Performance portability of a fluidized bed solver

V M Krushnarao Kotteda
Mechanical Engineering
University of Texas at El Paso
El Paso, USA
vkotteda@utep.edu

Vinod Kumar
Mechanical Engineering
University of Texas at El Paso
El Paso, USA
vkumar@utep.edu

William Spotz
Multiphysics Applications
Sandia National Laboratories
Albuquerque, USA
wfspotz@sandia.gov

Daniel Sunderland
Scalable Algorithms
Sandia National Laboratories
Albuquerque, USA
dsunder@sandia.gov

***Abstract*—Performance portability is a challenge for application developers as the source code needs to be executed and performant on various hybrid computing architectures. The linear iterative solvers implemented in most applications consume more than 70% of the runtime. This paper presents the results of a linear solver in Trilinos for fluidized bed applications. The linear solver implemented in our code is based on the Kokkos programming model in Trilinos, which uses a library approach to provide performance portability across diverse devices with different memory models. For large scale problems, the numerical experiments on Xeon Phi and Kepler GPU architectures show good performance over the results on Xeon (Haswell) computing architectures.**

***Index Terms*—performance, portability, fluidized bed, Kokkos, Trilinos, MFiX**

I. INTRODUCTION

Portability of scientific and engineering applications is complicated given the application programming interfaces, various programming models, and performance requirements. The performance of such applications with various programming models on various computing systems are different as the memory access, processor and shared memory architecture and other elements differ. In addition, a program optimized for one Co-processor/device may not be efficient or run on another Co-processor/device from a different vendor. For example, a program optimized for a Xeon Phi processor is different from the one optimized for a Kepler processor.

Programming models such as OpenMP, OpenACC, and OpenCL address the manycore parallelism but do not address the memory access patterns. Therefore, we choose the Kokkos [1] programming model in Trilinos as a tool for creating a performance portable linear solver. Trilinos [2]–[4], an open-source software framework, includes robust algorithms and enabling technologies for solving large-scale complex multi-physics problems besides others. It is developed at Sandia National Laboratories. As of the most recent release version 12.12, the library consists of approximately 60 packages including distributed-memory parallel linear algebra and solvers for large sparse linear systems. The modern linear solver packages in Trilinos offer portability and scalability. However, it leverages more than 100 third party libraries such as Boost, BLAS, CUDA, Hypre, LAPACK, Matlab, Metis, NetCDF, ParMetis, PETSc, ScaLAPACK and SuperLU. The library contains three different generations of packages. The first generation packages operate on Essential Petra [2] objects. The second-generation packages are a complete rewrite of the first generation packages to (i) solve problems with more than 2 billion unknowns, (ii) support new data types such as complex and extended precision, (iii) enable and simplify the addition of shared-memory parallelism (OpenMP/CUDA/Pthreads) to the distributed memory parallelism for many programming models [3]. These packages are based on Kokkos [1] computational kernels and operate on Templated Petra objects (Tpetra). The last generation packages are based on Kokkos programming model only.

There are few portable and performant multiphase solvers. Martineau et al. [5] evaluated emerging many-core parallel programming models including Kokkos for heat conduction problem. They observed that the performance of portable models is 5 to 20% slower than non-portable programming models. Carilli et al. [6] integrated Kokkos library with a FORTRAN based Cartesian CFD solver. It is observed that the integrated library improves the portability as well as the performance of the CFD solver.

We demonstrate that a linear solver based on Kokkos kernels can be executed on multicore, manycore and graphics processing units (GPU) and show good performance even without any architecture-specific optimizations. The paper is organized as follows. In Section 2, we present related work. Section 3 presents a brief overview of the linear iterative solvers in MFiX-Trilinos. Section 4 describes the hybrid computing architectures used for the performance evaluation tests. In Section 5, the performance evaluation of preconditioned BiCGStab method in MFiX-Trilinos application, an open source multi-phase flow solver, based on Kokkos node API is studied. The conclusions are presented in Section 6.

II. RELATED WORK

We describe CUDA, OpenMP, MPI and Kokkos programming models which provide portability across different computer architecture. In the present work, the linear solver code is developed with MPI and Kokkos programming models.

