The HPCG benchmark: analysis, shared memory preliminary improvements and evaluation on an Arm-based platform

Daniel Ruiz[1], **Filippo Mantovani**[1], **Marc Casas**[1], **Jesus Labarta**[1], **and Filippo Spiga**[2]

[1]Barcelona Supercomputing Center
[2]Arm Ltd.

Thursday 15th March, 2018

Abstract

The High-Performance Conjugate Gradient (HPCG) benchmark complements the LINPACK benchmark in the performance evaluation coverage of large High-Performance Computing (HPC) systems. Due to its lower arithmetic intensity and higher memory pressure, HPCG is recognized as a more representative benchmark for data-center and irregular memory access pattern workloads, therefore its popularity and acceptance is raising within the HPC community. As only a small fraction of the reference version of the HPCG benchmark is parallelized with shared memory techniques (OpenMP), we introduce in this report two OpenMP parallelization methods. Due to the increasing importance of Arm architecture in the HPC scenario, we evaluate our HPCG code at scale on a state-of-the-art HPC system based on Cavium ThunderX2 SoC. We consider our work as a contribution to the Arm ecosystem: along with this technical report, we plan in fact to release our code for boosting the tuning of the HPCG benchmark within the Arm community.

1 Introduction and related work

While the LINPACK benchmark still is driving the Top500 ranking, the High Performance Conjugate Gradient (HPCG) [1] is widely accepted as an alternative choice complementing LINPACK for evaluating the performance of large HPC systems. The main difference between the two benchmarks is that LINPACK focuses on matrix-matrix multiplications, while HPCG solves a symmetric sparse linear system of equations using the conjugate gradient method and the multigrid symmetric Gauss-Seidel as preconditioner. Differences between the two benchmarks translate into a different arithmetic intensity footprint, positioning HPCG in a less extreme or more realistic spot than LINPACK, and therefore making it a good candidate for representing the behaviour of a large fraction of today applications running on modern supercomputers [2, 3].

As for LINPACK, also for HPCG has started in recent years a race towards the fine-tuning and porting on different architectures, showing that sometimes topping the Top500 do not imply having good performance with HPCG. The example of the Sunway TaihuLight supercomputer, the fastest LINPACK machine in the world, but with poor performance in HPCG [4].

Several examples of optimization for specific architectures for HPCG can be found in the literature. In [5] Park et al. present the effort performed by Intel for optimizing both shared memory implementation and mitigating MPI communication overhead. GP-GPU implementation and evaluation can be found in [6]. At system level, it has been inspiring for us the work performed by RIKEN team on the K computer presented in [7]: while we use similar coloring approaches, we focused more on tuning algorithmic aspects like coloring parameters rather than focus on loops and math kernels. Other evaluations of HPCG on recent architectures can be found in [8] for the Tianhe-2 supercomputer and in [9] for the NEC SX-ACE. As background study about the numerical methods behind HPCG, we consider relevant for our work [10, 11, 12] which cover coloring and reordering. Worth mentioning the importance of [13] for modeling performance and [14] for performance analysis and power efficiency.

It is not a surprise anymore that Arm architecture is gaining momentum in the HPC panorama. Several research projects have promoted and evaluated the readiness of various Arm-based platforms for HPC [15, 16]. The interest towards Arm architecture increased recently after multiple announcements of competitive Arm-based SoC and mainstream system integrators adding in their product portfolio Arm-based solutions [17, 18, 19].

For these reasons, we begin a profiling and evaluation study of HPCG on publicly novel Arm platforms available on the market. We targeted the HPCG reference version as well as two preliminary improved versions which utilise OpenMP parallelization for the preconditioner. Our work aims to build a new portable version starting from the HPCG reference source code. While maintain a level of portability across architectures, our approach based on de-facto standards differs from other contributions like [20]. It also differs from [21] because we considered a different geometry of the blocks. Exposing a fraction of serial code that can take advantage of the single-thread performance allows us to increase cache reuse. Moreover, our block size does not impact too much the number of iterations needed to convergence, as already shown in [5]. By releasing our code to the Arm community we further enable additional optimizations targeting the Scalable Vector Extension (SVE) introduced by Arm [22, 23].

The main contributions of this paper are: *i)* to introduce two coloring techniques for the symmetric Gauss-Seidel preconditioner, that allow us to increase the performance of the OpenMP version of HPCG by 9.5×, matching the performance of the MPI-only version; *ii)* to present a detailed evaluation of HPCG on a cutting-edge ARMv8 processor, such as the Cavium ThunderX2.

The document is organized as follows: Section 2 briefly introduces high-level structure and profiling of the HPCG benchmark. In Section 3 we describe two techniques for improving the performance of HPCG in a shared memory programming environment, together with other potential opportunities of performance gain. In Section 4 we then evaluate on a state-of-the-art Arm-based platform the effects of our code changes, first at node level and then at scale, on a cluster of 8 nodes. We close with comments and conclusions in Section 5.

2 Benchmark characterization

The HPCG benchmark solves a symmetric sparse linear system mimicking the typical behaviour of a finite element code on a 3D semi-regular grid. As explained in [1] the problem uses an additive Schwarz preconditioner for a first domain decomposition, while each sub-domain is preconditioned using a symmetric Gauss-Seidel sweep that is local to the sub-domain. In its current implementation, the most important numerical kernel for HPCG performance-wise is the 27-point stencil. The benchmark is split in the following phases:

Problem setup and validation – During this phase, all the needed data structures to execute the benchmark are constructed: the memory is allocated to store the sparse matrix that represents the 3-dimensional grid, the right-hand side vector and the result vector are also allocated. Processor and process topologies are generated and stored. Once everything is setup, a validation process checks that the generated matrix fulfills the requirements of the benchmark.

Compute reference SpMV and SymGS – The reference algebraic kernels sparse matrix-vector (SpMV) multiplication and symmetric Gauss-Seidel (SymGS) are executed and benchmarked. The timings collected in this phase will be used afterwards to check the possible improvements obtained by user optimizations.

Compute a reference conjugate gradient (CG) – The reference version of the CG algorithm is executed for a fixed number of iterations (50) and the reduction residual is stored. The same reduction residual needs to be reached by the optimized version implemented by the user. This implies that the optimized version can use a different CG algorithm requiring a different number of iterations to reach the residual. Performance of the optimized version will be in fact recorded independently from the number of iterations whenever (and only if) the reduction residual is reached.

Setup optimized CG run and validation – The optimized CG is then executed and noted as t_{CG}. This timing is used to compute how many times the CG will be executed during the actual benchmarking phase. The number of repetitions is computed as $N = (rt/t_{CG}) + 1$. Where N is the number of repetitions, rt is the execution time provided as input parameter by the user. A validation is performed to make sure that, after all the modifications performed by the user to improve the performance, the matrix still fulfills the requirements of the benchmark.

Optimized CG run – Finally, the HPCG is executed and its performance is recorded. This performance is reported at the end of the execution.

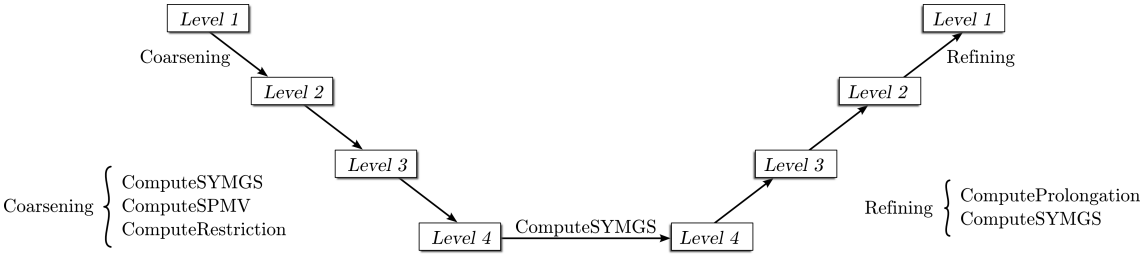

Figure 1: Representation of the V-Cycle multi-grid algorithm implemented in HPCG.

