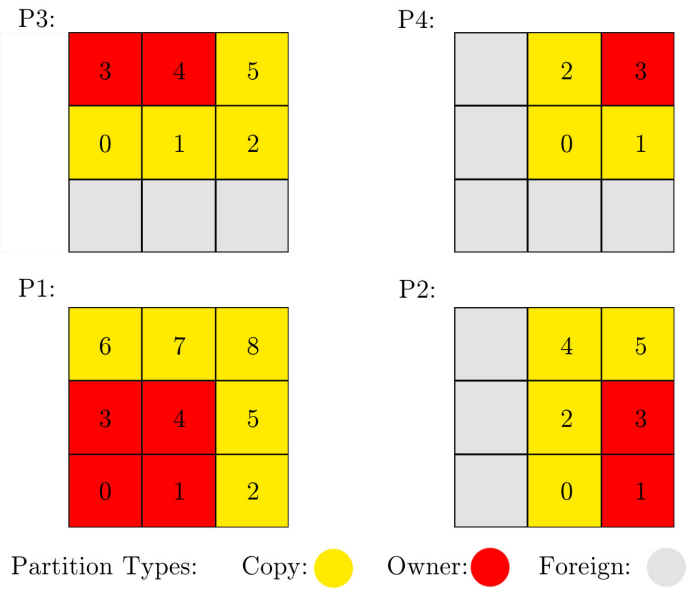

Figure 5: A partition for the 3x3 grid with respectif local indices

Overlapping Schwarz Algorithm. Solution of the local equation system and communication is done until global convergence is reached.

4. The final values of $\overrightarrow{(\psi_m)}$ are communicated and the Newton scheme proceeds.

Obviously, the major work of parallelization is to achieve the parallel solution of the linear equation system and to set up and administrate the grid partition. As both tasks can be done by DUNE methods and datastructures, the original task of this project was the integration of the DUNE components in the old solver's environment.

3 Description of the Project's Progress

3.1 The Linear Algebra (dune-istl)

The integration of the linear algebra methods contained in the *dune-istl* package was rather simple, thanks to the fact that the corresponding parts in Ippisch's original code had a highly modular design. A listing of the relevant implemented components follows:

- All vector and matrix data-structures in the context of the Linear Solver were homogenized to be of data types `Dune::BlockVector` and `Dune::BCRSMatrix`
- Original error handling was replaced by usage of `Dune::Exception`
- The `Dune::BiCGSTABSolver` was implemented using a `Dune::Amg::AMG` algebraic multigrid preconditioner with a `Dune::SeqSOR` coarse grid preconditioner.

Surprisingly, the simple substitution of the linear algebraic methods resulted in a speedup of the total solving process of 10% to 15%. Notice, that the original methods used, in principal, the same numeric algorithms as their DUNE substitutes.

3.2 The Grid (First Approach: dune-grid)

Although the integration of the YASP Grid constituted no principal problem, it soon became obvious that the sequence of calls from the Assembler to the YASP Grid would influence the run-time of the algorithm in an unfavourable way. While the original grid computed geometric information about the current element and all of its neighbours along with the incrementation of the element iterator, the Yasp Grid would not compute any geometric information until it is needed. This conflicts however with the behaviour of the Assembler calling a vast amount of face iterations to retrieve geometric informations about adjacent elements. To gain certainty, the implementation was completed:

- The `Dune::YaspGrid` was embedded in a wrapper class to synchronize extern calls with the convenient `YASPGrid` methods.
- Access to the geometric information of elements described by the `Dune::Entity` class was provided by the corresponding `Dune::Geometry` member. The `Dune::Entity` typed elements were handled by using appropriate iterators of type `Dune::LeafIterator` (for iterating the grid elements) and `Dune::IntersectionIterator` (for iterating the faces of a given element and access adjacent elements).
- The assignment of user data to `Dune::Entity` objects was established by the `Dune::LeafMultipleCodimMultipleGeomTypeMapper`.
- To achieve a better run-time, many calls of grid methods in the Assembler object were modified by instanciating iterators directly in the Assembler object, bypassing the grid wrapper class.