A. CUDA programming

CUDA (Compute Unified Device Architecture) is a parallel computing [7] architecture developed by NVIDIA. A heterogeneous computing system may contain both host/central processing units (CPUs) and massively parallel devices/graphics

processing units (GPUs). In CUDA programming, the computing system consists of a host such as Intel Xeon processor and devices such as Kepler K80. CUDA architectures support a range of computational interfaces including OpenCL. There are three CUDA specific keywords that specify the execution of a piece of code on host or device: *device, global and host*. The *device* and *global* keywords specify the execution on device while the *host* executes the code on the host. The three key abstractions possible through CUDA are a hierarchy of thread groups shared memories and barrier synchronization. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. These can be used to partition the problem into coarse sub-problems and then into finer pieces. This type of decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables transparent scalability since each sub-problem can be scheduled to be solved with the number of the cores available on a processor. Therefore, a compiled CUDA program can be executed on any number of processor cores, and only the runtime system needs to know the physical processor count. The CUDA kernel launch creates a grid of threads that execute the kernel function concurrently. The threads use a unique index, to identify the portion of the data to process.

B. OpenMP programming

OpenMP (Open Multi-Processing) [8], a shared memory architecture API [9], provides multi-threading capacity where a single core executes multiple processes or threads concurrently. A loop in a C++ program can be parallelized by invoking calls from OpenMP libraries and inserting the OpenMP compiler directives. Therefore, the threads get new tasks, the un-processed loop iterations, directly from local shared memory. OpenMP is an open specification for shared memory parallelism. The basic idea behind OpenMP is data-shared parallel execution. OpenMP is portable across the shared memory architecture. The unit of workers in OpenMP is the thread. It works well when accessing shared data costs nothing. Every thread can access a variable in the shared cache.

C. Pthreads programming

Pthreads programming model offers simple and portable way to express multithreading [10]. The "P" in Pthreads represents the POSIX (Portable Operating System Interface). The thread allows us to spawn a new concurrent process flow. Threads can be spawned by defining a function and it's arguments and are very effective on multi-processor systems. Threads require less overhead as the threads within a process share the same address space. MPI parallel programming model are used in a distributed computing environment while threads are limited to a single computer system.

D. MPI programming

Message Passing Interface (MPI) is a specification for message passing operations. It defines each worker as a process. MPI is currently the standard programming model for application on distributed memory architectures. It provides language bindings for C, C++, and FORTRAN languages. MPI [11] offers performance and functionality and includes point-to-point message passing and collective operations, all scoped to user-specified groups of processes. The present work uses Open MPI, an open source MPI implementation.

E. Kokkos

Kokkos [1], a software package and programming model, has been developed at Sandia National Laboratories and its objective is performance portability. It is widely used to develop performant and portable codes/software for scientific and engineering applications. It is designed to handle diverse devices and runs the applications on hybrid computing architectures. Kokkos computational kernels are executed in fine-grain data parallel within an execution space [1]. And, those kernels operate on multidimensional arrays residing in memory spaces and provide polymorphic data layouts similar to the Boost.MultiArray [12].

The Kokkos API isolates the application code from device-specific programming models [1]. This allows a choice of the most performant programming model for each manycore device and optimizes the programming model without impacting the code. The back-ends possible with Kokkos API are: Pthreads [13], OpenMP [14] and Cuda [15]. Pthreads or OpenMP back-ends can use the portable hardware locality library for explicit thread placement. Back-ends for Intel Xeon Phi coprocessor can use the self-hosted mode processes run exclusively on this "device" as opposed to using the offload model.

Kokkos separates Host memory space (CPU) from the Device memory space (NVIDIA GPUs, Intel Xeon Phi). Kokkos Device is an abstraction implemented as a C++ template parameter that specifies which memory space to use. Currently, this parameter can be CUDA for NVIDIA GPUs, OpenMP or Pthreads for Intel Xeon Phi. The integration of these abstractions enables the application code to satisfy multiple architecture specific memory access pattern performance constraints without having to modify the source code.

The basic changes of moving to Kokkos from C++ code is:

1) replace array allocation with Kokkos multidimensional arrays;
2) replace functions with Functors and run in parallel on CPUs;
3) enable dispatch for GPU execution;
4) optimize algorithms for threading;
5) specializes in kernels for specific architectures.

The implementation of Kokkos programming model in the present C++ code to solve various linear systems is similar to the procedure mentioned in [16].