Figure 1 illustrates how the recursive V-Cycle multi-grid algorithm is implemented in the HPCG benchmark. The idea is to define different levels of the same matrix and move to a lower/coarser level applying the combination of smoother (`ComputeSYMGS`), residual computation (`ComputeSPMV`) and restriction of residual (`ComputeRestriction`). The HPCG benchmark implements four of such coarsening steps and then an interpolation process (also called refinement) restores the finer grid, going up to the finest level in the V-Cycle (shown on the right side of Figure 1). Each refinement step is implemented performing two kernels: `ComputeProlongation` smoothed using `ComputeSYMGS`.

2.1 Profiling

To understand the HPCG reference benchmark behaviour, we used the profiling information reported by the application itself complemented with additional custom timers to achieve a finest breakdown of the execution time. We also used Extrae [24] to obtain an execution trace for a more advanced analysis using Paraver [25, 26]. Our first study is based on execution with 8 OpenMP threads of the original HPCG [27] with grid size $nx = ny = nz = 128$, which correspond to a problem size of ~ 1.3 GB of memory per process.

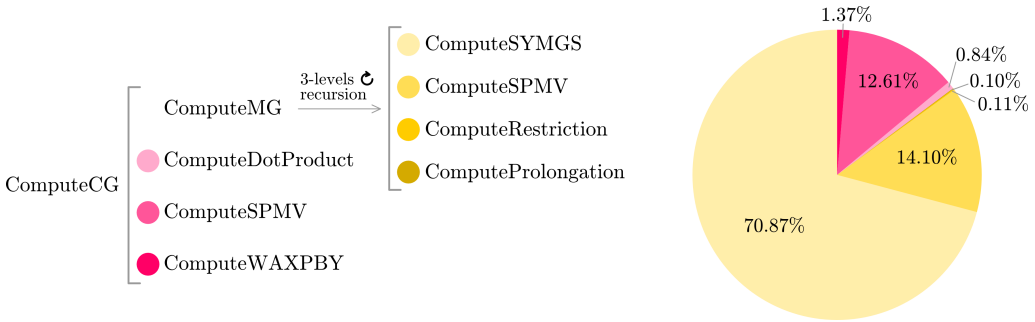

Figure 2: Breakdown of function calls during the conjugate gradient iteration (top) and percentage of their execution time (bottom) when running the serial reference HPCG benchmark.

Figure 2 (top) shows how calls from different compute kernels are related to each other. Figure 2 (bottom) reports the percentage of the total execution time spent in each kernel.

In our test, the multi-grid kernel (`ComputeMG`) consumes $\sim$ 85% of the execution time per iteration. It calls multiple functions, including `ComputeMG` recursively. The remaining $\sim$ 15% of the execution time, represented in Figure 2 (bottom) using green-scale, is consumed by `ComputeSPMV`, `ComputeDotProduct` and `ComputeWAXPBY`.

2.2 OpenMP parallelization

Figure 3 shows a timeline of the different OpenMP parallel regions executed by HPCG, using 8 threads each. On the x-axis we plot the execution time while in the y-axis we display threads progression. Different colors along the timeline mean different parallel compute, while empty (white) regions translate directly into the corresponding thread being idle. We observe that most of the time the OpenMP threads are idle and in our analysis we measured that $\sim$ 90% of the whole

execution time the code runs on a single thread. This is because the symmetric Gauss-Seidel, the most time consuming kernel of the benchmark, is a serial algorithm, therefore no OpenMP parallelization has been implemented in the reference version of HPCG.

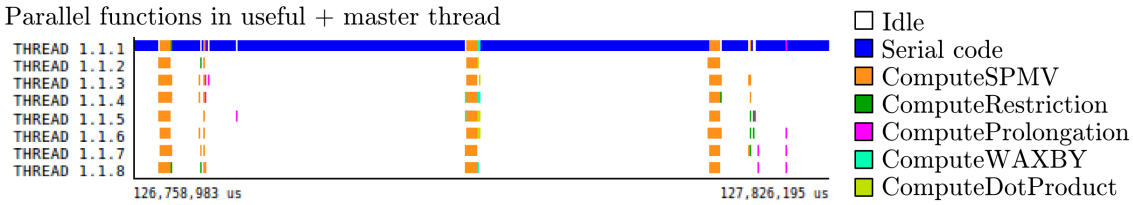

Figure 3: Timeline of the OpenMP parallel regions during the execution of `ComputeMG`.

3 Suggested improvements

As highlighted in the profiling study, the scalability of the reference OpenMP version is mostly limited by the serial preconditioning implementation. Parallelizing the preconditioning process implies a relaxation of the symmetric Gauss-Seidel algorithm that can introduce a penalization in terms of iterations needed to converge (i.e., trade-off between *fewer slower* and *more faster* iterations). By applying the techniques introduced in this section, we measured between 5% and 40% more iterations depending on the chosen implemented method and the parameters chosen. We provide more details about this overhead in the evaluation section (Sections 4.5 and 4.6).

The final benchmark performance, measured as number of floating operations per second, takes into account the execution time spent on the extra iterations required for convergence, due to the relaxation of the algorithm. However only the floating point operations of the first 50 iterations are considered to compute the performance score.

Parallelizing the preconditioner also required the modification of various data structures at run-time and the management of auxiliary variables. The portion of the runtime spent in such operations is considered when computing the final benchmark performance. From a productivity point of view, code changes are confined in 4 files and those increment the total number of lines by $\sim 2\%$.

3.1 Multi-color reordering

We start with the parallelization using OpenMP of the symmetric Gauss-Seidel kernel as described in [10]. The core idea behind this technique is to color each node in the graph in a way that nodes with the same color do not share edges among them. By doing so, in the preconditioning process each node needs values only from first nearest neighbors in the graph. As result, nodes with the same color can be processed in parallel. Figure 4 shows an example of how to color a 2 dimensional graph with 9 neighbors for each node. The initial graph is colored and then reordered, such that nodes with the same color have consecutive indexes. In our implementation, we used 8 colors which is also the smallest number of colors needed for a 27-point 3-dimensional stencil [28].

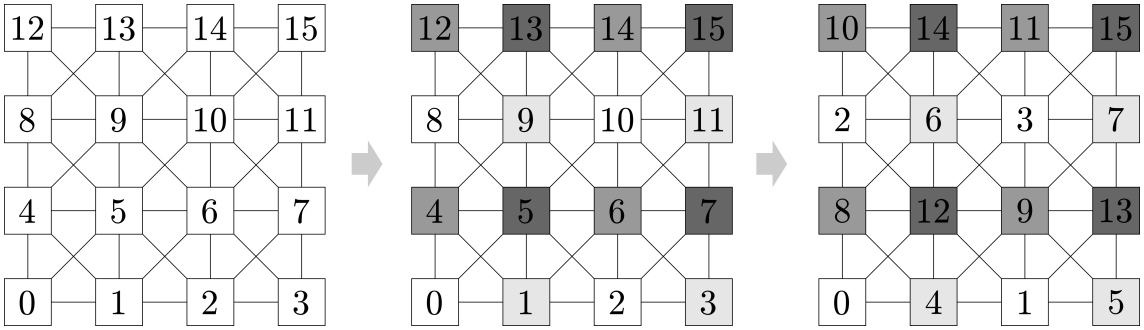

Figure 4: Example of coloring and reordering of a 2D regular graph with 16 nodes. Nodes not sharing edges can be colored with the same color and processed in parallel.

