INTRODUCING THE VOLSUNG GEOTHERMAL SIMULATOR: BENCHMARKING AND PERFORMANCE

Peter Franz[1], Jonathon Clearwater[1] and John Burnell[2]

[1]Flow State Solutions Ltd, 67 Hamon Place, Rotorua, New Zealand

[2]GNS Science, Private Bag 30-368, Lower Hutt 5040, New Zealand

Peter.Franz@FlowStateSolutions.co.nz

Keywords: Volsung, Simulator, TOUGH2, Benchmarking, Performance, Parallel Computing

ABSTRACT

In this paper we introduce the Volsung Geothermal Reservoir Simulator software package. At its core is a very fast numerical reservoir simulator based on the finite volume method (FVM). Its computational backbone is based on a hybrid method, where most of the numerically intensive calculations are performed on a CPU and only the memory bandwidth-limited linear solve operations are outsourced to an inexpensive, consumer-grade graphical processing unit (GPU). This approach enables vast performance enhancements when compared with traditional CPU-based systems while avoiding the hardware and software complexity of distributed memory architectures found in other high-performance-computing solutions.

We have validated Volsung versus test models from the Stanford 1980 Geothermal Model Intercomparison Study. In addition, we created simple test models to compare Volsung to results from the TOUGH2 simulator for selected problems of interest. Tests include a dual-porosity simulation problem which uses the Multiple Interacting Nested Continua (MINC) formulation, simulations using different equations of state (water, non-condensible gas and salt) and simulation runtime comparison. In all cases Volsung successfully reproduced the expected model results and achieved large speedups versus TOUGH2 for production size models.

1. INTRODUCTION

For years there have been few numerical simulators available on the free market for creating and running geothermal reservoir simulations. The TOUGH2 simulator (Pruess et al., 1999) has been the de-facto standard tool for more than two decades; only recently its successor TOUGH3 (Jung et al. 2016) has been released. Further the Waiwera (O'Sullivan et al., 2019) simulator is due for public release shortly.

We present here the new flow simulation package Volsung, which consists of three simulators for running fully coupled reservoir, wellbore and surface network models. Further, it contains a full graphical user interface for setting up models in 3D, a standalone wellbore simulator, python scripts for advanced data analysis and has features for off-loading computations to remote cloud servers.

As with any other new tool we need to verify and validate it. Verification is hard since only very few and simple two-phase models exist for which the analytical solution is known. In the first part of this paper we hence focus on comparing Volsung's output to TOUGH2 output for some select public models from the Stanford's 1980 Geothermal Model Intercomparison Study (Molloy, 1981). Further, for testing different equations of state and MINC, we use some test models from the TOUGH2 User's Guide (Pruess et al., 1999) or our own creations.

Since geothermal reservoir simulations using TOUGH2 often take hours to days to complete we are of course interested in Volsung's performance. The second part of this paper hence analyses the performance limits of the finite volume method (FVM) using different system architectures in order to investigate what speedups are technically feasible.

2. VALIDATION OF VOLSUNG

This section gives an overview of the validation test model runs comparing Volsung with TOUGH2. All models are available as examples in the Volsung package and where applicable contain the TOUGH2 files which can be used for testing other simulators[1].

Note when comparing results from Volsung to TOUGH2 we need to keep in mind that small differences are expected since Volsung uses the IAPWS-97IF steam tables while TOUGH2 uses IC67.

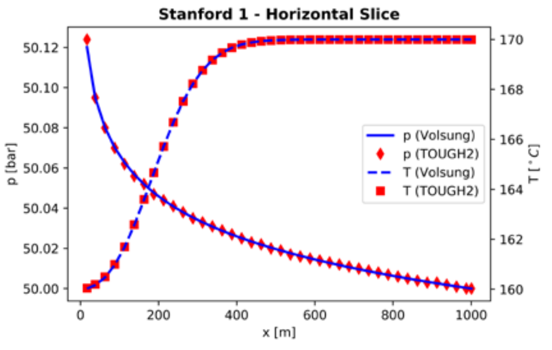
2.1. Stanford Problem 1

Stanford Problem 1 involves one-dimensional, radial, steady-state flow and unsteady heat transport in a single-phase liquid. The purpose is to test heat conduction and convection in a single-phase compressed water region. The results from the test are shown in Figure 1.

2.2. Stanford Problem 2

Stanford Problem 2 models radial flow to a line sink at the origin of a radial mesh and is divided into four sub problems. Case A remains single-phase liquid. Case B deals with two-phase conditions where both phases are mobile. In case C the fluid changes from compressed liquid to two-phase with flash-point propagation away from the well. The authors of the study mention that this case is hard to model numerically; indeed, the results from Volsung and TOUGH2 show some disagreement for this case for the blocks next to the sink. Results from problem 2 are shown in Figure 2. Croucher et.al.

[1] https://www.flowstatesolutions.co.nz/downloads



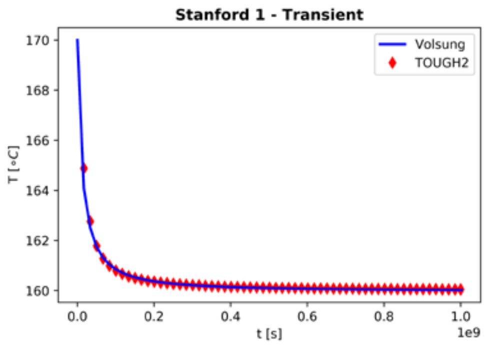


Figure 1: Comparison between Volsung and TOUGH2, Stanford Model Intercomparison Problem 1.

(2018) also investigated this problem and found that the simulators tested there deviated from the semi-analytical solution for this problem. While this does not explain the disagreement between Volsung and TOUGH2 here it demonstrates that this appears to be a hard problem and would merit further investigation for the different simulators.

2.3. Stanford Problem 3

The third problem from the Stanford code comparison study is a radial model with horizontal and vertical flow, including flow through a high permeability fracture and low permeability blocks. The problem definition is ambiguous and several participants in the code comparison study had different interpretations and hence different results.