Before any attempts of parallelization were made, the applied run-time tests showed that integrating the YASPGrid had resulted in an overall slowdown of the solver between 600% and 500% depending on how consequently the wrapper class methods were bypassed. This slowdown is most likely caused by the, in principal, unnecessary frequent use of the `LeafIntersectionIterator` which allows access to adjacent elements at high computational costs. However, no further analysis of the run-time and no attempt to parallelization was done as the reorganization of the Assembler object aiming towards a significant lesser use of the LeafIntersectionIterator would have posed a task of great effort in time. A more economical way to establish the parallel applicability of the orignal solver appeared to be the modification of the original grid, endowing it with the additional functionality needed by a selfcontained parallel grid.

3.3 The Grid (Second Approach: Parallelization of the Original Grid)

The transformation of the original grid to a parallel grid was done by the below steps:

- The grid object was renamed from `GridClass` to `ParallelGridClass` (A step of crucial psychological importance).
- A member function with the ability to calculate and set up the local partition depending on the current processor's index (as shown in Figure 3) was added to the class. It also creates an array containing information about the partition types of the local elements. The algorithm used for partitioning is simple, however it allows an arbitrary number of processors and tries to optimize the distribution considering different grid-point densities in different directions.
- As the partition information which is needed by the dune-istl functions can no longer be provided by the `YASPGrid`, the corresponding data structure `Dune::ParallelIndexSet` has to be established manually. This was combined with the process of partitioning.

- The grid functions responsible for iterating the grid have to be altered such that they no longer consider nonlocal elements and handle both local and global element indices.
- A `Dune::CollectiveCommunication` object of the dune-common package was used to implement an interface for gathering the whole global solution vector ensuring the reusability of the original interfaces for data-output.

In a first attempt, the new grid was implemented and tested in a sequential program run. As it was able to compute the correct solution with no measureable slowdown, the last changes necessary for the parallel solver were made:

- The jacobian matrix was embedded in a `Dune::OverlappingSchwarzOperator` object which is passed to the `Dune::AMG::Amg` preconditioner. Combining the latter with the parallel `Dune::BlockPreconditioner` ensures the correct application of the Overlapping Schwarz algorithm.
- Few changes to the Assembler were made to ensure the correct assembly of rows in the jacobian corresponding to elements with partition type Copy.
- The calculation of the global value for $\vec{d}(\vec{\psi_m})$ required additional communication in the Nonlinear Solver.
- Output had to be restricted to a single processor.

Figure 6 illustrates the final setup of the solver.