The algorithm we chose for coloring is the greedy coloring described in [11]. The computational cost of the coloring process using this approach introduces an overhead of ∼1%, therefore negligible compared to the overall execution time. We are aware that this method may affect badly the cache data reuse. In fact, as parallel accesses are performed on nodes that are non-contiguously stored, the coloring process harms locality.

3.2 Multi-block color reordering

The idea behind multi-block color reordering is similar to multi-color reordering but instead of coloring single nodes we color set of nodes, called *blocks*. We used the method introduced in [10] and applied in [7], but using different block geometry and colors. Once the size and geometry of the block is defined, we treat every block as single node with a connectivity list that can be handled with the greedy coloring algorithm as mentioned in Section 3.1. After applying this method, blocks with the same color can be computed in parallel, while nodes within the same block must be computed in a serial way. Figure 5.A illustrates how the blocking is performed and how the blocks are colored in a 2D graph with regular nearest neighbors connectivity.

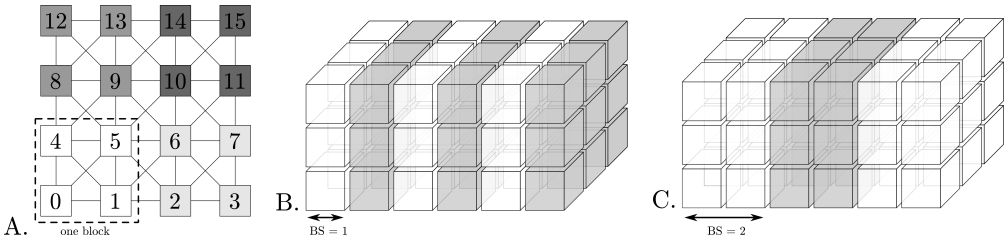

Figure 5: A. Example of multi-block color reordering of a regular 2D graph. **B.** 3D lattice with $BS = 1$, $C = 2$. **C.** 3D lattice with $BS = 2$, $C = 2$.

This method improves the convergence of the algorithm compared to the simple coloring, as shown in [5], therefore less iterations are needed in comparison with the multi-color reordering. It also improves the cache locality with respect to the multi-color reordering since each thread will access consecutive rows of the matrix.

For the block partitioning, we decided to group nodes into blocks following a 2D slice topology. BS is the block size, i.e., the *thickness* of the slice, while C represents the amount of colors. For exploratory purposes, in this study those parameters are chosen at compilation time, but adapting the code they can be set also dynamically. The idea is to be able to optimize the amount of colors and the size of the blocks that better maps to the underlying hardware and therefore provides a higher performance.

In Figure 5 we show an example using $BS = 1$ (B) and $BS = 2$ (C). In both cases the number of colors is $C = 2$. Varying these parameters has different effects on the performance: increasing the number of colors is beneficial for convergence but decreases the parallelism and imposes constraints on the geometry, because $nx \mod C$ should be 0 for keeping all threads busy all the time. To be noticed that increasing the block size can have both cache benefits and less OpenMP synchronizations, but it may lead to inefficiencies during the recursion steps, especially when the lattice size becomes smaller.

3.3 Use of Arm-optimized compiler and libraries

Since 2016, Arm distributes the Arm HPC Compiler [29], an LLVM-based HPC compiler bundled with at set of high-performance mathematical libraries, called Arm Performance libraries. Banchelli et al. [30], performed an early evaluation of those tools and they measured significant difference in performance between GCC and the Arm HPC Compiler. We as well evaluated HPCG with and without Arm proprietary tools.

Some of the HPCG kernels could be mapped directly to BLAS or LAPACK functions, as for example the dot-product and the `axpy` operation. Other kernels can be derived, as e.g., the `waxpby` kernel which computes $W = aX + bY$ (W, X and Y are vectors, while a and b are scalar). It is possible to rewrite the `waxpby` kernel as a combination of a scalar-product BLAS call that computes $X' = aX$ and then an `axpy` call for computing $W = X' + bY$.

All dot product computations performed within the symmetric Gauss-Seidel are also good candidates for being replaced by library calls. The vectors used for such operations are composed of the values of the neighbor nodes of a given row and the values of the solution vector for each of the neighbor nodes of the same row, therefore the number of elements of the vector is at most 27. Unfortunately we verified that the explicit use of math libraries on such small vectors has no performance benefits on the considered machine unless extra work is done to expose extra parallelism.

3.4 Vectorization

The symmetric Gauss-Seidel kernel does not take advantage of automatic SIMD vectorization. Since the floating point operations performed within the kernel consist mainly on a dot product where one of the arrays is not contiguous in memory, the compiler does not generate SIMD instructions.

We introduced a naive SIMD version of the `ComputeSPMV` and the `ComputeSYMGS` which uses NEON [31, 32] without sensible performance improvements. The same version can be used as starting point to test different SIMD extensions such as the Arm Scalable Vector Extension (SVE) [23], which include gather-load and scatter-store instructions, useful for stencil workloads as the one presented by the symmetric Gauss-Seidel algorithm.

4 Results and evaluation

The code development performance evaluation shown in Figure 10 has been performed on a dual socket compute node developed by Atos/BULL for the Mont-Blanc 3 project housing production silicon Cavium ThunderX2 processors with 24 ARMv8 cores at 2.00 GHz each and 256 GB DDR4 ECC memory.

The scalability study presented in Section 4.8 has been performed on a development cluster featuring 8 dual-socket nodes equipped with production silicon Cavium ThunderX2 processors with 28 ARMv8 cores at 2.00 GHz each. The nodes are connected using Mellanox Infiniband EDR.

As for the profiling data presented in Section 2.1, unless specified differently, our study is based on a local problem size $nx = ny = nz = 128$, which corresponds to $\sim$ 1.3 GB of memory per process.

In all OpenMP experiments, we set the environment variable `OMP_PROC_BIND` to ensure that the runtime binds threads to cores, resulting in better data locality.

The software stack used for our tests included: RHEL 7.4, GCC 7.1.0, Arm HPC Compiler 18.1, Arm Performance Libraries 18.1, and Open MPI 3.0.0.

4.1 IPC and Memory Intensity

We considered two main metrics: the *Instruction per Clock-cycle* (IPC) and the *Memory Intensity* ratio $M_I = I_{mem}/I_{tot}$, where I_{mem} is the number of instructions accessing the memory and I_{tot} is the total number of instructions.

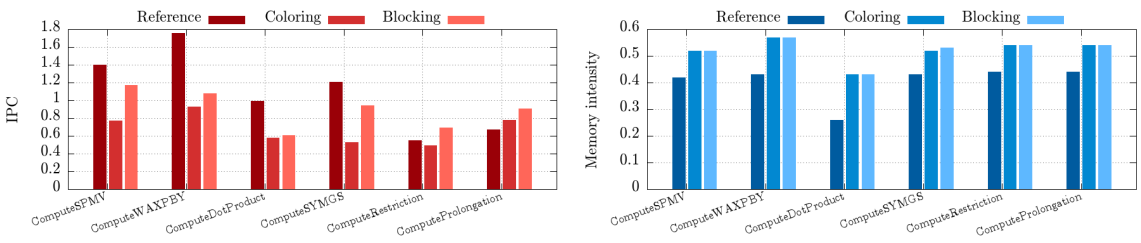

Figure 6: Average IPC and M_I per active thread of the most relevant HPCG functions. The darker bars show the metrics for the reference version of the code, while lighter bars show values for the improved versions introduced in Section 3.

