MB3 D6.4 – Report on application tuning and optimization on ARM platform
Version 1.0

Document Information

Contract Number	671697
Project Website	www.montblanc-project.eu
Contractual Deadline	PM24
Dissemination Level	Public
Nature	Report
Authors	Fabio Banchelli, Marta García, Marc Josep, Victor Lopez, Filippo Mantovani, Xavier Martorell, Daniel Ruiz, Xavier Teruel (BSC) Patrick Schiffmann (AVL), Alban Lumi (UGRAZ)
Contributors	
Reviewers	Roxana Rusitoru (ARM)
Keywords	Applications, performance analysis, co-design insight, OmpSs programming model, mini-apps, power consumption of HPC codes

Notices: *This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N^o 671697.*

Contents

Executive Summary

The current report refers to the work performed under T6.4 during the period from M12 to M24 of the Mont-Blanc 3 project.

This deliverable describes the results generated in porting and tuning the applications considered as part of the co-design process in Mont-Blanc 3 for ARM. We present our optimization experiences for 9 applications. For most of them we performed evaluations on Mont-Blanc 3 platforms and analyses on both ARM and Intel-based platforms. Building on Deliverable 6.1 [1] which tried to identify root causes of scaling issues, we continued with analysis and implemented optimizations to overcome these performance problems.

Overall load imbalance was identified as a very important issue. Most applications needed to address this either via improving algorithms and domain decompositions, distributing their ressources to fewer MPI ranks and more threads, or via dynamic load balancing (DLB).

For the HPCG benchmark, a performance analysis and an initial algorithmic optimization is presented. The work presented is not ARM-specific, but it has been tested on an ARM-based cluster by a team of students. Following the directive in [1], Lulesh has been ported to OmpSs and tested on the Mont-Blanc 3 mini-clusters.

In the ARM ecosystem the ARM Performance Libaries have been evaluated on a widely used scientific suite QuantumESPRESSO. The results indicate speedups when using ARMPL for linear algebra workloads, and highlight opportunities to improve the FFT functions.

In addition, the recently released ARM compiler was compared to GCC. Performance and usability were comparable, further investigation which compiler is preferable for which type of workload is suggested.

For the applications in cardiac modelling and mesh deformation we generally find optimizations stemming from analysis on Intel systems advantageous for ARM systems and vice versa, e.g. work to scale to the high core density ThunderX system proved valuable for performance many core x86 systems. For some of them, power measurements are presented: this numbers will be used as baseline when comparing powerfomance and power figures in the final Mont-Blanc 3 demonstrator under deployment in WP3.

1 Lulesh

1.1 Description

Lulesh is a hydrodynamics simulation application. An analysis on ThunderX [2] and MareNostrum when using MPI and OpenMP can be found on [1], together with a first approach using OmpSs.

1.2 OmpSs Port

Lulesh has been ported to OmpSs, as described on Deliverable D6.5 ([3]) and run on the Mont-Blanc mini-clusters Thunder-X and Jetson-TX. When running on Jetson-TX, using OmpSs provides an increase in performance in comparison when running with OpenMP, up to a 30% (Figure 1).

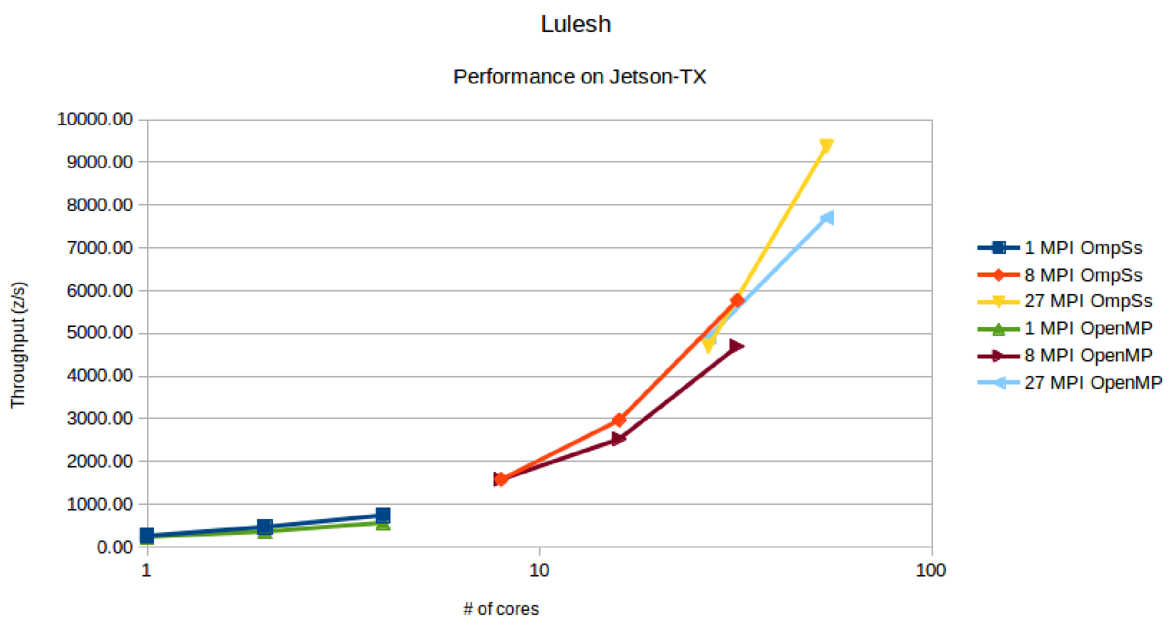


Figure 1: Performance when running Lulesh on Jetson-TX, using both OmpSs and OpenMP. The higher the better.

On Thunder-X, the performance is slightly lower when using OmpSs in comparison with OpenMP. The difference between them gets broader when increasing the number of threads per process (Figure 2).

Figure 3 shows the task execution of two iterations when using only OmpSs with 8 threads. Despite small granularity, there is not a huge overhead, and there is close to no imbalance between threads.

When increasing the number of threads, the performance significantly decreases, mainly due to insuficient amount of work for all cores, as can be seen on figure 4. When we increase the problem size, the principal cause of performance loss is task granularity; the tasks executing function ”EvalEOSForElems” take significantly much more time than the set of other tasks, and thus creating high imbalance on synchronization points (figure 5).

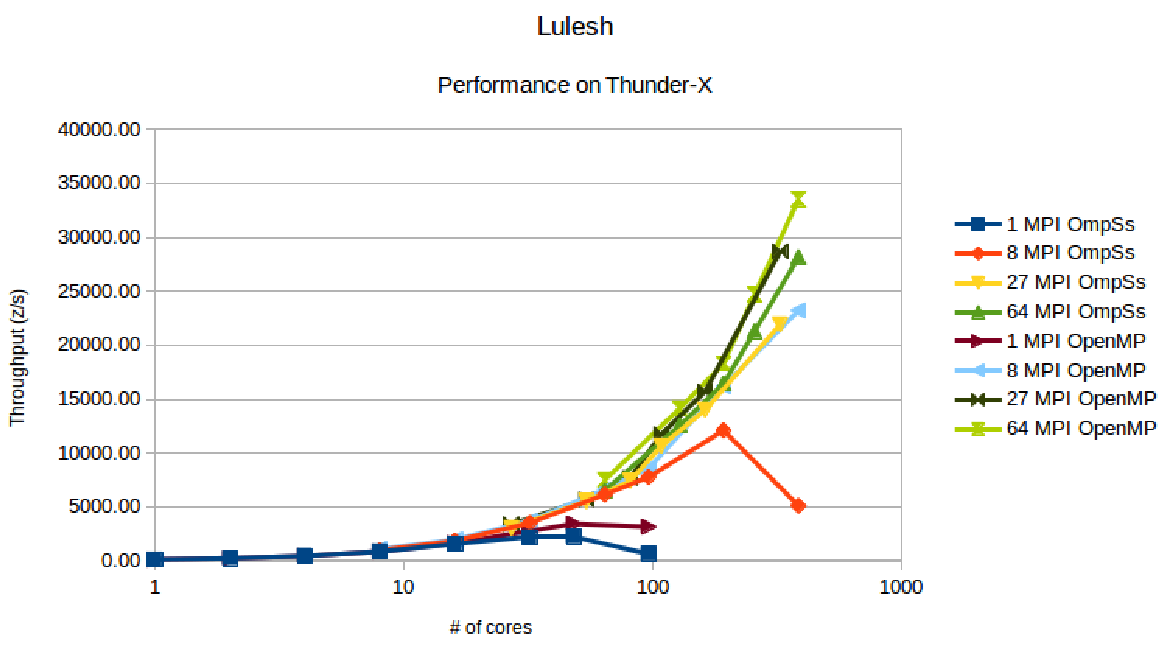


Figure 2: Performance when running Lulesh on Thunder-X, using both OmpSs and OpenMP. The higher the better.

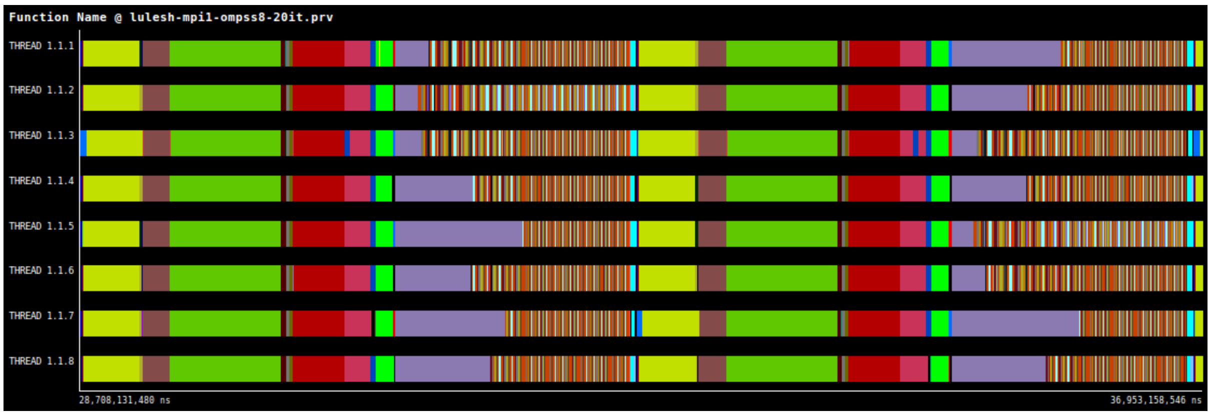


Figure 3: Tasks executed during two Lulesh iterations when running with 1 MPI process and 8 OmpSs threads.

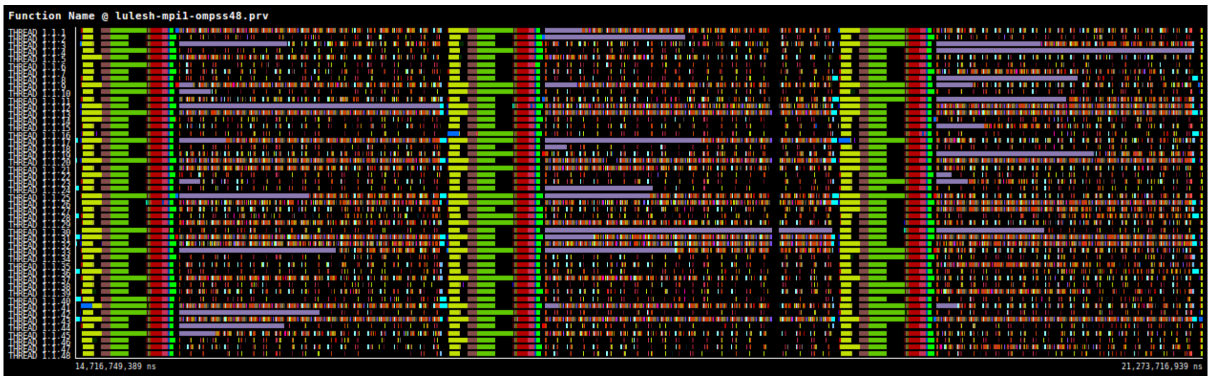


Figure 4: Tasks executed during three Lulesh iterations when running with 1 MPI process and 48 OmpSs threads.

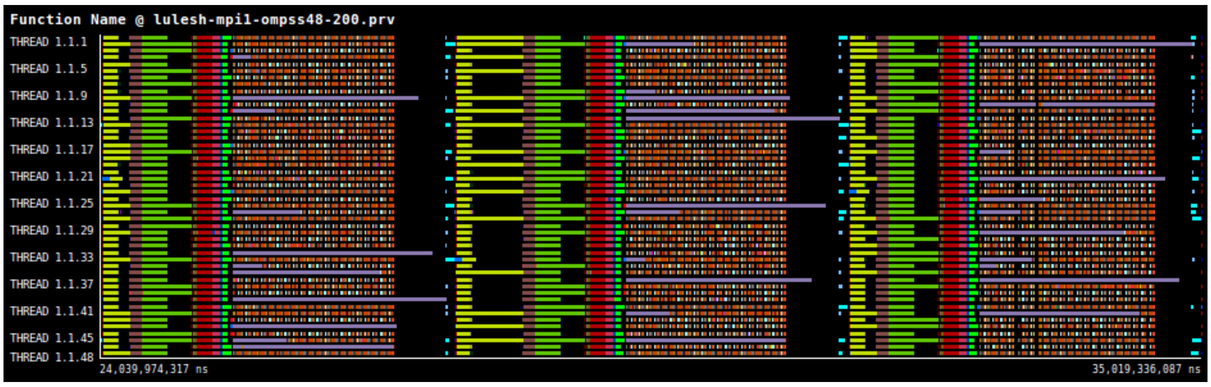


Figure 5: Tasks executed during three Lulesh iterations when running with 1 MPI process and 48 OmpSs threads and a higher problem size than 4. Purple bars depict function "EvalEOSForElems".

1.3 DLB

Lulesh is able to adjust the load balance of the execution using the -b command line option. This option will change the relative weight of regions within a domain. A large value will cause a great disparity between the region with the most elements and the region with the fewest, provoking more imbalance between domains. This is very interesting for doing performance tests with OmpSs + DLB, as the level of the load imbalance can be dynamically adjusted per experiment.

DLB [4, 5] can fix the load balance of the MPI processes in the same shared memory node by temporally distributing the CPUs when one process is idle and other processes can potentially increase its parallel execution. This load balance tests with Lulesh have been performed on the Mont-Blanc prototype, which only has two CPUs. With this restriction, there is only one scenario where we can exploit the DLB capabilities. Lulesh needs to be executed with two MPI ranks per node and only one thread per process, at least at the beginning of the execution, and only when one of the processes gets blocked due to an MPI blocking call, the other process in the node can increase its number of threads by one.

There is also another issue regarding the number of CPUs per node in this experiment. DLB is only able to fix the load imbalance inside the node, if only two MPI ranks are distributed per node, there is a very high chance that either two low load processes or two high load processes get placed in the same node, and this kind of imbalance cannot be fixed by DLB. This is not critical on regular HPC systems as they usually provide a high number of CPUs per node, but it is on the Mont-Blanc systems. To solve this issue, a previous run has been analyzed to detect the load imbalance of each rank so the highest load process is placed in the same node as the lowest load process, the second highest with the second lowest, and so on.

Figures 6 and 7 show the comparison of execution times of Lulesh with and without DLB in different configurations (B=1: low load imbalance, B=8: high load imbalance), and different problem sizes. Execution times with DLB are up to 5% faster in both configurations, except in weak scaling running with 256 MPI ranks, where rank distribution cannot be done efficiently due to the high number of ranks.

The improvement using DLB with Lulesh is shown in Figure 8. The first trace shows a single threaded execution with 8 MPI ranks. The blue rectangle highlights a chunk of code with high load imbalance, the red events correspond to tasks, and the green events correspond to MPI blocking calls. This chunk is only 30% of the iteration. The second trace shows a similar execution, using same resources and same input, but applying DLB in the unbalanced, also highlighted, chunk. By applying DLB, the less loaded ranks (5-8) can lend their CPUs to

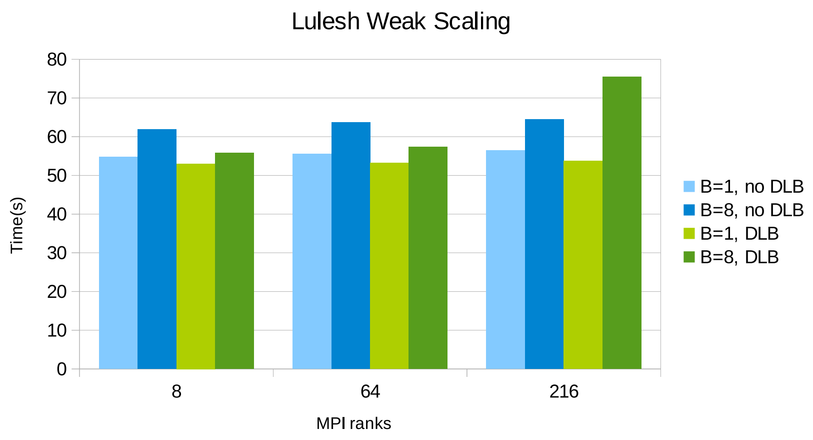


Figure 6: Lulesh execution time running on Mont-Blanc prototype, weak scaling.

