Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200059

Acceleration of Hydro Poro-Elastic Damage Simulation in a Shared-Memory Environment [1]

Harel Levin [a,b], Gal Oren [a,c], Eyal Shalev [d] and Vladimir Lyakhovsky [d]
[a] *Department of Physics, Nuclear Research Center-Negev, P.O.B. 9001, Beer-Sheva, Israel*
[b] *Department of Mathematics and Computer Science, The Open University of Israel, P.O.B. 808, Raanana, Israel*
[c] *Department of Computer Science, Ben-Gurion University of the Negev, P.O.B. 653, Beer Sheva, Israel*
[d] *Geological Survey of Israel, 32 Yeshayahu Leibowitz, Jerusalem, Israel*

Abstract. Hydro-PED [1] is a numerical simulation software which models nucleation and propagation of damage zones and seismicity patterns induced by well-bore fluid injection. While most of the studies in geo-physical simulation acceleration and parallelization usually focus on exascale scenarios which are translated into vast meshes, encouraging a distributed fashion of parallelization, the nature of the current simulations of Hydro-PED dictates amount of data that can conveniently fit on a single compute node - NUMA and accelerator memory alike. Thus shared-memory parallelization (such as OpenMP) can be fully implemented. In order to utilize this insight, Hydro-PED was interfaced with Trilinos [2] linear algebra solvers package, which enabled an evolution to iterative methods such as CG and GMRES. Additionally, several code sectors were parallelized and offloaded to an accelerator using OpenMP in a fine grained manner. The changes implemented in Hydro-PED gained a total speedup of x5-x12, which will enable Hydro-PED to calculate long-term simulation scenarios of hundreds of years in a feasible time - a few weeks rather than a year.

Keywords. Geological Simulation, Accelerators, Numerical Linear Algebra, Shared Memory, Trilinos

1. Introduction

On the past few decades, the increasing compute power encourages the development and usage of scientific computer simulations as a preliminary step before traditional experiments and industrial development. The geophysical research area is no exceptional as applications like FLAC [4], GEOS [5] and others provide numerical simulations for a

[1]This work was supported by the Lynn and William Frankel Center for Computer Science. Computational support was provided by the NegevHPC project [3].
Special thanks goes to A. Tarabay, M. Homel and J. White, our colleagues in Lawrence Livermore National Laboratory for their priceless support and guidance during this work.
Source code is accessible at `https://github.com/harellevin/Hydro-PED`.

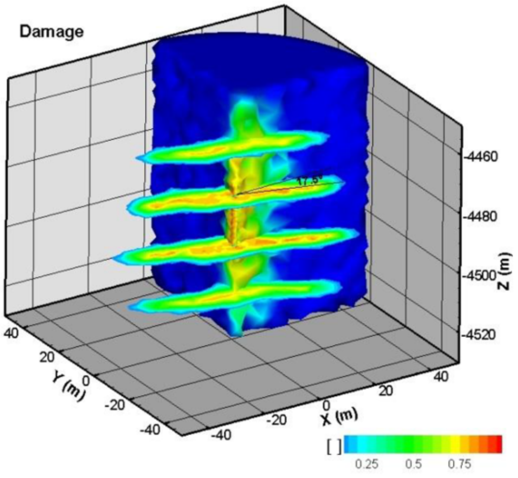

Figure 1. [1] Hydro-PED simulation of damage to granite wellbore after 80 hours of fluid injection.

wide range of geophysical scenarios [6] [7]. *Hydro-PED* [1] was developed in 2013 in order to simulate hydro fracturing [8].

Hydro-PED was first introduced as a serial program written in Fortran 90, which models nucleation and propagation of damage zones and seismicity patterns induced by wellbore fluid injection. The model formulation of Hydro-PED accounts for the following general aspects of brittle rock deformation: (1) Nonlinear elasticity that connects the effective elastic moduli to a damage variable and loading conditions; (2) Evolution of the damage variable as a function of the ongoing deformation and gradual conversion of elastic strain to permanent inelastic deformation during material degradation; (3) Macroscopic brittle instability at a critical level of damage and related rapid conversion of elastic strain to permanent inelastic strain; (4) Coupling between deformation and porous fluid flow through poro-elastic constitutive relationships incorporating damage rheology with Biot's poroelasticity. Figure 1 presents an output from Hydro-PED simulation of damage to granite wellbore after 80 hours of fluid injection, originally presented on [1].