In Figure 6 we show IPC and M_I for the most relevant HPCG functions. The darker bars show the values of IPC and M_I for the reference version of the code, while lighter bars show the same metrics for the improved versions with higher parallelism introduced in Section 3.

For the reference version, where most of the time only one thread is working, we observe a high value of IPC for those functions that are serially executed. On the contrary, the memory intensity M_I is lower in the reference version, compared to implementations with higher parallelism. In fact, while we can assume I_{mem} is approximately constant over different implementations, I_{tot} is bigger when running with the reference version. Therefore the ratio M_I is lower on more serialized code (reference version in Figure 6).

We can also notice that for both improved versions, Coloring and Blocking, the memory intensity in `ComputeSYMGS` is higher than the reference. Such observation justified a deeper study of cache miss-ratio.

4.2 Cache miss-ratio

We define the *Cache Miss Ratio* as $R_{Li} = M_{Li}/A$, where M_{Li} is the number of misses in the cache of level i and A is the total number of accesses to memory of our code (i.e., load+store instructions). On a complete execution of the reference version $R_{L1} = 7.50\%$ and $R_{L2} = 7.13\%$, while Figure 7 reports the R_{Li} ratio for each of the relevant functions. Looking at the values of R_{Li} for `ComputeSYMGS` it is clear the negative impact of the coloring techniques, which increases the miss ratio especially in the L2 caches. On the contrary, the values of R_{Li} for the blocking implementation of `ComputeSYMGS` show that our implementation greatly benefits of data locality within the block.

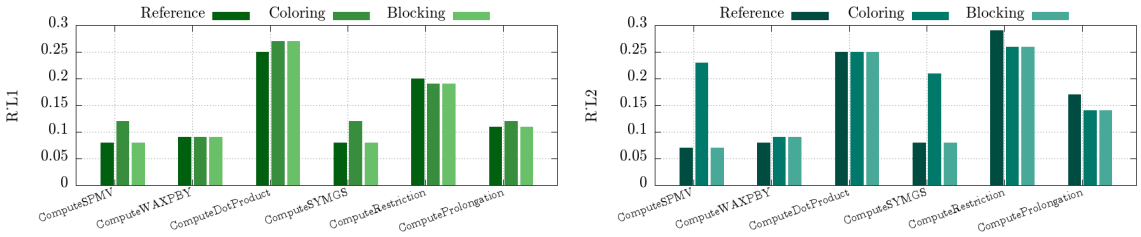

Figure 7: Average cache miss ratio R_{Li} per active thread of the most relevant HPCG functions. The darker bars show the values of R_{Li} for the reference version of the code, while lighter bars show R_{Li} values for the improved versions introduced in Section 3.

Similarly, we studied the *Misses Per Kilo-Instruction* (MPKI) ratio, defined as $\text{MPKI}_{Li} = M_{Li}/(I/1000)$, where I is the total number of instructions executed and M_{Li} is defined as the number of misses in the cache of level i. On a complete execution of the reference version $\text{MPKI}_{L1} = 28.95$ and $\text{MPKI}_{L2} = 27.50$. In Figure 8 we report MPKI for the most relevant functions for the reference version and each of the OpenMP implementations evaluated. We can notice that the data layout introduced in our two implementations penalizes the performance of e.g., `ComputeWAXPBY` or `ComputeDotProduct`.

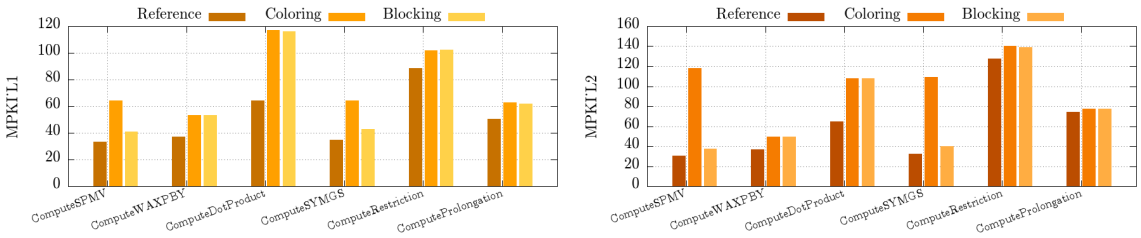

Figure 8: Average MPKI of the most relevant HPCG functions for L1 and L2 caches. The darker bars show the values of MPKI for the reference version of the code, while lighter bars show MPKI values for the improved versions introduced in Section 3.

4.3 Network communications

For analyzing the communication patterns and overheads, we executed the reference version of HPCG with 8 MPI ranks, $nx = ny = nz = 128$ and a single OpenMP thread per MPI rank. Table 1 shows the percentage of execution time spent in each of the MPI calls by each of the MPI processes (ranks). As expected, the most impacting MPI function is the `MPI_Allreduce`. Even

if the impact in a run with 8 MPI processes is only $\sim 1\%$ it is known (see e.g., [13]) that MPI collective operations become soon a bottleneck at large scale.

Table 1: Percentage of time spent in MPI calls by each MPI rank

	Outside MPI	MPI_Send	MPI_Irecv	MPI_Wait	MPI_Allreduce
Rank 1	98.68 %	0.31 %	0.06 %	0.06 %	0.86 %
Rank 2	99.14 %	0.21 %	0.06 %	0.06 %	0.50 %
Rank 3	98.76 %	0.27 %	0.06 %	0.09 %	0.80 %
Rank 4	98.90 %	0.23 %	0.06 %	0.06 %	0.72 %
Rank 5	99.33 %	0.27 %	0.06 %	0.07 %	0.24 %
Rank 6	99.17 %	0.18 %	0.06 %	0.06 %	0.50 %
Rank 7	98.96 %	0.25 %	0.06 %	0.09 %	0.62 %
Rank 8	98.75 %	0.24 %	0.06 %	0.08 %	0.85 %

Figure 9 show, for each message size, the percentage of the total exchanged messages in point-to-point and collective MPI functions and the percentage of cumulated time spent to transmit messages of that size. As expected, almost 70% of communication time is spent on packages of 8 Bytes, proving that small messages, mostly generated by collective operations, are impacting negatively on the overall communication time. This test has been performed within a single node, but the situation is expected to degrade even more when including network stack overheads.

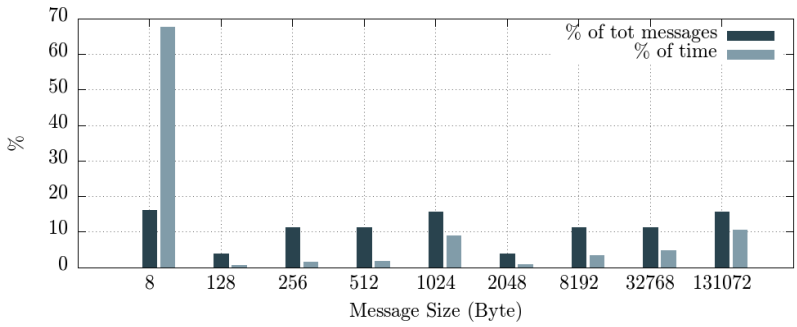

Figure 9: Percentage of number of messages and percentage of MPI communication time (including p2p and collectives) for each message size.

Table 2 shows a regular connectivity among MPI ranks, however, depending on the topological map of the 3D data structure on the MPI ranks, each MPI rank need to communicate to its neighbors a different amount of data, varying from a few tenths of Bytes up to hundreds of MB, when running with 8 MPI ranks and $nx = ny = nz = 128$.

