# PERFORMANCE PORTABLE HPCG

ZACHARY A. BOOKEY*, IRINA P. DEMESHKO†, SIVASANKARAN RAJAMANICKAM‡, AND MICHAEL A. HEROUX§

**Abstract.** The High Performance Conjugate Gradients Benchmark is an international project to create a more appropriate benchmark test for the world's largest computers. The current LINPACK benchmark, which is the standard for measuring the performance of the top 500 fastest computers in the world, is moving computers in a direction that is no longer beneficial to many important parallel applications. In this project we are developing a version of HPCG, using the Kokkos package found in Trilinos, that can be optimally executed across several distinct high performance computing architectures. This new code demonstrates an efficient programming approach that can be adopted by other programmers to write portable high performance software.

**1. Introduction.** After generations of using the High Performance Linpack (HPL), or Top 500, benchmark [1] to measure the performance of large computers it became necessary to use another benchmark to help better the direction that super computers were headed to more accurately reflect the types of applications that these machines were running.

HPL is a simple program that factors and solves a large dense system of linear equations using Gaussian Elimination with partial pivoting. While dense matrix - matrix multiplication and related kernels are commonly used in scientific applications, they are not representative of all the operations usually performed in scientific computing: computations with lower computation-to-data-access ratios(computational intensity) and with irregular memory access are also very common.

The High Performance Conjugate Gradient (HPCG)[5][4] was created to fill the gap that HPL had created. HPCG uses a preconditioned conjugate gradient to solve a system of equations, that executes both dense computations with high computational intensity and computations with low computational intensity such as sparse matrix-matrix multiplications.

Original HPCG benchmark doesn't exploit full parallelism available on existing Supercomputers which makes it unfair to use it for performance measurement. The goal of our project was to create a performance portable version of HPCG that gives reasonable performance on all existing supercomputers. We choose Kokkos[2] library from Trilinos[3] as a tool to provide performance portability in HPCG code.

**2. HPCG.** HPCG is a new and upcoming benchmark test to rank the worlds largest computers. On top of solving a large system of equations, HPCG also features a more irregular data access pattern so that data access affects results as well as matrix computations.

HPCG begins by creating a symmetric positive definite matrix and it's corresponding multi grid to be used in the preconditioning phase. For the preconditioner it uses a Symmetric Gauss-Seidel forward sweep and back sweep to solve the lower and upper triangular matrices. For the actual solve of $Ax = b$, HPCG uses the conjugate gradient method after the preconditioning phase. HPCG runs in seven major phases.

1. **Problem Setup:** This is the beginning of HPCG and is where we construct the geometry that is used to generate the problem. HPCG generates a sym-

*Saint John's University, zabookey@csbsju.edu
†Sandia National Laboratories, ipdemes@sandia.gov
‡Sandia National Laboratories, srajama@sandia.gov
§Sandia National Laboratories, maherou@sandia.gov

metric, positive definite, sparse matrix with up to 27 nonzero entries per row depending on the local location of the row.

2. **Validation Testing:** This portion of the program is to make sure any changes made produce valid results. Specifically it checks to make sure that both the unpreconditioned and preconditioned conjugate gradient converge in around 12 and 2 iterations respectively. It also makes sure that after performing both a sparse matrix vector multiplication and a symmetric Gauss-Seidel sweep that we preserve symmetry by using two pseudorandomly filled vectors and performing simple operations that should be zero due to the nature of our symmetric matrix A.
3. **Reference Sparse Matrix Vector Multiplication and Multigrid Timing:** This portion of the code times how long it takes to perform the reference versionts of SPMV and Symmetric Gauss-Seidel.
4. **Reference Conjugate Gradient Timing:** Here we run 50 iterations of the reference version of the conjugate gradient method and record the resulting residual. This residual must be attained by the optimized version of conjugate gradient no matter how many iterations are required.
5. **Optimized Conjugate Gradient Setup:** Runs one set of the optimized conjugate gradient and determines the number of iterations required to reach the residual found before. Then figures out how mamy times to reach the desired residual to fill in the requested benchmark time.
6. **Optimized Conjugate Gradient Timing:** Runs the optimized conjugate gradient the required amount of times. Records time for each timed section to report out later.
7. **Report Results:** Writes out log files for debugging and creates the .yaml file to display the results which can then be submitted if all the requirements are met.