Most of the studies in geo-physical simulation software acceleration and parallelization usually focus on exascale scenarios which simulate mesh simulation while focusing on either wide areas or high resolution. As a result, most of the simulations result in a vast and dense mesh of cells, encouraging software developers to adopt a distributed fashion of parallelization in order to divide the computational load between as many computational cores as possible. However, this is not the case in current Hydro-PED simulated scenarios as even a relatively coarse grained mesh with static pre-defined fine area near the wellbore edges, provides valuable insights about the simulated scenario. Therefore, the nature of the current simulations of Hydro-PED dictates the amount of data that can conveniently fit on a single compute node - NUMA and accelerator memory alike. thus shared-memory parallelization (such as OpenMP) can be fully implemented. Nevertheless, while Hydro-PED also supports simulations of a short-range of time, the *long-term* simulations are of special interest in the context of damage estimation. However, the time dimension is not subject to parallelization since each timestep depends on its precestor timestep. When it came to simulation of hundred of years in high definition, the run-

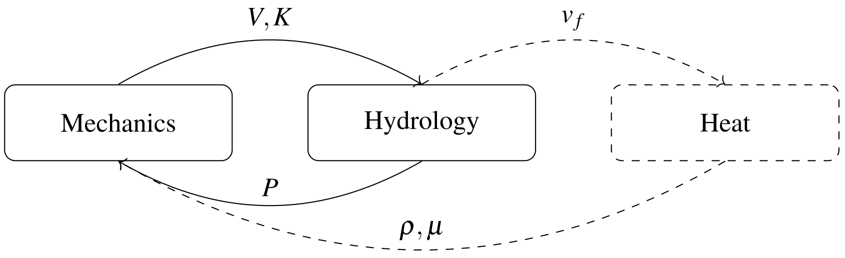

Figure 2. Schematic overview of the relations between Hydro-PED modules. The Thremodynamics module is currently under development.

time of Hydro-PED became unfeasible. Consequently, it was crucial to inspect various approaches in order to accelerate each simultaion timestep on a single node.

2. Hydro-PED - Hydro Poro-elastic Damage simulation

Hydro-PED consists of two modules: (1) Mechanical-Damage module and (2) Hydrological module. The mechanical module iterates over all simulation cells and solves relevant geo-physical equations using Explicit Finite Difference Lagrangian Method (EFDLM), while the hydrological uses the Finite Element Method (FEM) to transform the diffusion equations into a linear equations system. Later, the module initiates a third party direct solver (HSL [9]). A tetrahedral mesh is used to describe the physical area throughout all simulation modules. Each tetrahedron is referred as *element* and each of its vertices is referred as *node*. Another module which couples heat equations is currently under development. Figure 2 provides a schematic overview of the relations between the modules. These relations will be explained on the following subsection.

2.1. EFDLM Mechanics Module

Explicit Finite Difference Langrangian Method (Hence EFDLM) is a fully explicit numerical method relies on a large-strain explicit Langragian formulation originally developed by Cundall [10].

The module solves the force balance equation for each node in the mesh. The forces over the body are induced from the underground stresses and a Frequency Independent Damping. Using the force balance, the velocity of each node is obtained. Later, the strain tensor of each element is being calculated, which is induced by the combination of elastic and plastic strain. The strain tensor enables the calculation of the new coordinates of each node. Using this data, the module produces the volume (V) and permeability (K) of each element which will be used in the hydrological module.

The mechanical module uses an adaptive timestep. This timestep is calculated at each iteration such that simulation cycles will be more frequent whenever a rapid changes occur, and will be rare when there are no notable changes on rock's form. The damage state of the rock is calculated at each step. Whenever a failure occurs, the simulation walks into a subroutine called *drop* which performs several healing actions.

Prior to the current work, some OpenMP directives were implemented in certain parts of the mechanical module of Hydro-PED. However, the acceleration achieved by these directives were insufficient.

2.2. FEM Hydrology Module

On each hydrological step, the corresponding module receives the current volume and permeability level of each element. The module solves a differential equation which describes the diffusion of fluid pressure through the rock. In order to do so, the module uses the well-known finite elements method (FEM) which yields a system of linear algebraic equations. For a given N elements, the equations are of type $A\vec{x} = \vec{b}$, where A is a $N \times N$ sparse symmetric positive matrix, b is a pre-calculated vector of size N representing the flux on each element, and x is the target solution vector of size N. The vector x represents the pressure (P) on each element. These values will be used later by the mechanical module to simulate the next timestep.