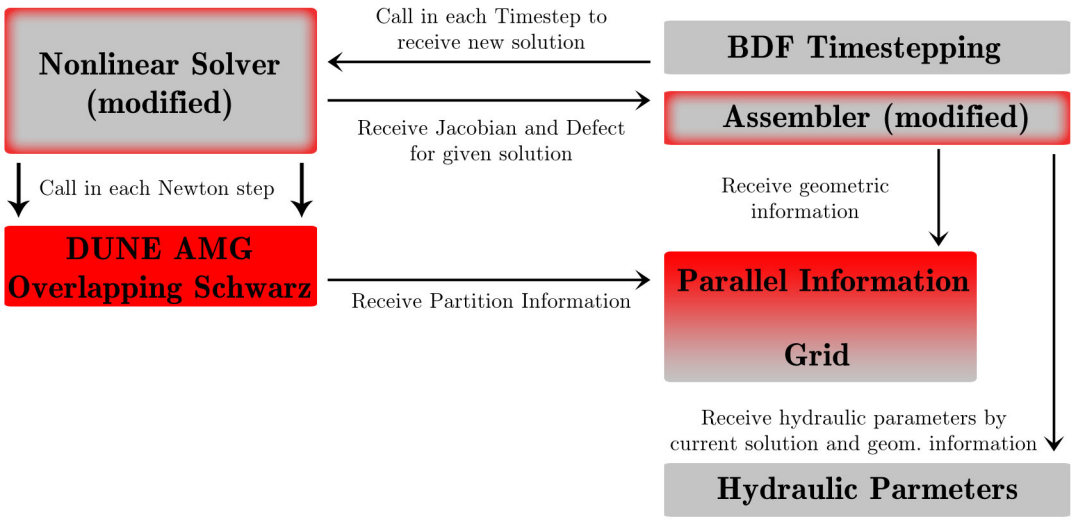

Figure 6: The final scheme of the parallel solver

4 Testing the Parallel Solver

The solver was tested with a simulation of a 2D infiltration of initially unsaturated soil ($0.75x0.4m$) with a heterogenous structure consisting of three different materials. This was realized by a setup of initially constant ψ_m with "no flow" Neumann boundary conditions at the side boundarys and Dirichlet conditions at the top and bottom. The following sequence of images shows the infiltration