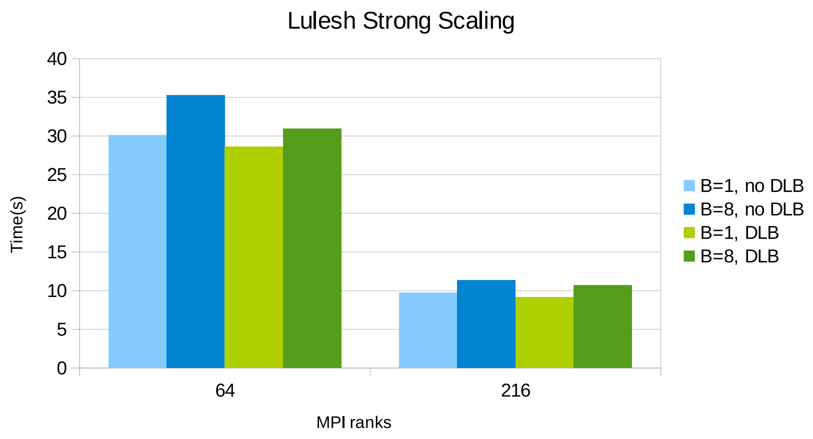


Figure 7: Lulesh execution time running on Mont-Blanc prototype, strong scaling.

the most loaded (1-4). This is done each iteration, completely transparent to the application and achieving a 1.52 of speedup over the same highlighted chunk in the upper trace. Based on Amdahl's law $S(p, s) = \frac{1}{(1-p)+\frac{p}{s}}$, this local speedup will limit the maximum achievable speedup for the application: $S(p = 0.3,\ s = 1.52) = 1.11$, close enough to the 10% speedup shown in the previous charts.

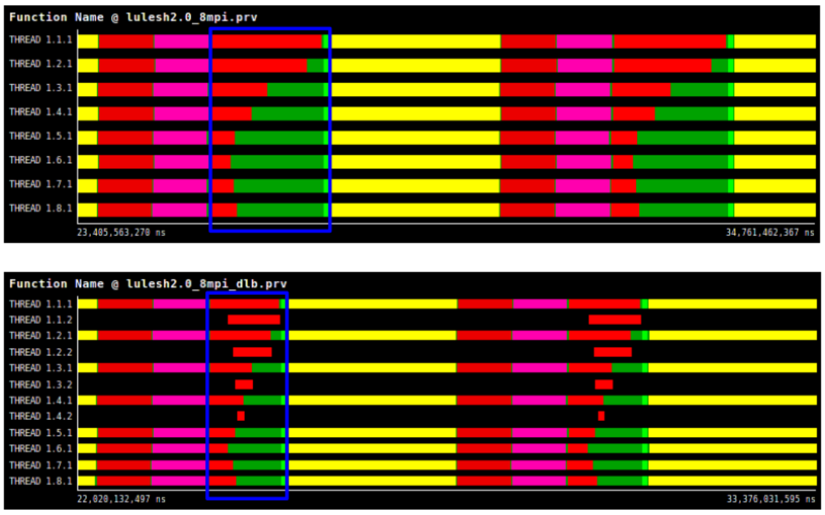


Figure 8: Lulesh trace comparison with and without DLB.

2 HPCG

2.1 Description

High Performance Conjugate Gradient (HPCG) [6] is a synthetic benchmark with the intent to complement the High Performance LINPACK (HPL) benchmark currently used to rank the TOP500 computing systems.

HPCG is a complete and standalone code that measures the performance of several linear algebra kernels with different memory access patterns. Within the benchmark the following basic operations are performed:

- Sparse matrix-vector multiplication
- Vector updates
- Global dot products
- Local symmetric Gauss-Seidel smoother
- Sparse triangular solve (as part of the Gauss-Seidel smoother)

The reference implementation is written in C++ with MPI and OpenMP support.

This application was selected to be executed for the Student Cluster Competition held during the International Supercomputing Conference [7]. During the competition, a team from the *Universitat Politècnica de Catalunya* had to optimize the application. The team was supported by the Mont-Blanc project and advised by project members at BSC. As the benchmark represents a large class of applications, we include here the effort invested on it and a preliminary optimization.

2.2 Analysis

HPCG execution can be divided into 5 discrete steps which include problem generation, testing and execution.

- Problem setup phase
- Reference timing phase
  - Sparse matrix-vector multiplication and Multi-Grid (MG)
  - Conjugate Gradient (CG) algorithm
- Optimized problem setup
  - Data structures optimizations by user
  - Validation of the modifications
  - CG setup
- Benchmarking of optimized CG
  - Sparse matrix-vector multiplication
  - Symmetric Gauss-Seidel
- Results report

An initial profiling of the different benchmarked phases is presented in Table 1. The most expensive part of the benchmark is the MG kernel, on which sparse matrix-vector multiplication and symmetric Gauss-Seidel are performed. The execution was performed by using 1 MPI rank and 8 OpenMP threads on an AMD Seattle SoC and running the benchmark for 10 minutes.

	OpenMP only		MPI only	
Kernel	**GFlops**	**Time (s)**	**GFlops**	**Time (s)**
DDOT	1.62	2.17	0.73	17.51
WAXPBY	1.58	2.23	0.98	12.93
SpMV	1.50	20.68	1.62	70.46
MG	0.26	657.37	1.22	519.76
Total	**0.30**	**682.47**	**1.24**	**620.73**

Table 1: Performance and execution times for the different kernels executed on HPCG. Execution was performed with the reference version on a AMD Seattle with 1 MPI rank and 8 OpenMP threads.

We also executed and traced with Extrae the reference code on the nodes of the ThunderX cluster (4 Cavium ThunderX SoC's for a total of 192 cores). The execution used 4 MPI ranks and 48 OpenMP threads per rank.

Figure 9 show the trace obtained. There we can see two complete iterations of the Conjugate Gradient algorithm. On the top, blue means that the specified core was doing useful work (i.e. executing code), while no color means the specific core was executing nothing. On the middle

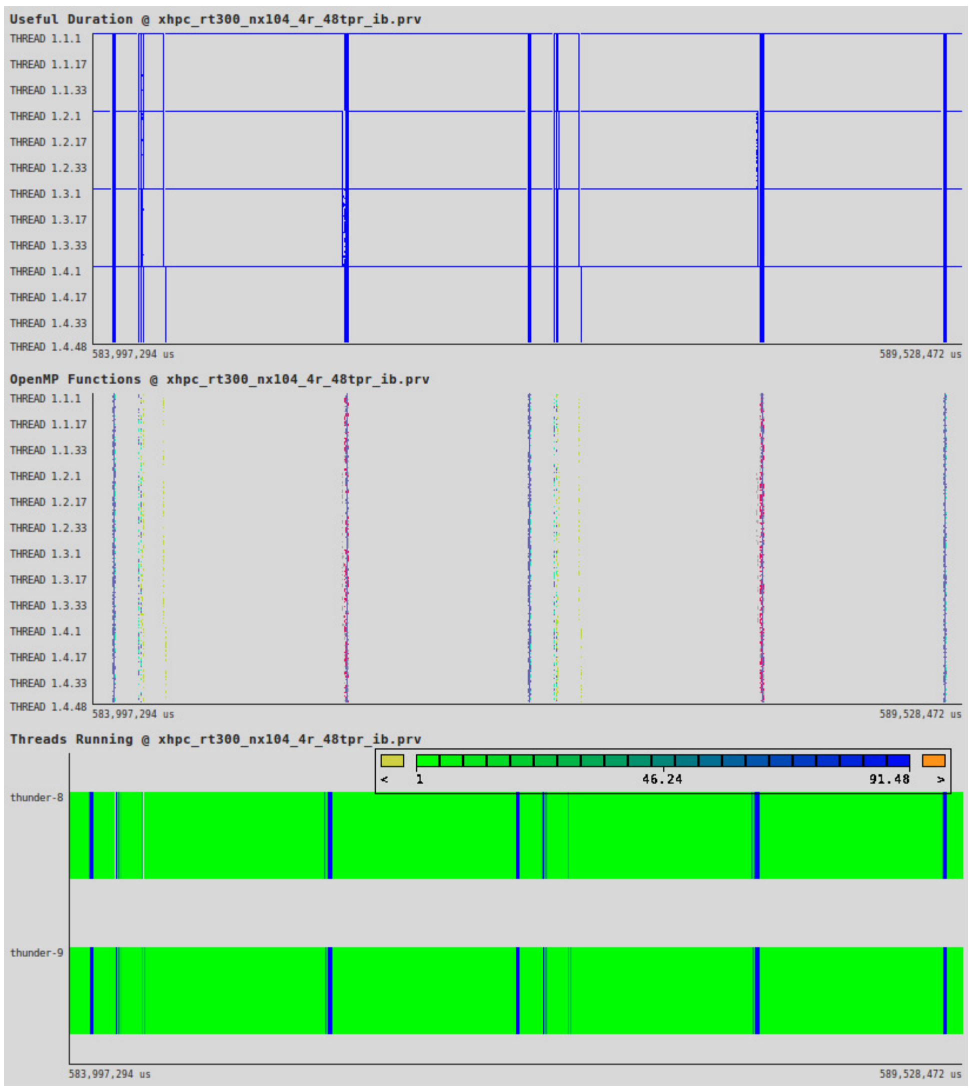


Figure 9: Paraver trace of two full iterations in the optimized Conjugate Gradient

	OpenMP only		MPI only	
Kernel	**GFlops**	**Improvement**	**GFlops**	**Improvement**
DDOT	1.67	1.03	0.89	1.22
WAXPBY	1.61	1.02	0.98	0.99
SpMV	1.57	1.05	1.70	1.05
MG	0.47	1.81	1.61	1.32
Total	**0.53**	**1.76**	**1.57**	**1.27**

Table 2: Performance and speed up, against reference execution with same configuration, of the optimized HPCG version

we can see on different colors the different OpenMP sections, which mostly map to the blue sections on the top (except for the master thread). On the bottom, it is shown how many cores were used per node (i.e. on the two MPI ranks of each node) during the execution, again, it can be seen that green areas (low count of cores executing useful code) is too low most of the time. This behavior leads to a bad usage of the hardware resources when using OpenMP, which at the end translates into having to use MPI only, which increases the total amount of communication that has to be done.

2.3 Optimizations

We based our optimizations on the ones presented by the RIKEN during SC14 [8]. These optimizations can be basically split in:

- Improve memory allocations by allocating contiguous memory
  - Should improve all kernels in general
- Block multicoloring of the indirected graph to parallelize the symmetric Gauss-Seidel
  - This way we could decrease the amount of communications performed by decreasing the number of MPI ranks and increasing the amount of OpenMP threads used by each rank
- Loop optimizations
  - Except the symmetric Gauss-Seidel, the rest of the kernels can benefit from this

In our case, we modified the code to allocate the data structures in a contiguous way in terms of memory. Also, we performed loop unrolling on the SpMV, the DDOT and WAXPBY kernels. Table 2 shows the improvement achieved against the reference version.

The multicoloring of the graph to parallelize the symmetric Gauss-Seidel, it is still on going. This technique is meant to reorder the sparse matrix in a way that nodes of the graph that do not depend on each other are known (i.e. we assign them the same color) so they can be processed in parallel. Considering that the graph generated on the HPCG benchmark is a 27-stencil, then 8 colors can be used [9].

3 QuantumESPRESSO

3.1 Description

QuantumESPRESSO is an integrated suite of codes for electronic-structure calculations and materials modeling at the nanoscale. It is based on density-functional theory, plane waves and pseudopotentials. It is mostly written in Fortran and uses MPI and (optionally) OpenMP to exploit parallelism.

QuantumEspresso uses BLAS, LAPACK and FFT functions for its most heavy computational parts. Therefore, it is a good candidate to analyze the performance of the ARM Performance Libraries (ARM PL) against the optimized libraries already included on the Mont-Blanc software stack.

Table 3 shows the different library configurations used. All the executions were performed on the AMD Seattle platform deployed at BSC facilities.

Execution name	BLAS library	LAPACK library	FFT library
ARMPL	ARM PL v2.2.0	ARM PL v2.2.0	ARM PL v2.2.0
ATLAS	ATLAS v3.11.39	ATLAS v3.11.39	FFTW v3.3.6
OpenBLAS	OpenBLAS v0.2.20	OpenBLAS v0.2.20	FFTW v3.3.6
ARMPL + FFTW	ARM PL v2.2.0	ARM PL v2.2.0	FFTW v3.3.6

Table 3: Library configurations used for QuantumESPRESSO executions on a AMD Seattle with 8 MPI ranks

3.2 Results

For the analysis of QuantumESPRESSO we executed the pwscf-small benchmark input set provided by its developers [1]. This set consists of 4 different inputs. The first one is the smallest of all while the third and fourth are the biggest. The third input is more BLAS focused while the fourth is more FFT focused.

Figure 10 shows the normalized performance for the different inputs as well as the execution time of the more important routines within the execution. For each input, different executions are reported. It can be seen that while using the ARM PL alone does not provide a better performance. This changes when using the ARM PL for BLAS and LAPACK routines while using FFTW library for the FFT routines. Then, the performance is always similar or better at the one obtained with other optimized libraries.

Figure 11 shows, for each input, the normalized execution time against the ARMPL execution for each of the most relevant routines reported by QuantumESPRESSO. Table 4 we can see what executes each of the routines. As we already mentioned, the ARM PL seems to lack performance for the FFT functions. As for the BLAS-centered routines, the performance is similar or better with compared to other BLAS implementations, as can be seen at Figure 11.

3.3 Conclusions

QuantumESPRESSO can benefit from properly optimized BLAS, LAPACK and FFT libraries. Of course, this is not only true for this application. Therefore, the results observed for Quantu-

[1] `http://www.qe-forge.org/gf/project/q-e/frs/?action=FrsReleaseBrowse&frs_package_id=36`

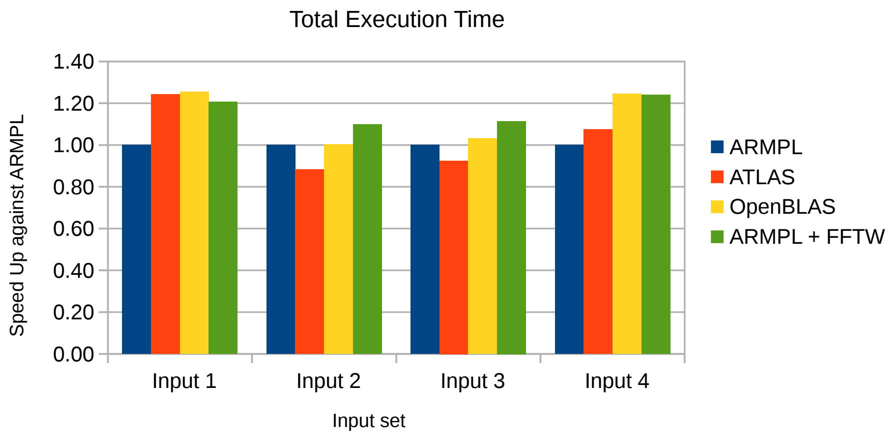


Figure 10: Normalized performance against the ARMPL execution for the different inputs from the pwscf-small input set

Routine name	BLAS + LAPACK	FFT	MPI	IO
calbec	×			
fft		×		
ffts		×		
fftw		×		
interpolate		×		
davcio				×
fft_scatter			×	

Table 4: Operations performed at each of the QuantumESPRESSO reported routines

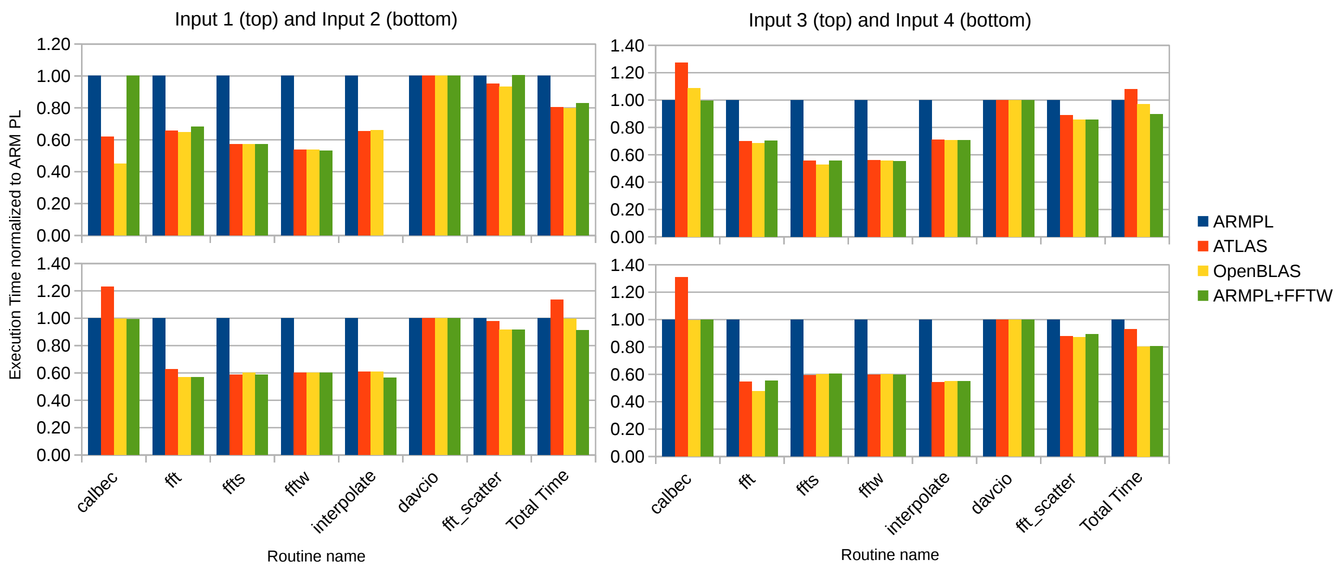


Figure 11: Normalized execution time against the ARMPL execution for the most relevant routines executed within the inputs from the pwscf-small input set

mESPRESSO can and should be used in other applications with similar compute characteristics.