III. MFiX-Trilinos

In the present work, MFiX [17] is integrated with the modern linear solvers, which are based on Kokkos programming model, in Trilinos [2] via an interface. MFiX contains

three different suites of models to simulate flow in fluidized beds: Two-fluid (TFM), Discrete element (DEM) and Particle in Cell (PIC) models. The first and the last models are Eulerian-Eulerian framework models whereas DEM is an Eulerian-Lagrangian framework model. In TFM, the fluid and solid phases are assumed to be the continuum and are modeled with separate sets of conservation equations with proper interaction terms. In this model, convergence problems occur when particles are not completely fluidized. In DEM, we track all the solid particles. In PIC model, the fluid is assumed to be continuum while using 'parcels' to represent groups of particles with similar characteristics. This model offers reduced computational cost over DEM. It is noticed that the framework to simulate the solid phase is different on different models. However, the fluid phase exists in the Eulerian framework in all the three models. Therefore, in the present study, the linear system of equations for the fluid phase is solved with an iterative method in MFiX and MFiX-Trilinos. The equations for the solid phase are solved with a built-in solver in MFiX. Therefore, MFiX-Trilinos can be used to simulate a fluid using any of three models in MFiX. In the present study, we considered TFM model to simulate flow in fluidized beds. However, the integrated solver can be used to solve the system of equations for the fluid or solid or in both phases. The current version of MFiX lacks advanced iterative solvers. Therefore, the integration of those solvers in Trilinos with MFiX can improve the capabilities of MFiX and enable the solution of larger and more complex problems.

MFiX is written in procedural Fortran and cannot be directly integrated with Trilinos libraries which are object-oriented, and the memory layouts in C++ and Fortran are different. Hence, an interface is required to exchange the information across the Software and the library. Therefore, we developed an interface that consists of MFiX, Fortran, C and C++ wrappers to integrate MFiX with Trilinos. In the present study, the interface is tested on various computer architectures. In general, this interface can be applied to integrate software and libraries written in Fortran, C, or C++ [18], [19]. The MFiX wrapper is a Fortran code that understands the structure of the linear system of equations from MFiX. Arrays required for forming the linear system of equations are passed to the Fortran Wrapper. The Fortran wrapper transfers the arrays to the C wrapper. The C wrapper acts as a mediator and changes the memory layouts of the arrays in Fortran to C++. The linear system constructed in the C++ wrapper is solved with Jacobi preconditioned Krylov methods such as CG, GMRES, and BiCGStab.

IV. EVALUATION ENVIRONMENT

We evaluated the performance of preconditioned CG methods based in Kokkos kernels on Hansen at Sandia National Laboratories and Stampede2 at Texas Advanced Computing Center clusters. Stampede2 is used for Intel Xeon Phi tests, whereas Hansen is used for Xeon (Haswell) as well as NVIDIA Kepler tests. Testbed configuration details are listed in Table I. The device in these configurations refers to a single Xeon Phi or a single Kepler GPU, respectively. Results presented in this paper are for pre-production Intel Xeon Phi processors (Knights Landing) and pre-production NVIDIA Kepler co-processor.

TABLE I
CONFIGURATIONS OF TESTBED CLUSTERS

Name	Hansen	Stampede2
Nodes	3	32
CPU	2 Xeon E5-2698 2.30GHz HT-on	Xeon Phi 7250 1.4GHz HT-on
Co-Processor	K80m	
Interconnect	FDR IB	OPA
Memory	512 GB	96GB DDR4 + 16GB MCDRAM
Compiler	gcc 4.9.3 + CUDA 8.0	ICC 17.0.4
MPI	open MPI 1.10.6	Intel MPI 2017 3
OS	RedHat 6.5	CentOS 7.3

V. PERFORMANCE EVALUATION RESULTS

Performance evaluation results for the Kokkos implementation of the code to solve various sparse linear systems are presented in Figure 1. The Kokkos backend node type is passed to `Tpetra MatrixMarket Reader` to read the sparse matrix from a file. In this code, the sparse matrix A is read from a matrix-market file, `matrixFileName`. The `comm` initializes the communication among the MPI processors. Therefore, the matrix is distributed among the processors. A preconditioner, which approximately equals to A, is formed with `Ifpack2` package in Trilinos. The parameters which are required to set the preconditioner are specified in `ifpack2par` XML file. The left-hand side (x) and right-hand side (b) vectors of the linear system are formed via the row map of the sparse matrix A. The entries of b are made with randomizing method in `Tpetra Multivector`. The linear problem is set with these `Tpetra` objects (A, x and b). And, the problem is solved with an iterative method in `Belos` package. The solver settings are passed to the `solver` method via `belosList`. The `solve` method in `Belos` solves the linear system and finds the unknown vector x. The pseudo code to solve the linear system for a fluidized bed problem is similar to the code shown in Fig. 1. In the fluidized bed problem, the non-zero entries in the matrix from MFiX are populated via `InsertGlobal` and `FillComplete` methods in `Tpetra CrsMatrix` class. The right hand side vector is filled via `InsertGlobal` method. The steps for formation of the linear system and solving that with a preconditioned method are same as in Fig. 1.