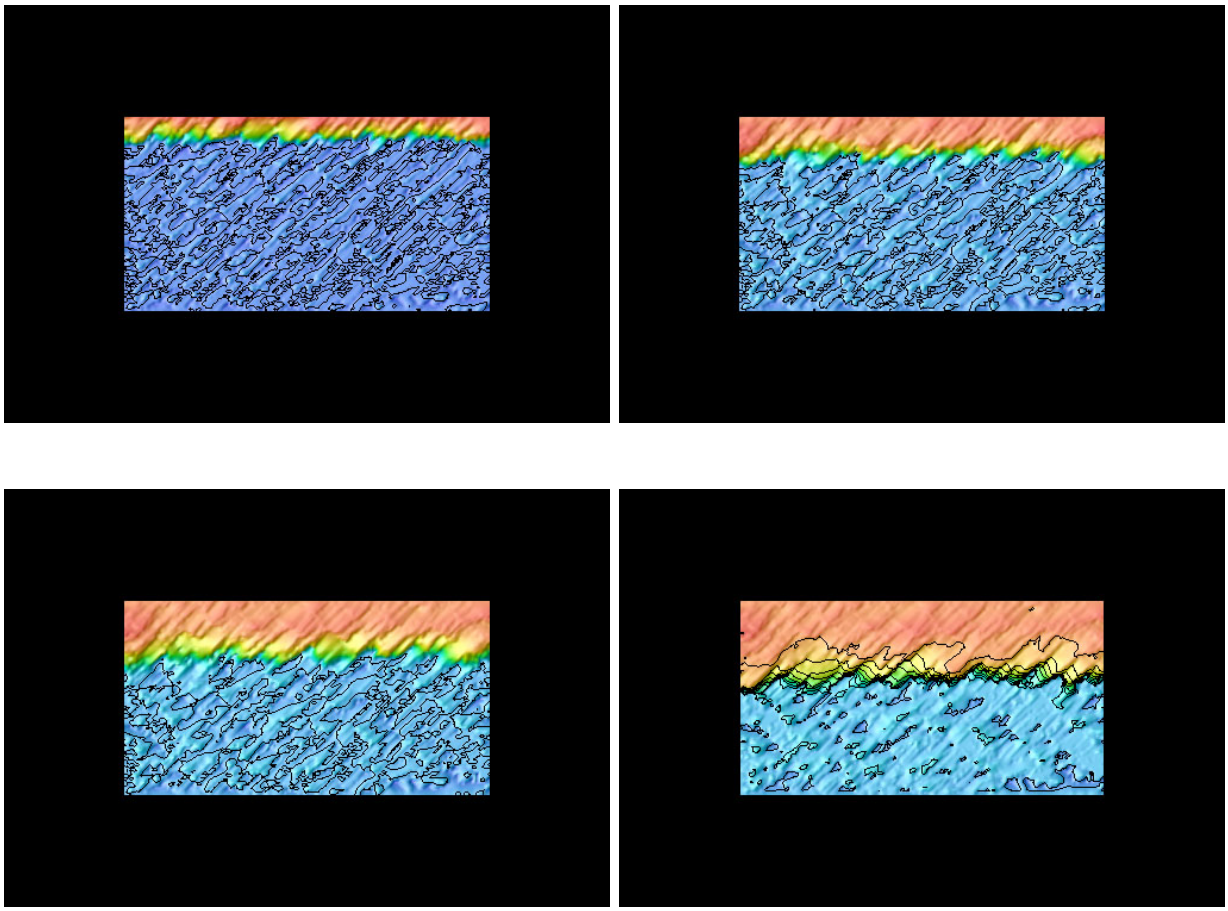

Figure 7: The Infiltration Front: The matric potential ψ_m for the infiltration front ranges from about -1060 Pa (red) to -2203 Pa (blue)

front entering the soil for time intervalls of 100 , 300, 500 and 800 seconds. The solutions of the parallel and the sequential solver do not differ optically. This example simulation was executed for different combinations of processor number and grid points to apply two different kind of scalability tests:

4.1 Static Problem - Variable Number of Processors

The run-time differs from the optimum only when the local grid sizes get too small (Figure 8). Considering that no significant difference in sequential run-time between the solver with parallel grid and the solver with the original grid could be detected, this result is quite satisfying.

The parallelization appears to not influence the stability of the nonlinear convergence as the number of linear and nonlinear iterations shows only weak fluctuations. (Figure 9)

4.2 Constant Number of Gridpoints per Processor

For a optimal parallel algorithm, increasing the total load while keeping the load per processor constant, should result in no increase in run-time. However, the given algorithm applies the Newton - in principle - to the global grid. The number of these non-linear iterations are not independent of the global grid size. Hence, we have to expect a slow-down proportional the increase in non-linear iterations. Figure 10 shows an increase in run-time far above this optimum as can be seen by comparison with Figure 11. An additional test was applied which halfened the time-step every time the

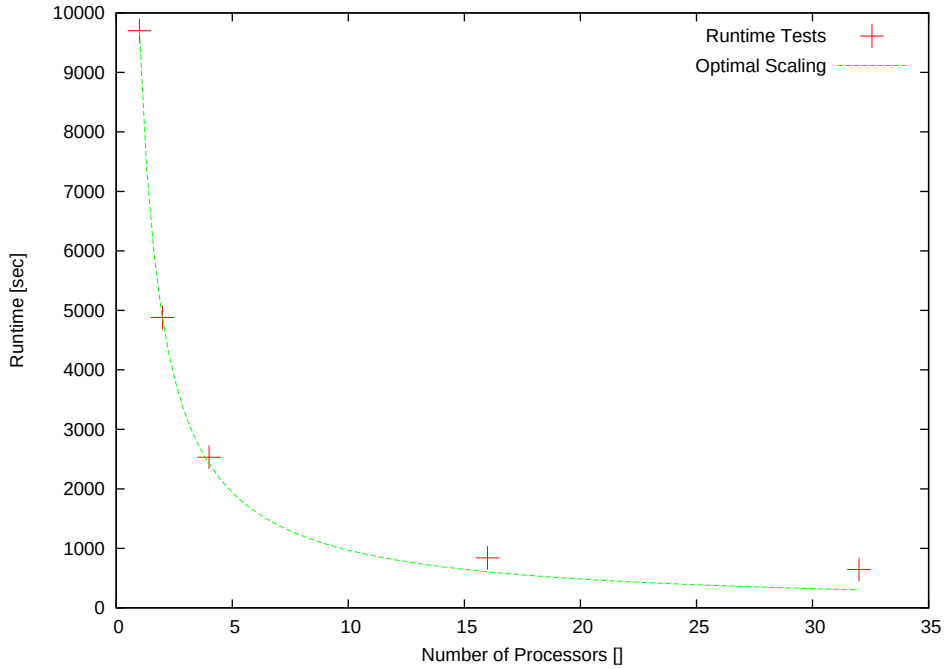

Figure 8: Runtime measurements for "Static Problem - Variable Number of Processors" test