To simplify the problem, we implemented a rectilinear grid with a high porosity, high permeability well block, a high permeability fracture and low permeability blocks above the model's fissure. In Figure 3 we show the pressure, gas saturation and flowing enthalpy transients of the well block.

2.4. Stanford Problem 4

Problem 4 is a 1km square reservoir with 20 single block layers each 100m thick. The initial reservoir temperature is prescribed and a flow rate of 100kg/s is extracted from the bottom layer for a 40 year period. Figure 4 shows the pressure and gas saturation profiles at the end of the production period as well as the transient of the production fluid enthalpy.

2.5. Stanford Problem 5

The fifth problem from the Stanford code comparison study is a 2D model covering a horizontal 300×200×100m^3 grid. The model is initialized with a temperature gradient as described in the model study. One block produces 5kg/s for

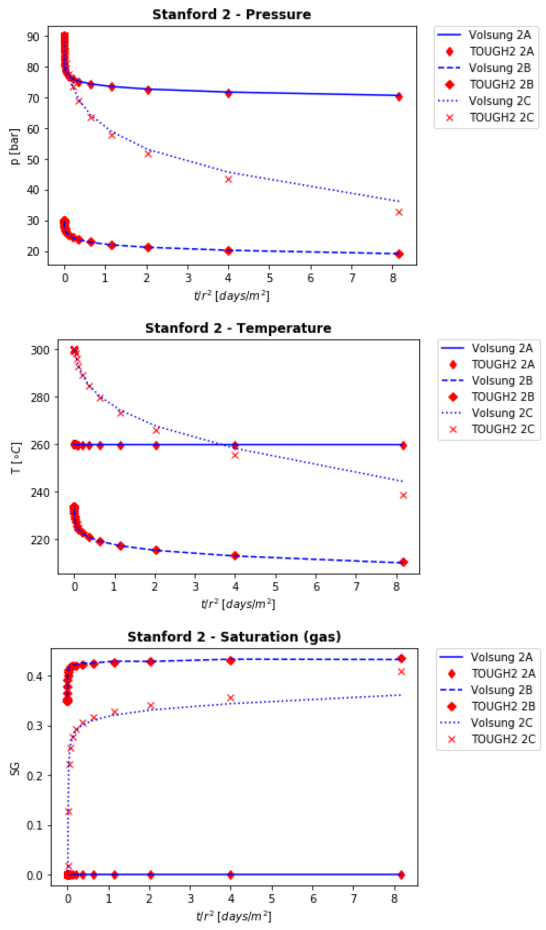
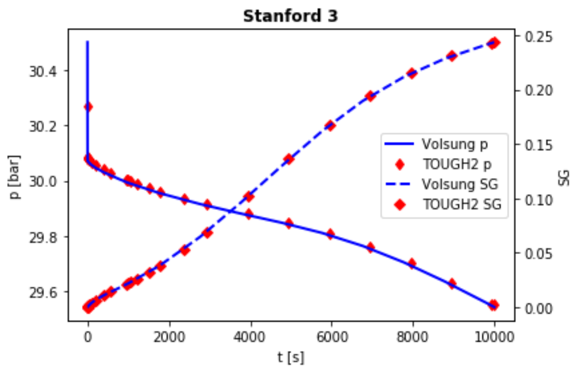


Figure 2: Comparison between Volsung and TOUGH2, Stanford Model Intercomparison Problem 2.



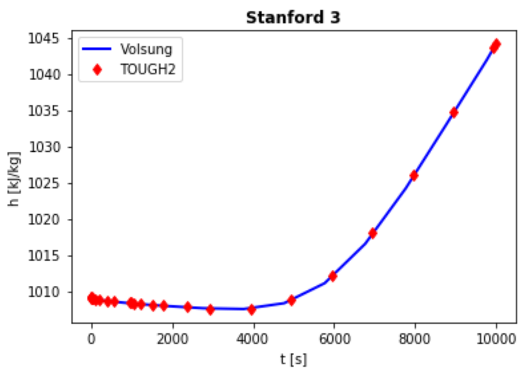
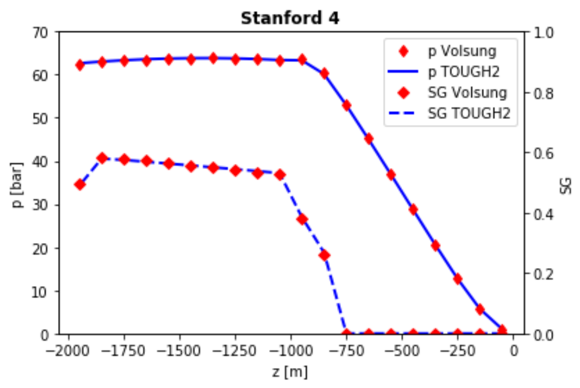


Figure 3: Comparison between Volsung and TOUGH2, Stanford Model Intercomparison Problem 3.



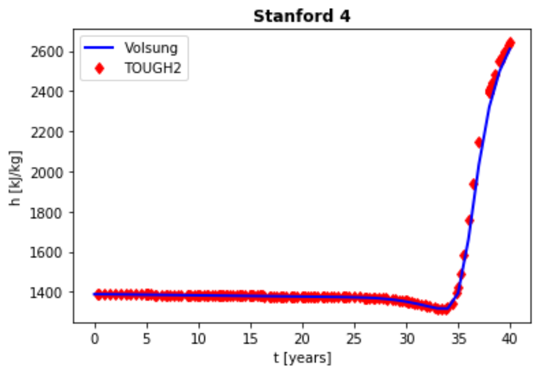


Figure 4: Comparison between Volsung and TOUGH2, Stanford Model Intercomparison Problem 4.

10 years. Figure 5 shows the pressure, gas saturation in the production block and produced enthalpy transients for the model.