Table 2: MPI ranks connectivity: total amount of MB sent.

	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6	Rank 7	Rank 8
Rank 1	0.00	107.89	107.89	1.08	107.89	1.08	1.08	0.01
Rank 2	107.89	0.00	1.08	107.89	1.08	107.89	0.01	1.08
Rank 3	107.89	1.08	0.00	107.89	1.08	0.01	107.89	1.08
Rank 4	1.08	107.89	107.89	0.00	0.01	1.08	1.08	107.89
Rank 5	107.89	1.08	1.08	0.01	0.00	107.89	107.89	1.08
Rank 6	1.08	107.89	0.01	1.08	107.89	0.00	1.08	107.89
Rank 7	1.08	0.01	107.89	1.08	107.89	1.08	0.00	107.89
Rank 8	0.01	1.08	1.08	107.89	1.08	107.89	107.89	0.00

4.4 Vectorization

We evaluated automatic vectorization obtained using GCC (version 7.1.0) and LLVM-based Arm HPC Compiler (version 18.1). For GCC we used the following flags: `-O3 -mcpu=native -ffast-math -ftree-vectorize -ftree-vectorizer-verbose=0 -fopenmp -std=c++11 -funroll-loops`. For Arm HPC Compiler we used the following flags: `-O3 -mcpu=native -ffast-math -fvectorize`

`-fopenmp -std=c++11 -ffp-contract=fast`. In both cases no vector instructions are generated in the `ComputeMG` portion of the code.

As mentioned in Section 3, we implemented a naive version of the HPCG benchmark which uses Arm NEON instructions to parallelize the sparse matrix vector multiplication and the symmetric Gauss-Seidel kernels. We observed an increment of the SIMD instructions executed during the Gauss-Seidel kernel and a moderate increment during the SPMV kernel. In the first case, the performance was 9% worse than in the reference implementation of the HPCG. In the second kernel we observed a 7% gain. The final performance was decreased by a 14%. We did not invest additional time in improving this naive implementation, but we made it part of our code with the goal of enabling future research focus on Arm Scalable Vector Extensions.

4.5 Multi-coloring evaluation

Figure 10 shows the performance obtained for different OpenMP thread configurations of the multi-coloring version (green lines) described in Section 3.1 and the multi-block coloring (blue lines) introduced in Section 3.2. For comparison we report also the performance of the MPI-only implementation on one socket (black dot) and the OpenMP reference version of the HPCG benchmark (in red lines) that, as expected, does not show any scalability.

The evaluation presented in the rest of this section is based on executions of HPCG with fixed grid size of $nx = ny = nz = 192$, which corresponds to a problem size of ~ 4.9 GB of memory per process.

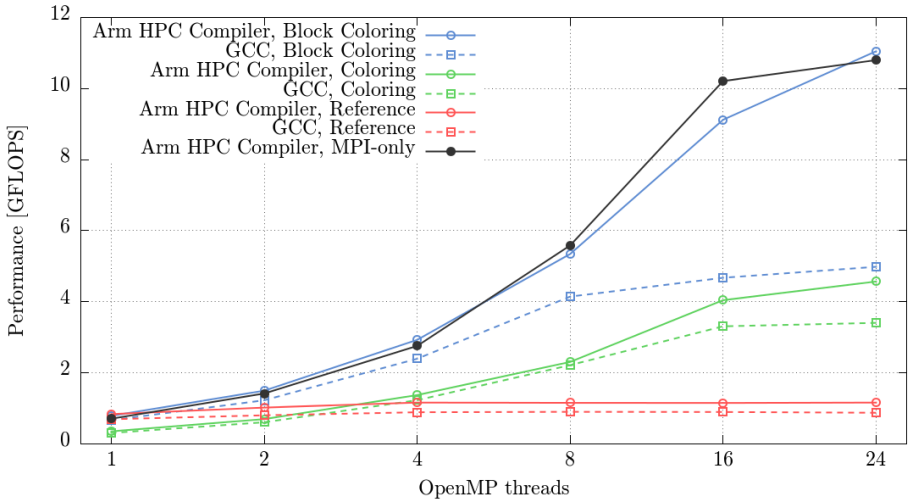

Figure 10: Performance of HPCG multi-block colored version, multi-colored version and reference version

The performance improvement is significant, even if the scalability is far from optimal. The main issue with the multi-color reordering approach is the resultant low instruction per clock cycle (IPC). This is the result of two concurrent factors:

1. Due to the computation order imposed by the reordering of the matrix, almost all memory accesses on the symmetric Gauss-Seidel are to non-contiguous addresses. This impacts negatively on the cache miss ratio.

2. Due to the higher number of threads accessing the memory, the L2 is more stressed both by a higher number of requests and a heavier coherency traffic.

Table 3 shows the cache miss ratio in L1 and L2 for a complete execution and for the `ComputeMG` function: the L2 cache miss ratio increases from 7.67% of the reference version to 21.23%. Note that the hardware counter mapped into the PAPI event `PAPI_L2_DCM` in Cavium ThunderX2 Performance Monitor Unit implementation includes cache refills generated by hardware prefetcher. This explains why higher cache miss rate in L2 than in L1 is observed.

Since the computation order is modified, the multi-coloring reordering implementation increases the parallelism. However it requires 20% to 38% more iterations depending on the geometry of the input-set to achieve the convergence, negatively affecting final overall performance.

Table 3: Figures of performance and duration for the implementations of HPCG analyzed in this document.

		R_{L1}	R_{L2}	MPKI$_{L1}$	MPKI$_{L2}$	IPC	Time
Reference	Overall	7.50%	7.13%	28.95	27.50	1.40	146.12 s
	`Compute_MG`	7.97%	7.67%	34.30	33.01	1.36	814 μs
Coloring	Overall	10.44%	16.03%	49.24	75.62	0.72	94.65 s
	`Compute_MG`	12.35%	21.23%	64.64	111.14	0.57	232 μs
Blocking	Overall	7.63%	7.06%	35.00	32.38	1.09	75.13 s
	`Compute_MG`	8.16%	7.61%	42.97	40.08	0.98	131 μs

Looking at compiler effects, we observed that the Arm HPC Compiler generates averagely a faster binary than GCC. Figure 11 shows a timeline of 24 OpenMP threads executing the same portion of the code (`GenerateProblem` in pink) for GCC (up) and Arm HPC Compiler (down). Because the time scale used in the Figure is identical, it is clearly visible that GCC generates code $\sim 6\times$ slower compared to Arm HPC Compiler. It is clearly visible threads unbalance in the GCC version. This unbalance was also present when using the same OpenMP scheduling policy with both compilers.

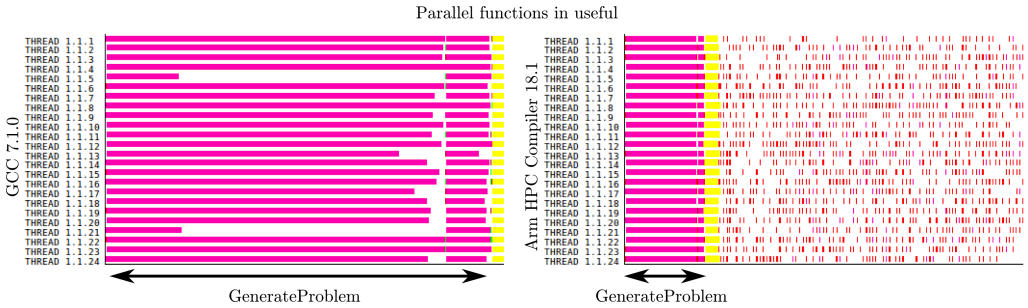

Figure 11: Paraver timeline of the initial section of HPCG. Same time scale.