A lack of performance on the FFT routines implemented at the ARM PL can impact the final performance. This effect can be mitigated by preloading a more performant FFT library such as FFTW. In any case, the BLAS and LAPACK routines implemented at the ARM PL perform similar or better to other optimized libraries. This makes the ARM PL a good candidate for applications that can benefit from an optimized BLAS implementation.

The ARM PL are currently in its second major version. And for the results obtained, looks like more effort was put in the BLAS and LAPACK implementations rather than in the FFT one. So room for improvement is available at the FFT routines. Future ARM PL releases could boost the performance of the FFT routines. This means that the work being done here will be expanded as new ARM PL versions are released.

4 Compiler evaluation

Within the Mont-Blanc project, we adopted since 2011 the GNU Compiler Collection (GCC) suite as the main compiler since it was the one providing the widest support for ARM architecture. This hypothesis needs to be validated again with the release of the new ARM HPC Compiler. The ARM HPC Compiler is developed by ARM and it is based on LLVM [10]. It provides C, C++ and Fortran compilers at the current version, which makes a necessity to consider it as an alternative, or even the new main compiler, on our software stack. We report here a preliminary comparison between the ARM HPC Compiler (v. 1.3) and GCC (v. 7.1.0).

4.1 Methodology

For this study we choose four different applications. Two of them are mini-apps already used within the Mont-Blanc project as Lulesh and CoMD. The third one is Polybench. This benchmark suite contains kernels from various application domains such as linear algebra, physics simulation, dynamic programming, etc. Therefore, it was a good candidate to test how well each compiler performs when generating a wide amount of different types of codes. The fourth and last application is QuantumESPRESSO, already analyzed on Section 3. The idea with QuantumESPRESSO is to see the feasibility of using the ARM HPC Compiler with real-world applications and to see how good the generated code is.

For each of these applications, we used only OpenMP implementations (if available), pulling out of the equation MPI implementations since we want to see only the performance obtained by the serial code as well as the performance obtained by the specific OpenMP runtime of each compiler. In this sense, GCC uses GOMP as an OpenMP runtime whereas the ARM HPC Compiler uses KMP runtime. For QuantumESPRESSO though, we used the MPI-only version in order to see also how good the MPI implementation is when compiled as well with the ARM HPC Compiler.

The ARM platforms used for running our experiments are a dual socket node with a 48-core Cavium ThunderX SoC Pass2 on each socket [11] and a single socket node with a 8-core AMD Seattle SoC [12]. QuantumESPRESSO has been executed only on the AMD Seattle SoC. The reason for this choice was to try a complete custom ARMv8 implementation as well as a SoC featuring ARM Cortex-based architecture. In all cases, the compilers optimization flags used have been `-O3 -mcpu={cortex-a57,thunderx}` depending on the platform used for the evaluation. The data gathered in this study will be used as reference while studying the performance of the Mont-Blanc demonstrator platform based on ThunderX-2 deployed by WP3.

4.2 Results

4.2.1 Lulesh

Lulesh is a shock hydrodynamics code developed at Lawrence Livermore National Laboratory. It is written in C++ and has an OpenMP implementation which was used during our experiments. We perform a strong scaling study using the following application parameters: `-s 80 -i 100`, i.e., performing 100 iterations on a data size of 80 elements.

Figure 12 shows performance, scalability and parallel efficiency of Lulesh when compiled with GCC version 7.1.0 and when compiled with the ARM HPC Compiler version 1.3. It can be seen that, even though scalability is poor, when the number of OpenMP threads is higher or equal than 48, the ARM HPC Compiler scales better than GCC, leading to the idea that the OpenMP runtime featured at the ARM HPC Compiler scales better than the one at GCC. Nevertheless, comparing both versions when the number of threads is still low shows that the performance achieved with GCC is slightly better.

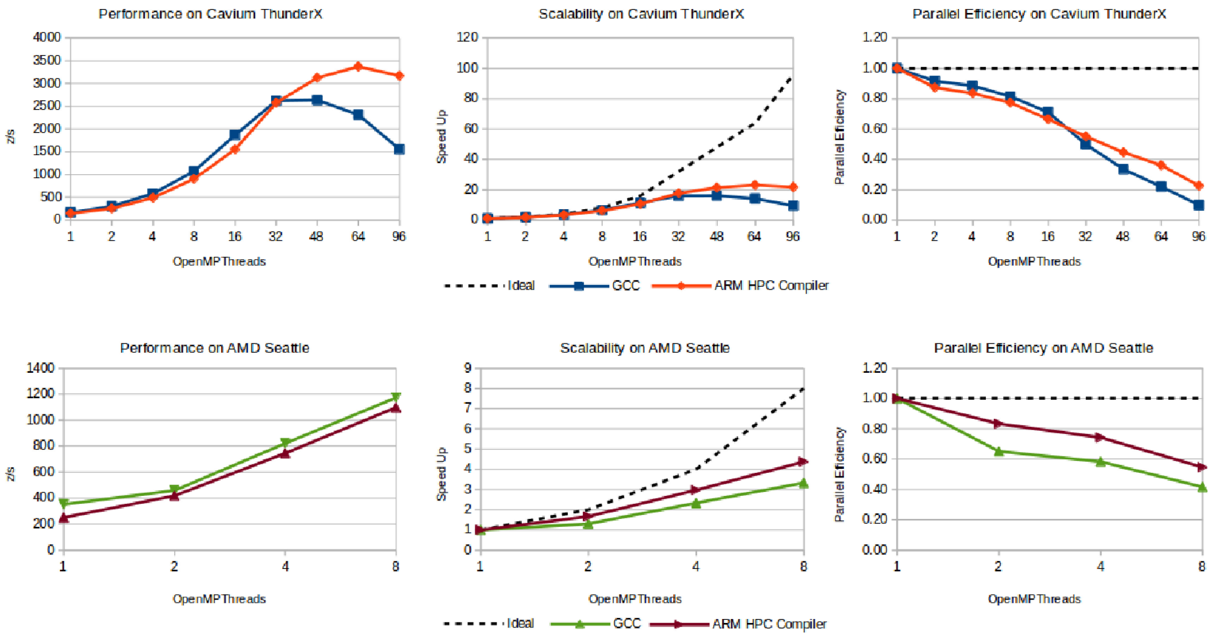


Figure 12: Performance, scalability and parallel efficiency of Lulesh executed in ThunderX and compiled with GCC v7.1.0 and ARM HPC Compiler 1.3

We collected execution traces with Extrae and visualized them with Paraver in order to discover what was producing these differences in performance and scalability. In the case of Lulesh with 16 OpenMP threads, two executions have been performed, one with the binary of Lulesh generated by GCC and the other with the binary generated by the ARM HPC Compiler.

We note that the ARM HPC Compiler generates ∼45% more instructions than GCC. Looking at the execution time, it takes 6% more total execution time, but 26% more time per iteration. The detailed measurements are shown in Table 5 for ARM HPC Compiler and Table 6 for GCC.

These observations led us to look at the distribution of the Instructions Per Cycle (IPC) depicted in Figure 13 and Figure 15. Both figures are histograms of the IPC value: on the x axis we show bins of IPC growing in steps of 0.01, from 0.06 up to 0.93, while on the y axis we show the threads. The value of each cell/pixel is color coded from green (lower occurrence) to blue (higher occurrence). Figure 13 clearly shows that the code generated by the ARM HPC

Compiler tends to run at higher IPC than GCC. This mitigates the difference between the overall execution time and the total number of instructions executed.

Another factor that can contribute to compensate the aforementioned difference between HPC ARM Compiler and GCC is the load imbalance within iterations. In Figures 14 and 16 we show one iteration of Lulesh in a timeline: on th x axis we represent the time in μs, on the y axis we plot each of the 16 threads while in color code, from green (low value) to blue (high value) the number of instructions execution in a given portion of code. Within an iteration there are small differences of execution time between threads although they execute a similar number of instructions. In these pictures it is also visible the aforementioned difference of number of instructions comparing the upper bound of the color scale: in the case of GCC blue represents $\sim 33 \times 10^6$ instructions while for the ARM HPC Compiler blue is $\sim 52 \times 10^6$ instructions.

Overall figures	**Value**
Total Instructions	4.36×10^9
Average Per Thread	0.27×10^9
Avg/Max	0.93
Total Execution Time	51.35 s
Time Per Iteration	495.29 ms

Table 5: Number of instructions and execution time for Lulesh compiled with the ARM HPC Compiler.

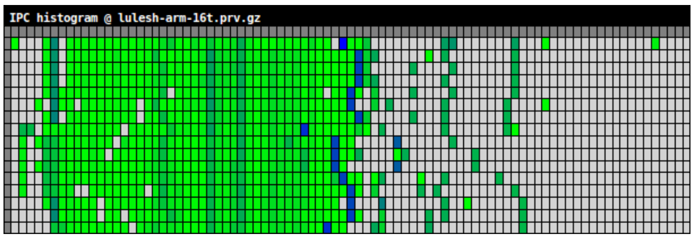


Figure 13: Histogram representing the IPC distribution over the threads while running Lulesh compiled with the ARM HPC Compiler.

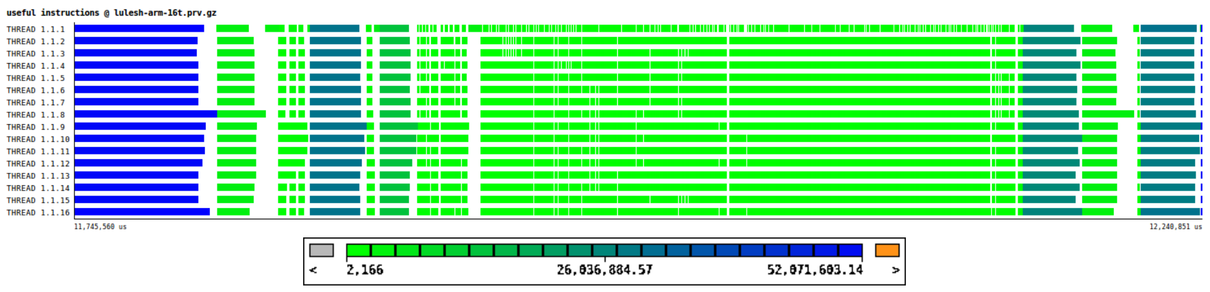


Figure 14: Timeline representation of one iteration of Lulesh compiled with the ARM HPC Compiler.

Overall figures	**Value**
Total Instructions	2.99×10^9
Average Per Thread	0.19×10^9
Avg/Max	0.93
Total Execution Time	48.27 s
Time Per Iteration	393.04 ms

Table 6: Number of instructions and execution time for Lulesh compiled with GCC.

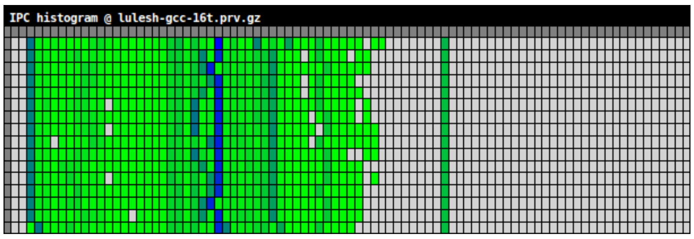


Figure 15: Histogram representing the IPC distribution over the threads while running Lulesh when compiled with GCC.

It is interesting to analyze in a similar way the Extrae trace with 64 OpenMPI threads, as from 12 it seems that the performance of the ARM HPC Compiler takes over GCC. Indeed, even if the number of total instructions of one execution of Lulesh compiled with the ARM HPC Compiler is higher than the number of total instructions generated by GCC, the IPC histogram show a slight improvement for the ARM HPC Compiler. In Figure 17 we show the histogram of IPC where on the x axes we place bins from 0.01 (left) to 0.9 (right) with bin size

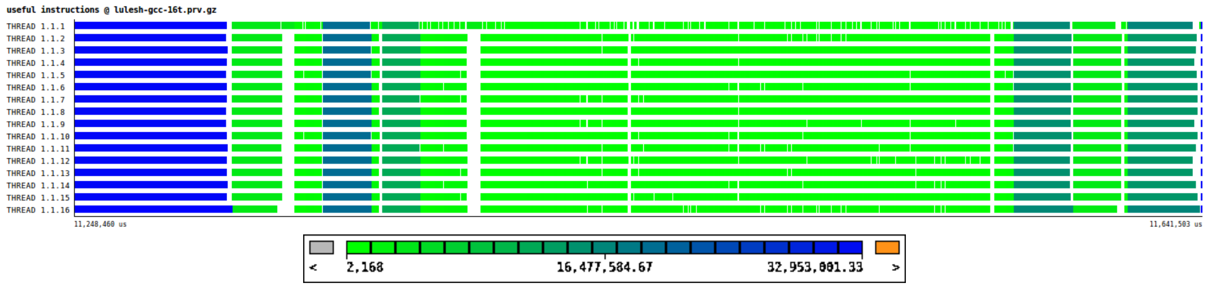


Figure 16: Timeline representation of one iteration of Lulesh compiled with GCC.

of 0.01. One can see that for the GCC compiler (bottom part of Figure 17) not only the blue pixels concentrate at lower IPC, but also their distribution looks more sparse, resulting in a less efficient code compared to the ARM HPC Compiler.

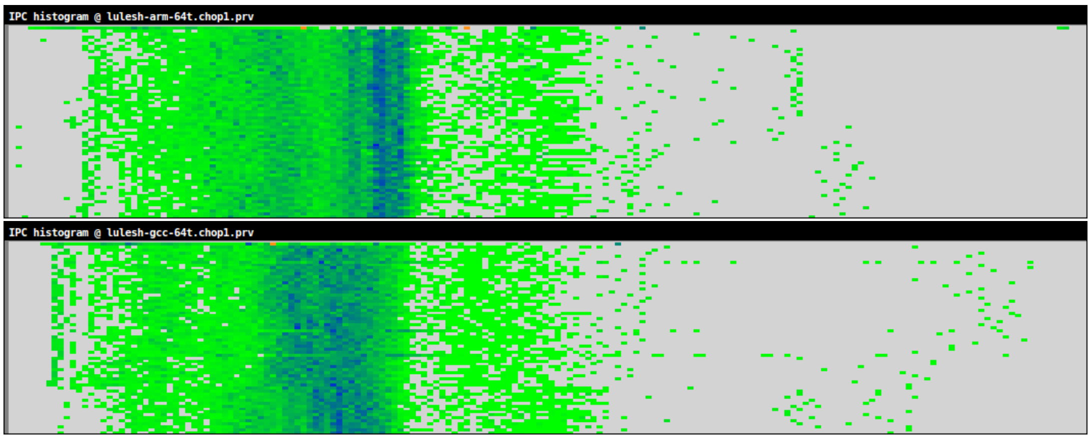


Figure 17: Histogram representing the IPC distribution over the threads while running Lulesh compiled with ARM HPC Compiler (top) and with GCC (bottom).

Figure 18 shows different histograms for the cases with 16 and 64 OpenMP threads and with the binary generated by GCC and the ARM HPC Compiler. Now, the first row of each histogram represent the different parallel regions at the code. The second row represents with a gradient color code how well-balanced that parallel region was (i.e., how properly distributed was the computation within the specific parallel region across the OpenMP threads): the more blue the more balanced the parallel region was while the more green the less balanced. It can be seen that, for the executions with 16 OpenMP threads (i.e., where GCC outperforms the ARM HPC Compiler) the load balance is better with the GCC binary. At the other side, for the execution with 64 OpenMP threads, the binary generated with the ARM HPC Compiler presents better load balance, which leads to the better scalability showed before.