HPCG gives you the option to run with MPI, OpenMP, both, or in serial. Running with MPI adds an extra dimension to the problem and requires processes to exchange values on their borders to perform. This results in a tradeoff between more overhead and more parallelism.

**3. Kokkos.** As different computer architectures are better with certain applications than others it has become increasingly difficult to write code that will perform well across many different types of architectures. One solution to this problem is the C++ package, Kokkos. Kokkos acts as a wrapper around your code to allow you to specify at compile time where and how you want to run your application. Kokkos executes computational kernels in fine-grain data parallel within an `Execution space`. Currently Kokkos supports the following `Execution spaces`:

- `Serial`
- `PThreads`[6]
- `OpenMP`[8]
- `Cuda`[7]

Kokkos has two main features: `Kokkos::View` polymorphic Multidimensional Arrays and parallel dispatch. `Kokkos::View` is essentially a wrapper around an array of data that gives you the option to specify which execution space you want to store the data on and allows you to choose what sort of memory access traits you wish this data to have. `Kokkos::View` also handle their own memory management via reference counting so that the view automatically deallocates itself when all of the variables that reference it go out of scope, thus making memory management much

simpler across multiple devices.

There are three main parallel dispatch operations in Kokkos: `parallel_for`, `parallel_reduce`, and `parallel_scan`. All of these serve their own purpose and act as wrappers over how you would execute a section of code in parallel over the respective execution space. For all of the data parallel executions kernels you initiate the kernel by passing in a functor that performs the desired parallel operation, as of host to device.

`Parallel_for` is simply a generic for loop that will run all of the context of the loop in parallel. This works well for parallel kernels like vector addition. Parallel_reduce is for simultaneously updating a single value, this function guarantees that you avoid race conditions with the updated values. `Parallel_reduce` works well for parallel kernels like finding the dot product of two vectors. `Parallel_scan` is for taking a view and creating a running sum of values to replace the values of the view. Although `parallel_scan` is useful it was only really needed for setup phases in our HPCG.

Kokkos allows for nested parallelism that involves creating a league of teams of threads. With this tool, a developer could launch a `parallel_reduce` kernel that uses a `parallel_for` to update some value that later gets added to the value that was initially included to be reduced. Although this has not been implemented in our project yet, there are places in our code that could and should benefit from nested parallelism and thus we intend to include it at a later time.

**4. HPCG + Kokkos.** The goal for our project was to create a version of HPCG that produces valid results across many architectures without sacrificing performance. We believe that Kokkos library is the best available tool to provide performance portability for the C++ code, we choose to refactor HPCG to use it.

General strategy for Kokkos re-factoring includes:

- replacement custom data types with Kokkos multidimensional arrays;
- replacement the parallel loops with Kokkos parallel kernels;
- code optimizations to avoid an unnecessary communication overheads and improve usage of co-processors.

**4.1. Replacement custom data types with Kokkos multidimensional arrays.** Restructuring the code involved a whole rewrite of HPCG to change how all of the structs stored their values. We replaced every array that would be used in a parallel kernel with an appropriate view. Once this was functional we had to go back to some of the compute algorithms and change how the data was accessed as to not try to access device data from the host or the other way around.

While restructuring we decided to change how our SparseMatrix stored the data and implemented it as a sort of overlying structure on top of a Kokkos CSRMatrix. This change required us to again go back and change how most of our computational kernels worked and created a noticeable increase in performance. At this point the code was functional across all of Kokkos execution spaces but took a severe performance hit while trying to run on Cuda.

**4.2. Replacement the parallel loops with Kokkos parallel kernels.** Rewriting the parallel kernels involved replacing the parallel loops with the correct type of Kokkos parallel kernel. This part of re-factoring was heavily focused on converting the computation algorithms into functors and lambdas. Completing this task didn't affect portability at all and due to some Kokkos restrictions actually caused a slight reduction of performance in the function ComputeResidual. Other kernels would later

have to be changed to accommodate the fact that data was stored on a device but was being run on the host.