This behavior is present in all executions so it explains the systematic higher performance of the Arm HPC Compiler. Proof of this, is the fact that if we consider the performance as in HPCG v2.4 (i.e., ignoring setup time) GCC and Arm HPC Compiler deliver roughly the same performance (within 10% fluctuation).

4.6 Multi-block coloring evaluation

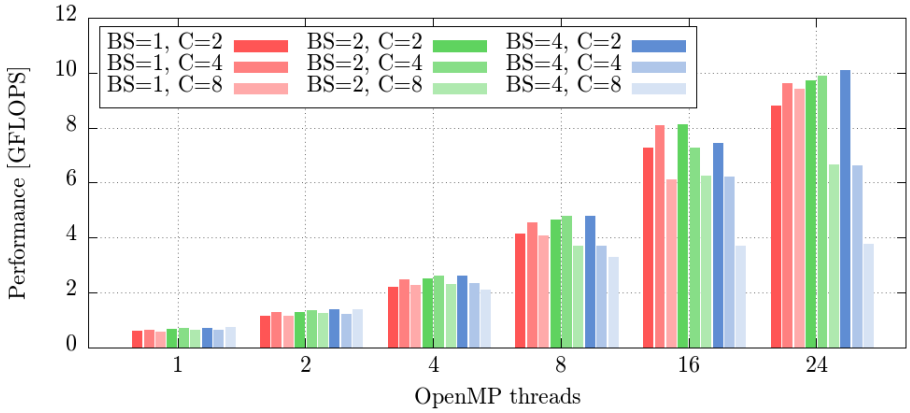

Figure 12: Performance of HPCG multi-block colored version for different block sizes (BS) and number of colors (C). We performed the study for both compilers, Arm HPC Compiler and GCC, but we report here only the figures of the Arm HPC Compiler as it outperforms GCC by $10-20\%$

As described in Section 3.2, we use the parameters BS and C in order to test different block sizes and color configurations. Figure 12 summarizes various performance obtained combining

block sizes and number of colors. Considering the size of the grid $192 \times 192 \times 192$, the amount of nodes of the graph included in each block is $192 \times 192 \times BS$. We observed that, while increasing the block size and the amount of colors led to a less number of iterations to reach convergence, it also led to poorer performance due to the lower level of exploitable parallelism.

If we analyze the case $\{BS = 4, C = 8\}$, the choice of $BS = 4$ implies $nz/BS = 192/4 = 48$ blocks in order to be processed using 8 colors. This means that the number of threads working at the same time in the outer level of recursion is $nz/BS/C = 6$. Whenever we assign more threads to this configuration, we will have unused resources, leading to suboptimal performance. This explains the light-blue bars in Figure 12. On the other hand, a small block size and small amount of color led to a higher number of iterations performed, thus penalizing the final performance. We implemented therefore what we called *dynamic slicing*, i.e., the possibility of computing and configuring BS and C at runtime with the goal of maximizing the parallelism each time we change the refinement of the grid. We verified that this approach unlocks enough compute parallelism without affecting too much the number of iterations needed to converge. Our tests shown only 10% more iterations than the reference execution. As observed in the multi-color reordered version, the Arm HPC Compiler generates a faster binary than GCC in the case of multi-block coloring as well.

Figure 10 (blue lines) show the performance of the multi-block coloring method that improves the one of multi-coloring by 2.5× and the reference versions by 9.5×. For comparison we also report the performance of the MPI-only version of HPCG (black line) when solving the same problem size on a single socket production node with 24 cores. We can observe that the multi-block coloring implementation can reach the performance obtained with a pure MPI implementation.

We analyzed the reasons of potential performance degradation in our implementation:

Setup time – The HPCG performance is computed as:

$$GFLOPS = \frac{OP_{FP}}{T_{bench} + \frac{T_{opt} + T_{setup}}{10} N_{CG}}$$

where T_{bench} is the execution time of the benchmark phase, T_{opt} is the time spent in optimizing the data structures, T_{setup} is the time spent in initializing the data structures, N_{CG} is the amount of CG sets executed within the benchmark phase and OP_{FP} are the floating point operations performed within the benchmark phase. We organize our tests such that each MPI process of the MPI-only version handles the same amount of memory as each OpenMP thread (i.e., we change nx, ny, nz of the OpenMP runs in order to match the amount of memory used by the correspondent MPI-only case). As consequence, each MPI process must initialize a chunk of memory that is $1/np$ of the total memory. On the contrary, the process handling the OpenMP execution have to allocate the total amount of memory. We noticed that T_{setup} does not scale linearly when increasing the input size, therefore, OpenMP versions have an extra overhead.

IPC during computation phases – We observed a systematic lower IPC during computation phases on our OpenMP versions. This could be caused by a better data cache locality on pure MPI implementations since each process executes serial code on its own local grid. In all OpenMP versions, each thread executes serial code on different parts of the grid, thus affecting cache locality. Unfortunately we can not verify this hypothesis because we have currently no direct access to uncore hardware counters, including last level cache information.

Number of instructions per thread – We observed that, in terms of instructions, all OpenMP versions execute more instructions than the pure MPI implementation. This is due to a non-linear scaling of the number of instructions executed per thread when increasing the number of OpenMP threads.

Iterations per CG set – We performed a relaxation of the symmetric Gauss-Seidel algorithm on the OpenMP versions. This relaxation generates a penalization in terms of iterations executed per CG set, negatively affecting the final performance.

This last method also implies better data locality, that translates into a better use of both levels of caches, as shown in the last two lines of Table 3 and in Figures 7 and 8.

Figure 13 compares the memory bandwidth reported by the HPCG benchmark (blue line) with $BS = 1$ and $C = 8$ and the one achieved by the STREAM benchmark (gray lines). On HPCG, the bandwidth increases till reaching 24 cores. That is not the case of STREAM, where the performance of 8 cores is similar to 24 cores. Therefore, the poor scalability of our implementation

is not a lack of memory bandwidth, but a lack of parallelism due to the size of the blocks (higher size implies less blocks within the same color) and the amount of colors (higher number of colors implies blocks need to be split between more groups, therefore, less blocks per color), and the fact that the amount of parallelism is proportional to the amount of blocks within the same color.

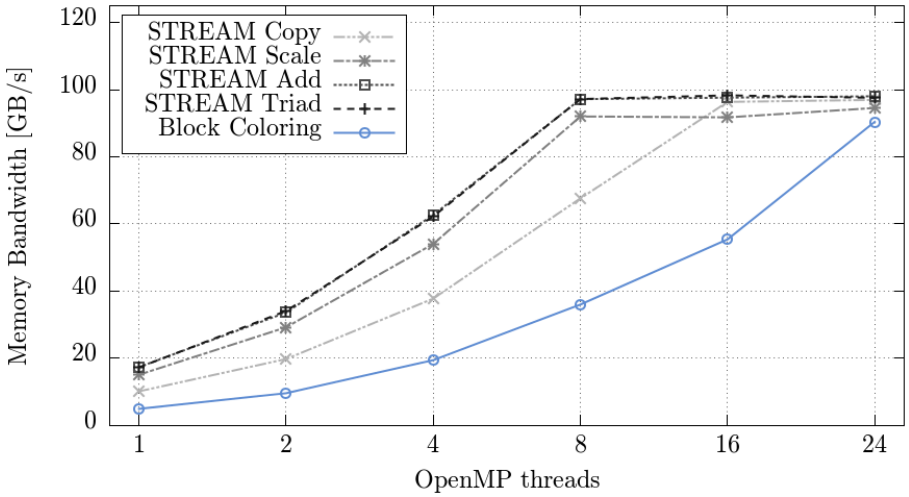

Figure 13: Memory bandwidth achieved with the HPCG with $nx = ny = nz = 192$ and the STREAM benchmarks

4.7 Arm-optimized mathematical libraries evaluation

We implemented a version of the *ComputeDotProduct* that uses the BLAS function *ddot*. Figure 14 shows the performance obtained by using the Arm Performance Libraries as a BLAS implementation (green lines) compared to the reference version without mathematical libraries (orange lines). In both cases OpenMP is used to parallelize the kernel. We can observe that the use of the Arm Performance Libraries almost always improves the performance. Also, combining GCC with the Arm Performance Libraries still adds some benefit (dashed green line).