Since Hydrological changes in the wellbore tends to be less extensive, a hydrological step will occur after every couple of mechanical steps, excepts when the mechanical timestep is too long. Due to the adaptive timestep mechanism implemented in Hydro-PED, the geological changes between each two consecutive steps are relatively small. Consequently, the numerical errors generated by the simulation are minor and can be neglected.

2.3. FEM Heat Module (under development)

The heat module will be called at every hydrological step. It shall receive the fluids velocity (v_f) and calculate the diffusion and advection of the temperature over the fluids. Using the finite elements method, the heat equations can be translated into a system of linear algebraic equations, $A\vec{x} = \vec{b}$. However, due to the advection, matrix A will be asymmetric. This fact enforces a usage of slightly different solution methods. The solution of the linear system yields the density (ρ) and viscosity (μ) of the fluid.

2.4. Implications of the input file

The nature of the input file implicate the concrete execution in several ways. First, it dictates the size of the grid which strongly influences the actual size of the arrays. By Amdahl's law, bigger problem size derives greater parallelization potential. Hence, the input file implicates the speedup factor.

Second, the material type and the balance of forces defined in the input file influences the size of the timestep of both the mechanics and the hydrology modules. A fractional crystalized rock suffering from great pressure forces, tends to have rapid nucleation and propogation of fraction, which in turn increases the amount of mechanical timesteps. On the other hand, porous rocks will require additional hydrological steps. Hence, the speedup factors of each module will dictate total speedup of hydro-PED which will be different for each simulation scenario.

2.5. Bottlenecks and Challenges

Considering the common scenario, most of the computation time of Hydro-PED is spent by the solution process of the hydrological linear algebraic system. After the construction of the the system (i.e. constructing matrix A, and vector $\vec{b}$, and after allocating memory space for the solution vector $\vec{x}$, the system was sent to a third-party solver library called HSL [9]. This solver uses an algorithm which has time complexity of $O(n^3)$ (for matrix

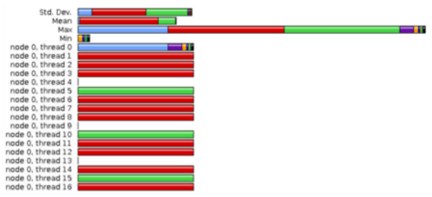

Figure 3. TAU profile results for the mechanical module. Red bars represents execution time of subroutine *derivation*.

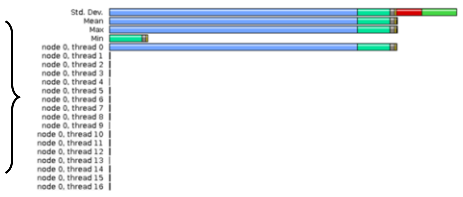

Figure 4. TAU profile results for *move_grid*'s execution time per call. Most of the time was spent by the main thread.

of size $n \times n$), as will be explained on section 4. Hence, finding more efficient way to solve the linear system, given the modern heterogeneous system architectures, was one of the first necessary steps.

Another disadvantage of using HSL was its lack of support for asymmetric matrices. Keeping on mind that the next challenge will be the coupling of the heat module with the current modules, which will yield an asymmetric matrix, forced a search after new solvers.

Yet another aspect which was investigated is how to boost the performance of the mechanical module itself. As mentioned before, some parts of the module were parallelized using OpenMP directives in a fine-grained manner. However, we were curious whether the power of accelerators can be utilized to achieve another speedup, which, as mentioned before, was crucial in order to shorten the amount of time the run spent on each timestep.

3. Speedup Using Explicit Asynchronous Offload

In order to find how to exploit better performance from the mechanical module, a parallelization-driven profile of the code was conducted. The profile focused the search to a subroutine which calculated the displacement of mesh elements coordinates. While the execution-time per-call of the subroutine was relatively small, this subroutine was called over and over, resulting in a consumption of on not less than 41% of module's total runtime. By implementing both parallelization and time-sharing offload, extra speedup was achieved.