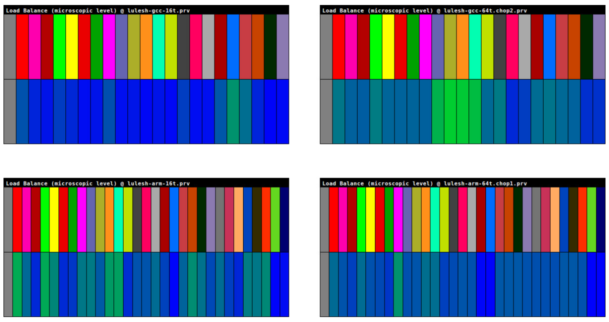


Figure 18: Histogram showing the load balance on the different parallel regions found within the computation parts of the Lulesh execution

4.2.2 CoMD

CoMD is a reference implementation of a typical classical molecular dynamics algorithms and workloads. It is written in C and features an OpenMP implementation which was used during our experiments. We perform a strong scaling study using the following application parameters: `-N 10 -n 5 -e -i 1 -j 1 -z 1 -x 60 -y 60 -z 60`, i.e., performing 10 time steps, including the computation of `eam` potential, over a cube of $60 \times 60 \times 60$ unit cells, using one rank for each direction and dumping output every 10 time steps, so only at the end.

Figure 19 shows performance, scalability and parallel efficiency of CoMD when compiled with GCC version 7.1.0 and when compiled with the ARM HPC Compiler version 1.3. It can be seen that performance is always better with GCC as well as scalability. This behaviour is different if compared with Lulesh, where the scalability was better with the ARM HPC Compiler. For both applications we considered the Figure of Merit (FOM) provided by each in order to make the studies. As for the scalability, we compare against the single thread execution.

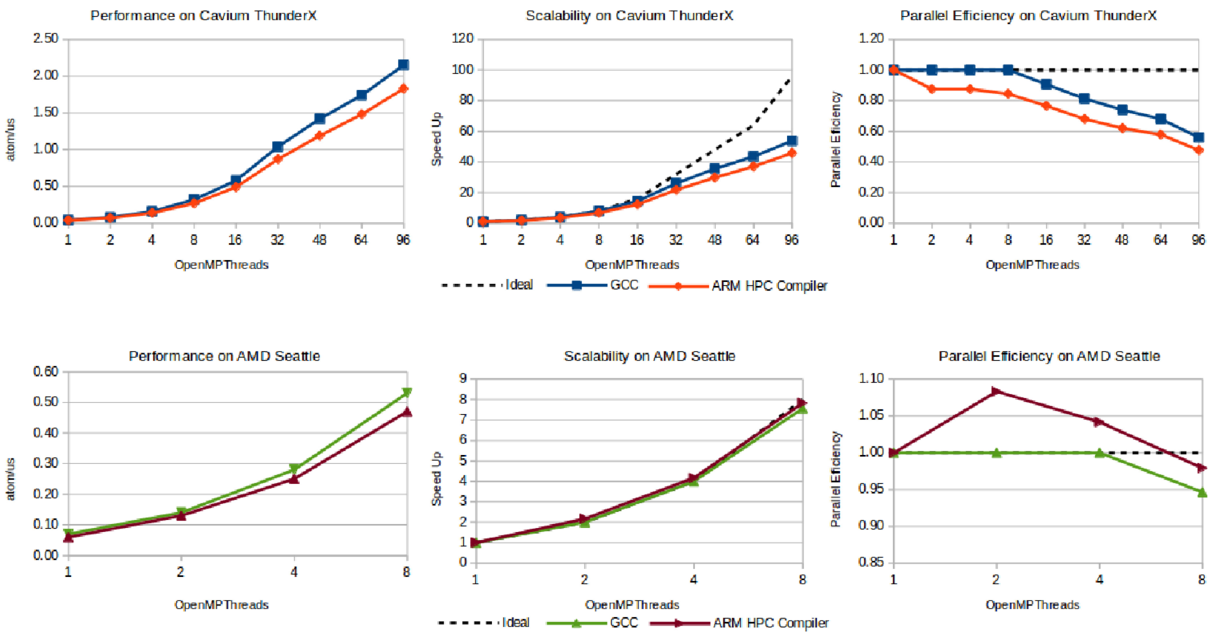


Figure 19: Performance, scalability and parallel efficiency of CoMD executed in ThunderX and compiled with GCC v7.1.0 and ARM HPC Compiler 1.3

The analysis of several CoMD Extrae trace shows once more the trend revealed while studying Lulesh: the ARM HPC Compiler generates $\sim 23\%$ more instructions than GCC for executing the same code. However, when we analyze the histograms of IPC in two executions of CoMD using 16 and 64 OpenMP threads we notice a more disperse distribution of the IPC when executing with the ARM HPC Compiler. It seems reasonable to assume that the less noisy distribution of the IPC for GCC codes explains its slightly higher performance. The graphical analysis of the IPC is reported in the histograms of Figure 20 for the configuration using 16 OpenMP threads and Figure 21 for 48 OpenMP threads.

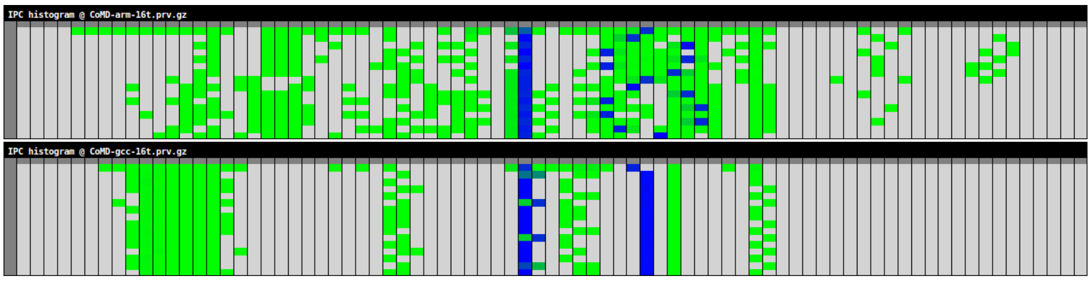


Figure 20: Histogram representing the IPC distribution over 16 OpenMP threads while running CoMD compiled with ARM HPC Compiler (top) and with GCC (bottom).

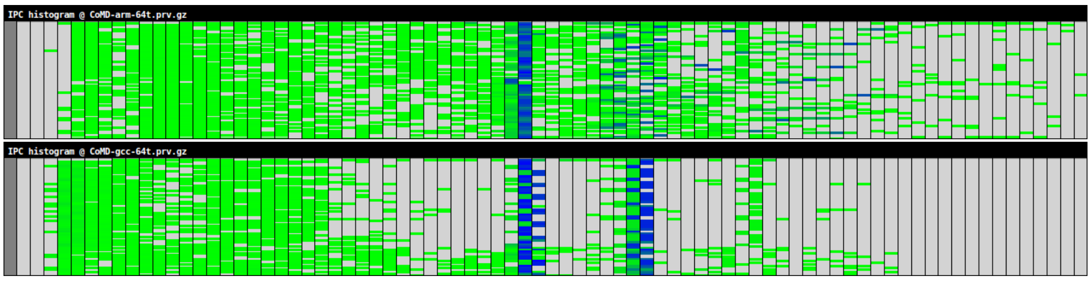


Figure 21: Histogram representing the IPC distribution over 64 OpenMP threads while running CoMD compiled with ARM HPC Compiler (top) and with GCC (bottom).

Figure 22 shows the load balance (second row, green is lower load balance while blue means higher load balance) of each parallel region (different color on first row) for each execution (i.e., 16 OpenMP threads with GCC and ARM HPC Compiler and the same with 64 OpenMP threads). We can see that in both OpenMP configuration, GCC generates a code that ends up with a higher load balance.

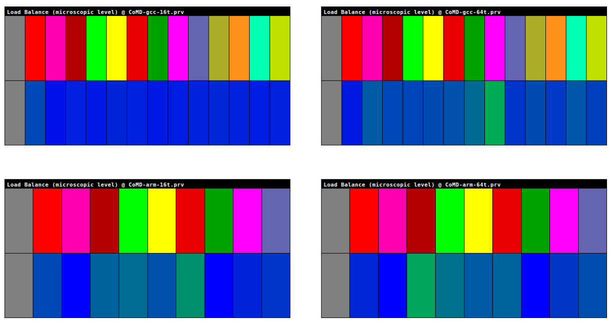


Figure 22: Histogram showing the load balance on the different parallel regions found within the computation parts of the CoMD execution. On the left we can see the load balance for an execution of CoMD using 16 OpenMP threads, compiled with GCC (top) and compiled with the Arm HPC Compiler (bottom). On the right the load balance for an execution with 64 OpenMP threads with GCC (top) and the Arm HPC Compiler (bottom)

4.2.3 Polybench

Polybench contains a total of 30 different kernels split in three categories. Each category differs from the other in the kind of computation it performs. Table 7 shows a summary of the kernels as well as what they compute, please note that Linear Algebra category has been split in three since it is actually split in three subcategories.

Category	Kernel	Summary
Datamining	correlation	Correlation computation
	covariance	Covariance computation
Linear Algebra - BLAS	gemm	Matrix-multiply C=alpha*A*B+beta*C
	gemver	Vector multiplication and matrix addition
	gesummv	Scalar, vector and matrix multiplication
	symm	Symmetric matrix-multiply
	syr2k	Symmetric rank-2k operations
	syrk	Symmetric rank-k operations
	trmm	Triangular matrix-multiply
Linear Algebra - Kernels	2mm	2 Matrix multiplications (alpha * A * B + beta * D)
	3mm	3 Matrix multiplications ((A*B)*(C*D))
	atax	Matrix transpose and vector multiplication
	bicg	BiCG Sub Kernel of BiCGStab Linear solver
	doitgen	Multiresolution analysis kernel
	mvt	Matrix vector product and transpose
Linear Algebra - Solvers	cholesky	Cholesky decomposition
	durbin	Teoplitz system solver
	gramschmidt	Gramschmidt
	lu	LU Decomposition
	ludcmp	LU Decomposition
	trisolv	Triangular solver
Medley	deriche	Edge detection filter
	floyd-warshall	Find shortest path in a weighted graph
	nussinov	Dynamic programming algorithm for sequence alignment
Stencils	adi	Alternating direction implicit solver
	fdtd-2d	2-D Finite different time domain kernel
	heat-3d	Heat equation over 3D data domain
	jacobi-1d	1-D Jacobi stencil computation
	jacobi-2d	2-D Jacobi stencil computation
	seidel-2d	2-D Seidel stencil computation

Table 7: Polybench kernels summary

For each kernel, we generated two binaries, one with GCC version 7.1 and another with the ARM HPC Compiler version 1.3. Then, for each binary, we executed it 10 times in order to compute the average execution time of each kernel. This average time was used to compute the normalized execution time against GCC binary's execution. This procedure was done for each ThunderX-based node as well as for the Seattle-based node. The reason is that since they feature different ARMv8 cores, specific optimization flags can be passed to the compiler to allow more architecture-aware optimizations of the binary.

Figure 23 show the results obtained. Please note that, for each category of kernels, we show only two metrics. One is the normalized execution time of the ARM HPC Compiler's binary on the ThunderX against the execution time of the GCC's binary on the ThunderX. The other metric is the same but executed on the Seattle SoC.

By looking at the normalized execution time, we can see that even within the same category, there is not a clear trend: some of the kernels the binary generated by GCC performs better while for some others it is the other way around. Nevertheless, there are a few cases where the ARM HPC compiler binary performs very well compared to the GCC counterpart as for the *gemm* kernel executed on ThunderX or the *gesummv* and *bicg* kernels on the AMD Seattle. As for the rest, there are some other cases where GCC's binary greatly outperforms the ARM HPC Compiler one as in the *durbin*, *gessumv*, *trisolv*, *atax* and *jacobi-1d* kernels on the ThunderX and the *mvt* kernel on the Seattle.

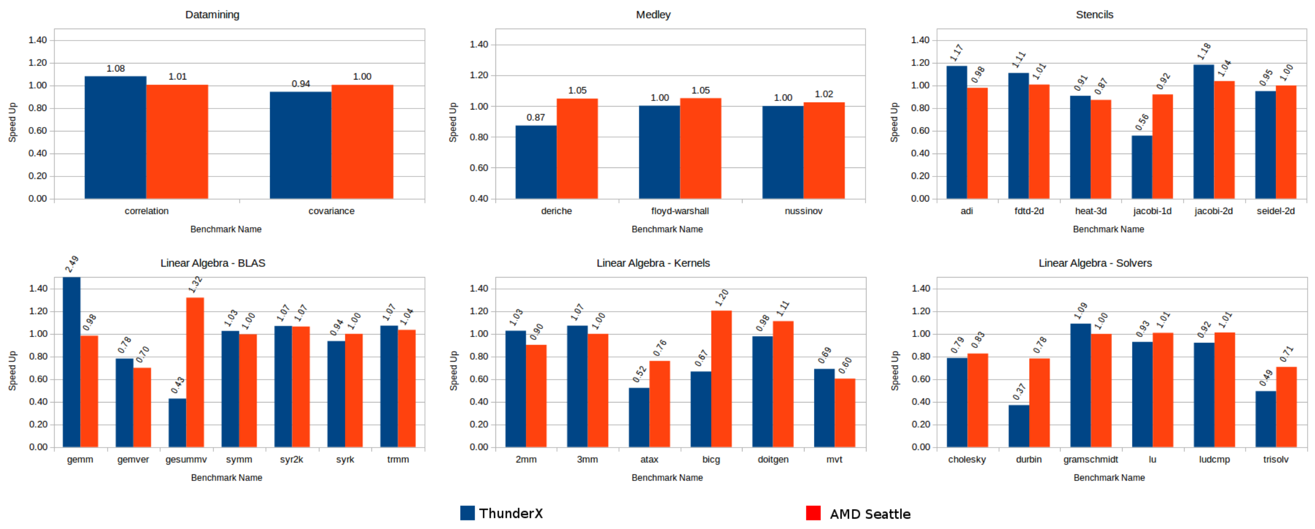


Figure 23: Normalized execution time against GCC binaries of Polybench kernels against ARM HPC Compiler binaries for Cavium ThunderX and AMD Seattle SoC's

4.2.4 QuantumESPRESSO

As already explained in Section 3, QuantumESPRESSO is an integrated suite of codes for electronic-structure calculations and materials modeling at the nanoscale. It is writen mostly in Fortan and uses MPI for exploiting parallelism.

For our experiments, we choose 4 different configurations:

- GCC v7.1.0 + MPICH v3.2 + ARM Performance Libraries v2.2.0 + FFTW v3.3.6
- GCC v7.1.0 + OpenMPI v2.1.1 + ARM Performance Libraries v2.2.0 + FFTW v3.3.6
- ARM HPC Compiler v1.3.0 + MPICH v3.2 + ARM Performance Libraries v2.2.0 + FFTW v3.3.6
- ARM HPC Compiler v1.3.0 + OpenMPI v2.1.1 + ARM Performance Libraries v2.2.0 + FFTW v3.3.6

We are only using the combination of the ARM Performance Libraries with the FFTW library since it was the one presenting the better performance in the study presented in Section 3.

In Figure 24 we can see the normalized performance against the GCC + MPICH configuration. We chose this version as baseline since it is a typical compiler configuration used in the clusters deployed at BSC. The variation in the overall performance is not significant (below 10%).

If we look at each of the most significant routines we can see some differences though. In Figure 25 we can see the normalized execution time against the GCC + MPICH configuration for each of those routines. As before, no significant differences can be appreciated except for the MPI focused routines (i.e., *fft_scatter*), where the OpenMPI version, no matter if along with GCC or the ARM HPC Compiler, performs worse than MPICH. This is only for two inputs, so we think that this behavior is due to some issue with the specific message size in communications or a lack in intra-node communications performance for specific messages sizes.