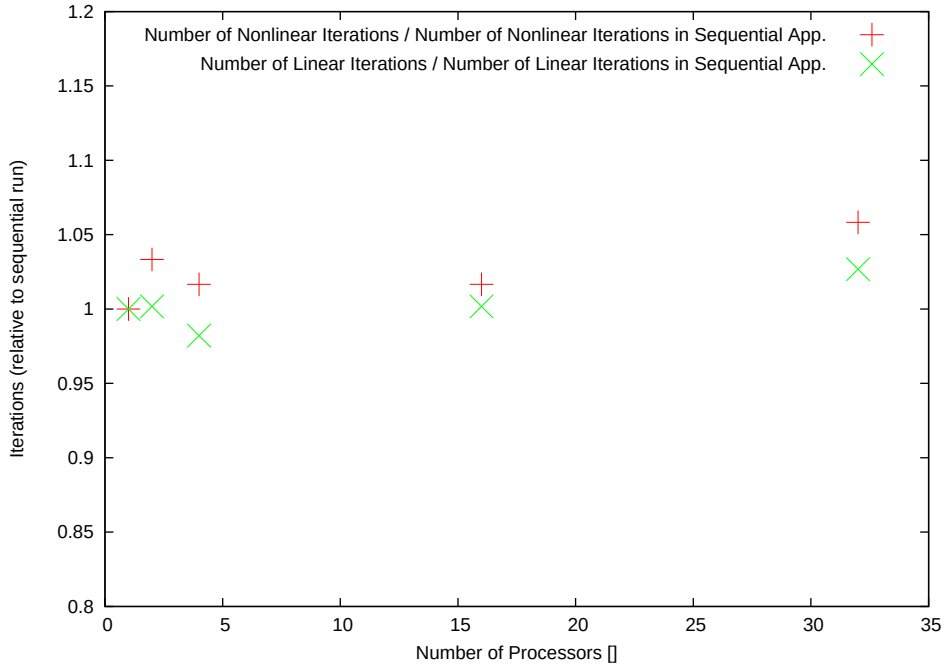

Figure 9: Relative changes in the number of linear and nonlinear iterations

number of grid points in each dimension were doubled. As in the previous test, the relative increase in linear iteration outruns the relative increase in nonlinear iterations (Figure 12 and 13).