Even if the benefit of using the Arm Performance Libraries are sensible, it is important to note that the execution time spent on the *ComputeDotProduct* kernel represents less than a 5% of the total benchmark time. The overall gain using them is performance wise almost negligible at this stage. There is still a potential performance gain if computing the dot products within the symmetric Gauss-Seidel using Arm-optimized libraries. The only difficulty for achieving that is the need of remapping the arrays on contiguous memory addresses. Such operation could have noticeable additional overheads. This step is still under investigation.

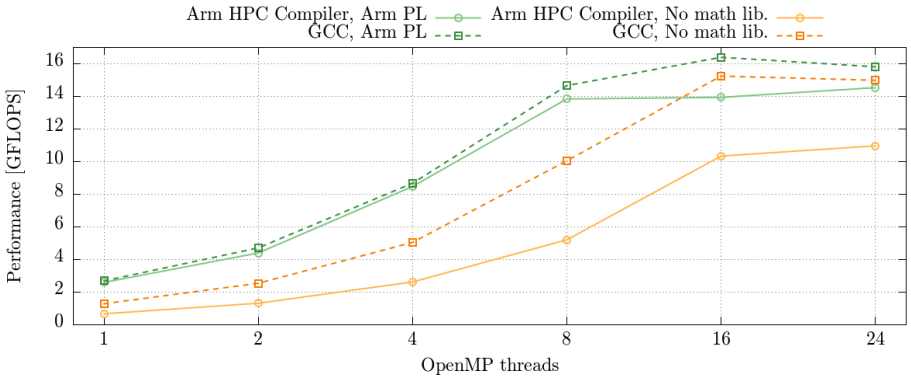

Figure 14: Performance of the *ComputeDotProduct* kernel using the Arm Performance Libraries compared with the reference version that does not make use of any math library.

4.8 Performance at scale

We evaluate scalability of HPCG up to 8 dual socket Cavium ThunderX2 nodes. Each node is equipped with dual socket Cavium ThunderX2 with 56 ARMv8 cores and 256 GB of memory in total. Infiniband EDR is provided by Mellanox MT4115 ConnectX-4 adapters. The cluster is deployed using OpenHPC 1.3 [33] and it runs Red Hat Enterprise Linux 7.4. The relevant software stack used for this work is composed by the following packages and libraries: GNU 7.1.0, Arm HPC Compiler 18.2.0, Mellanox OFED 4.3-1.0.1.0, UCX 1.3 [34], Open MPI 3.0.1 built with UCX support and Slurm 17.02.9.

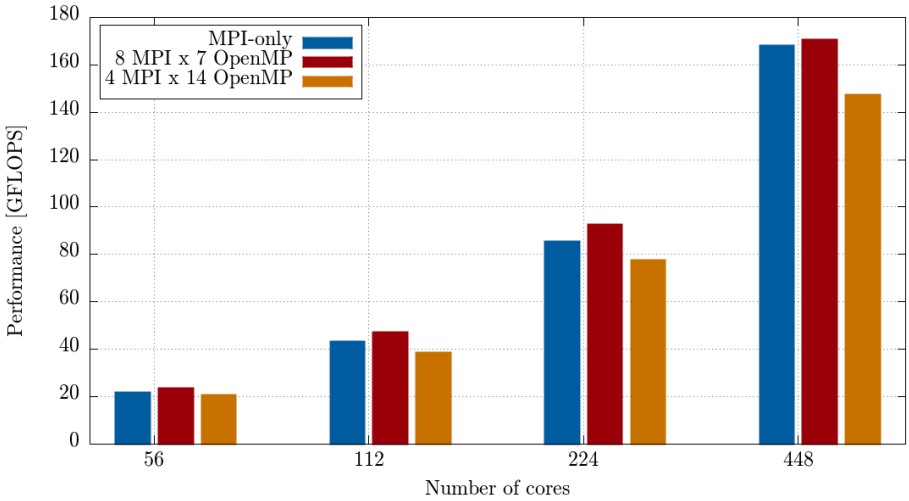

Figure 15: HPCG performance of the best hybrid MPI+OpenMP code implementation (multi-block color reordering) and MPI-only up to 8 dual-socket nodes.

Figure 15 compares the MPI only version with two hybrid MPI+OpenMP executions modes: 8 MPI processes per node each using 7 OpenMP threads (red bars) and 4 MPI processes per node each using 14 OpenMP threads (yellow bars). Block-coloring parameters BS and C are dynamically chosen by the application at run-time based on problem size and executions modes. The problem size, defined by grid dimensions $\{nx, ny, nz\}$, have been chosen to operate in weak scaling regime, allocating the same amount of memory at node level for all executions ($\sim$ 126.98 GB). All tests exploit appropriate process placement and thread binding via Open MPI (via the "`-map-by`" option). UCX is explicitly used as point-to-point management layer (via the "`-mca pml ucx`" option).

This preliminary study shows that the MPI+OpenMP implementation with block-coloring can, in certain conditions and for certain local problem sizes, outperform the MPI-only version.

5 Conclusions

In this paper we introduce a profiling of the HPCG benchmark and we analyze different strategies to parallelize the symmetric Gauss-Seidel preconditioner from the HPCG benchmark. We focus on shared memory first and we relay on OpenMP as de-facto standard for express parallelism within a HPC node. Focusing on the parallelization of the Gauss-Seidel, we implemented two approaches: *multi-color reordering* and *multi-block color reordering*.

The work presented in this paper is a preliminary set of activities which will be used as a starting point for further improvements. As the multi-block coloring approach has been proved to deliver higher performance, we will focus on optimizing the block layout in that implementation. Also, we will further explore vectorization and new data access patterns. All of these are extremely important to further improve the performance of the HPCG benchmark. Optimized memory access pattern could improve the effective memory bandwidth by making better use of the memory hierarchy, manual vectorization could help in increasing the floating point operation per cycle ratio whereas different block layouts could impact on the amount of parallelism exposed within the Gauss-Seidel kernel.

We evaluated our parallelization techniques on Cavium ThunderX2 systems, state-of-the-art ARMv8 architecture targeting HPC. It is important to point out that our main objective it is not

to optimize the HPCG benchmark for a specific processor technology but to implement a base version which takes advantage of shared memory techniques. This version that can be used as starting point for specific optimizations based on various micro-architecture, including different Arm-based processors. Concerning the Arm ecosystem, it is encouraging to note that, on average on this particular benchmark, Arm HPC Compiler delivers better performance than GCC.