First, we studied the portability of a linear solver which is based on Kokkos programming model on different architectures (NVIDIA K80 GPU, Intel Xeon Phi, and Intel Xeon) for various sparse matrix systems [20]. The application code is same for different programming models. However, the configuration settings for Kokkos library is different on different computer architectures. Next, we studied the portability of a linear solver in MFiX-Trilinos which is based on Kokkos node

```
typedef KokkosClassic::DefaultNode::
DefaultNodeType node_type;

node_type node(defaultParameters);
sparse_mat_type A (matrixFileName, comm, node);

multivector_type x(A->getRowMap(), 1);
multivector_type b(A->getRowMap(), 1);
b->randomize();

linear_problem_type Problem(A, x, b));
Problem->setProblem();

RCP<belos_solver_manager_type> solver;

prec_type Preconditioner (prectype, A);
Preconditioner->setParameters(ifpack2par);
Preconditioner->initialize();
Preconditioner->compute();

Problem->setRightPrec(Preconditioner);
belos_stdcg_manager solver((Problem, belosList));

solver->solve();
```

Fig. 1. A pseudo code to solve a linear system with CG solver in Belos along with a preconditioner in Ifpack2. The sparse matrix is obtained from the SuiteSparse Matrix Collection [20].

TABLE II
MATRICES USED FOR NUMERICAL EXPERIMENTS. N IS THE NUMBER OF ROWS, NNZ IS THE NUMBER OF NONZEROS. THE SYMMETRIC LINEAR SYSTEM IS REPRESENTED WITH sym WHILE THE UNSYMMETRIC ONE IS REPRESENTED WITH $unsym$.

Matrix	N	NNZ	NNZ/N	Type
af_3_k101	503,625	17,550,675	34.8	
af_shell3	504,855	17,562,051	34.8	
audikw_1	943,695	77,651,847	82.3	
bone010	986,703	47,851,783	48.5	
Bump_2911	2,911,419	127,729,899	43.9	
ecology2	999,999	4,995,991	5.0	
Flan_1565	1,564,794	114,165,372	73.0	
G3_circuit	1,585,478	7,660,826	4.8	sym
Geo_1438	1,437,960	60,236,322	41.9	
Hook_1498	1,498,023	59,374,451	39.6	
ldoor	952,203	42,493,817	44.6	
Queen_4147	4,147,110	316,548,962	76.3	
Serena	1,391,349	64,131,971	46.1	
StocF-1465	1,465,137	21,005,389	14.3	
thermal2	1,228,045	8,580,313	7.0	
atmosmodd	1,270,432	8,814,880	6.9	
atmosmodj	1,270,432	8,814,880	6.9	
atmosmodl	1,489,752	10,319,760	6.9	unsym
atmosmodm	1,489,752	10,319,760	6.9	
circuit5M-dc	3,523,317	14,865,409	4.2	
Freescale1	3,428,755	17,052,626	5.0	

API on different computer architectures for 3D a fluidized bed problem.

A. Sparse Matrix collection

We studied the portability of the CG linear solver with Jacobi relaxation preconditioner. First, we solved various symmetric linear systems. The symmetric matrices in the linear systems from the sparse matrix collection [20] are read via `Tpetra` Matrix reader. The linear system formed with the `Tpetra` objects is solved with CG solver in `Belos`. The preconditioner used along with the linear solver is Jacobi. The symmetric matrices considered in this study are listed in Table II. The "Hook_1498" matrix contains the maximum number of equations (N), whereas "Queen_4147" has the maximum number of nonzeros in the sparse matrix (NNZ). The CG solve time with fully MPI only, MPI + Kokkos-OpenMP, and MPI + Kokkos-CUDA programming models are shown in Fig.2. The solver with Kokkos-OpenMP as well as Kokkos-CUDA backend programming models performs better compared to that with fully MPI only. This can be seen in the figure. The Kokkos-CUDA backend solver is relatively fast compared to the Kokkos-OpenMP backend solver. Further, we considered one Xeon Phi processor and compared the solver time with MPI processors and threads. And, it is observed that the CG solve time for a number of MPI processors and the same number of OpenMP threads are in good agreement. We also compared the performance of CG solver time with Kokkos-OpenMP and Kokkos-Pthreads programming models. A good agreement between the solver times with these two programming models is observed.