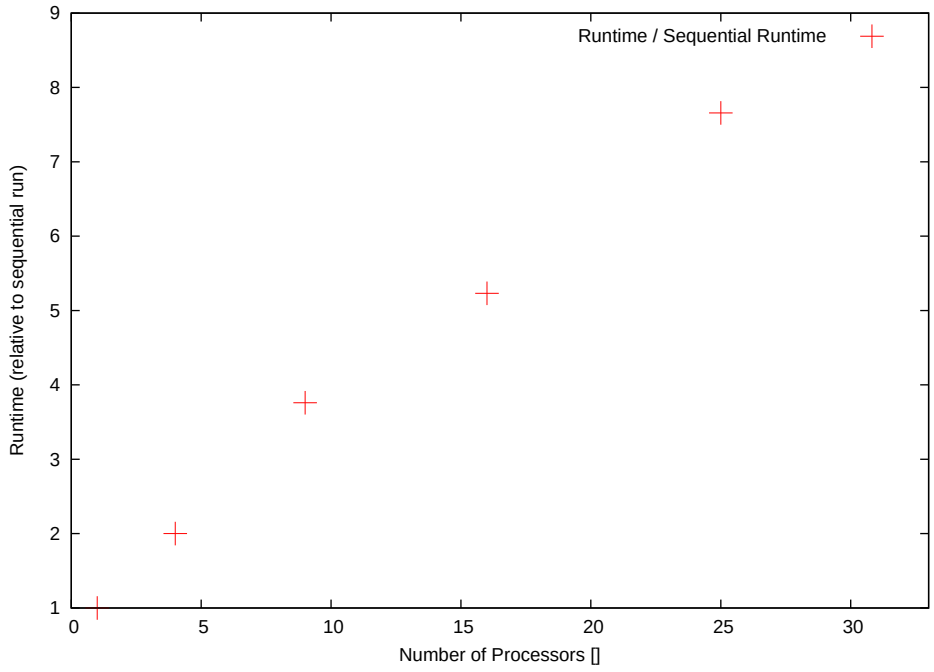

Figure 10:
Relative run-time measurements for "Constant Number of Gridpoints per Processor" test

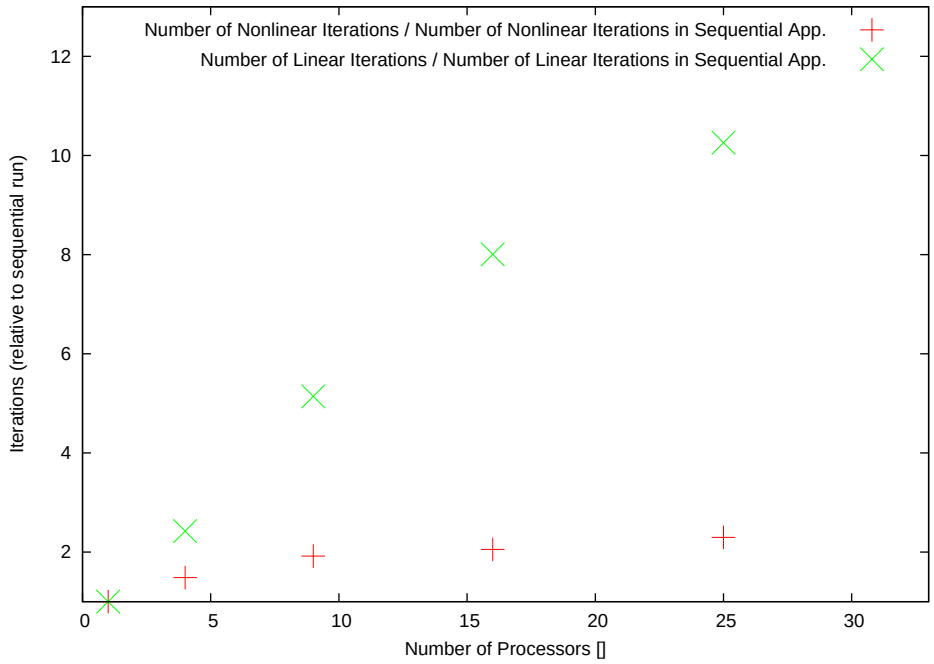

Figure 11:
Relative changes in the number of linear and nonlinear iterations for "Constant Number of Gridpoints per Processor" test

