EMBARGO PERFORMANCE PORTABLE HIGH PERFORMANCE CONJUGATE GRADIENT BENCHMARK

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

Abstract. The High Performance Conjugate Gradient Benchmark (HPCG) 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. HPCG is designed to exercise computations and data access patterns more commonly found in applications. The reference version of HPCG exploitx only some parallelism available on existing supercomputers and the main focus of this work was to create a performance portable version of HPCG that gives reasonable performance on hybrid architectures.

1. Introduction. The High Performance Conjugate Gradient (HPCG)[5][4] is emerging as a complement to the High Performance Linpack (HPL) benchmark for ranking the top computing systems in the world. 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.

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 world's 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 multigrid 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 symmetric, 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 randomly filled vectors and performing simple operations that should be zero due to the nature of our symmetric matrix A.

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

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 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 trade-off between more overhead and more parallelism.

3. Kokkos. As computer architectures differ in their features for best parallel performance 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, such as from 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.

One of the useful features of Kokkos is that parallel operations can be nested. This allows us to run up to three level parallelism inside a single kernel. This happens by creating a league of thread teams so that each team of threads has a specified number of threads. Then we can assign vector lanes to each thread that can run in parallel as well. This allows us to call a parallel kernel on the league and then another parallel kernel on the teams and finally a parallel kernel on the vector lanes of the thread. Depending on the type of problem nested parallelism can significantly improve performance of the Kokkos kernels, but, at the same time, is introduces some overheads that can be significant for the problems with no enough parallelism. We'll explore this further in HPCG later.

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 re-factor HPCG to use it.

General strategy for Kokkos re-factoring includes:

- Replace custom data types with Kokkos multidimensional arrays;
- Replace the parallel loops with Kokkos parallel kernels;
- Code optimizations to 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 `Kokkos::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.

An example of "nested loop to Kokkos" kernel conversion is presented in the Figure 4.1. Here we replace outer loop with the `Kokkos::parallel_for` and put internal part of the loop to the Kokkos kernel (see right part of the Figure 4.1).

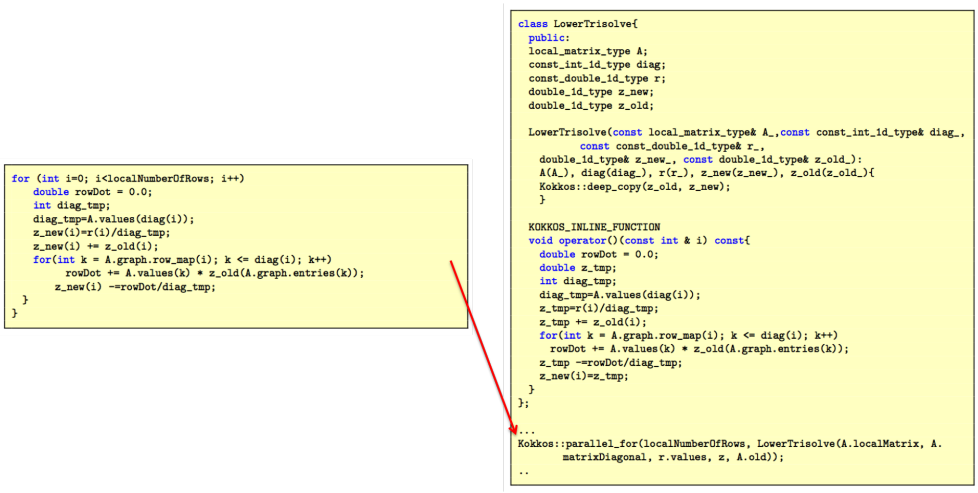


Fig. 4.1: An example of "nested loop to Kokkos kernel" conversion.

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*[1] 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 number of sweeps 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

[1] We understand that the inexact solve violates the policies of the benchmark, but is included for comparison anyways.

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* is not fully implemented. We are in the process of finishing it to provide it as part of HPCG+Kokkos implementation. 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.

5. Performance evaluation. We evaluate performance for our implementation of the HPCG code on our Shannon testbed cluster. Shannon has 32 nodes with 2 Intel SnadyBridge CPUs and 2 Nvidia K40/K80 GPUs per node.