3.1. Profiling

TAU Performance System [11] is a commonly-used profiler which is aimed to the task of profiling runtime of parallel applications. TAU shows how much time was consumed by each subroutine on each MPI rank and by each thread. Profiling Hydro-PED using TAU provided the bar-chart provided on figure 3 (the bars representing threads #17-#31 were deleted as they showed just the same behavior as the other threads). It is clear from the chart, that most of the runtime is consumed by the subroutine indicated by dark-red color. This subroutine turned to be a subroutine called *derivation* which is initiated from the subroutine *move_grid*. The subroutine *move_grid* is used to calculate the displacement of elements' coordinates (based on the velocities induced by the pressure vectors). Closer inspection of the subroutine showed that each invocation of the subroutine is relatively fast (figure 4). However, this subroutine is used on frequent occasions which explains its vast time consumption.

```fortran
subroutine move_grid(dt)
  ...
  ! calculate new coordinates based on velocities
  do i=1,nodes_count
      cord(i) = cord(i) + vel(i)*dt
  end do

  do i = 1,elements_count
  ! derivation of basic functions
    call derivation(i,cord,dr)
    strain(i) = calculate_new_strain(strain(i), dr)
    ! update fluid pressure
    pf_el(i) = calculate_new_pressure(pf_el(i), dr, dt)
  end do
return
end subroutine move_grid
```

Listing 1: The implementation of *move_grid* subroutine.

Listing 1 provides the implementation of subroutine *move_grid*. Note that parts of the code were encapsulated in subroutines (i.e. *calculate_new_pressure* and *calculate_new_strain*) for simplicity. After close inspection of the called subroutines, making sure that *move_grid* is a SIMD (Single Instruction Multiple Data) calculation, the entire content of the subroutine was wrapped in OpenMP's parallel-SIMD directive. In order to further improve the runtime, we considered offload to Intel® Xeon-Phi® 5110p co-processor (formerly known as Knights Corner - KNC) [12]. The performance on newer Xeon-Phis should outperforms our results. Offload to GPGPU accelerators using OpenMP 4.5 is not yet fully supported by most common Fortran compilers. As it will be implemented, we will be able to perform the same offload to NVIDIA® accelerators as well.

3.2. Asynchronous Accelerator Offload

In order to use the accelerator we consider a *host-target model* where the NUMA cores of the machine are considered as the *host* which executes the application. Whenever needed, the host can offload part of the computation to one of the accelerators attached to it. Hence, the accelerators are referred as the *targets*. The offload is done by special system calls who can be initiated either by run-time API or compiler directives (such as in OpenMP 4.5 [13]).

In order to perform a calculation on an accelerator, the input data should be offloaded to the accelerator first. Later, after the accelerator completes its calculations, the results should be sent back to the host. The time consumed by the communication between the host and the target is relatively large and is correlated linearly with the size of the data. Consequently, the effectiveness of of the offload is usually subject to the computational load of the calculation itself. Nevertheless, while *move_grid*'s computational load is pretty light, one can benefit from splitting the calculation between the host and the accelerator. The basic idea is to send the input arrays to the accelerator asynchronously as soon as possible, even before *move_grid* was called. Than, when the application initiates the subroutine, *cord* array will be calculated by both the host and the target simultaneously. Then, the host will calculate part of *strain* and *pf_el* array, while the target

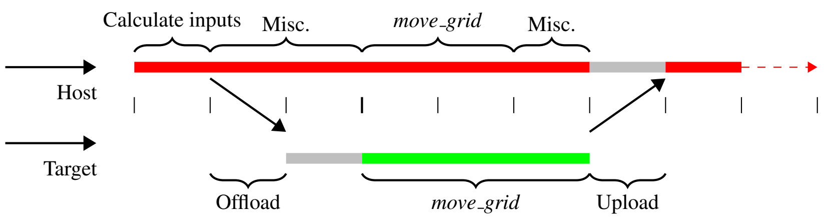

Figure 5. A timeline that demonstrates a common execution scenario where *move_grid* runs simultaneously on both host and target. The communication between the host and the target is asynchronous.

will calculate the rest of these arrays. Whenever the host finished calculating his part, he will continue the execution of the program to the point where the value of *strain* or *pf_el* are necessary. Whenever the target finishes his part of the computation he will send his output back to the host asynchronously.