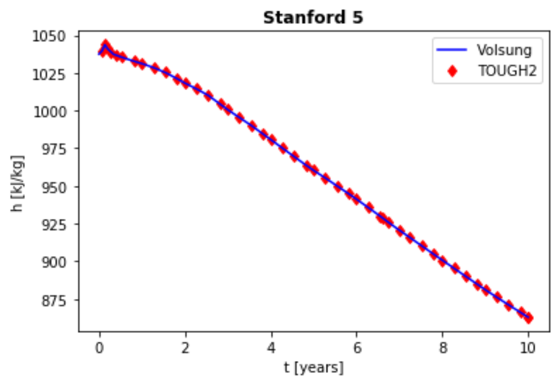
2.6. Stanford Problem 6

The sixth problem from the Stanford code comparison study is a three-dimensional model with single phase liquid at depth sitting under a two-phase zone with immobile steam. At the top of the model is a zone of colder single-phase water. Production is from below the two-phase zone at a rate of 1000 kg/s for the first 2 years, 2500 kg/s for the next 2 years, 4000 kg/s for the following 2 years and 6000 kg/s for the final 2 years for a proposed 8-year forecast.

In both Volsung and TOUGH2, and in the simulators used in the original study, the 6000 kg/s production rate caused catastrophic pressure decline in the production zone terminating the simulations. A comparison between production enthalpy, pressure and gas saturation in the well block is shown in Figure 6. The difference between Volsung and TOUGH2 at the final simulation time is that TOUGH2 fails to find a solution and halts while Volsung continues to generate output until near vacuum is achieved in the block.

2.7. MINC

To test Volsung's MINC capabilities (Pruess et al., 1999; Thunderhead, 1999) we created a simple 2D test model, 500×500m^2 with a single 100m thick layer and uniform 20m cell spacing. The model is set to 100bar and 304ºC initial conditions. A cold fluid injector with constant injection rate and injection enthalpy of 12.6kg/s and 196kJ/kg, respectively, is put close to the centre of the model. A producer is located



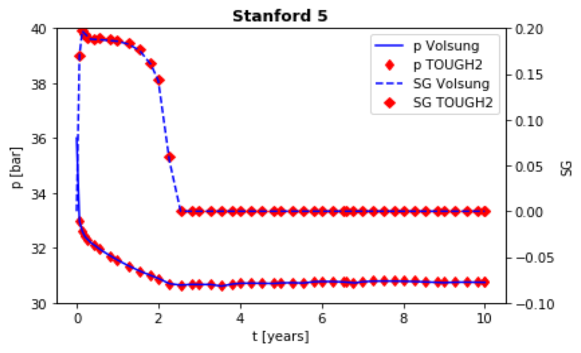


Figure 5: Comparison between Volsung and TOUGH2, Stanford Model Intercomparison Problem 5.

140m away from the injector towards the side of the model and set to produce 12.6kg/s, i.e. the same amount as injected. The model was run for 10 years.

We tested this model with two different fracture spacings, 300m and 50m, and using one fracture and two matrix layers with 2%, 4% and 94% volume fraction, respectively. The MINC mesh was generated internally by Volsung and TOUGH2 using these same basic parameters.

Figure 7 shows the transient behaviour of the model for the block containing the producer. The temperature in all 3 MINC layers and the enthalpy of the produced fluid is shown.

2.8. EOS with CO_2

Volsung has capabilities to use an equation of state (EOS) module which accounts for a non-condensible gas (NCG); currently CO_2 and air have been implemented. The same basic equations as in TOUGH2 are used; only some minor improvements to aid numerical stability have been made.

To test the thermodynamic behaviour versus a TOUGH2 model we wanted to focus on the basic thermodynamics rather than on transport phenomena. Hence we created a very simple model consisting of one block with a sink attached. The model is started in compressed liquid state but then traverses through two phase and finally single-phase gas state; it finally runs out of fluid and the simulation collapses at this stage. Figure 8 shows the comparison between Volsung and TOUGH2; both simulators show the same transient behaviour and agree well on the phase partitioning of CO_2.

2.9. EOS with Salt

Volsung features an EOS containing NaCl which is based on the method published by Battistelli et al. (1997); however, at

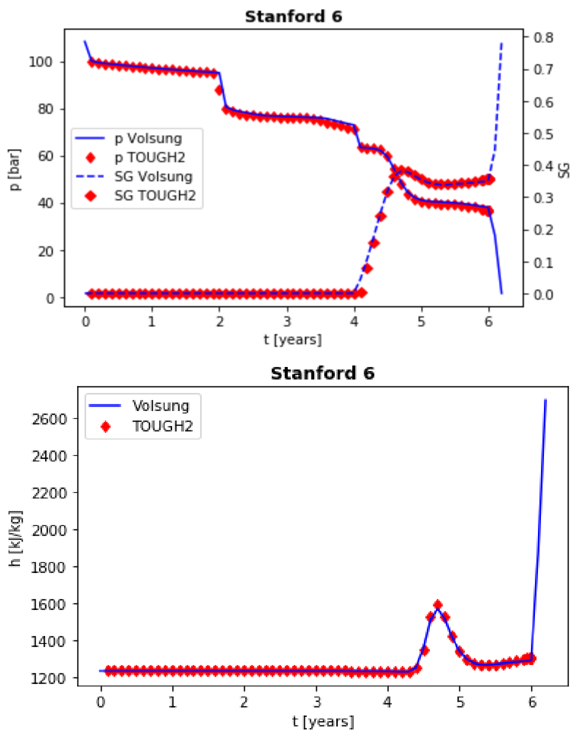


Figure 6: Comparison between Volsung and TOUGH2, Stanford Model Intercomparison Problem 6.

this stage it does not contain NCG yet. The general idea of having an EOS with salt but no NCG is to provide a method to support modelling of a salty tracer for models in which mixing or boiling occurs. It is planned to add NCG support in the near future.