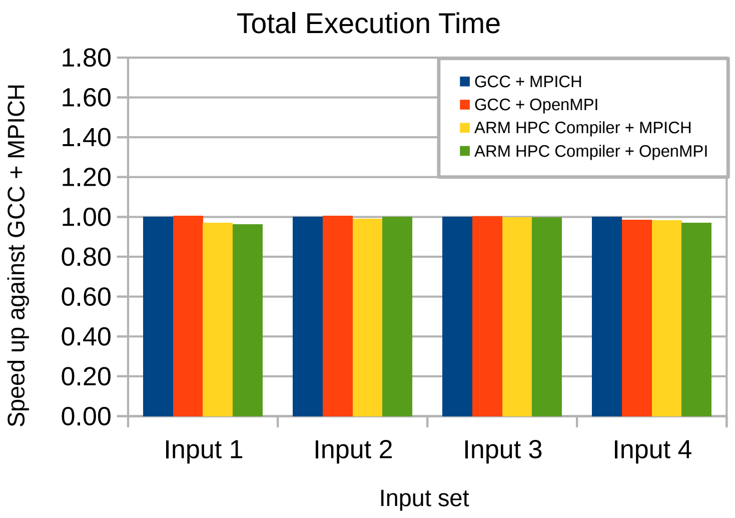


Figure 24: Normalized performance against the GCC + MPICH execution for the different inputs from the pwscf-small input set

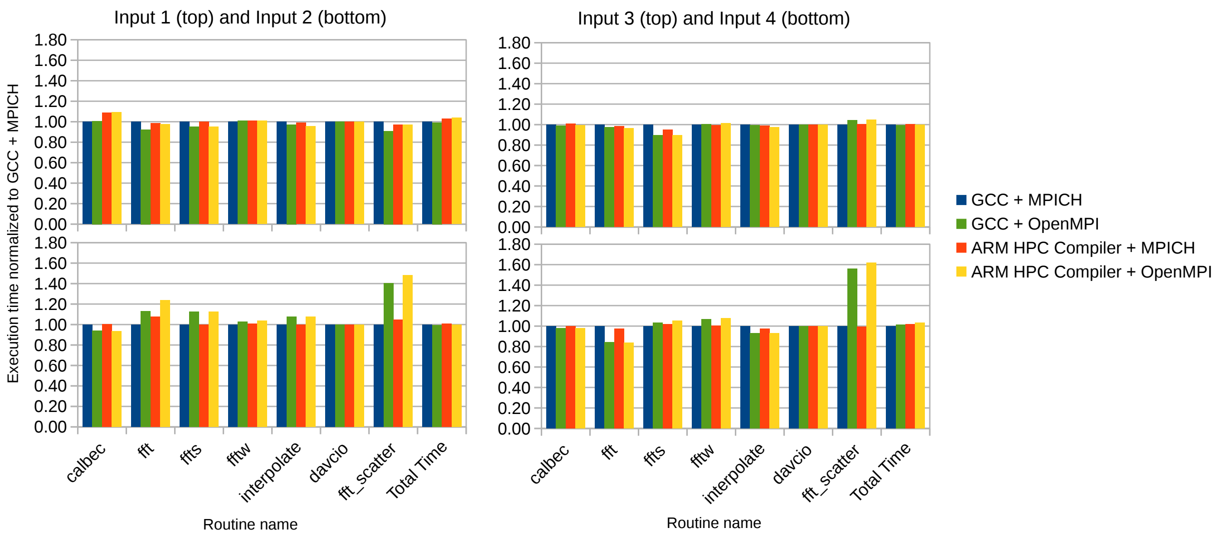


Figure 25: Normalized execution time of most relevant QuantumESPRESSO routines against the GCC + MPICH execution for the different inputs from the pwscf-small input set

4.3 Conclusions

With the results obtained on the three benchmarks, we cannot say whether GCC or the ARM HPC Compiler generate better binaries. In any case, we noticed that when using OpenMP, the runtime featured by each compiler suite could have a noticeable impact on the final performance as seen in Lulesh. As a summary, we observed the following:

- GCC generated better performing serial code than ARM HPC Compiler:
  - At least for real world alike workloads as the one found in Lulesh and CoMD.
  - Compiler optimizations should be further studied since both GCC and the ARM HPC Compiler do not enable the same optimizations by default.
- ARM HPC Compiler's OpenMP runtime seems to perform better when the parallel regions are small: this is the case of Lulesh and indeed it is not the case of CoMD, where granularity of threads is bigger.
- ARM HPC Compiler works for real-world applications such as QuantumESPRESSO, performing the same as GCC. Next iterations of the compiler might change this

As expected in such a complex system, there is no "clear winner" in this preliminary comparison. This work rises indeed several questions that are worth further investigation. It seems that the performance degradation of ARM HPC Compiler is due to a larger number of instruction executed: *i)* Is this always the only cause of performance degradation? *ii)* Why and where the ARM HPC Compiler generates makes the CPU executing more instructions? *iii)* Which kind of instructions are executed more compared to GCC? *iv)* Which kind of optimization flags can be used to better control the compilation process with the ARM HPC Compiler? During August 2017, at the end of the writing of this document, a new version of the ARM HPC Compiler has been released [10]. We will continue investigating these questions in the time frame of the project in close collaboration with ARM, as partner of the consortium.

5 Eikonal Solver - UGRAZ

5.1 Description

The Eikonal equation and its variations (forms of the static Hamilton-Jacobi and level-set equations) are used as models in a variety of applications. These applications include virtually any problem that entails the finding of shortest paths, possibly with inhomogeneous or anisotropic metrics. The Eikonal equation is a special case of non-linear Hamilton-Jacobi partial differential equations (PDEs). In this work, we consider the numerical solution of this equation on a 3D domain with an inhomogeneous, anisotropic speed function:

$$\begin{cases} H(x, \nabla\varphi) & = \sqrt{(\nabla\varphi)^\top * M * \nabla\varphi} = 1, \forall x \in \Omega \subset R^3 \\ \varphi(x) & = B(x), \forall x \in B \subset \Omega \end{cases}$$

Where Ω is a 3D domain, φ(x) is the travel time at position x from a collection of given (known) sources within the domain, M(x) is a 3 * 3 symmetric positive-definite matrix encoding the speed information on Ω, and B is a set of smooth boundary conditions which adhere to the consistency requirements of the PDE. We approximate the domain Ω by a planar-sided tetrahedralization denoted by Ω_T. Based upon this tetrahedralization, we form a piecewise linear approximation of the solution by maintaining the values of the approximation on the set of vertices V and employing linear interpolation within each tetrahedral element in Ω_T.

5.2 Analysis

In order to benchmark the Eikonal solver two different meshes are used the rabbit heart around three millions tetrahedrons and human heart mesh around twenty-four millions tetrahedrons. The volumetric mesh specifying the geometry of the heart is mandatory for any simulation. The mesh definition is split in three different files. The first file contains the nodes of the mesh, the second contains the volumetric elements and the third the fibre orientation per volumetric element. The file extension corresponding to these three files are, .pts, .elem and .lon. Scalability results for Eikonal solver are gathered from executing it on:

Intel platforms:

- Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
- Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz

ARM platforms:

- ThunderX mini-cluster
- JetsonTX-1 cluster

The new OpenMP version of the Eikonal solver is three times faster than the old version on ARM platforms and two times faster on Intel platforms. This performance improvements came as a result of some algorithmic changes, such as the replacement of the ”active list” set find function, logarithmic in size, with a bitset array with constant complexity and critical blocks removal. The accumulation of ”active list” is now done in parallel and there is no conversion from active set to vector since the vector is filled in parallel from the set during the accumulation phase.

The memory footprint reduction of the solver improved the performance furthermore. In this method the number of memory transfers is reduced by precomputing all the needed inner products for each tetrahedron in reference orientation, 18 floats in total. A second step replaces 12 memory accesses per tetrahedron by on-the-fly computations from 6 given data per tetrahedron which reduces the memory footprint to these 6 numbers in total[13].

A set of different tests where performed on different platforms using TBunnyC and H4C440 meshes, with 3,073,529 and 24,400,999 tetrahedral elements respectively. Table 8 shows the execution time, the speed-up and the efficiency for the old version of the Eikonal solver, parallelized solely by OpenMP, using TBunnyC mesh on the Mont-Blanc prototype. Table 9 shows

Table 8: Old version of the Eikonal Solver execution on the Mont-Blanc prototype

Time using 1-Thread (sec.)	**Time using 2-Thread (sec.)**	**Speed-Up**	**Efficiency (%)**
106.31	74.03	1.43	71.81

the execution time, the speed-up and the efficiency for the new version of the Eikonal solver, parallelized solely by OpenMP, using TBunnyC mesh on the Mont-Blanc prototype. As we see on the execution time respectively the new version is approximately 2 times faster than the old version and the efficiency is better, this is as result of many improvements of the code during the last months.

Tables 11 and 12 show the execution time, the speed-up and the efficiency of the Elikonal application running on the ThunderX mini-cluster and on the Xeon Phi.

Table 9: New version of the Eikonal Solver execution on the Mont-Blanc prototype

Time using 1-Thread (sec.)	Time using 2-Thread (sec.)	Speed-Up	Efficiency (%)
57.2	34.1	1.67	83.84

Table 10: Eikonal Solver execution for the rabbit heart mesh on ThunderX

#Cores	#Execution time (sec.)	#Speed-Up	#Efficiency
1	49.43	1	1
2	27.44	1.78	0.89
4	14.73	3.35	0.84
8	7.54	6.55	0.82
16	3.95	12.51	0.78
32	1.98	24.96	0.78
48	1.43	34.56	0.72
64	1.21	40.85	0.64

Table 11: Eikonal solver execution for the human heart mesh on ThunderX

#Cores	#Execution Time (sec.)	#Speed-Up	#Efficiency
1	381.01	1	1
2	251.58	1.51	0.75
4	138.06	2.75	0.68
8	72.51	5.25	0.65
16	38.10	10.01	0.62
32	19.44	19.59	0.61
48	13.15	28.97	0.60
64	12.70	30.00	0.47

Table 12: Eikonal solver execution for the rabbit heart mesh on Xeon Phi

#Cores	#Execution Time (sec.)	#Speed-Up	#Efficiency
1	64.1	1	1
2	35.7	1.97	0.89
4	20.4	3.14	0.78
8	10.6	5.50	0.75
16	5.5	11.65	0.73
32	2.81	22.81	0.71
48	1.85	34.64	0.72
64	1.42	45.14	0.70
96	1.09	58.8	0.61
128	0.90	71.22	0.55
256	0.68	94.2	0.36

The old version of Eikonal solver was not so good with respect to scaling where the efficiency reaches 30% using only 12 cores. With the latest changes it also scales on Xeon Phi processors as in Figure 26.

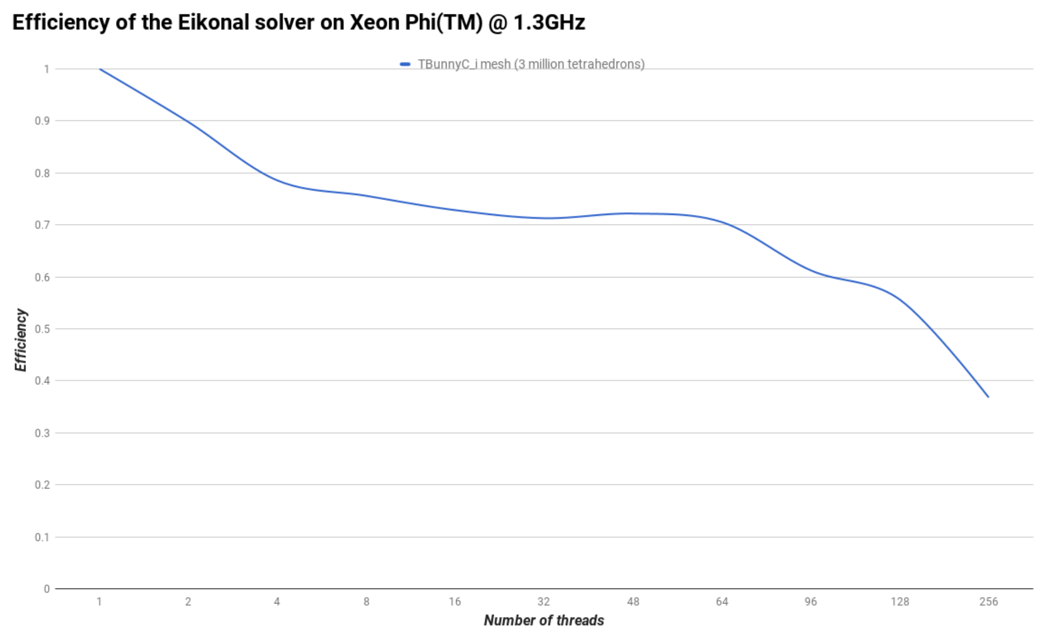


Figure 26: Efficiency of the Eikonal solver on the Intel Xeon Phi.

As we see in the Figure 27 it scales on the first socket and when we jump from the first socket to the second we see that the efficiency starts to drop in the levels around 60%. This version is three times faster than the old version on the ThunderX platforms.

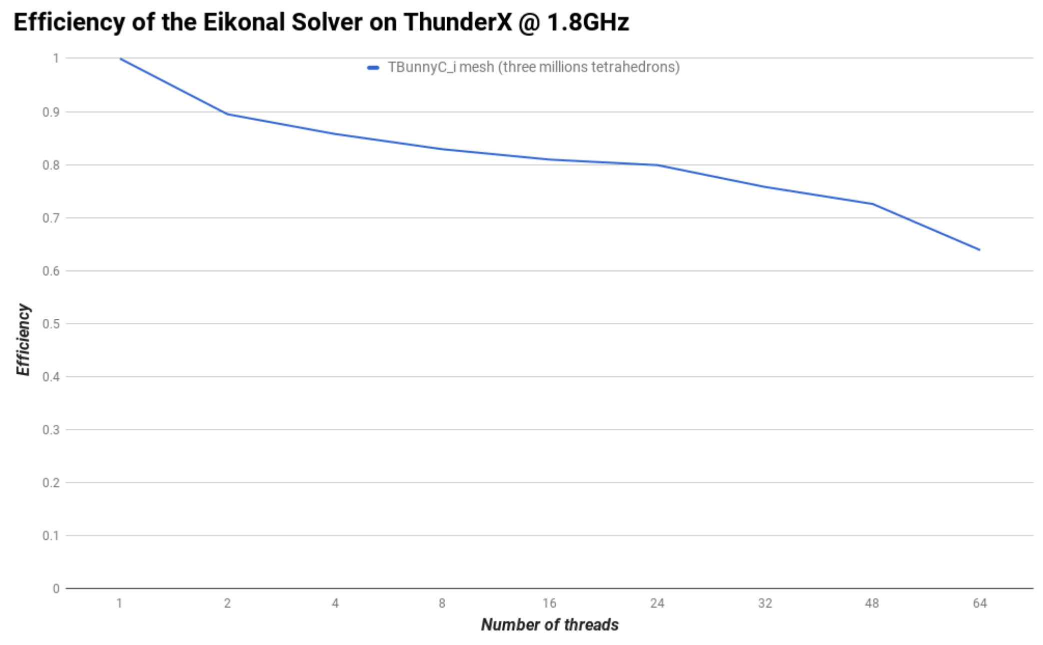


Figure 27: Eficiency of the Eikonal solver on ThunderX.

Profiling of the Eikonal application is performed using the Extrae tool to generate trace files and Paraver to visualize and analyse the traces. Traces are generated by Extrae on the Merlin node, running the Eikonal solver using 8 cores.

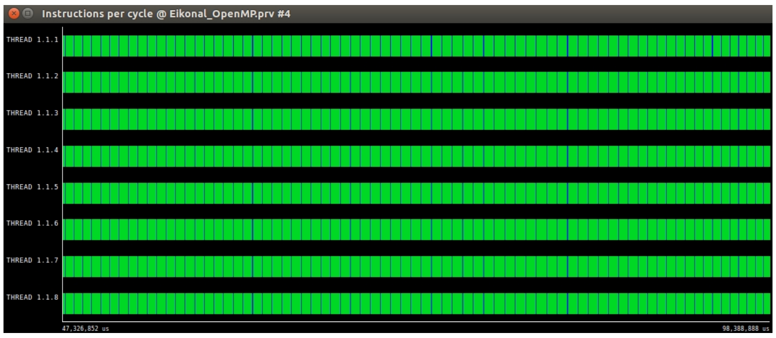


Figure 28: Instruction per cycle of Eikonal on Merlin.

We also identify in the traces a region that limits the scalability at very large core counts as shown in Figure 29. The active list contains relatively small numbers of the nodes on the domain, and it seems very difficult for improvements at very large core counts because than the threads do not have enough work.

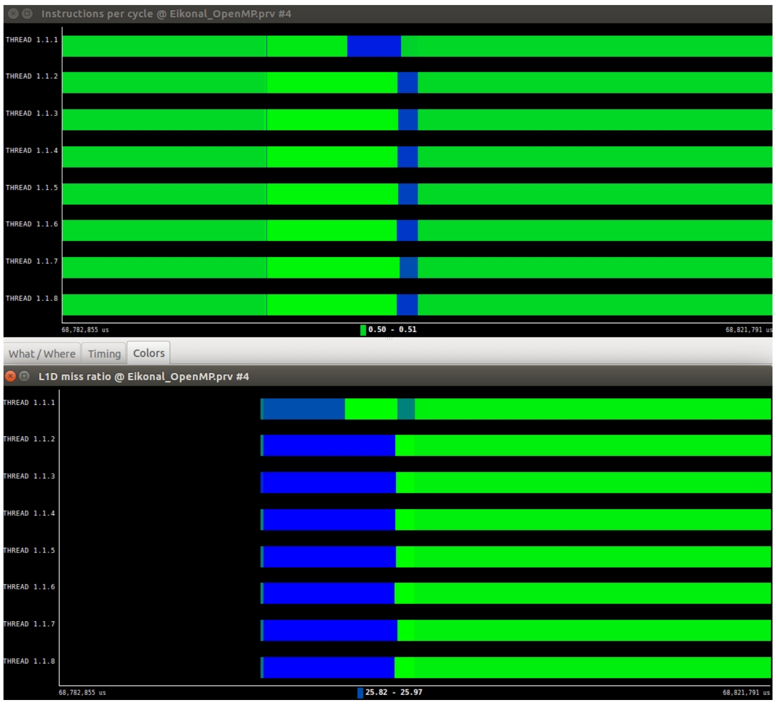


Figure 29: Instruction per cycle zoom-in and L1D cache misses of Eikonal on Merlin.