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 5.1 the standard 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.

	Standard 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 5.1: GFLOPS Results for various SYMGS on Problem Size 16^3

As seen in Table 5.2 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.

	Standard 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 5.2: GFLOPS Results for various SYMGS on Problem Size 64^3

Table 5.3 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.

	Standard 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 5.3: GFLOPS Results for various SYMGS on Problem Size 128^3

In Table 5.4 we tested our preconditioning algorithms on a problem size of 192^3. As expected our standard Gauss-Seidel performed at the same level as before and our level solve starts to slow down its performance increases. The inexact solve left us with some abnormally high results that remained valid. For executing on OpenMP we witnessed results of 100+ GFLOPS and for Cuda we found high 90's. We are still investigating what may have caused these results.

	Standard 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 5.4: GFLOPS Results for various SYMGS on Problem Size 192^3

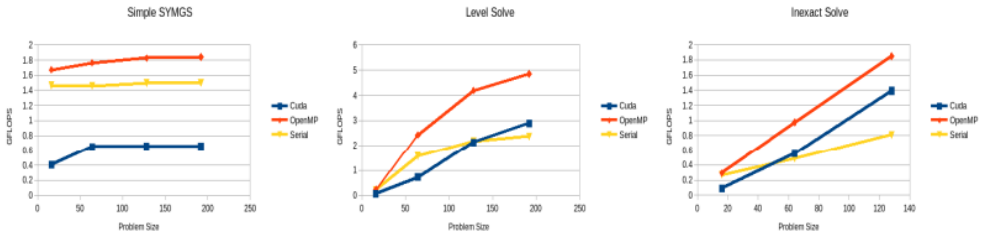
Now we're going to look at how each preconditioning algorithm performs overall on each execution space.

As seen in Fig 5.1(a) 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 hindrance.

In Fig 5.1(b) 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 achieve better performance using this preconditioner than Cuda. However this could be because we haven't done much cuda optimizations in terms of memory.

Looking at Fig 5.1(c) it seems we have a trend that increases our performance based on problem size. We are confident this has to do with the fact that when our problem size is larger our matrix is less dense and so we need less jacobi iterations to produce an answer exact enough to pass the symmetry tests. However we opt to leave out problem size 192^3 due to the abnormal results mentioned before.

5.1. Optimizations. Aside from the symmetric Gauss-Seidel optimizations already made, we are looking to further these optimizations in the future by exploiting hierarchical parallelism. For example, in the *inexact solve* we can implement a three level parallelism technique that breaks our rows into chunks that a team of threads each gets assigned. Now our teams of threads work on our chunk while in parallel



((a)) Simple SYMGS ((b)) Level Solve ((c)) Inexact Solve

Fig. 5.1: Plots of GFLOPS over the Various SYMGS Algorithms

we perform the matrix vector computation in parallel using the threads in the team. A near identical method can be applied to our *level solve algorithm* but since the level solve deals with fewer rows at a time we won't see as much of a performance increase in here. We're confident that these aren't the only places that can benefit from hierarchical parallelism and we plan to explore these options at a later time.

If we start by looking at our original LowerTrisolve kernel used in the inexact solve algorithm (see Figure 4.1) we see that there is a for loop inside of our parallel kernel that doesn't update any outer values and could benefit from being parallelized. If you are even more clever you can take advantage of the third level by assigning chunks to teams as described above. Thus taking full advantage of three level parallelism we are left with the following.