Not having a NCG makes direct comparison to TOUGH2's EWASG hard since it is difficult to set up the model in both Volsung and TOUGH2 to match the same state. We created a model very similar to the RHBC model described in the TOUGH2 manual (Pruess et al., 1999), featuring vapour pressure lowering due to salinity, appearance of the solid halite phase and decreasing permeability due to reduction of the free pore space via a permeability modification function. Results for this model are shown in Figure 9; the agreement in all parameters except for a small temperature offset is very good but one needs to keep in mind that the TOUGH2 model still contains traces of CO_2. We will revisit this model once Volsung also supports NCG.

3. PERFORMANCE ANALYSIS

In the geothermal modelling community, the most accepted method for solving the transport equations for mass and heat is the finite volume method (FVM). All simulators mentioned here—TOUGH2, TOUGH3, Waiwera and Volsung—are based on this method. In brief, the method can be summarized as follows:

- Given an initial state at time t we want to calculate the state at time t+dt.
- We calculate a residual vector R(t+dt, X) by considering mass and heat conservation equations and using a trial set of primary variables X.
- We then calculate the Jacobian matrix J by estimating the derivatives of R with respect to the primary variables. This is usually done using finite differencing.
- The solution for the state at t+dt is improved using Newton's method; for this we need to solve the linear system J * (-dX) = R.
- These Newton iterations are repeated until a solution fits the convergence criteria. The accepted solution is the state at t+dt.

The computational time intensive tasks in this method are:

1. Repeatedly calculating the residual vector R using different primary variables. Since the Jacobian matrix is generally determined using finite differencing, we need to calculate the residual N_{PV}+1 times per Newton iteration. Here N_{PV} is the number of primary variables per element and can be expressed as $N_{PV}=N_C+1$ with

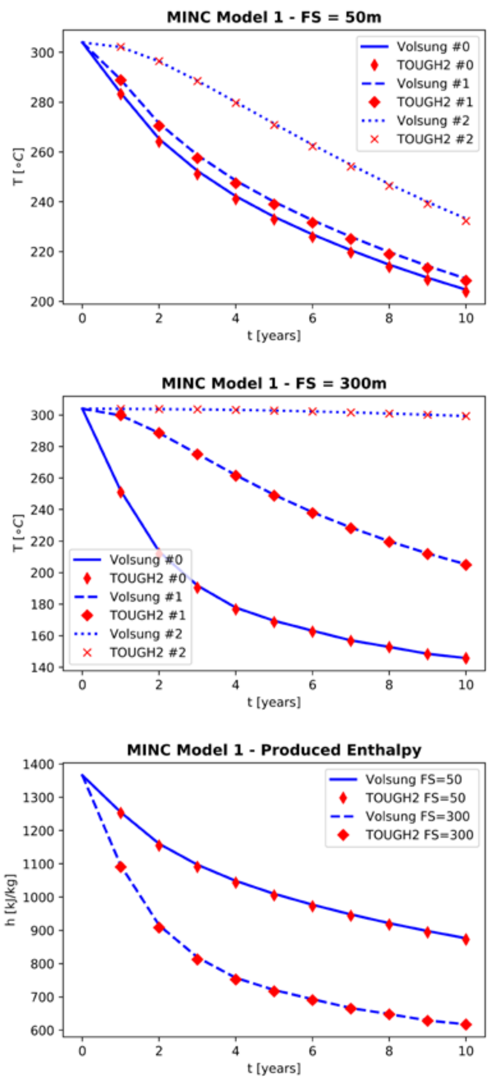


Figure 7: Comparison between Volsung and TOUGH2, MINC Problem with 50m and 300m fracture spacing (FS). #0 indicates fracture, #1 and #2 indicate matrix layers.

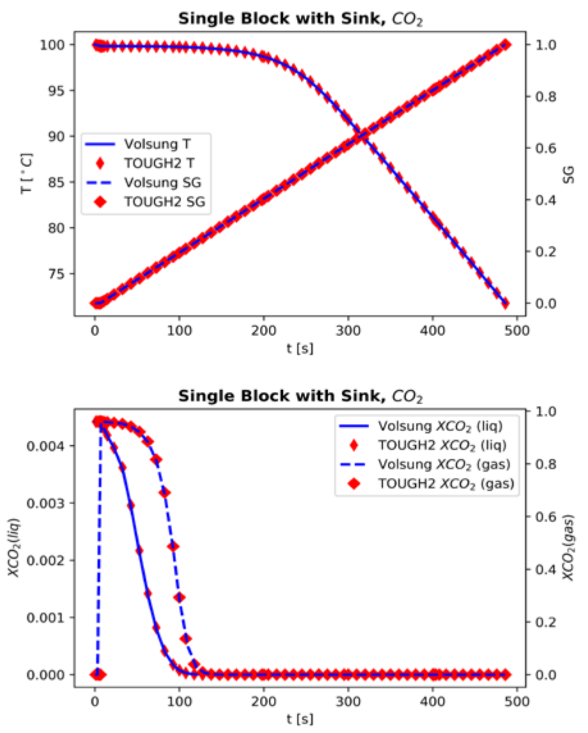


Figure 8: Comparison between Volsung and TOUGH2, single block with sink containing CO_2.

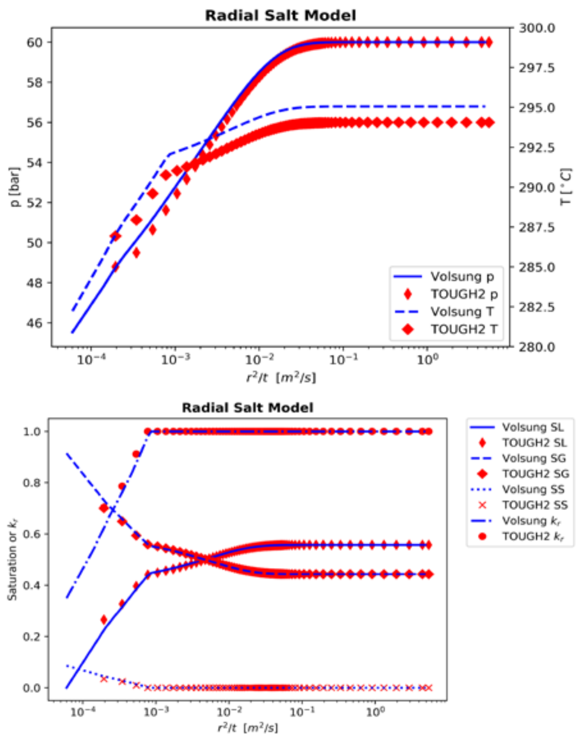


Figure 9: Comparison between Volsung and TOUGH2, Model containing NaCl.