On the ThunderX mini-cluster we also see some noise during the execution of the Eikonal solver, which might be caused by the ThunderX node itself because we do not observe that on the Intel platforms.

5.3 Energy Measurements

There are various solutions to measure power and energy consumption in HPC. Approaches to measure power consumption usually use power meters or microcontroller-based meters measuring the current and voltage. Recent processors have built-in energy consumption measuring capabilities. Starting with the Sandy Bridge processors, for example, Intel provides the RAPL (Running Average Power Limit) interface. However, these internal performance meters often can not match the precision of external power meters. The correlation between the RAPL interface measurement and the reference measurement depends on the workload type. Libraries such as pmlib and light-weight tools such as Likwid or interfaces such as PAPI are using the RAPL interface to manage the energy consumption. To calculate the energy and the power that is consumed on the Intel platforms we use what is available on the device on-chip energy counters. We have used different tools to calculate the energy that is consumed during the runtime of our applications on the Intel platforms, such as: PAPI, perf, PMU, power governer, and all this method reports correct results and are widely used nowadays.

The value used on this deliverable are gathered using PAPI, reading RAPL counters that are collected locally on the machine under test. These counters have the following restriction, the energy counters are reported every millisecond that makes this counters to produce errors for the application runtime around millisecond, and also the reported energy does not start with the starting of the application, therefore we need to synchronize the start of our application. To compare the values if they are correct, we gathered values also using the perf tool that comes with the Linux kernel while uses the perf event interface. At the same time we can gather other hardware performance counter values including cycles and cache misses. On the Intel Platforms we count the energy measurements which are reported by the energy counters that are restricted to the processor, uncore and memory. Power monitor/controlling is available for:

- Package power plane (PKG)
- Cores power plane (PP0)
- uncore devices (PP1)
- DRAM power plane

In the Figure 30, we see the difference of the energy measurements for two of the Eikonal solver versions, with and without memory footprint and the difference on energy consumption.

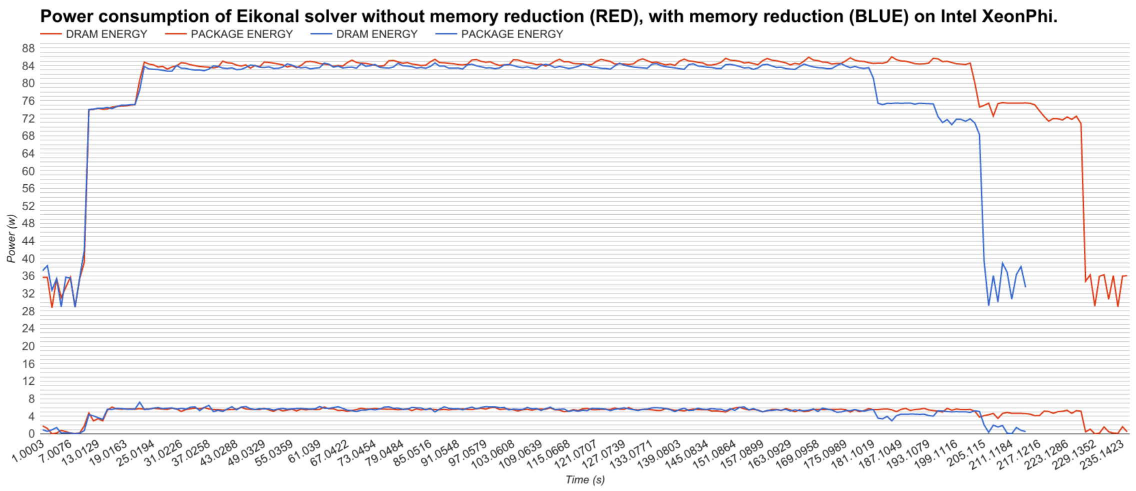


Figure 30: Eikonal solver on the Intel Xeon Phi @ 1.3GHz

The measurements on the ARM platforms are done through the external dedicated power monitoring tool called Yokogawa WT230 Digital Power Meter. It collects power data (V, W, A) from the 4 Thunder nodes since they only use one power supply for all of them. To make a fair comparison between different platforms we calculate the total energy consumed by our application during the runtime excluding the energy that is consumed by idle status. A detailed view of these reporting is shown in Figure 31, when we have executed two versions of Eikonal solver using 20 threads in one node of the ThunderX mini-cluster.

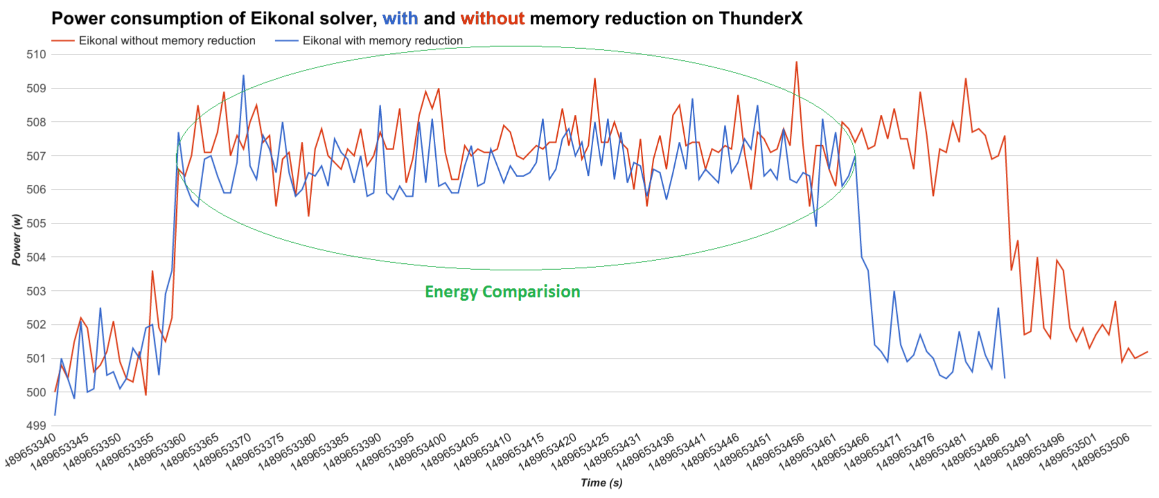
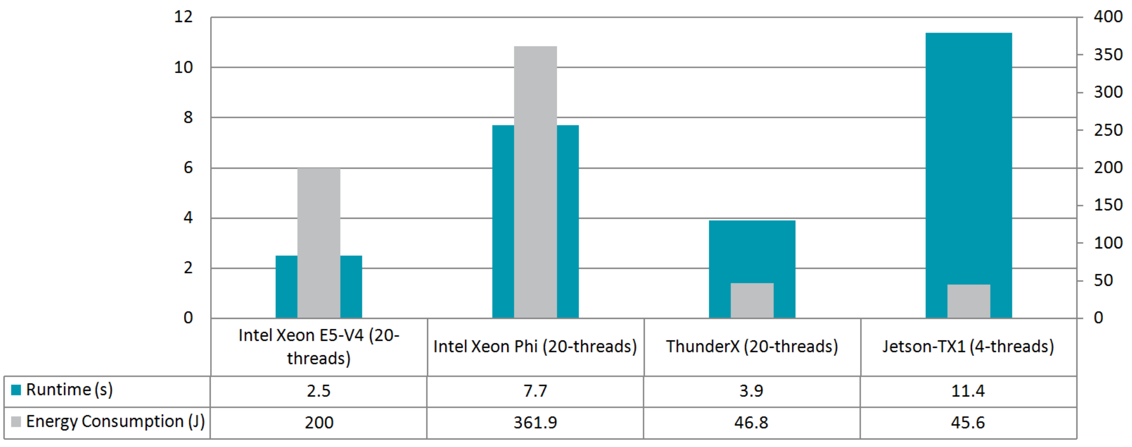


Figure 31: Eikonal solver on the ThunderX @ 1.8GHz

Runtime and energy consumption of the Eikonal solver on different machines are shown on Figure 32. We can see that ThunderX is 1.5x slower than Xeon E5. With respect to energy consumption ThunderX is approximately 4.3 times more efficient.



	Intel Xeon E5-V4 (20-threads)	Intel Xeon Phi (20-threads)	ThunderX (20-threads)	Jetson-TX1 (4-threads)
🟦 Runtime (s)	2.5	7.7	3.9	11.4
⬜ Energy Consumption (J)	200	361.9	46.8	45.6

Figure 32: Runtime and energy consumption of the Eikonal solver on Intel

.

5.4 Conclusions

- The new OpenMP version of the Eikonal solver scales better on both Intel and ARM platforms achieved by:
  - Memory footprint reduction[13].

  - Replacement of "active list" set find function and critical block removal.
  - Parallel accumulation of active list.
  - Parallel filling of active vector.

- Eikonal solver is three times faster than the old version.
- Energy consumed during the execution of Eikonal solver on ARM platforms is 4 times less than on Intel platforms.
- Performance is 1.5x lower on ThunderX compared to Intel E5.
- Ongoing domain decomposition approach.

6 CARP - UGRAZ

6.1 Description

The Cardiac Arrhythmia Research Package (CARP) [14], which is built on top of the MPI-based library PETSc [15], was used as a framework for solving the cardiac bidomain equations in parallel. PETSc [15] served as the basic infrastructure for handling parallel matrices and vectors. Hypre [16], advanced algebraic multigrid methods such as BoomerAMG, and ParMetis [17], graph-based domain decomposition of unstructured grids, were compiled with PETSc as external packages. An additional package, the publicly available Parallel Toolbox (pt) library (http://paralleltoolbox.sourceforge.net), [18] which can be compiled for both CPUs and GPUs, was interfaced with PETSc.

The parallelization strategy was based on recent extensions of the CARP framework [19]. Briefly, achieving good load balancing, where both computational load and communication costs are distributed as evenly as possible, is of critical importance. While this is achieved for structured grids with relative ease [20], since the nodal numbering relationship is the same everywhere, mirroring the spatial relationship, it is far more challenging in the more general case of unstructured grids, which are preferred for cardiac simulations [21]. For general applicability, unstructured grids are mandatory to accommodate complex geometries with smooth surfaces, which prevent spurious polarizations when applying extracellular fields. To obtain a well-balanced grid partitioning, ParMeTis computes a k-way element-based partition of the mesh's dual graph, to redistribute finite elements among partitions. Depending on whether PETSc or pt was employed, two different strategies were devised.

6.2 Analysis

As we have seen on Deliverable D6.1 [1] the program consists of three main components: a parabolic solver, an ionic current component, and an elliptic solver. The parabolic solver is responsible for determining the propagation of electrical activity, by determining the change in transmembrane voltage from the extracellular electric field and current state of the transmembrane voltage. The elliptic solver unit determines extracellular potential from transmembrane voltage at each time instant. The ionic model component is a set of ordinary differential equations and is computed from a separate library, which must be linked in at compile time.

Since the CARP application is memory bound, the best combination to run it on the ThunderX mini-cluster is to use 1 MPI per socket and the number of threads per MPI process up to 16 threads per MPI. As we see from the Tables 13 and 14, CARP scales very well on the

Table 13: CARP execution on ThunderX for the rabbit heart mesh

#Cores	#Total time (sec)	#Parabolic	#ODE	#Efficiency
1	681	642.7	33.7	1
2	327	307.7	16.6	1.04
4	144	134.0	8.42	1.18
8	62	56.5	4.32	1.37
16	33	30.2	2.19	1.28
32	19	16.9	1.13	1.12
64	11	10.6	0.59	0.96
128	8	7.6	0.33	0.66

Table 14: CARP execution including Elliptic solver on ThunderX for the rabbit heart mesh

#Cores	#Total time (sec)	#Elliptic	#Parabolic	#ODE
1	6382	5680	642.7	33.7
2	4656	4313	307.7	16.6
4	2675	2519	134.0	8.42
8	1390	1316	56.5	4.32
16	1080	1036	30.2	2.19
32	901	871	16.9	1.13
64	810	787	10.6	0.59
128	777	756	7.6	0.33

ThunderX mini-cluster. We can say that in ThunderX nodes the efficiency is around 96% for parabolic solver and ODEs solver using 64 cores, then it drops to 70% for parabolic solver and 90% for ODEs solver.

In figure 33 we see strong scaling execution of CARP code that runs on SuperMUC with up to 8192 cores and approximately 150 Mill. tetrahedral finite elements.

6.3 Energy Measurements

We measure the energy for overall CARP code running on Intel E5 processors and ThunderX with the same methods as described on subsection 5.3. Below we present a test case running it with 8 MPI and 2 OpenMP threads each on both platforms.

- Intel E5-2660:
  - CORES power:
    - Average power on idle status: 6.53 (W)
    - Average power consumed: 52.94 (W)
    - Average power consumed by the program excluding idle status: 46.41 (W)
  - SOCKET power:
    - Average power on idle status: 15.05 (W)
    - Average power consumed: 65.04 (W)
    - Average power consumed by the program excluding idle status: 49.99 (W)
  - DRAM power:
    - Average power on idle status: 12.29 (W)

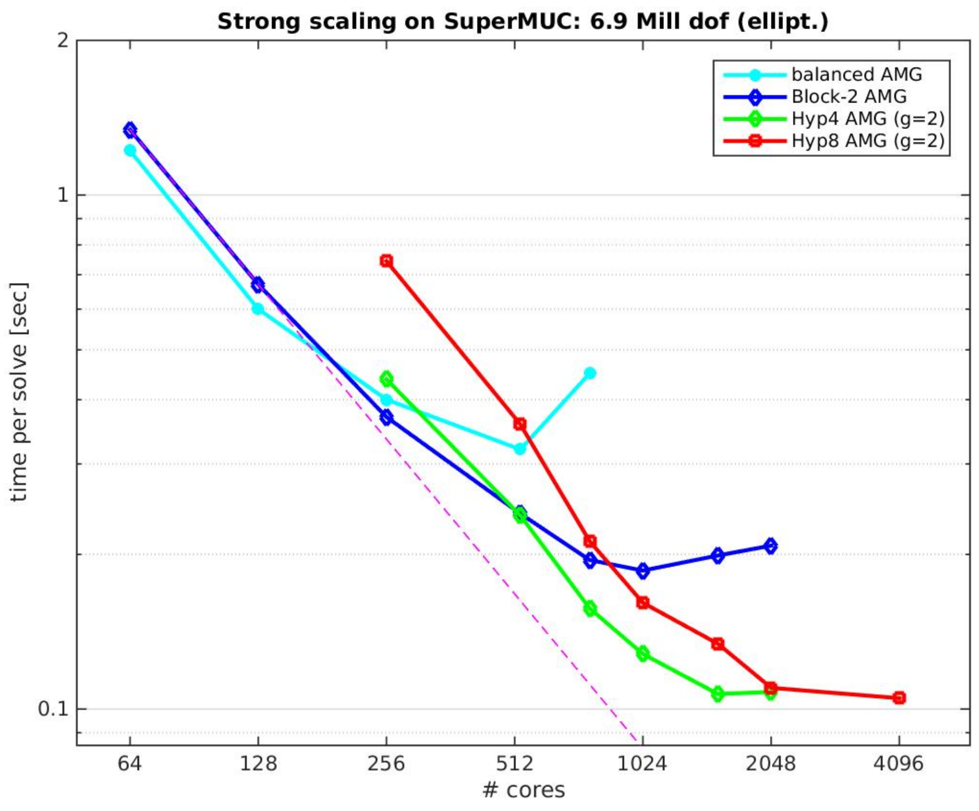


Figure 33: CARP scaling on SuperMUC.

    * Average power consumed: 33.64 (W)
    * Average power consumed by the program excluding idle status: 21.35 (W)

Total energy during the runtime of the program in this case is the power consumed by memory plus the power consumed by sockets multiplied by the total runtime of application.

  - Total energy during the runtime of CARP is:
    * Total power consumed by CARP: 71.34 (W)
    * Runtime: 1372 seconds
    * Total energy consumed by CARP: 97.89 (kJ)

- ThunderX mini-cluster:
  - Average power on idle status: 506.06 (W)
  - Average power consumed: 512.97 (W)
  - Average power consumed by the program excluding idle status: 6.91 (W)

Total energy during the runtime of the program in this case is the power consumed by memory plus the power consumed by sockets multiplied by the total runtime of application.

  - Total energy during the runtime of CARP is:
    * Total power consumed by CARP: 6.91 (W)

- ∗ Runtime: 6141 seconds
- ∗ Total energy consumed by CARP: 42.46 (kJ)

- Comparison between Intel Xeon E5-2660 and the ThunderX:
  - Runtime of the CARP code is 4.4x slower on ThunderX than Intel E5-2660
  - Energy during the runtime of the CARP code is 2.3x less energy on ThunderX than Intel E5-2660
- One core performance between Intel Xeon E5-2660 and ThunderX:
  - Runtime of the CARP code on ThunderX: 14838 seconds
  - Runtime of the CARP code on Intel E5-2660: 772 seconds
  - Performance during the runtime of the CARP code on ThunderX is 19 times slower than on the Intel E5-2660.