We also considered the non-symmetric matrices [20] to form the linear system. "circuit5M-dc" sparse matrix contains the maximum number of equations as well as maximum non zero rows among the other matrices considered in this study. The procedure followed to solve the linear system is same that for the symmetric linear systems. The solver time with fully MPI only programming model, MPI + Kokkos-OpenMP programming models and MPI + Kokkos-CUDA programming models are shown in Fig. 3. The solver with MPI + Kokkos-CUDA and MPI + Kokkos-OpenMP programming models are relatively fast compared to the solver with the MPI/Kokkos-Serial backend.

B. 3D Fluidized bed problem

Fluidized beds are used in various applications such as chemical [21], [22], mineral [23], [24], pharmaceutical [25]–[27], petroleum [28]–[30] and fossil fuel power plants [31]. For example, gasification of a feedstock in a fluidized bed increases the efficiency of the fossil fuel plant and reduces the greenhouse gases. The flow in fluidized beds is multi-physics in nature. Therefore, understanding of such complex flow is still limited. The existing computer applications lack the capabilities for designing, optimizing, and controlling of industrial-scale fluidized bed reactors. Sophisticated models can improve the capabilities of such applications. To improve the performance as well as portability of MFiX (Multiphase Flow with Interphase eXchanges), it is integrated with state-of-the-art linear solvers in Trilinos.

The fluidized bed with the central jet in three-dimensions is simulated via MFiX-Trilinos. The fluidized bed has width of $10cm$, height of $100cm$, and thickness of $10cm$. The computational domain is discretized with 100 equi-spaced cells in the axial direction and 10 equi-spaced cells in the normal direction. The number of cells uniformly distributed in the widthwise direction is 10. The velocity of the jet is

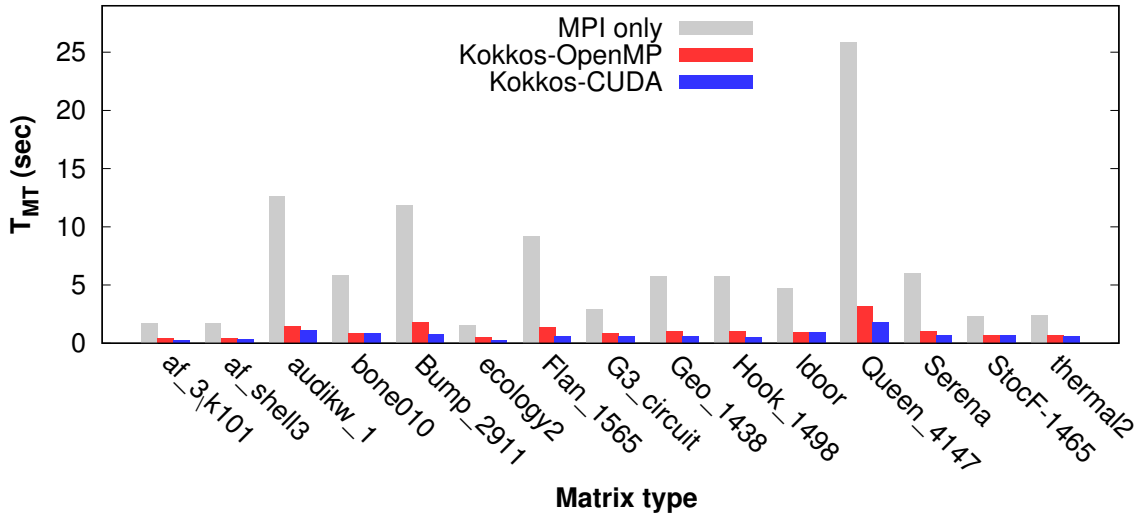

Fig. 2. Solve time of preconditioned CG method with various programming models on different computer architectures for solving various symmetric linear systems [20].