Computing the preconditioner using a symmetric Gauss-Seidel was initially done in serial and thus moving to Kokkos required us to copy memory from the device to the host every time we ran it, which was the reason performance was lost. We tried implementing many different ways to perform a sweep of symmetric Gauss-Seidel in parallel to eliminate the need of copying data. We implemented an inexact solve, a level solve, and a coloring algorithm.

The inexact solve works by performing a few triangular matrix solves. We split our matrix $A$ into parts $L$, $U$, and $D$ where $L$ is the lower triangular matrix including the diagonal, $U$ is the upper triangular matrix including the diagonal and $D$ is the diagonal. The inexact solve is done by solving three different equations. Given our problem $Ax = b$, to simulate a symmetric Gauss-Seidel sweep we solve $Lz = b$ then $Dw = z$ and finally $Ux = w$. All of these solves are done using a parallel Jacobi solve. Although this version of symmetric Gauss-Seidel is run in parallel and avoid all device to host memory copies it still performs slowly since each Jacobi solve needs to be run a certain amount of time depending on problem density and results in more parallel kernels being launched than is ideal.

Our level solve algorithm splits our matrix $A$ into parts $L$, $U$, and $D$ where $L$ is the lower triangular part of $A$ that includes the diagonal, $U$ is similar to $L$ since it is just the upper triangular part of $A$ that includes the diagonal. $D$ is just the diagonal of A. For this algorithm we introduce another data structure, levels, to our SparseMatrix that stores all of the data needed for sorting the matrix. When we optimize the problem we find dependencies in solving for $L$ and sort based on those dependencies in a way that a row will only be solved if all of it's dependencies have been solved. We repeat this process for solving for $A$ and store all of this data into levels. Now when we compute our Symmetric Gauss-Seidel we solve just like we do for the inexact solve except we start by solving for the rows in level 1 in parallel and then the rows in level 2 in parallel and repeat until we solve all of our levels. In the end we have a Symmetric Gauss-Seidel that has introduced a deal of parallelism and performs well compared to the original implementation.

At the time of writing our coloring algorithm does not work. We are in the process of debugging and determining what a good fix will be. The algorithm works by coloring our matrix $A$ so that all rows with a certain color have no dependencies on one another. This way we can run a sweep of symmetric Gauss-Seidel in parallel over each color. While similar to the level solve, this method requires more iterations to converge due to the fact that although no two rows in the same color depend on each other it is likely that they depend on a row in a color that will be solved in a later iteration.

### 4.3. code optimizations to avoid an unnecessary communication overheads and improve usage of co-processors.

**5. Performance evaluation result.** We evaluate performance for our implementation of the HPCG code on our Compton and Shannon testbed clusters. Compton is used for Intel Xeon and Intel Xeon Phi tests, and Shannon is used for NVIDIA Kepler (K20x) tests. Testbed configuration details are given in Table 5.1. Note that in these configurations device refers to a dual socket Xeon node, a single Xeon Phi, and a single Kepler GPU respectively. Results presented in this paper are for pre-production Intel Xeon Phi co-processors (codenamed Knights Corner) and pre-production versions of Intel's Xeon Phi software stack. Performance and configuration

of the co-processors may be different in final production releases.

| Name | Compton | Shannon |
|---|---|---|
| Nodes | 32 | 32 |
| CPU | 2 Intel E5-2670 HT-on | 2 Intel E5-2670 HT-off |
| Co-Processor | 2 Intel Xeon Phi 57c 1.1 GHz | 2 K20x ECC on |
| Memory | 64 GB | 128 GB |
| Interconnect | QDR IB | QDR IB |
| OS | RedHat 6.1 | RedHat 6.2 |
| Compiler | ICC 13.1.2 | GCC 4.8.2+CUDA 6.5 |
| MPI | IMPI 4.1.1.036 | MVAPICH2 1.9 |
| Other | MPSS 2.1.3653-8 | Driver: 319.23 |

TABLE 5.1
*Configurations of testbed clusters*

Performance results coming soon!