N_C being the number of components (e.g. H_2O, CO_2, NaCl) in the system.

2. Solving the linear system J * (-dX) = R. This is generally done by employing an iterative solver like the BiCGStab (van der Vorst, 1992) or GMRES (Saad and Schultz, 1986) method.

It is generally advantageous to use as large time steps as possible for the simulation since this reduces both the number of times the residual has to be computed as well as the number of times the linear system needs to be solved. However large time steps come at a cost: Since the off-diagonal elements of the Jacobian matrix scale with the time step the matrix becomes non-diagonally dominant and will require many more iterations than when using small time steps. Additionally, the numerical solvers can break down when they run out of numerical precision, which happens quite frequently with time steps larger than ~10^{12}s. An ideal time stepping method would aim to determine the approximate time for solving the linear system a priori and choose the time step accordingly; however, the current time stepping methods used in the simulators are quite crude.

For the following discussion we will set the time step length issue aside for a moment and focus on how to efficiently solve the two main computational tasks.

3.1. Computing the Residual

The residual vector needs to be computed repeatedly to determine both the right-hand side vector and the Jacobian matrix. Using a set of primary variables (e.g. pressure and temperature) we need to calculate the secondary variables (e.g. density, enthalpy, and viscosity of each phase) of each model element and from there its accumulation terms. The residual is then calculated using the accumulation terms from the previous time step and by taking sources/sinks and fluxes to/from neighbouring elements into account. With the exception of flux terms no calculations require knowledge of another element's thermodynamic state, and even the calculation of the flux terms can be arranged in such a way that all numerical calculation can be performed in a completely data parallel manner.

Advances in computing technology over the last couple of years have no longer focused on increasing the CPU clock rate by significant amounts. Computational speedup today is usually achieved by parallel computing, where a computational task is performed by multiple arithmetic units or cores. These can be located either inside a single CPU, or we can also employ multiple CPUs in either shared or distributed memory configurations.

Data parallelism is an easy problem for which we can employ multiple-core systems. For large models, we expect linear scaling in terms of the number of cores N_{cores}, i.e. the computational time for calculating the residual is approximately

$$t_{residual} = t_{residual,serial} / N_{cores}$$

where $t_{residual,serial}$ is the time required when performing the calculation using a single core.

The number of cores available on CPUs has grown significantly over the last years. Quad cores are nearly a standard these days; CPUs with 8 cores are available for high-end consumer machines. Top end CPUs feature more than 24 cores, and some computational accelerators like the Intel

Xeon Phi can have more than 72 cores, although at a reduced clock rate versus a CPU. Hence desktop PCs can achieve great speedup related to the calculation of the residual, provided one invests in suitable CPU hardware.

All of the simulators under discussion here with the exception of TOUGH2 use parallel methods, either through OpenMP or MPI libraries. Excepting TOUGH2 (for which also parallel versions exist) we hence do not expect large differences between the simulators' performance relating to calculating the residual, though of course it is always possible that a simulator may employ non-optimal algorithms.

3.2. Solving the Linear System

3.2.1. General Approach

The linear system is in general solved using a sparse iterative solver. Internally these solvers use basic linear algebra operations, i.e. once per iteration the solver will solve several of the following operations: SpMV (sparse matrix-vector multiplication), AXPY (vector scaling plus addition), DOT (vector dot products), and SpTSv (sparse triangular solves).

All these linear operations can be parallelized; however, the SpTSv can prove challenging (see discussion further below). However, as it turns out we need to keep the total memory size for the linear system in mind; this is the total size of all matrices and vectors which need to be stored in RAM and are accessed by the solver during each iteration. The size N_B in bytes for storing the Jacobian matrix can be estimated as

$$N_B \sim (6+1) * N_E * (N_C + 1)^2 * (B+4)$$

The size depends on the number of connections per element (typically 6 for a regular tartan grid), the number of elements N_E, the number of components N_C and the number of bytes per coefficient B (8 for double precision). In addition to storing the Jacobian matrix we also need to store another matrix of the same size when using ILU0 as a preconditioner. For pure water (N_C=1) we can hence estimate the minimum size of RAM required as

$$N_B \sim 2 * 336 * N_E$$

In addition to this a number of vectors need to be stored as well; the exact number depends on the type of solver used and can be substantial (e.g. for GMRES). However, the above estimate suffices for our discussion here.

When starting the sparse iterative solver, the system will load the required data from RAM into the CPU's cache; this is a relatively slow process. The cores then access the CPU cache to load the data into the arithmetic units and perform the necessary calculations; this process is typically two orders of magnitude faster than loading data from RAM into the cache.

For the second and subsequent solver iterations we can distinguish two cases: If the size of the linear system is small compared to the CPU cache size then the CPU will still have a copy of all the necessary data in the cache. Since no data needs to be loaded from RAM into the cache this iteration will be much faster than the first iteration. Since the cores can access and process the data from the cache, parallelization works and the calculation is sped up.

On the other hand, if the linear system size exceeds the CPU cache size then the system needs to reload the data from RAM each iteration of the iterative solver. It now does not matter that the computation can be solved in parallel since even the speed of performing the computations on a single core far outstrips the speed of loading data from RAM. The process of solving the linear system now becomes bandwidth limited.