Below is a code snippet that demonstrates how we used Kokkos hierarchical parallelism in the lower trisolve kernel for the inexact solve version of the symmetric Gauss-Seidel.

```
class LowerTrisolve{
    public:
    local_matrix_type A;
    const_int_1d_type diag;
    const_double_1d_type r;
    double_1d_type z_new;
    double_1d_type z_old;
    int localNumberOfRows;
    int rpt = rows_per_team;

    LowerTrisolve(const local_matrix_type& A_,const const_int_1d_type& diag_, const const_double_1d_type& r_,
           double_1d_type& z_new_, const double_1d_type& z_old_, const int localNumberOfRows_):
           A(A_), diag(diag_), r(r_), z_new(z_new_), z_old(z_old_), localNumberOfRows(localNumberOfRows_){
           Kokkos::deep_copy(z_old, z_new);
           }

    KOKKOS_INLINE_FUNCTION
    void operator()(const team_member & thread) const{
        int row_indx=thread.league_rank()* rpt;
        Kokkos::parallel_for(Kokkos::TeamThreadRange(thread, row_indx, row_indx+rpt), [=] (int& irow){
            double rowDot = 0.0;
            double z_tmp;
            int diag_tmp;
            diag_tmp=A.values(diag(irow));
            z_tmp=r(irow)/diag_tmp;
            z_tmp += z_old(irow);
            const int k_start=A.graph.row_map(irow);
            const int k_end=diag(irow)+1;
            const int vector_range=k_end-k_start;
            Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(thread, vector_range),
                   KOKKOS_LAMBDA(const int& lk, double& lrowDot){
                   const int k=k_start+lk;
                   lrowDot += A.values(k) * z_old(A.graph.entries(k));
               }, rowDot);
            z_tmp -=rowDot/diag_tmp;
            z_new(irow)=z_tmp;
        });
    }
};

..
  const int team_size=localNumberOfRows/rows_per_team;
```

```
const team_policy policy( team_size , team_policy::team_size_max( LowerTrisolve(A.localMatrix, A.matrixDiagonal, r.values, z, A.old,
        localNumberOfRows) ),vector_lenght);
Kokkos::parallel_for(policy, LowerTrisolve(A.localMatrix, A.matrixDiagonal, r.values, z, A.old, localNumberOfRows));
```

Using the above implementation of nested parallelism in our inexact solve method for symmetric gauss-seidel, we see performance results as in Figures 5.2, and Tables 5.5 and 5.6. It is clear that Cuda receives a huge benefit from this as it allows us to finely tune how we want to parallelize this method. This nested implementation works as described above and for our figures it chooses chunks of size n for a problem of size n^3. This assures us that we actually work with every row in our matrix and gives us decent results.

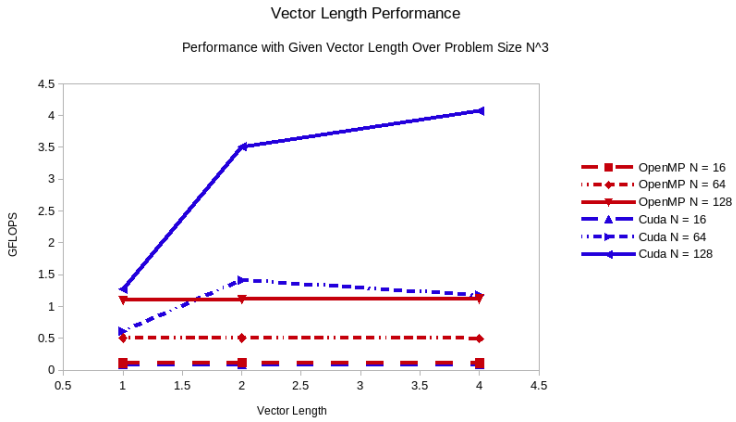


Fig. 5.2: Plot of GFLOPS for different vector lengths used in lowerTrisolve

Vector Levels:	1	2	4
$N = 16$	0.087516	0.0857736	0.0834751
$N = 64$	0.611932	1.41309	1.17325
$N = 128$	1.2713	3.51299	4.07047

Table 5.5: GFLOPS for Number of Levels and Problem Size N^3 Affect Cuda Performance

Vector Levels:	1	2	4
$N = 16$	0.110544	0.111938	0.111116
$N = 64$	0.50259	0.502229	0.495932
$N = 128$	1.09584	1.11385	1.11723

Table 5.6: GFLOPS for Number of Levels and Problem Size N^3 Affect OpenMP Performance

These results are highly preliminary and we are going to investigate performance

for different combinations of the number of rows per team, vector length and problem size.

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 plan to fix a few performance bottlenecks and utilize hierarchical parallelism to fully take advantage of Kokkos kernels.

We also plan 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.
[9] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, 2003.