Comparing different variations of the preconditioner requires us to consider different problem sizes since some of our implementations will perform better on larger more sparse matrices than they will on smaller and denser matrices.

As you can see in Table 5.2 the serial version of symmetric Gauss-Seidel performs best since the level solve and the inexact solve only have limited parallelism. This is due to the fact that there aren't enough non-dependent rows to provide sufficient parallelism on each level and the inexact solve needs 18 iterations to get a solution for each triangular matrix that is close enough to the exact solution to maintain symmetry.

| | Serial SYMGS | Level Solve | Inexact Jacobi Solve |
|---|---|---|---|
| Cuda (Shannon) | 0.408776 | 0.0634977 | 0.091606 |
| OpenMP (Shannon) | 1.67202 | 0.233525 | 0.295145 |
| Serial (Shannon) | 1.46676 | 0.22152 | 0.269744 |

TABLE 5.2
*SYMGS on Problem Size* $16^3$

As seen in Table 5.3 the level solve begins to become the optimal preconditioner. This is similar to the issue with size $16^3$ where now our matrix is large and sparse enough that the level solve starts to see an increased amount of parallelism. The inexact method still lags behind since even though our matrix is less dense it still needs 12 Jacobi iterations to have a solution exact enough to pass the symmetry test.

| | Serial SYMGS | Level Solve | Inexact Jacobi Solve |
|---|---|---|---|
| Cuda (Shannon) | 0.64954 | 0.720434 | 0.560632 |
| OpenMP (Shannon) | 1.76462 | 2.42687 | 0.97734 |
| Serial (Shannon) | 1.45816 | 1.56839 | 0.485523 |

TABLE 5.3
*SYMGS on Problem Size* $64^3$

Table 5.4 shows us still that level solve is our most optimal preconditioner for similar reasons as stated before. However it makes sense to note that the inexact solve is steadily increasing performance as we our problem size increases. With a problem size of $128^3$ we only need 5 jacobi iterations to achieve a near exact solution to each triangular solve.

| | Serial SYMGS | Level Solve | Inexact Jacobi Solve |
|---|---|---|---|
| Cuda (Shannon) | 0.660526 | 2.1373 | 1.39231 |
| OpenMP (Shannon) | 1.82678 | 4.19081 | 1.84878 |
| Serial (Shannon) | 1.49575 | 2.18794 | 0.810011 |

TABLE 5.4
*SYMGS on Problem Size* $128^3$

Coming back to Table 5.5 later. DON'T FORGET...

| | Serial SYMGS | Level Solve | Inexact Jacobi Solve |
|---|---|---|---|
| Cuda (Shannon) | 0.659346 | 2.89654 | ??? |
| OpenMP (Shannon) | 1.842 | 4.858 | ??? |
| Serial (Shannon) | 1.5048 | 2.37896 | ??? |

TABLE 5.5
*SYMGS on Problem Size* $192^3$

Now we're going to look at how each preconditioning algorith performs overall on each execution space.

As seen in Fig 5.1 the vanilla symmetric Gauss-Seidel is dominant on the non-cuda execution spaces. This has to do with the fact every time this method is called we had to run a lot of memory copies between the host and device. When we aren't using Cuda all of our memory is located in one space and thus the memory copying is avoided and we don't see a huge performance hinderance.

In Fig 5.2 the results between execution spaces while using the level solve preconditioning method vary quite a bit. First, notice that there is a trend with the results and it appears that our max performance received from this preconditioner starts to taper off. With this trend in mind it appears that OpenMP will acheive better performance using this preconditioner than Cuda. However this could be because we haven't done much cuda optimizaitons in terms of memory.

Reference Fig 5.3 soon. But not right now.

**6. Conclusion.** In the end we have worked towards creating a version of HPCG that works alongside the Kokkos package found in Trilinos. This version of HPCG will be a useful for being able to run the reference version of HPCG out of the box and not need to configure the code to be compatible with the specific machine being benchmarked.

Our work here is not completed but we have made great headway on this project and currently have code that produces similar results across all of the Kokkos execution spaces. In the future we want to fix a few performance bottlenecks and utilize hierarchical parallelism to fully take advantage of Kokkos kernels.We also want to fix our coloring algorithm for the symmetric Gauss-Seidel so we can choose at compile time which algorithm to use for preconditioning. It will be interesting to compare performance between all of our preconditioning algorithms.