As it has been observed in Section 4.6, a static block layout evaluated in our multi-block coloring implementation does not always express a high degree of parallelism, which directly translates into not fully utilizing all the resources available on the processor for all refinement stages of the multi-grid. We notice in fact that, when increasing the block size, the performance drops. Therefore, a new approach to dynamically vary the geometry of the blocks has been implemented, evaluated and proved capable of reaching and, in some cases, even outperforming the pure-MPI performance. Comparing the memory bandwidth achieved with our implementation and the one measured with the STREAM benchmark (see Figure 13) we notice that we are using a significant fraction of the bandwidth, but there is still space for improvement. In view of this, it is not only important to increase the level of parallelism of the Gauss-Seidel, but also to improve the cache usage and try to increase the number of floating point operations performed for each memory access.

We noticed that the multi-coloring algorithm significantly degrades the cache hit-ratio. This is due to the fact that the different nodes of the grid are accessed using a pseudo sparse pattern decreasing the locality of the data structures used on the HPCG benchmark. This effect appears mitigated on the multi-block coloring approach since each block is processed sequentially: within a block the access to the data is identical to the reference version of the benchmark.

The optimizations flags used during the experiments, even if coherently chosen during the benchmark campaign, could be sub-optimal preventing to achieve the best performance on Cavium ThunderX2. For this reason, further close discussion with the SoC provider and the developers of the compilers tested in this work will continue. The same applies for optimized mathematical libraries: we measured that using the Arm Performance Libraries can deliver a better performance, but probably improving basic sparse matrix operations (e.g., SParse-Matrix Vector (SPMV)) will bring even a higher benefit.

As a final note, the authors of this paper release open-source the improved version of the HPCG benchmark evaluated in this paper in order to establish a baseline version that can be used for future algorithmic improvements, low-level fine-grain optimizations and also reproducible performance comparisons between Arm and non-Arm platforms. The code is available at `https://gitlab.com/arm-hpc`.

References

[1] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC challenge benchmark suite. *Lawrence Berkeley National Laboratory*, 2005.

[2] Jack Dongarra and Michael A. Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[3] Jack Dongarra and Piotr Luszczek. HPCG technical specification. *Sandia National Laboratories, Sandia Report SAND2013-8752*, 2013.

[4] Jack Dongarra. Sunway TaihuLight supercomputer makes its appearance. *National Science Review*, 3(3):265–266, September 2016.

[5] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D. Kalamkar, Xing Liu, Md. Mosotofa Ali Patwary, Yutong Lu, and Pradeep Dubey. Efficient Shared-memory Implementation of High-performance Conjugate Gradient Benchmark and Its Application to Unstructured Matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 945–955, Piscataway, NJ, USA, 2014. IEEE Press.

[6] Everett Phillips and Massimiliano Fatica. A CUDA Implementation of the High Performance Conjugate Gradient Benchmark. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Lecture Notes in Computer Science, pages 68–84. Springer, Cham, November 2014.

[7] Kiyoshi Kumahata, Kazuo Minami, and Naoya Maruyama. High-performance conjugate gradient performance improvement on the K computer. *The International Journal of High Performance Computing Applications*, 30(1):55–70, February 2016.

[8] Xianyi Zhang, Chao Yang, Fangfang Liu, Yiqun Liu, and Yutong Lu. Optimizing and Scaling HPCG on Tianhe-2: Early Experience. In *Algorithms and Architectures for Parallel Processing*, Lecture Notes in Computer Science, pages 28–41. Springer, Cham, August 2014.

[9] R. Egawa, K. Komatsu, Y. Isobe, T. Kato, S. Fujimoto, H. Takizawa, A. Musa, and H. Kobayashi. Performance and Power Analysis of SX-ACE Using HP-X Benchmark Programs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 693–700, September 2017.

[10] T. Iwashita and M. Shimasaki. Algebraic multicolor ordering for parallelized ICCG solver in finite-element analyses. *IEEE Transactions on Magnetics*, 38(2):429–432, March 2002.

[11] Yousef Saad. *Iterative Methods for Sparse Linear Systems: Second Edition*. SIAM, April 2003. Google-Books-ID: ZdLeBlqYeF8C.

[12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. JHU Press, December 2012.

[13] Vladimir Marjanović, José Gracia, and Colin W. Glass. Performance Modeling of the HPCG Benchmark. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Lecture Notes in Computer Science, pages 172–192. Springer, Cham, November 2014.

[14] E. Vermij, L. Fiorin, C. Hagleitner, and K. Bertels. Boosting the Efficiency of HPCG and Graph500 with Near-Data Processing. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 31–40, August 2017.

[15] Michael A Laurenzano, Ananta Tiwari, Adam Jundt, Joshua Peraza, William A Ward, Roy Campbell, and Laura Carrington. Characterizing the performance-energy tradeoff of small arm cores in hpc computation. In *European Conference on Parallel Processing*, pages 124–137. Springer, 2014.

[16] Nikola Rajovic, Alejandro Rico, et al. The Mont-Blanc prototype: an alternative approach for HPC systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 444–455. IEEE, 2016.

[17] Timothy Prickett Morgan. Cavium is truly a contender with one-two Arm server punch. https://www.nextplatform.com/2017/11/27/cavium-truly-contender-one-two-arm-server-punch/, 2017.

[18] Nicole Hemsoth. Arm benchmarks show HPC ripe for processor shakeup. https://www.nextplatform.com/2017/11/13/arm-benchmarks-show-hpc-ripe-processor-shakeup/, 2017.

[19] Nicole Hemsoth. Cray ARMs Highest End Supercomputer with ThunderX2. https://www.nextplatform.com/2017/11/13/cray-arms-highest-end-supercomputer-thunderx2/, 2017.

[20] Kokkos: HPCG benchmark, 2018.

[21] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito Takahashi. Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in ICCG method. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 474–483. IEEE, 2012.

[22] Nigel Stephens, Stuart Biles, et al. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, 2017.

[23] Alejandro Rico, José A Joao, Chris Adeniyi-Jones, and Eric Van Hensbergen. ARM HPC ecosystem and the reemergence of vectors. In *Proceedings of the Computing Frontiers Conference*, pages 329–334. ACM, 2017.

[24] BSC performance analysis tools: Extrae.

[25] BSC performance analysis tools: Paraver.

[26] Filippo Mantovani and Enrico Calore. Performance and power analysis of HPC workloads on heterogeneous multi-node clusters. *Journal of Low Power Electronics and Applications*, 8(13), 2018.

[27] Official HPCG benchmark source code. (commit `0281412` on 15 Nov 2017).

[28] Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

[29] Arm HPC tools and libraries.

[30] Fabio F Banchelli Gracia, Daniel Ruiz Muñoz, Ying Hao Xu Lin, and Filippo Mantovani. Is Arm software ecosystem ready for HPC? In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[31] Gaurav Mitra, Beau Johnston, Alistair P Rendell, Eric McCreath, and Jun Zhou. Use of SIMD vector operations to accelerate application code performance on low-powered ARM and intel platforms. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1107–1116. IEEE, 2013.

[32] Nikola Rajovic, Paul M Carpenter, Isaac Gelado, Nikola Puzovic, Alex Ramirez, and Mateo Valero. Supercomputing with commodity CPUs: Are mobile SoCs ready for HPC? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 40. ACM, 2013.

[33] Karl W Schulz, C Reese Baird, David Brayford, Yiannis Georgiou, Gregory M Kurtzer, Derek Simmel, Thomas Sterling, Nirmala Sundararajan, and Eric Van Hensbergen. Cluster computing with OpenHPC. *HPC Systems Professionals Workshop (HPCSYSPROS'16)*, 2016.

[34] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar R. Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. UCX: an open source framework for HPC network apis and beyond. In *23rd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2015, Santa Clara, CA, USA, August 26-28, 2015*, pages 40–43, 2015.