To find out at what problem size this bandwidth limitation becomes an issue, we can compare the typical cache sizes of modern CPUs with the size of the linear system. For example high-end CPUs like from Intel Xeon family have ~25MB cache; hence the maximum model size they can accommodate is around

$$N_E = 25 * 1024^2 / (2 * 336) \sim 39{,}010$$

Other more commonly used CPUs like Intel i7s have only about 6 to 12MB of cache and hence can accommodate much smaller systems. In addition, the operating system and other processes will occupy part of the cache and further reduce this number. Also, if using more than one component (e.g. NCG or salt), this number decreases dramatically due to the quadratic (N_C + 1) term.

Two strategies can be employed to overcome or improve the bandwidth limitation. The first is to use distributed memory systems, where the linear system is distributed over the RAM of multiple nodes, i.e. each computer will work on certain rows of the system. Whenever required the nodes exchange intermediary results, e.g. via a network connection. This method allows to add up the bandwidths of the individual nodes; also, if the linear system is split into sufficiently small parts, then these will again fit into the caches of the CPUs of the nodes. The simulators based on the PETSc library, i.e. TOUGH3 and Waiwera, make use of this method. For example, O'Sullivan et al. (2019) made use of 320 CPUs for their 2,300,000-element sized Lihir model. Using the above equations for estimating the size requirement we see that they required about 4.6MB per CPU, i.e. they were able to push the problem below the bandwidth limitation threshold.

The major disadvantage of the distributed memory method is the high number of nodes required to run large models; further, one requires access to an adequate computer cluster or the high capital cost of purchasing such a system. A second disadvantage is the complexity of SpTSv operations when using distributed memory; we will discuss this further down.

The second strategy is to push the bandwidth limited operations of the linear solve to a computational device which has higher bandwidth than a CPU. Graphical Processing Units (GPUs) have become popular to solve parallelized computational problems over the last decade. A great advantage of these GPUs is that their bandwidth far exceeds the bandwidth of a CPU. For example high end consumer grade GPUs like NVIDIA's GeForce TITAN RTX obtain bandwidths of up to 672GB/s at a cost of US$2,500; other high end GPUs reach up to 1000GB/s. For comparison, Intel Xeon CPUs typically reach only ~30GB/s. Hence depending on expenditure GPUs can achieve speedups of up to factors of 30 versus CPUs for the linear solve operation while employing normal desktop or laptop computers. NVIDIA supports linear solver development by providing dense and sparse library packages for basic linear algebra operations[2].

The Volsung package provides two solver architectures: One CPU based solver for small models or for PCs which are not fitted with an NVIDIA GPU, and a fully parallelized GPU solver architecture featuring BiCGStab and GMRES solver types.

3.2.2. Preconditioner Issues

Sparse iterative solvers typically employ preconditioners to improve their performance, i.e. to reduce the number of iterations required. The most often employed preconditioner is incomplete lower/upper with zero backfill (ILU0). This preconditioner has the same matrix stencil as the Jacobian matrix and hence has the same storage requirement. It is based on finding an upper and lower triangular matrix pair such that

$$(L*U) = J$$

However, the above L/U matrices would be dense and require a huge amount of space; hence ILU0 only calculates the coefficients on the Jacobian matrix stencil. This reduces the above equation to an approximation, i.e. $(L*U) \sim J$. An approximate solution is all that is required within the iterative solver; however, the better this approximation the smaller the number of iterations required. Hence the less coefficients are used for L/U the more iterations may be required by the solver.

ILU is a highly serial problem since finding the solution for one row in the system requires knowledge of previous parts of the solution. Parallelization can be achieved in two ways. The first method is to use additional block stencils over the matrix diagonal, as it is done in Jacobi-Block or ASM preconditioners. Each block can then be solved in parallel on different nodes. However, using this method means even less coefficients than for ILU0 are considered and the preconditioner suffers from degradation effects with increasing number of stencil blocks.

The second method is making use of the fact that in ILU0 some rows can be solved independently of each other. Rows are then combined into solve levels; within each level rows can be solved independently in parallel. The preconditioner is analyzed once at the beginning of a simulation to determine the solve levels; since the levels do not change over the simulation, they can be re-used many times. The advantage of this strategy is that the full ILU0 can be used, i.e. the quality of the preconditioner does not degrade. However, since the number of solve levels is quite high this method does not work very well on distributed memory systems since it requires a high number of communications between the nodes. For GPUs on the other hand the communication part is not required, hence making this architecture suitable for parallel ILU0.

3.2.3. Other Considerations

So far, we have only considered situations where the linear system to be solved did not change, i.e. we assumed different simulators would go through exactly the same calculation steps.

However, there are other factors at play. For example we need to carefully consider numerical truncation errors when calculating the Jacobian matrix. This becomes important at large time-steps since parts of the equations for calculating the residual vector scale with the time step while other parts don't. Exactly how a simulator will perform this calculation will determine the numerical truncation; if not done properly it may become harder to find numerical solutions to the linear system, i.e. more iterations are required.

Similarly, it is quite common to see the number of iterations of the solver increase with large time steps since the matrix becomes non-diagonal dominant. At large time-steps solver breakdown becomes an issue when the internal precision of the solver is exceeded; this usually means that the current time step is to be aborted and the step is repeated with a reduced time step; this comes at a high computational time penalty.

Lastly, it is also common to see super-linear increases in the number of iterations when approaching the solver breakdown limit. In this case doubling the time step may result in more than twice the time required for solving the linear system. It would be of high interest to find better criteria for time step adaption in order to improve simulator performance.

3.3. Overall Parallel Performance

When employing parallelization, the overall speedup achieved is limited by Amdahl's law (Amdahl 1967). For our example here where we have two processes—one for calculating the thermodynamics (TD) and one for linear solves (LS)—we can calculate the maximum possible speedup as

$$S_{tot} = 1.0 / (p_{TD}/S_{TD} + p_{LS}/S_{LS})$$

Here p stands for the fraction of the computational time the process needs when run in serial, and S is the speedup for the process by employing parallelization. We can now calculate expected speedups; we focus here on typical values encountered and consider only shared memory systems.

