

PERFORMANCE PORTABLE HPCG

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

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) benchmark 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) 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, MPI-only, HPCG benchmark doesn't exploit full parallelism available on existing Supercomputers which makes it unfair to use it for performance measurement. And the goal of our project was to create a performance portable version of HPCG that gives reasonable performance on all existing supercomputers, by using Kokkos library from Trilinos.

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, zabokey@csbsju.edu

†Sandia National Laboratories, ipdemes@sandia.gov

‡Sandia National Laboratories, srajama@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 versions 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 many 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. Currently Kokkos supports the following execution spaces:

- Serial
- PThreads
- OpenMP
- Cuda

Kokkos has two main features, views and parallel kernels. A 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. Views 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 kernels: `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 parallel 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 uses Kokkos features to be able to produce valid results across many architectures without sacrificing performance. First we needed to rewrite the parallel kernels to use Kokkos parallel kernels, then we needed to restructure all of the code to work with Kokkos views, and finally we needed to make a few optimizations to avoid an unnecessary amount of memory copying from the host to the device and vice versa.

4.1. Kokkos re-factoring. 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 accomodate the fact that data was stored on a device but was being run on the host.

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.

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 colouring 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 its 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 is going wrong. 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.

4.2. Performance evaluation result. Performance results coming soon!

5. Conclusion.

6. Acknowledgements.

7. Conclusion.

REFERENCES

- [1] M. A. HEROUX, J. DONGARRA, AND P. LUSZCZEK, *Hpcg technical specification*, tech. rep., Sandia National Laboratories.
- [2] M. MENTOR, *On chocolate pudding consumption rates*, Int. J. Quantum Foodstuffs, 42 (2006), pp. 42–142.