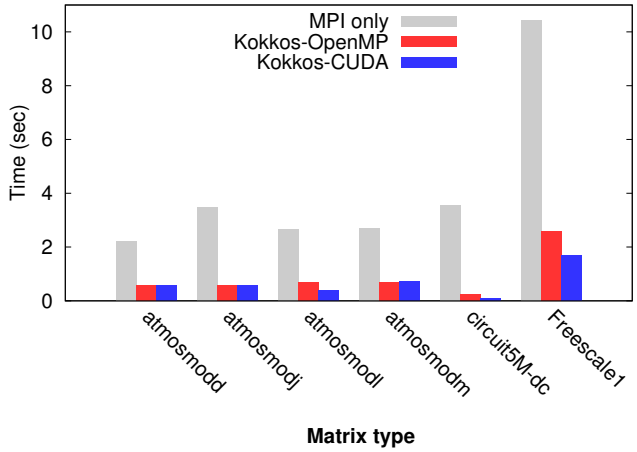

Fig. 3. Solve time of preconditioned CG method with various programming models on different computer architectures for solving various non-symmetric linear systems [20].

$61.7cm/s$ while the velocity of the fluid from the distributor plate is $25.9cm/s$. Constant mass flow and pressure boundary condition are specified on the bottom and top boundary, respectively. Initially, the half of the bed is filled with sand particles of size $0.04cm$ and density of $2g/cm^3$. The void fraction at minimum fluidization of the bed is 0.42. The sand particles are fluidized with the air density of $0.0012g/cm^3$ and viscosity of $0.00018Poise$. The flow is assumed to be laminar as well as incompressible. The equations governing the fluid in the Cartesian coordinate system are solved with MFiX-Trilinos. The linear system of equations is solved with the BiCGStab method. The validation of the flow in the fluidized bed from MFiX-Trilinos can be found in [19]. The flow with MFiX and MFiX-Trilinos are qualitatively same. The wall clock time to solve the problem size of 1M with various Jacobi preconditioned iterative methods on various computer architectures is shown in Fig. 4. For a programming model, the CG solver is relatively fast compared to BiCGStab and GMRES solvers. However, GMRES solvers are relatively fast compared to BiCGStab solvers for Kokkos-Serial as well as Kokkos-OpenMP programming models. For an iterative method, fully MPI only programming solvers are relatively fast compared to that with MPI + Kokkos-CUDA programming models. We will investigate this further in the future. However, Kokkos-OpenMP backend CG/BiCGStab solvers are relatively fast compared to fully MPI only and Kokkos-CUDA backend CG/BiCGStab solvers.

Further, we solved the flow in the fluidized of different sizes with the BiCGStab solver. Fig. 5 shows the variation of solver time with problem size for different programming models. For the problem size $<$ 5M, the fully MPI only model solver is relatively fast compared to that with hybrid programming models (MPI + Kokkos-OpenMP/CUDA). However, the Kokkos-OpenMP backend solvers performance is comparable with the fully MPI only solvers. It is observed that the hybrid solvers are relatively fast compared to fully MPI only programming solver for the problem size of $>$5M. This could be due to efficient utilization of host and device in the computational environment. Therefore, we considered MPI + Kokkos-OpenMP backend BiCGStab solver to solve the linear system of size 20M. Fig. 6 shows the variation of solver time with number of processors and threads on stampede2. As expected, the solver time decreases with an increase in number of processors as well as the threads. For a fixed number of MPI processors, the solver time decreases with an increase in the number of threads and it is minimum for the number of threads per processor is 4.

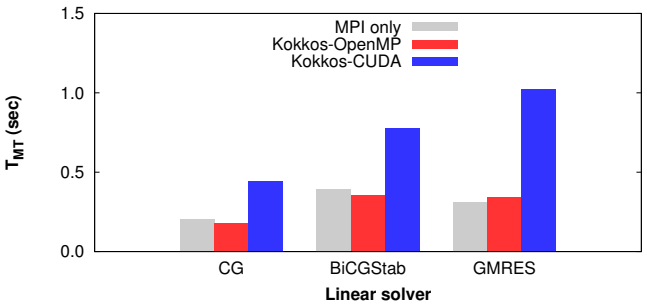

Fig. 4. Flow in the 3D fluidized bed: Jacobi preconditioned solvers time on various computer architectures for the problem size of 1M.