**7. Acknowledgments.** Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This paper is cross-referenced at Sandia as SAND

REFERENCES

[1] J. Dongarra and et al., *Top 500 supercomputer sites*. `http://www.top500.org`, 1999.
[2] 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, (2014).
[3] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An overview of the trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.
[4] M. A. Heroux and J. Dongarra, *Toward a new metric for ranking high performance computing systems*, tech. rep., Sandia National Laboratories, 2013.
[5] M. A. Heroux, J. Dongarra, and P. Luszczek, *Hpcg technical specification*, Tech. Rep. SAND2013-8752, Sandia National Laboratories, 2013.
[6] laise Barney, *Posix threads programming*. `https://computing.llnl.gov/tutorials/pthreads/`.
[7] NVIDIA, *Cuda programming guide version 3.0*, tech. rep., Nvidia Corporation, 2010.
[8] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 1023.

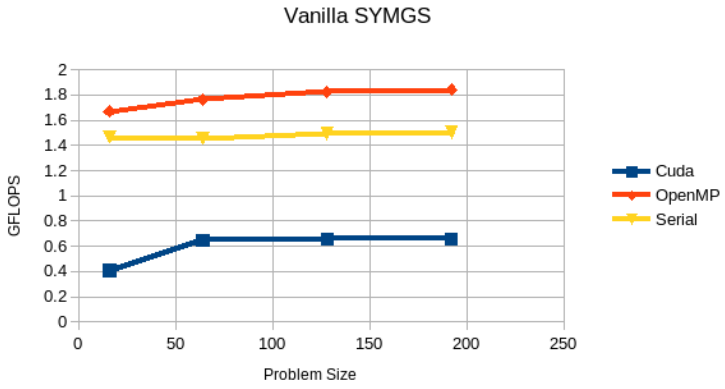


FIG. 5.1. *Plot of GFLOPS for Vanilla SYMGS on each execution space.*

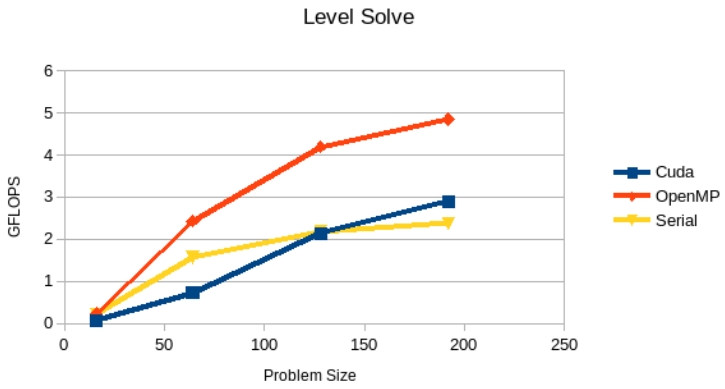


FIG. 5.2. *Plot of GFLOPS for level solve on each execution space.*

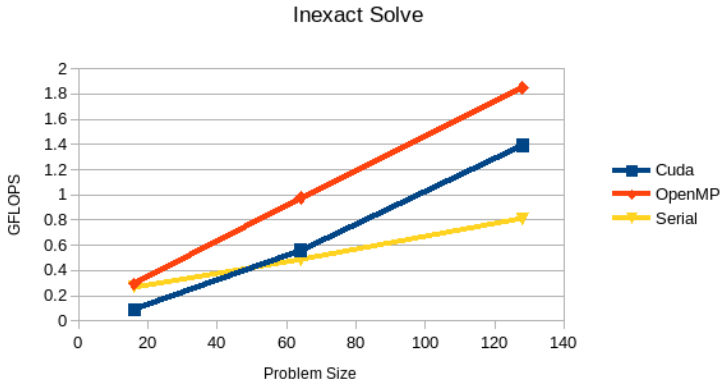


FIG. 5.3. *Plot of GFLOPS for inexact solve on each execution space.*