This concept is demonstrated by the timeline presented on figure 5. The red line above represents the host, and the green line below represents the target. The timeline starts with the calculation of the arrays which are crucial for the calculations performed on *move_grid*. Whenever these arrays are ready, they will be offloaded to the accelerator, while the host performs another calculations. When the host steps into *move_grid*, the target will initiate the calculation on his part of the data. The host may continue with miscellenous calculations while the target still calculates his part. However, whenever the host reaches a part of the code where either *strain* or *pf_el* arrays are crucial, he will make sure the fresh data from the target was received (otherwise, he will wait).

```
subroutine move_grid(dt)
  ...
  !dir£ offload begin target(mic:0) wait(vel_signal) nocopy(cord : REUSE
   ↪ RETAIN) out(strain(1:phi_length): REUSE RETAIN) nocopy(vel : REUSE
   ↪ RETAIN) nocopy(m_biot : REUSE RETAIN) nocopy(nop : REUSE RETAIN)
   ↪ nocopy(de,dr,i,j,ii,nn,n) in(dt,nodes_count,phi_length)
   ↪ signal(strain_signal)
  call move_grid_inline(1,phi_length)
  !dir£ end offload

  !dir£ offload begin target(mic:1) wait(vel_signal) nocopy(cord : REUSE
   ↪ RETAIN) out(strain(phi_length+1:2*phi_length) : REUSE RETAIN) nocopy(vel
   ↪ : REUSE RETAIN) nocopy(m_biot : REUSE RETAIN) nocopy(nop : REUSE RETAIN)
   ↪ nocopy(de,dr,i,j,ii,nn,n) in(dt,nodes_count,phi_length)
   ↪ signal(strain_signal)
  call move_grid_inline(phi_length+1,2*phi_length)
  !dir£ end offload

  ! Host Part
  call move_grid_inline(2*phi_length+1,ne)
  return

  contains
    subroutine move_grid_inline(offset_begin,offset_end)
end subroutine move_grid
```

Listing 2: The implementation of the asynchronous offload in *move_grid* subroutine.

Listing 2 shows how the asynchronous offload was implemented using OpenMP. The main logic of *move_grid* subroutine was moved to another subroutine called *move_grid_inline* (listing 3) which gets two parameters which indicate the beginning and end indices of the strains array which should be handled on the current invocation. The subroutine *move_grid* calls *move_grid_inline* three time: one time for each MIC and one time for the host itself. The commands which call the MICs' *move_grid_inline* are wrapped in an offload directive which waits to a *vel_signal* signal indicating the calculation of the velocities array has already been done. The directive states the identity of the traget MIC, and dictates allocation and transport of the part of the strains array which is being handled. Whenever the MIC finishes calculating the strains vector, it sends a *strain_signal* signal.

```
subroutine move_grid_inline(offset_begin,offset_end)
  !dir£ attributes forceinline :: move_grid_inline
  !dir£ attributes offload:mic :: move_grid_inline
  integer :: offset_begin,offset_end

  !£OMP PARALLEL
  !£OMP DO SIMD
  do i=1,nodes_count
    cord(i) = cord(i) + vel(i)*dt
  end do
  !£OMP DO SIMD PRIVATE(dr,ii,de,j,nn)
  do i = (offset_begin),(offset_end)
    call derivation(i,cord,dr)
    strain(i) = calculate_new_strain(strain(i), dr)
  end do
  !£OMP END PARALLEL
end subroutine
```

Listing 3: The implementation of *move_grid_inline* subroutine.

3.3. Results

We implemented an asynchronous offload of *move_grid* subroutine. The results were tested on a machine with two sockets Intel® Xeon® CPU E5-2660 v2 processors. Each of them has 10 cores with clock rate of 2.2GHz. The machine contains two Intel® Xeon-Phi® 5110p co-processors. The total runtime of *move_grid* was measured on three settings: (1) Host only mode. That is, all the calculations were performed on the host itself. (2) Offload of about two-thirds of the array to one accelerator. (3) Offload of $5/11$ of the array to each of the two accelerators. Figure 6 shows the measured runtime for each of the settings. The offload to two coprocessors gained a speedup of about factor two compared to the host-only setting. This results suggests that asynchronous offloading is beneficial. As offload latency of accelerators is getting smaller ans smaller in modern architectures, asynchronous offload and host-target shared computations may be implemented on computations to enhance their performance.

We should note that the NUMA nature of the node is undifferentiated by the current implementation of the EFDLM module. That is, the memory is allocated on the master thread's socket memory. This implementations derives some overhead when remote access is done from the second socket. Obviously, when all the threads are allocated on a

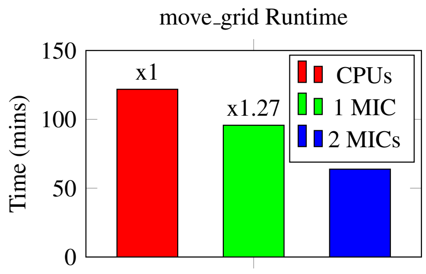

Figure 6. The runtime (in minutes) of subroutine *move_grid* using: (1) only CPUs, (2) CPUs and one Xeon-Phi accelerator, (3) CPUs and two Xeon-Phi accelerators.

single UMA (such as if a compact affinity was defined), this overhead should not occur. However, when the grid defining the scenario is sufficiently large, the speedup gained by distributing the problem to more than one socket's amount of threads overcomes the time overhead.