During a steady state simulation, the time step is usually not constrained; due to the non-diagonal dominance of the matrix this means that typically more than 90% of the computational time is spent in the linear solve. We can now consider the speedup employing parallelization using a modern desktop CPU, for example an octa-core, versus running the simulation on a serial simulator like TOUGH2. If we assume that the model is large and hence bandwidth limited (i.e. no linear solve speedup) we have

$$S_{tot} = 1.0 / (0.1 / 8 + 0.9 / 1.0) = 1.10$$

On the other hand, if we now also speed up the linear solve process by using a GPU with 15 times the bandwidth of the GPU we obtain

[2] https://docs.nvidia.com/cuda/cuda-samples/index.html#cusolversp-linear-solver-

$$S_{tot} = 1.0 / (0.1 / 8 + 0.9 / 15.0) = 13.8$$

If the time step is constrained during calibration or scenario runs then the matrix is usually diagonal-dominant and the linear solve requires less iterations. Typically, the computational time is now split evenly between calculating the thermodynamics and performing the linear solve. When only employing parallelization for the thermodynamic calculations we will have

$$S_{tot} = 1.0 / (0.5 / 8 + 0.5 / 1.0) = 1.8$$

or, when using our GPU example,

$$S_{tot} = 1.0 / (0.5 / 8 + 0.5 / 15.0) = 10.4$$

The above calculations suggest that significant speedup on large models can only be had by breaking the bandwidth limitation of the CPU, whether by employing distributed memory computer clusters or by using GPUs. They also illustrate that, when using GPUs, we can expect the largest speedups while running steady-state simulations. During calibration/scenario runs speedup will be more in line with the number of cores employed for solving the thermodynamic states, however, the GPU still provides speedup beyond this limit.

4. PERFORMANCE TESTS

4.1. Steady-State Performance

In this test we wanted to investigate the bandwidth dependency of the linear solve operation. We chose a 100,000 block model (50×50×40 blocks) with block dimensions 20×20×10 metres, set to a cold initial state (300bar, 20ºC); this model more than exceeds the CPU cache size of the CPUs we tested it on. One bottom corner block was set as a fixed state with high temperature and pressure (500bar, 250ºC); across the diagonal we produced 1kg/s of fluid from a block in the top layer. The model used single porosity, 50mD permeability, specific heat of 1000J/(kg*K), rock density 2650kg/m^3, heat conductivity of 2W/(K*m) and 1% porosity. The simulation was run for 1 million years with no constraints on the time step. This model was chosen for its simplicity, i.e. it doesn't have phase changes.

We ran this model on different simulators—TOUGH2, TOUGH3 and Volsung; using Volsung we also ran it using different CPUs and GPUs for solving the linear system and used different operating systems (Linux and Windows 10). We then recorded total run time, the time spent on solving the linear systems[3] and the number of time steps it took the simulator to solve the simulation.

Table 1 summarizes the run results. A direct comparison of the times is difficult since the different simulators took a different number of time steps to move through the simulation time. TOUGH2 took nearly twice the number of steps (162)

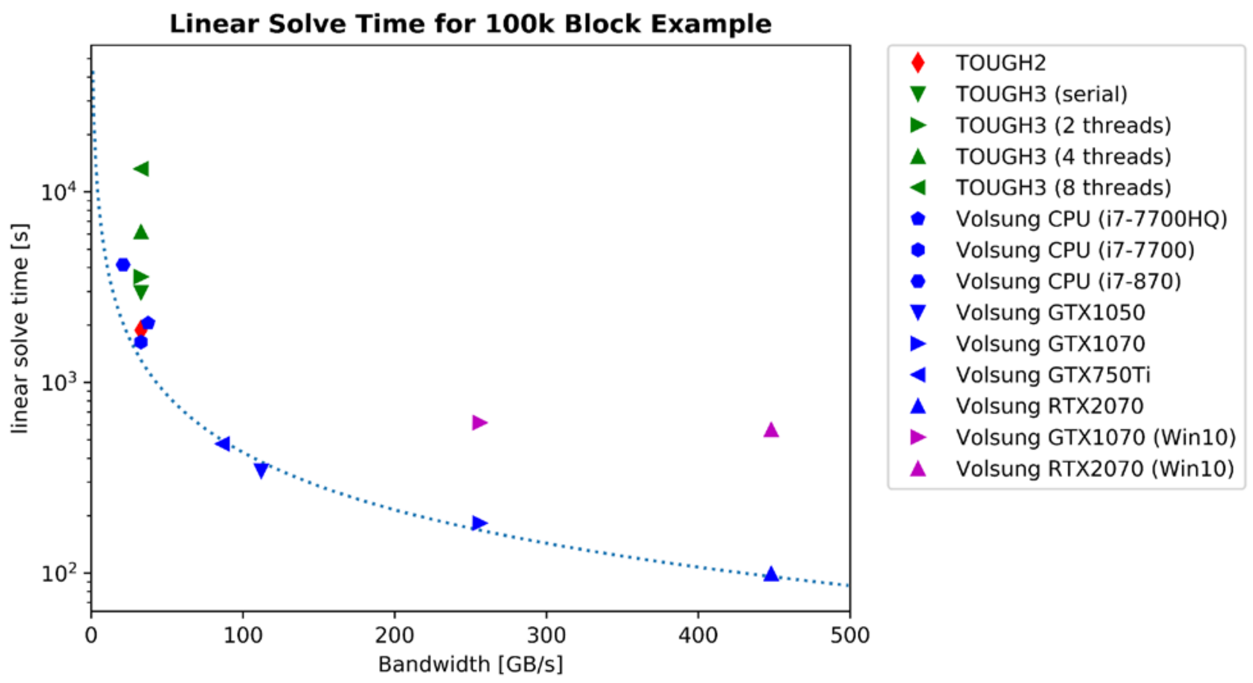


Figure 10: Performance benchmarking versus TOUGH2 and TOUGH3 using different CPU and GPU architectures for Volsung. The dotted line demonstrates the 1/bandwidth dependency of the linear solve operation for large models. The slower solution times for Volsung using Windows is understood to be graphics driver issue.