The overall CARP project was executed on the Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz and ThunderX @ 1.80GHz. ThunderX is 4.4x slower than Intel E5-2660. On the other hand it consumes 2.3x less energy than Intel E5-2660.

6.4 Conclusions

- Scales on SuperMUC up to 16k cores.
- Energy consumed during the execution of CARP on the ARM platforms is 2.3 times less than on the Intel platforms. However, the performance is 4.4x lower on the ARM platform versus the Intel one.
- Overall CARP, including the elliptic and parabolic solvers, are limited by memory bandwidth.
- Reduced memory transfer in MPI communication by reordering the unknowns.
- Ongoing work on:
  - Performance analysis for individual components of CARP.
  - Having a look on Hierarchical Hybrid Grids discretization methods.
  - Parallelization in time via XBraid.

7 Non-Newtonian Fluid solver - UGRAZ

7.1 Descriptions

PDEs are ubiquitous in many application fields. Traditionally, some of the most efficient and widely used methods to solve discretized PDEs are those from the class of multigrid methods [22, 23]. However, composing and tuning such a solver is highly non-trivial and usually dependent on a diverse group of parameters ranging from the actual problem description to the target hardware platform. To attain high performance, scalability and performance portability, new code generation techniques can be used in conjunction with DSLs, which provide the means of specifying salient features in an abstract fashion. ExaStencils[2] aims at realizing this vision for the domain of geometric multigrid solvers.

Of course, many (real world) applications are conceivable. One that is just as relevant as it is challenging is given by the application of simulating non-Newtonian fluids. Even without taking the extended use case of non-Newtonian behavior into account, solving the underlying coupled fluid and temperature equations is non-trivial. In this work, we present our recent accomplishments in generating highly optimized geometric multigrid solvers for this application. For this, we employ a finite volume discretization on a staggered grid with varying grid spacing which is solved using the SIMPLE algorithm [24, 25]. SIMPLE stands for Semi-Implicit Method for Pressure Linked Equations. It is a guess-and-correct procedure for the calculation of pressure on a staggered grid arrangement. The generated implementation is already OpenMP parallel and includes models for the incorporation of non-Newtonian properties.

7.2 Viscoplastic Non-Newtonian Fluids

Viscoplastic fluids are those non-Newtonian fluids characterized by a yield-stress, defined as a threshold after which a fluid readily flows [26, 27]. In general, yield-stress fluids are suspensions of particles or macromolecules, such as pastes, gels, foams, drilling fluids, food products and nanocomposites. Processes with viscoplastic fluids are of great importance in mining, chemical and food industry. For instance, several works have shown that rheological properties of fruit juices [28, 29], mining pulps [30, 31] and nanofluids [32, 33] are well described by viscoplastic non-Newtonian models such as Herschel-Bulkley, Bingham and Casson. Nanofluids are colloidal dispersions of nanometric-sized (<100 nm) metallic or non-metallic particles in a base fluid. The addition of nanoparticles improves the thermal conductivity and increases the viscosity of the fluid. Nanofluids containing spherical nanoparticles are more likely to exhibit Newtonian behavior and those containing nanotubes show non-Newtonian behavior. Furthermore, nanofluids show non-Newtonian behavior at higher shear rate values [34]. The applications of interest of our numerical experiments are related to the flow with Bingham plastic fluids due to changes in buoyancy forces caused by thermal effects and mixed convection effects. It has been found that under certain conditions water suspensions with nanoparticles such as SiO_2/TiO_2 [32], $BaTiO_3$ [35], ITO [36] behave as Bingham fluids. The presentation of the application problem follows closely the description in [37] and we will focus especially on the natural convection example therein.

The formal mathematical description of the coupled problem is similar to the Navier-Stokes equations for incompressible fluids together with an additional equation regarding the (scaled) temperature θ, an additional term in the Navier-Stokes equations and non-linear material coefficients regarding the non-Newtonian fluid behavior. Let us combine the three velocity components in one velocity column vector $\vec{v} = (U, V, W)^T$ and the gradient operator ∇ is also considered as column vector.

$$-\nabla^T \left(H(\dot{\Gamma}) \nabla \vec{v} \right) + D \cdot \left(\vec{v}^T \cdot \nabla \right) \vec{v} + \nabla p \quad + D \begin{bmatrix} 0 \\ \theta \\ 0 \end{bmatrix} = 0 \tag{1}$$

$$\nabla^T \vec{v} = 0 \tag{2}$$

$$-\nabla^T \left(\nabla \theta \right) + G \cdot \left(\vec{v}^T \cdot \nabla \right) \theta = 0 \tag{3}$$

Broadly speaking, Bingham fluids behave as a Newtonian fluid under the influence of a shear stress higher than the yield stress ($\tau > \tau_y$). When the yield-stress falls below τ_y (unyielded

[2]http://www.exastencils.org

region) a solid structure is formed. In the present work, the numerical implementation of the Bingham model is based on the bi-viscosity model [38], considering low values of τ_y according to experimental observations [32].

7.3 Numerical solution of the coupled problem

The coupled system of PDEs (1)-(3) is non-linear due to the material terms H, D, G that depend on velocity $\vec{v}$ and on temperature θ. Even in case of constant material terms, the equations (1) and (2) represent the Navier-Stokes equations (non-linear because of $\vec{v}^T \cdot \nabla$) with an additional term from the temperature. PDE (3) is only non-linear w.r.t. to the coupling via $\vec{v}^T \cdot \nabla$, not regarding the temperature θ.

The discretization uses a staggered grid in order to fulfill the LBB stability condition, see in Figure 34 and illustrations in [39].

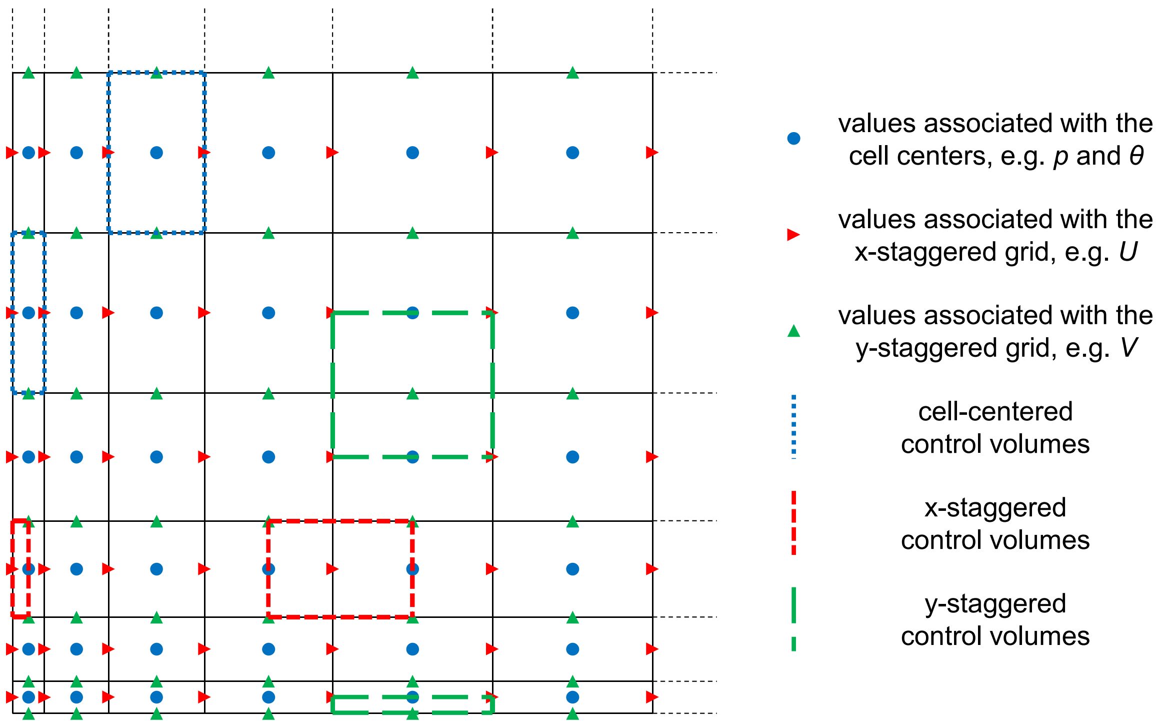

values associated with the cell centers, e.g. p and θ
values associated with the x-staggered grid, e.g. U
values associated with the y-staggered grid, e.g. V
cell-centered control volumes
x-staggered control volumes
y-staggered control volumes

Figure 34: 2D illustration of the lower left part of a non-equidistant, staggered grid. Velocity components are associated with the centers of edges (resp. faces in 3D). Staggered control volumes get halved at the boundary.

The non-linear terms $\mathcal{L}(x)$ are quasi-linear, i.e., they can be expressed as $L(x) \cdot x$. This allows to rewrite the system of PDEs (1)-(3) as block system with non-linear sub-blocks.

$$\begin{pmatrix} A(\theta, \vec{v}) & B & C(\theta) \\ B^T & 0 & 0 \\ 0 & 0 & T(\theta, \vec{v}) \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p \\ \theta \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{4}$$

Fixing temperature θ^{old} we get the stationary Navier-Stokes equations

$$\begin{pmatrix} A(\theta^{\text{old}}, \vec{v}) & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p \end{pmatrix} = \begin{pmatrix} -C \cdot \theta^{\text{old}} \\ 0 \end{pmatrix} \tag{5}$$

and the additional temperature equation

$$T(\theta, \vec{v}) \cdot \theta = 0 \ . \tag{6}$$

Finally we end up with the *SIMPLE algorithm* [24] for (7) containing the *pressure correction step* (9):

$$\begin{pmatrix} A & B \\ B^T & 0 \end{pmatrix} \cdot \begin{pmatrix} \vec{v} \\ p' \end{pmatrix} = \begin{pmatrix} g - B \cdot p^* \\ 0 \end{pmatrix} \tag{7}$$

$$\text{Solve} \qquad A\vec{v}^* = g - B \cdot p^* \tag{8}$$

$$\text{Solve} \qquad -\Delta t \nabla^T \nabla p' = -B^T \cdot \vec{v}^* \tag{9}$$

$$\text{Solve} \qquad \vec{v}' = -D^{-1} B \cdot p' \tag{10}$$

$$\text{Update} \qquad \begin{pmatrix} \vec{v} \\ p \end{pmatrix} = \begin{pmatrix} \vec{v}^* \\ p^* \end{pmatrix} + \begin{pmatrix} \vec{v}' \\ p' \end{pmatrix} \tag{11}$$

7.4 Scaling and Efficiency results

Currently, the code generation tool ExaStencils is used to generate automatically an Hybrid MPI+OpenMP parallelized multigrid solver for the coupled problem above. At first we have a hierarchical partitioning of the domain where the whole domain is divided into blocks and each of these blocks usually is assigned to an MPI process. These blocks gets subdivided into fragments and we could assign these fragments into OpenMP (or CUDA) threads. In between these fragments data have to be exchanged. Usually it is better to have one big fragment and multiple OpenMP threads that run across this single fragment as in the following test cases.

Table 15: Non-Newtonian Fluid solver on ThunderX

#MPIxOpenMP	#Execution time (sec.)	#Speed-Up	#Efficiency
1x1	745819	1	1
1x4	148380	5.0	1.25
2x4	92491	8.0	1.00
4x4	69728	10.6	0.66
8x4	54860	13.5	0.42
16x4	55269	13.5	0.21
32x4	57077	13.0	0.10

The Non-Newtonian Fluid solver code that is used for benchmarks on the Tables 15 and 16 is generated without vectorization. We faced an issue with the code generation tool on the part that generate vectorized code on the ARM platforms and we are working on solving it.

Table 16: Non-Newtonian Fluid solver on Xeon Phi

#MPIxOpenMP	#Execution time (sec.)	#Speed-Up	#Efficiency
1x1	305210	1	1
1x4	80232.1	3.8	0.95
2x4	50591.6	6.0	0.75
4x4	36857.2	8.2	0.51
8x4	34704.1	8.7	0.27
16x4	29586.1	10.3	0.16
32x4	23924.7	12.7	0.10

Tables 15 and 16 show the execution time, the speed-up and the efficiency of the Non-Newtonian Fluid solver running on the ThunderX mini-cluster and on Xeon Phi. For this test case the total problem size is 262,144 cells, and moving from 16 MPI to 32 MPI the 4 OpenMP threads work on only 4096 cells, this explain also the very poor efficiency when we perform strong scaling on both Xeon Phi and ThunderX platforms. From the Figure 35 we see that the Non-Newtonian Fluid solver scales very bad in both platforms, on ThunderX efficiency is a bit better. Regarding the runtime, ThunderX is 2 times slower than Xeon Phi.

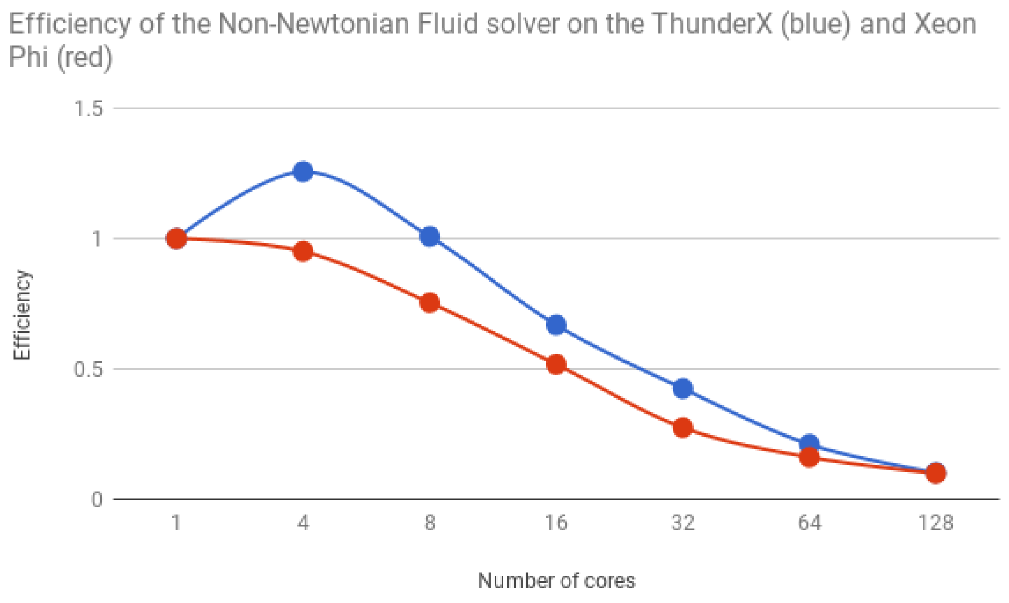


Figure 35: Efficiency of the Non-Newtonian Fluid solver on the ThunderX and Xeon Phi

7.5 Future work

The planned work for Non-Newtonian Fluid solver application will be as follow:

- Using larger problem sizes.
- OpenMP/MPI version will be traced on Mont-Blanc clusters.
- Analysis using BSC tools, extrae, paraver, dimemas and ARM Code Advisor.
- Adaption of code generation tool with respect to the analysis on ThunderX.
- Detailed Energy measurements.

8 Mesh Interpolation Mini-App

This mini-app was extracted from the AVL Fire codebase to allow faster optimization iterations and cooperation within Mont-Blanc 3 while maintaining a representative code and workload. This section largely follows [40].

8.1 Description

Mesh deformation is a performance critical part of many problems in fluid dynamics. Radial basis function (RBF) interpolation based methods for mesh deformation have addressed the increasing complexity for larger data sets.

Given a computational mesh, treated as point cloud in $\mathbb{R}^3$, a subset of N points $X = \{x_i\}_{i=1}^N$ and associated function values $f_i = f(x_i)$ for these points, an interpolating function s such that: $s|_X = f|_X$ of the form $s(x) = \sum_{i=1}^N \lambda_i\, \phi(\|x - x_i\|) + p(x)$ is sought, where p is a polynomial. In this implementation a multiquadric biharmonic RBF $\phi(r) = \sqrt{r^2 + c^2}$, with a scaling parameter $c \in \mathbb{R} \setminus \{0\}$, is used, which requires $p(x) = \alpha \in \mathbb{R}$.

Requiring the interpolation condition in all given points and demanding a side condition on the coefficients of the polynomial term leads to a symmetric system of linear equations for the determination of the coefficients $\vec{\lambda}$ and α:

$$\sum_{i=1}^{N} \lambda_i \phi(\|x_i - x_k\|) + \alpha = f(x_k), \qquad 1 \le k \le N,$$

$$\sum_{i=1}^{N} \lambda_i = 0. \tag{12}$$

8.2 Original Implementation and Performance Analysis

The original implementation of our multilevel distributed RBF code described in [41] is considered as baseline and described here. Since the introduction of the multilevel method was mainly concerned with the efficient distribution of work across distributed memory machines, the local solver algorithms for CPU and GPU are largely based on [42].