4. Speedup Using Advanced Linear Algebra Solvers

The task of solving linear algebraic systems of equations is one of the fundamental problems in scientific computing. During the past decades a lot of methods were devised in order to tie the best solution method to each problem given any hardware architecture. While initiation of third-party libraries of solvers for this task is usually the preferable step, it should be taken into account that not all the solvers were created equal. Each solver may use different methods which may applicable for different type of matrices [14]. Furthermore, not all the solvers implemented in a way that exploits the features and characteristics of the underlying hardware. The last fact is even more true when it comes to heterogeneous hardware and accelerators [15]. Consequently, when we looked for a way to speedup the hydrology module, the linear solver was the usual suspect.

4.1. Methods to Solve Linear Algebraic Equations

Roughly speaking, all the algorithms to solve linear systems can be divided to two types: (1) Linear methods, which translate the original system into equivalent more simple system, and then solve the equivalent system. (2) Iterative methods, which starts with initial guess ($\vec{x}_0$) for the solution vector $\vec{x}$. Later, the iterative algorithm calculates the residual which is the "distance" between the guess and the correct solution, i.e. $|A\vec{x}_0 - A\vec{x}|$. The algorithm will try to refine the initial guess until the residual will be lower than a predefined threshold. Each linear algorithm is distinct by the permutations and factorizations it adopts to achieve the equivalent system. Each iterative algorithm is distinct by the way it performs the refinements.

In order to achieve the simple equivalent system in linear methods, there are several techniques to manipulate the original system. These techniques usually evolves all the vector and matrix cells in the entire system. Therefore, the time complexity of the equivalent system calculation step is usually larger than $O(n^2)$ (considering a system with matrix A of size $n \times n$). The time complexity of the solution step is usually $O(n^2)$. One way to achieve a simple equivalent system is to perform a *LU-factorization* [16], where $A = LU$ such that L (respectively U) is a matrix which have non-zero values only on its lower (respectively upper) triangle. Prior to these work, Hydro-PED used the linear solver HSL_MA87 [17] which uses Cholesky factorization where $A = LL^T$ such that L is a matrix which have non-zero values only on its lower triangle, and L^T is transposed L. Both LU-factorization and LL^T-factorization have time complexity of $O(n^3)$.

While the calculation of the equivalent system in linear methods evolves the entire vector and matrix cells in the system, the calculation of the residual in iterative method can be done by using only the non-zero cells solely. Consequently, using iterative methods on sparse matrices (i.e. $\left|\{(i,j)|A_{i,j} \neq 0\}\right| = O(n)$), the time complexity of each iteration will be $O(n)$ plus the time complexity of the solution refinement step which is usually $O(n)$ too. In conclusion, the time complexity of iterative method which is applied on matrix A of size $n \times n$ and which is converged after k iterations will be $O(nk)$.

4.2. Trilinos - Solver for Heterogeneous Systems

Trilinos [2] is a collection of open source libraries which are used as building blocks. Following a recent work about usage of second-generation Trilinos [18], we used several libraries which we interfaced with Hydro-PED:

- Techos - Provides wrappers for BLAS and LAPACK, smart pointers and parameter lists [19].
- Kokkos - Implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms. It supports MPI, OpenMP, Pthreads and CUDA [20].
- Tpetra - Implements linear algebra objects which are built on Kokkos [21].
- Belos - Implements most of the common iterative solution methods of linear systems [22].

The combination of these building blocks provides a strong and versatile framework to solve linear systems. The main additive value of Trilinos is the usage of Kokkos as an encapsulated framework which enables the programmer to exploit different types of HPC architectures and technologies without any major changes in the code. This fundamental feature makes Trilinos optimal for heterogenous systems which contains traditional CPUs along with GPGPUs and Xeon-Phis. Moreover, Trilinos uses blocking (i.e. tiling) methods in order to achieve better performance by dwelling each block in a single UMA. This NUMA-aware approach gains greater speedups as shown in [23].

We used *Belos* package to implement the well known iterative algorithm *Conjugate-Gradients* (CG) [24] which shows relatively rapid convergence for symmetric matrices (as the matrix yielded by the diffusion equations of the hydrology module). We may use the *Generalized minimal residual* (GMRES) method [25] in the future when we will deal with asymmetric matrices on the heat module.