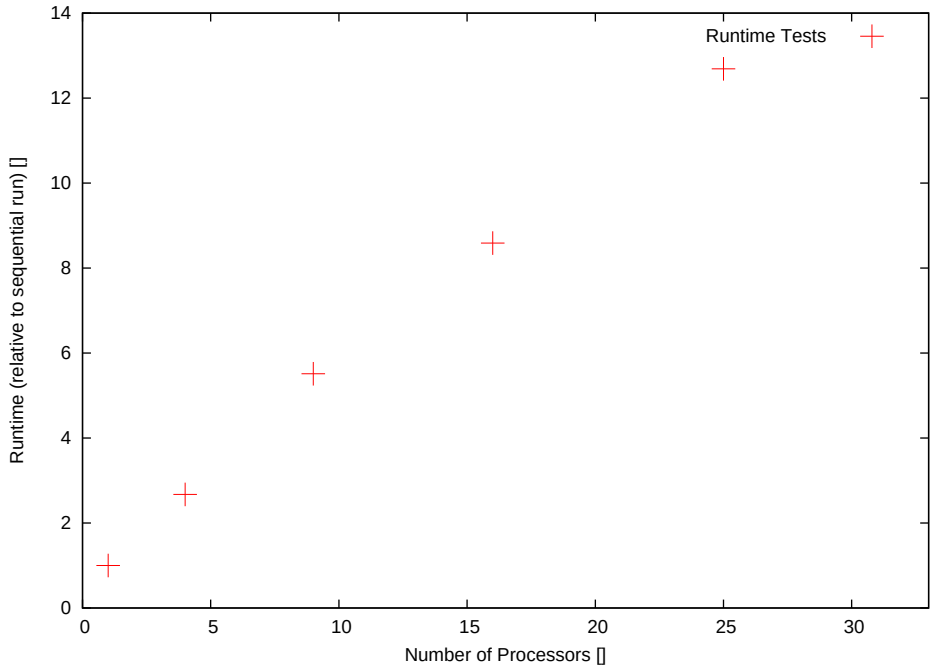

Figure 12:
Relative run-time measurements for "Constant Number of Gridpoints per Processor" test with variable time steps

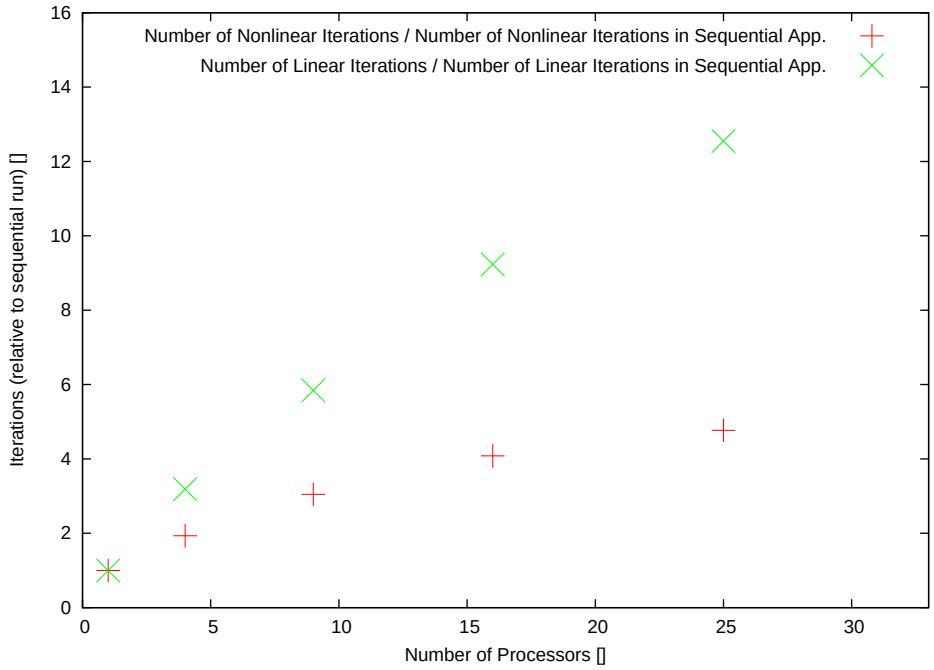

Figure 13:
Relative changes in the number of linear and nonlinear iterations for "Constant Number of Gridpoints per Processor" test

References

[1] Bastian, P. 1999. Numerical Computation of Multiphase Flows in Porous Media. Habilitationsschrift. Christian-Ablrechts-Unicersität, Kiel.

[2] Dune Homepage: http://www.dune-project.org/

[3] Ippisch, O. Coupled Transport in Natural Porous Media. Dissertationsschrift (2001). Ruprecht-Karls Universität, Heidelberg.

[4] Nocedal J., Wright S., Numerical Optimization, Springer Verlag (2006)

[5] Homepage: http://crd.lbl.gov/ xiaoye/SuperLU/