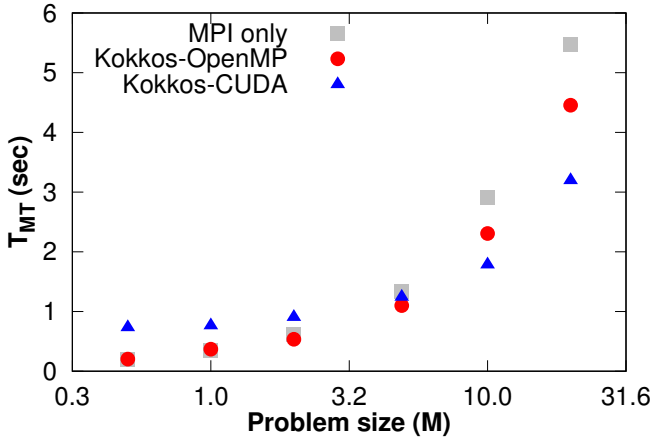

Fig. 5. Flow in the 3D fluidized bed: the time spent in solving the linear system of different sizes with Jacobi precondictioned BiCGStab solver.

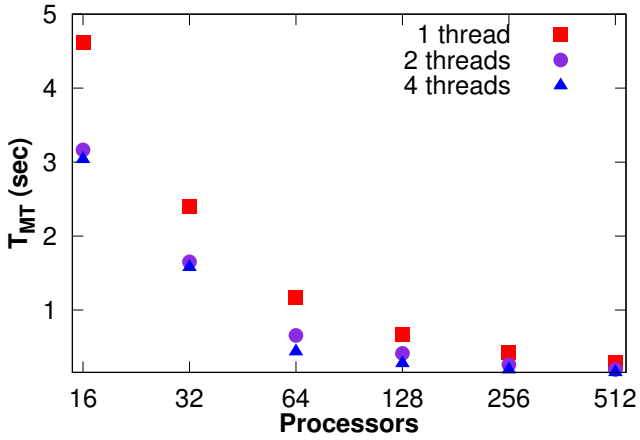

Fig. 6. Flow in the 3D fluidized bed: the time spent in solving the linear system of size 20M with different number of processors and threads on Stampede2.

VI. Conclusions

We studied the performance and portability of a linear solver which is based on Kokkos API on various hybrid parallel architectures. The Kokkos implementation is a single code which is performant on diverse HPC architectures. The Kokkos refactoring strategy is used to create a thread parallel and portable version of the code.

We solved the various symmetric linear system with Jacobi preconditioned CG solver. The symmetric matrices in the linear systems from the Sparse matrix collection [20] is read via `Tpetra Matrix reader`. The solver time with various programming models is compared. It is observed that the solver with Kokkos-OpenMP as well as Kokkos-CUDA backend programming models performs better compared to that with fully MPI only model. The Kokkos-CUDA backend solver is relatively fast compared to the Kokkos-OpenMP backend solver. We also compared the performance of CG solver with Kokkos-OpenMP and Kokkos-Pthreads programming models. A good agreement between the solver times with these two programming models is observed. Further, non-symmetric matrices from the matrix collection [20] are considered as well to form the linear system. The solver time with MPI + Kokkos-CUDA and MPI + Kokkos-OpenMP programming models are relatively fast compared to the solver time with fully MPI only model.

Further, we solved the flow in the fluidized of different sizes with Jacobi preconditioned BiCGStab solver. For the problem size $<$ 5M, the fully MPI programming model solver is relatively fast compared to that with hybrid programming models (MPI + Kokkos-OpenMP/CUDA). However, the Kokkos-OpenMP backend solvers performance is comparable with the fully MPI only solvers. The hybrid solvers are relatively fast compared to fully MPI only programming model solver for the problem size of $>$ 5M. This could be due to efficient utilization of host and device in the computational environment. Therefore, we considered Kokkos-OpenMP backend BiCGStab solver for the strong scaling of problem size 20M. The solver time decreases with an increase in the number of processors as well as the threads per processor. The solver time is low for the number of processors = 512 and threads per processors is 4.

Acknowledgement

This work is supported by the U.S. Department of Energy (DOE) National Energy Technology Laboratory (NETL) under Grant Number $DE-FE\_0026220$, XSEDE computational resources including Stampede2 and Sandia National Laboratories, New Mexico. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access

patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[2] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley, "An overview of the Trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[3] P. Lin, M. Bettencourt, S. Domino, T. Fisher, M. Hoemmen, J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, and S. Kennon, "Towards extreme-scale simulations for low Mach fluids with second-generation Trilinos," *Parallel processing letters*, vol. 24, no. 04, p. 1442005, 2014.

[4] M. A. Heroux, "Software challenges for extreme scale computing: Going from petascale to exascale systems," *The international journal of high performance computing applications*, vol. 23, no. 4, pp. 437–439, 2009.