4.3. Reuslts

In Hydro-PED's hydrology model, each timestep yields a new matrix which is slightly different from the previous matrix. Therefore, we would not be able to exploit the advantage of one-time factorization in the linear method. Furthermore, for a matrix of size $n \times n$ yielded by Hydro-PED, there will be about $12n$ non-zero values (due to geographical and geometrical considerations [1]). Moreover, the differences in the solution vector

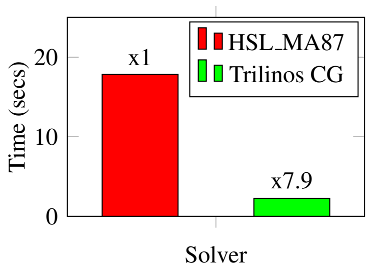

Figure 7. Solvers overall runtime with 11e4 nodes using HSL_MA87 and Trilinos.

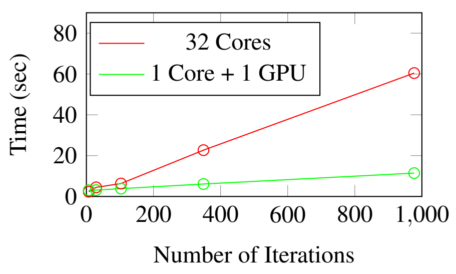

Figure 8. Trilinos overall runtime as a function of iterations amount, on multi-core CPUs and on GPU.

$\vec{x}$ between each two consecutive steps is relatively small such that the solution of previous timestep may be used as a good initial guess for the following timestep. As a result of these consideration, HSL's linear solver was replaced with the implementation of CG given by Trilinos' Belos package.

We tested both HSL and Trilinos on a common scenario of a mesh with 110,000 nodes. Both tests ran on 32 cores provided by machine with two sockets of Intel® Xeon® Gold 6130. Figure 7 shows the overall runtime of each solver (in seconds). The average number of iterations needed until Trilinos solver converged was 160. Trilinos showed speedup of almost x8 comparing to HSL_MA87.

We investigated the influence of GPU accelerators on the runtime of Trilinos. We used a relatively big (but still applicable) scenario of a mesh with 8.5 million nodes which took about 2GB of GPU memory capacity. We ran several simulations with this mesh, each takes different number of iterations to converge. The simulations ran on a machine with two sockets of Intel® Xeon® Gold 6130 and one NVIDIA® Tesla® V100 GPGPU. Figure 8 shows the overall runtime of Trilinos solver using different amount of iterations. We can learn from the trends of the chart that while the initialization of the solver using the GPU took about 3 times more than the initialization without accelerator (probably due to memory offload), the runtime of each iteration on the GPU was 15 times faster than on the CPUs. Consequently, the usage of GPU started to pay-off starting from 20 iterations.

5. Conclusion

In this paper we show several useful techniques to profile, analyze, and enhance the performance of scientific applications on a shared-memory environment, using geophysical application as a test-case. Hydro-PED's code was built in a heterogeneous and modular fashion which dictated different kinds of treatment. In the mechanics module, the explicit manner of calculations directed us to profile the runtime and find bottlenecks. In the hydrology module, which was based on FEM and linear systems, we used an of-the-shelf solution. However, choosing this solution should be done carefully, as we demonstrated.

Both on the hydrology module and the mechanics module, the usage of accelerators gained an extra speedup which are shown in figure 9. These speedups were gained either by synchronous or asynchronous offload. These techniques can be implemented in many other places, both in Hydro-PED and in another applications.

As both the balance between hydrological and mechanical steps, and the amount of iterations until convergence of Trilinos changes between different scenarios, the total

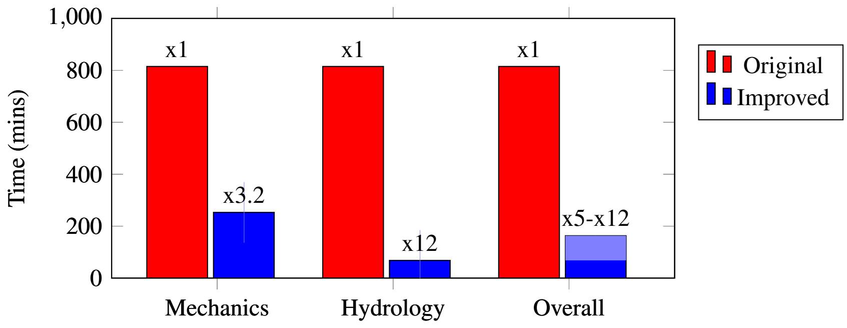

Figure 9. The speedup gained by: (1) asynchronous offload in the mechanics module, (2) using Trilinos in the hydrology module, and (3) overall. Note that the overall speedup of Hydro-PED is dictated by the balance between these modules (see 2.4).

speedup of our work is expected to be between x5-x12 comparing to the original execution. Using this speedup, a large-scale simulation which would have taken a year, will finish on a few weeks.