Algorithm 1: Solver Algorithm Overview

```
while not converged do
    for level in MultiLevel-DD do
        for box in DD[level] do        Outer Loop
            if num points in box < threshold then
                solveDirectly(box)
            else
                solveFMM(box)
            end
        end
        communicateResults()
        calcLevelResidual()
    end
    calcGlobalResidual()
end
```

In this project, we only discuss work on the iterative solver, which is the dominant part. The outer loop highlighted in algorithm 1 is the first parallel section of the solver. There is an implicit barrier after each iteration of this loop. Parallelism is exploited at two stages. First, the domain of points is decomposed in cubic boxes using an octree structure. The boxes are statically assigned to MPI ranks, thus there should be at least as many non-empty boxes as MPI ranks. The dominant operations within each box are dense matrix vector products, which are either performed using brute-force or are approximated by a fast multipole method (FMM). The inner products are the second stage of parallelism, both are implemented in OpenMP and CUDA. The second parallel section of the solver is the update of the current residual where a dense matrix vector product over all points of the current level has to be computed. Again, the data is distributed using an octree structure and assigned to MPI ranks. The calculations per rank exploit shared memory parallelism yet again.

8.3 Theoretical Performance Analysis

The main work of this algorithm is the solution of subproblems in the multilevel loop. Level L contains at 8^L non-trivial subproblems. Due to refinement, each subproblem across all levels is approximately an equal amount of work. By choosing the number of refinement levels, the subproblem size can be adjusted. A minimum size per subproblem is necessary for good convergence.

In the original code, this level loop distributes subproblems via MPI in a round robin fashion. Assuming equal work per subproblem and ignoring network effects, an upper limit for scaling given a number of refinement levels (corresponding to a minimum problem size) can be derived. The lines in Figure 36 are labelled with the refinement level they represent and the minimal number of points required for this level, i.e. the topmost line has four refinement levels in addition to the full point cloud and requires more than eight million points to converge well.

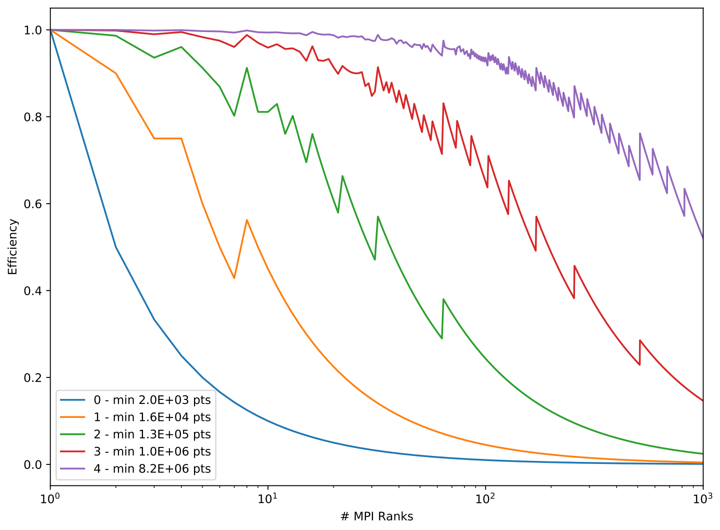


Figure 36: Theoretical ideal strong scaling efficiency for multi level iterative solver algorithm 1. Lines are labelled with the number of refinement levels and minimum number of points required at this level.

For typical problem sizes of this application, up to 1 million points, we find good (95%+) efficiency only up to ten ranks, at 100 ranks efficiency degrades to 60%. Consequently improving the parallelisation efficiency is a primary concern, when scaling beyond very small systems.

8.4 Optimization

8.4.1 Auto Tuning the Brute-Force vs. FMM Threshold

A specialised matrix vector product used to solve sub problems is the a core kernel. Two versions of this kernel are available, a brute-force algorithm of complexity $\mathcal{O}(n^2)$ and variant using the fast multipole method (FMM) of complexity $\mathcal{O}(n \log n)$.

The constant threshold was chosen during development on an Intel platform. Experiments showed the ideal threshold for a comparable Intel CPU has moved by a factor of 3, and there was a factor of 2 between a Cavium ThunderX and Intel Xeon, thus a auto tuning step for this threshold was implemented. This yields a speed-up of approximately 2, compared to using a single value across all systems.

8.4.2 Hybrid Parallelism on the Outer Loop

The strong scaling analysis in subsection 8.3 shows, it is impossible to scale practical problem sizes to very high numbers of MPI ranks. The baseline application already used OpenMP to parallelise the brute-force and FMM kernels, but did not scale well to a full socket or node (48/96 threads on ThunderX) because the kernels contain complex data access patterns including writing to shared resources, many synchronisation points and little (μs timescale) work per iteration.

Parallelising the loop of algorithm 1 yields two benefits over the previous results. First, on high refinement levels when the outer loop has many iterations, the application now scales equally well using MPI or OpenMP or a combination. In addition, the memory footprint is reduced, as most data only needs to be replicated once per process, not per thread. Second, when the outer loop has fewer iterations than threads, nested parallelism is used to keep all threads busy while keeping the work per task as large as possible.

8.5 Results

Three test systems were used to evaluate the optimizations. *Xeon* is a single dual socket node equipped with Intel(R) Xeon(R) E5-2690 v4 CPUs. An identical node with a NVIDIA Pascal P-100 GPU was used for the heterogeneous results. These are identified as the *Xeon + GPU* test system. Only processor (CPU, GPU, DRAM) power was measured on this system. Intel Running Average Power Limit (RAPL) counters, accessed via the `perf_event` were used to measure CPU power. The Nvidia System Management Interface (nvidia-smi) sampled with 10 Hz was used for the GPU power.

The *ThunderX* test system consists of four boards with two sockets per board. The ThunderX has 48 ARMv8-a cores at 1.8 GHz per socket. The interconnect is 10 Gigabit Ethernet. Power is measured by an external Yokogawa Power Meter (10 Hz sampling, 0.1% precision), which is read from the cluster head node via a serial interface. This setup includes the common power supply for all four nodes but excludes infrastructure, i.e. network switches, network storage, cooling. Since there is only one measurement for all four boards combined, the active power for jobs not using the full system has to be calculated. We calculate the power of the active nodes as the difference of total power and idle power per node times the number of idle nodes.

The results in this section refer only to the iterative solver phase. Input and output (IO), data generation and initialisation of supporting structures for the multilevel preconditioner are excluded. This reflects our use case, where the mesh deformation solver is called frequently during a CFD simulation, all IO is handled via memory without copying, and the support structures are only set up once.

The reported results are for a sphere of 262144 approximately equidistant points. For each evaluated core count, all valid domain decomposition refinement levels and distributions of physical cores to OpenMP threads and MPI ranks were measured. For the Xeon+GPU tests, all cores were assigned as MPI ranks.

Table 17: Results for ThunderX Test System

Cores	Energy	Time	Level	Rel. Energy Gain	Speedup
1	391.0	3103.2	3	1.0	1.0
2	198.9	1568.9	3	2.0	2.0
4	202.5	1599.7	3	1.9	1.9
8	52.3	406.5	3	7.5	7.6
16	29.3	223.8	3	13.4	13.9
32	18.2	134.0	3	21.5	23.2
48	14.9	108.2	3	26.3	28.7
96	10.5	72.0	3	37.3	43.1

For the ThunderX system, the results in Table 17 show similar and acceptable scaling behaviour for both energy and time to solution. Due to the high number of cores available, only refinement level three was tested.

As shown in Table 17 energy to solution was always improved by using as much of the node as possible, thereby reducing unused idle power. In general use, however, a node would typically be either fully used or the unused cores would be used by another job. A more practical question for production use is, whether energy and time to solution are always minimised at the same time, or if they are orthogonal goals. For our problem and given a fixed system both goals are very similar, we can not say if after further work the goals will diverge.

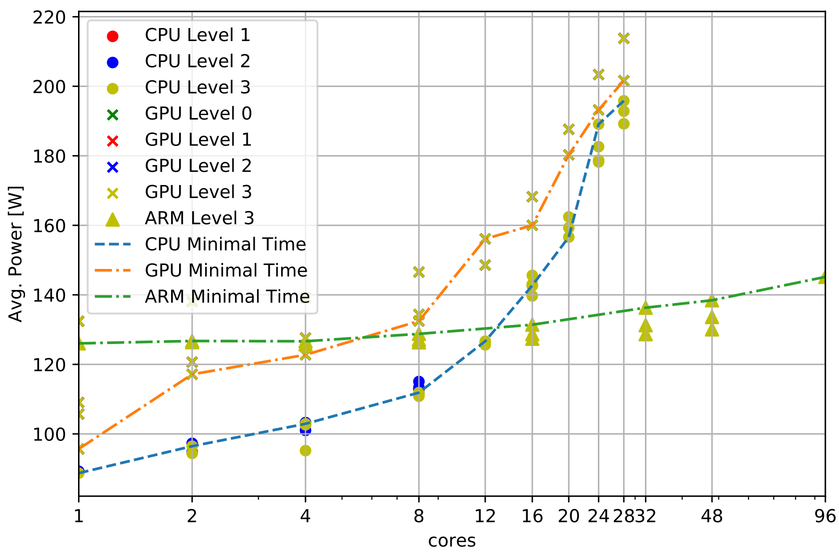


Figure 37: Average power per run for each test system for all configurations

A related concept is the idea of *race to sleep*, that it is often beneficial for energy to solution to solve a problem fast at the cost of increased power. In Figure 37 we show the average power draw of all systems and highlight the fastest solutions. Our results indicate agreement with this idea on the ThunderX system, where time to solution was very variable for different configurations. On the Xeon systems where the tested configurations were more similar in runtime, there are cases where lower power use can yield faster time to solution.

References

[1] Report on profiling and benchmarking of the initial set of applications on arm-based hpc systems. Deliverable D6.1 of the Mont-Blanc 3 project, 2016.

[2] Report on the experimentation of new technologies. Deliverable D7.5 of the Mont-Blanc 3 project, 2016.

[3] Initial report on automatic region of interest extraction and porting to openmp4.0-ompss. Deliverable D6.5 of the Mont-Blanc 3 project, 2017.

[4] M. Garcia, J. Corbalan, and J. Labarta. Lewi: A runtime balancing algorithm for nested parallelism. In *2009 International Conference on Parallel Processing*, pages 526–533, Sept 2009.

[5] Marta Garcia, Jesus Labarta, and Julita Corbalan. Hints to improve automatic load balancing with lewi for hybrid applications. *Journal of Parallel and Distributed Computing*, 74(9):2781 – 2794, 2014.

[6] Jack Dongarra, Michael A. Heroux, and Piotr Luszczek. Hpcg benchmark: a new metric for ranking high performance computing systems. Technical report, Electrical Engineering and Computer Science Department, Knoxville, Tennesse, 2015.

[7] Mont-Blanc consortium. Student cluster competition 2017. `https://goo.gl/4BBkGs`.

[8] Kiyoshi Kumahata and Kazuo Minami. Hpcg performance improvement on the k computer. `http://www.hpcg-benchmark.org/downloads/sc14/HPCG_on_the_K_computer.pdf`, 2014.

[9] 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.

[10] Arm Limited. Arm compiler for hpc. `https://goo.gl/yeUnt8`, 2016.

[11] Report on the experimentation of new technologies. Deliverable 7.5 of the Mont-Blanc 2 project, 2016.

[12] Intermediate report on deployment and evaluation of mini-clusters. Deliverable 4.5 of the Mont-Blanc 3 project, 2017.

[13] Daniel Ganellari and Gundolf Haase. Reducing the memory footprint of an Eikonal solver. In *2017 International Conference on High Performance Computing Simulation (HPCS)*. IEEE, 2017. accepted.

[14] E. Vigmond, M. Hughes, G. Plank, and L. Leon. Computational tools for modeling electrical activity in cardiac tissue. *J Electrocardiol*, 36:69–74, 2003.

[15] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne Nat. Lab., 2008.

[16] Hypre team. Hypre - high performance preconditioners User's Manual. Technical Report Software Version: 2.0.0, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, https://computation.llnl.gov/casc/hypre/download/hypre-2.0.0_usr_manual.pdf, Dec. 2006.

[17] Karypis G. and Kumar V. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System. http://www.cs.umn.edu/ metis, University of Minnesota, Minneapolis, MN, 2009.

[18] Gundolf Haase, Manfred Liebmann, Craig C. Douglas, and Gernot Plank. A parallel algebraic multigrid solver on graphics processing units. In Wu Zhang, Zhangxin Chen, Craig C. Douglas, and Weiqin Tong, editors, *HPCA (China), Revised Selected Papers*, volume 5938 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2009.

[19] S. Niederer, L. Mitchell, N. Smith, and G. Plank. Simulating human cardiac electrophysiology on clinical time-scales. *Front Physiol*, 2:14, 2011.

[20] M. Munteanu, L.F. Pavarino, and S. Scacchi. A scalable Newton-Krylov-Schwarz method for the bidomain reaction-diffusion system. *SIAM J. Sci. Comput.*, 2009.

[21] E. J. Vigmond, R. Weber dos Santos, A. J. Prassl, M. Deo, and G. Plank. Solvers for the cardiac bidomain equations. *Prog Biophys Mol Biol*, 96(1-3):3–18, 2008.

[22] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer-Verlag, 1985.

[23] U. Trottenberg, C. W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[24] S. V. Patankar and D. B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer*, 15(10):1787–1806, 1972.

[25] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Series in Computational Methods in Mechanics and Thermal Sciences. McGraw-Hill, New York, 1980.

[26] H. A. Barnes. The yield stress—a review—everything flows? *Journal of Non-Newtonian Fluid Mechanics*, 81:133–178, 1999.

[27] H. Zhu, Y.D. Kim, and D. De Kee. Non-newtonian fluids with a yield stress. *Journal of Non-Newtonian Fluid Mechanics*, 129:177–181, 2005.

[28] A. C. A. Gratão, V. Silveira Jr., and J. Telis-Romero. Laminar flow of soursop juice through concentric annuli: Friction factors and rheology. *Journal of Food Engineering*, 78:1343–1354, 2007.

[29] J. Telis-Romero, V. R. N. Telis, and F. Yamashita. Friction factors and rheological properties of orange juice. *Journal of Food Engineering*, 40:101–106, 1999.

[30] J. Yue and B. Klein. Influence of rheology on the performance of horizontal stirred mills. *Minerals Engineering*, 17:1169–1177, 2004.

[31] A. Merve Genc, I. Kilickaplan, and J. S. Laskowski. Effect of pulp rheology on flotation of nickel sulphide ore with fibrous gangue particles. *Canadian Metallurgical Quarterly*, 51:368–375, 2012.

[32] W. R. Richmond, R. L. Jones, and P. D. Fawell. The relationship between particle aggregation and rheology in mixed silicatitania suspensions. *Chemical Engineering Journal*, 71(1):67 – 75, 1998.

[33] A. Katiyar, A. N. Singh, P. Shukla, and T. Nandi. Rheological behavior of magnetic nanofluids containing spherical nanoparticles of feni. *Powder Technology*, 224:86 – 89, 2012.

[34] A. K. Sharma, A. K. Tiwari, and A. R. Dixit. Rheological behaviour of nanofluids: A review. *Renewable and Sustainable Energy Reviews*, 53:779 – 791, 2016.

[35] W. J. Tseng and S.-Y. Li. Rheology of colloidal batio3 suspension with ammonium polyacrylate as a dispersant. *Materials Science and Engineering: A*, 333(12):314 – 319, 2002.

[36] W. J. Tseng and F. Tzeng. Effect of ammonium polyacrylate on dispersion and rheology of aqueous {ITO} nanoparticle colloids. *Colloids and Surfaces A: Physicochemical and Engineering Aspects*, 276(13):34 – 39, 2006.

[37] J. Banaszek, Y. Jaluria, T. A. Kowalewski, and M. Rebow. Semi-implicit fem analysis of natural convection in freezing water. *Numerical Heat Transfer, Part A: Applications*, 36(5):449–472, 1999.

[38] E. J. O'Donovan and R. I. Tanner. Numerical study of the bingham squeeze film problem. *Journal of Non-Newtonian Fluid Mechanics*, 15(1):75 – 83, 1984.

[39] S. R. Djeddi, A. Masoudi, and P. Ghadimi. Numerical simulation of flow around diamond-shaped obstacles at low to moderate reynolds numbers. *American Journal of Applied Mathematics and Statistics*, 1(1):11–20, 2013.

[40] Schiffmann Patrick, Martin Dirk, Haase Gundolf, and Günter Offner. Optimizing a rbf interpolation solverfor energy on heterogeneous systems. Accepted for ParCo 2017, Bologna, Italy.

[41] G. Haase, D. Martin, P. Schiffmann, and G. Offner. A domain decomposition multilevel preconditioner for interpolation with radial basis functions. In I. Lirkov, S. Margenov, and J. Wasniewski, editors, *Large Scale Scientific Computing LSSC'17*, volume xx of *Lecture Notes in Computer Science*, page 8. Springer, 2017. accepted, peer-review.

[42] Gundolf Haase, Dirk Martin, and Günter Offner. *Towards RBF Interpolation on Heterogeneous HPC Systems*, pages 182–190. Springer International Publishing, Cham, 2015.