[5] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An evaluation of emerging many-core parallel programming models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'16. New York, NY, USA: ACM, 2016, pp. 1–10.

[6] M. F. Carilli, N. L. Mundis, and V. Sankaran, "Acceleration of Real Gas Physics Routines on Parallel Architectures using Kokkos," in *23rd AIAA Computational Fluid Dynamics Conference*, ser. AIAA AVIATION Forum. American Institute of Aeronautics and Astronautics, Jun. 2017.

[7] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. Buijssen, M. Grajewski, and S. Turek, "Exploring weak scalability for fem calculations on a GPU-enhanced cluster," *Parallel Computing*, vol. 33, no. 10, pp. 685–699, 2007.

[8] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008, vol. 10.

[9] P. Alonso, R. Cortina, F.-J. Martínez-Zaldívar, and J. Ranilla, "Neville elimination on multi-and many-core systems: Openmp, mpi and cuda," *The Journal of Supercomputing*, vol. 58, no. 2, pp. 215–225, 2011.

[10] B. Nichols, D. Buttlar, J. Farrell, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[11] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[12] B. Schäling, *The boost C++ libraries*. Boris Schäling, 2011.

[13] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[14] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[15] C. Nvidia, "Programming guide," 2010.

[16] I. Demeshko, H. C. Edwards, M. A. Heroux, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, and C. R. Trott, "Towards architecture aware performance portable finite element code." Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2014.

[17] M. Syamlal, W. Rogers, and T. J. OBrien, "MFIX documentation: Theory guide," *National Energy Technology Laboratory, Department of Energy, Technical Note DOE/METC-95/1013 and NTIS/DE95000031*, 1993.

[18] V. M. K. Kotteda, A. Chattopadhyay, V. Kumar, and W. Spotz, "A framework to integrate MFiX with Trilinos for high fidelity fluidized bed computations," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sept 2016, pp. 1–6.

[19] V. M. K. Kotteda, A. Chattopadhyay, V. Kumar, and W. Spotz, "Next-Generation Multiphase Flow Solver for Fluidized Bed Applications," no. 58066, p. V01CT16A013, 2017.

[20] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[21] Y. Cheng, J. Jae, J. Shi, W. Fan, and G. W. Huber, "Production of renewable aromatic compounds by catalytic fast pyrolysis of lignocellulosic biomass with bifunctional ga/zsm¡90¿5 catalysts," *Angewandte Chemie*, vol. 124, no. 6, pp. 1416–1419, 2011.

[22] J. G. Yates and P. Lettieri, *Introduction*. Cham: Springer International Publishing, 2016, pp. 1–21.

[23] A. J. Minchener, "Coal gasification for advanced power generation," *Fuel*, vol. 84, no. 17, pp. 2222 – 2235, 2005.

[24] K.-C. Xie, *Coal Gasification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 181–241.

[25] W. R. Ketterhagen, "Modeling the motion and orientation of various pharmaceutical tablet shapes in a film coating pan using dem," *International journal of pharmaceutics*, vol. 409, no. 1-2, pp. 137–149, 2011.

[26] J. Rantanen and J. Khinast, "The Future of Pharmaceutical Manufacturing Sciences," *Journal of Pharmaceutical Sciences*, vol. 104, no. 11, pp. 3612–3638, Nov. 2015.

[27] R. Šibanc, M. Turk, and R. Dreu, "An analysis of the mini-tablet fluidized bed coating process," *Chemical Engineering Research and Design*, vol. 134, pp. 15 – 25, 2018.

[28] I. Babich and J. Moulijn, "Science and technology of novel processes for deep desulfurization of oil refinery streams: a review," *Fuel*, vol. 82, no. 6, pp. 607 – 631, 2003.

[29] L. C. Castaeda, J. A. Muoz, and J. Ancheyta, "Current situation of emerging technologies for upgrading of heavy oils," *Catalysis Today*, vol. 220-222, pp. 248 – 273, 2014.

[30] A. Stanislaus, A. Marafi, and M. S. Rana, "Recent advances in the science and technology of ultra low sulfur diesel (ulsd) production," *Catalysis Today*, vol. 153, no. 1, pp. 1 – 68, 2010.

[31] J. Ruiz, M. Jurez, M. Morales, P. Muoz, and M. Mendvil, "Biomass gasification for electricity generation: Review of current technology barriers," *Renewable and Sustainable Energy Reviews*, vol. 18, pp. 174 – 183, 2013.