References

[1] Shalev, E., Lyakhovsky, V.: Modeling reservoir stimulation induced by wellbore fluid injection. In: Thirty Eighth Workshop on Geothermal Reservoir Engineering, Stanford University Stanford, California (2013)

[2] Heroux, M., Bartlett, R., Hoekstra, V.H.R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., Williams, A.: An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories (2003)

[3] NegevHPC homepage. `http://www.negevhpc.com` Accessed: 2019-02-19.

[4] FLAC homepage. `https://www.itascacg.com/software/flac` Accessed: 2019-06-24.

[5] Settgast, R., Johnson, S., Fu, P., Walsh, S., Annavarapu, C., Hao, Y., White, J., Ryerson, F.: Geos: a framework for massively parallel multi-physics simulations. theory and implementation. (2014)

[6] Homel, M., Herbold, E.: Fracture and frictional contact in the material point method using damage-field gradients for velocity-field partitioning. (2015)

[7] Annavarapu, C., Settgast, R.R., Vitali, E., Morris, J.P.: A local crack-tracking strategy to model three-dimensional crack propagation with embedded methods. Computer Methods in Applied Mechanics and Engineering **311** (2016) 815–837

[8] Shalev, E., Lyakhovsky, V.: The processes controlling damage zone propagation induced by wellbore fluid injection. Geophysical Journal International **193**(1) (2013) 209–219

[9] HSL: collection of fortran codes for large-scale scientific computation. See http://www.hsl.rl.ac.uk (2007)

[10] Cundall, P.A.: Numerical experiments on localization in frictional materials. Ingenieur-archiv **59**(2) (1989) 148–159

[11] Shende, S.S., Malony, A.D.: The tau parallel performance system. The International Journal of High Performance Computing Applications **20**(2) (2006) 287–311

[12] Chrysos, G.: Intel® xeon phi coprocessor (codename knights corner). In: 2012 IEEE Hot Chips 24 Symposium (HCS), IEEE (2012) 1–31

[13] OpenMP application program interface version 4.5. `https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`

[14] Sarkar, T., Siarkiewicz, K., Stratton, R.: Survey of numerical methods for solution of large systems of linear equations for electromagnetic field problems. IEEE Transactions on Antennas and Propagation **29**(6) (1981) 847–856

[15] Li, R., Saad, Y.: Gpu-accelerated preconditioned iterative linear solvers. The Journal of Supercomputing **63**(2) (2013) 443–466

[16] Cryer, C.W.: The lu-factorization of totally positive matrices. Linear Algebra and its Applications **7**(1) (1973) 83–92

[17] Hogg, J.D., Reid, J.K., Scott, J.A.: Design of a multicore sparse cholesky factorization using dags. SIAM Journal on Scientific Computing **32**(6) (2010) 3627–3649

[18] Lin, P., Bettencourt, M., Domino, S., Fisher, T., Hoemmen, M., Hu, J., Phipps, E., Prokopenko, A., Rajamanickam, S., Siefert, C., et al.: Towards extreme-scale simulations for low mach fluids with second-generation trilinos. Parallel processing letters **24**(04) (2014) 1442005

[19] Bartlett, R.A.: Teuchos c++ memory management classes, idioms, and related topics, the complete reference: a comprehensive strategy for safe and efficient memory management in c++ for high performance computing. Technical report, Sandia National Laboratories (2010)

[20] Edwards, H.C., Trott, C.R.: Kokkos: Enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (xsw 2013), IEEE (2013) 18–24

[21] Baker, C.G., Heroux, M.A.: Tpetra, and the use of generic programming in scientific computing. Scientific Programming **20**(2) (2012) 115–128

[22] Bavier, E., Hoemmen, M., Rajamanickam, S., Thornquist, H.: Amesos2 and belos: Direct and iterative solvers for large sparse linear systems. Scientific Programming **20**(3) (2012) 241–255

[23] Rajamanickam, S., Boman, E.G., Heroux, M.A.: Shylu: A hybrid-hybrid solver for multicore platforms. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE (2012) 631–643

[24] Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Volume 49. NBS Washington, DC (1952)

[25] Saad, Y., Schultz, M.H.: Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on scientific and statistical computing **7**(3) (1986) 856–869