[3] TOUGH2 did not record the linear solve time; we used the total run time as an estimate since the other simulators indicated that it was by far the dominant computational task

Table 1: Performance benchmarking of Volsung versus TOUGH2 and TOUGH3.

Label	Steps	Total Time [s]	Linear Solve Time [s]
TOUGH2 (i7-7700HQ)	162	1883.0	1883.0
TOUGH3-2 threads (i7-7700HQ)	845	4342.0	3587.0
TOUGH3-4 threads (i7-7700HQ)	845	6999.0	6177.0
TOUGH3-8 threads (i7-7700HQ)	845	14869.0	13233.0
TOUGH3-serial (i7-7700HQ)	845	3698.0	2969.0
Volsung CPU (i7-7700HQ)	93	2154.0	2054.0
Volsung CPU (i7-7700)	93	1696.0	1632.0
Volsung CPU (i7-870)	93	4311.0	4155.0
Volsung GTX1050	94	420.0	344.0
Volsung GTX1070	93	227.0	183.0
Volsung GTX750Ti	93	579.0	477.0
Volsung RTX2070	93	145.0	99.5
Volsung GTX1070 (Win10)	93	689.0	616.0
Volsung RTX2070 (Win 10)	93	631.0	566.0

than Volsung (93[4]); this enabled it to stay competitive versus Volsung's simple CPU solver (which is parallelized but not optimized yet) since the smaller time steps require less iterations in the linear solver (see discussion in 3.2.3).

TOUGH3 took 845 steps; since it is based on TOUGH2 we don't quite understand why it requires so many more time steps than TOUGH2. We also ran it using 1, 2, 4 and 8 threads on a quad core CPU and it showed significant runtime degradation with increasing number of threads. We believe this is due to the aforementioned preconditioner degradation, i.e. more and more matrix coefficients are dropped when increasing the number of stencil blocks/threads.

Since Volsung took the same number of steps we can use its linear solve time to illustrate the bandwidth dependency of this operation. Figure 10 shows the linear solve time as a function of the bandwidth of the architecture used. The dotted line in this figure highlights the 1/bandwidth dependency of the operation over a wide range of bandwidths. Using an RTX2070 GPU (448GB/s) Volsung solved this model approximately 18 times faster than TOUGH2, about 29 times faster than TOUGH3 (serial) and 133 times faster than TOUGH3 (8 threads) through the combination of longer time steps and faster solver iterations.

We also noted that the Volsung GPU solver requires significantly longer solve times under Windows 10 than under Linux operating systems; it appears that this is due to a limitation in the Windows driver for NVIDIA graphics cards when using the GPU for both displaying the screen and performing GPU calculations at the same time. Further tests will show if this limitation can be overcome when adding a second graphics card to a PC for displaying the screen.

4.2. Calibration/Scenario Performance

We performed another set of tests with variations of the #4 test model from the Stanford intercomparison study. We discretized this model using 18,000, 50,000 and 98,000 blocks and limited the time step to simulate behaviour similar as encountered in a calibration or scenario run of a simulation. We tested these models using TOUGH2 and Volsung and recorded the total run times and number of steps taken for the run. TOUGH2 and Volsung solved this model using a comparable number of time steps. Volsung achieved speedups of 3, 5 and 7 for this model, with speedup increasing with model size and hence the size of the linear system. These speedups are in line with expectations with the number of cores (4) and GPU (256GB/s) used as discussed in section 3.3.

5. CONCLUSIONS

We demonstrated the suitability of the new Volsung software package by validating it against the industry standard TOUGH2 simulator. Validation results confirm very good agreement between the simulators over all problems tested.

Limits to performance enhancement using parallel processes were discussed. Volsung uses GPUs to address the bandwidth limitation of the linear solve operation and parallelizes calculation of thermodynamic states using multi-core CPUs. The speedups observed qualitatively and quantitatively demonstrate the performance improvements possible when solving the finite volume method on shared memory systems.

REFERENCES

Amdahl, G. M. (1967). Validity of the single processor approach to achieve large scale computing capabilities. AFIPS Conference Proceedings, 30, 483–485

Battistelli, A., Calore, C., & Pruess, K. (1997). The Simulator TOUGH2/EWASG for Modelling Geothermal Reservoirs with Brines and Non-Condensible Gas. Geothermics, 26(4), 437–464.

Croucher, A., O'Sullivan, J.O., Yeh, A. & O'Sullivan, M.. Benchmarking and experiments with Waiwera, a new geothermal simulator. Proceedings, 43rd Workshop on Geothermal Reservoir Engineering Stanford University, Stanford, California, February 12-14, 2018

Jung, Y., Pau, G. S. H., Finsterle, S., & Pollyea, R. M. (2016). TOUGH3: A new efficient version of the TOUGH suite of multiphase flow and transport simulators. Computers and Geosciences, 108(November), 2–7.

Molloy, M. W. (1981). Geothermal reservoir engineering code comparison project. Proceedings Special Panel on Geothermal Model Intercomparison Study 1980, Stanford University, Stanford, California.

O'Sullivan, J., Croucher, A., Popineau, J., Yeh, A., & O'Sullivan, M. (2019). Working with Multi-million Block Geothermal Reservoir Models. Proceedings 44th

[4] One run in Volsung took 94 instead of 93 time steps. This is a behaviour commonly seen when applying parallelization; the results become slightly non-deterministic due to truncation and order-of-execution differences.

Workshop on Geothermal Reservoir Engineering, Stanford University.

Pruess, K., Oldenburg, C., & Moridis, G. (1999). TOUGH2 User's Guide, Version 2.0., LBNL-43134.

Saad, Y., & Schultz, M. H. (1986). GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. SIAM J. Sci. Comput., 7(3), 856–869.

Thunderhead (1999). Understanding and Using MINC. https://www.thunderheadeng.com/files/com/petrasim/Understanding and Using MINC.pdf

van der Vorst, H. A. (1992). Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. SIAM J. Sci. Comput., 13(2), 631–644.