D5.3 – Second-year Report on Proof-of-Concept Activities
Version 1.0

Document Information

Contract Number	676553
Project Website	www.pop-coe.eu
Contractual Deadline	Month 24, September 2017
Dissemination Level	Public
Nature	Report
Authors	José Gracia (HLRS)
Contributors	Nick Dingle (NAG), Anastasia Shamakina (HLRS), Stephan Walter (HLRS), Brian Wylie (JSC), Bo Wang (RWTH), Huan Zhou (HLRS), Michael Wagner (BSC), Jesus Labarta (BSC), Mathias Nachtmann (HLRS)
Reviewers	Nick Dingle (NAG)
Keywords	Proof-of-Concept; MPI; OpenMP; CUDA

Notices: *The research leading to these results has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No n° 676553.*

Change Log

Version	Author	Description of Change
v0.1	José Gracia	Initial structure of document
v0.2	José Gracia	Added initial draft on sections for DFTB, BPMF OMP, COOLFluiD, EPW, ArgoDSM, BPMF MPI
v0.3	José Gracia	Added initial draft on QE
v0.4	José Gracia	Ongoing activities; Update of QE
v0.5	José Gracia	Draft conclusions; overview and summary
v0.6	José Gracia	Revisions according to internal review
v1.0	José Gracia	Released to the EC

Contents

Executive Summary

In this document we report the current state of Proof-of-Concept activities. This type of activity is the central service of Work Package 5. The aim of the Proof-of-Concept activity is to assist users in implementing complex code refactoring or applying advanced parallelisation techniques. Proof-of-Concept activities may be initiated as a result of performance assessments in Work Package 4, but unlike them, are expected to take significantly longer to conclude.

In the course of the second year, the POP has concluded 7 Proof-of-Concept activities. As reported in the rest of the document, these have resulted in substantial performance increase of more than 10% in 5 cases; thereof in 3 cases even very significant performance increase of more than 50%. Only 2 activities have resulted in marginal or no performance increase. The proposed changes to the code have already been incorporated in some cases into the production code by the customer; in a few cases, this is still being evaluated.

In addition, another 4 Proof-of-concept activities are currently in progress. Their current state is briefly summarised in this document.

1 High-level Overview of Proof-of-Concept Activities

The project POP offers three distinct service activities. Two of them, Audit and Performance Plan, are performance analysis services hosted in work package 4. The Audit is an initial assessment of an applications performance characteristics and identification of possible bottlenecks, while the Performance Plan does more detailed root cause analysis of bottlenecks. One of the outputs of theses assessments, in particular from the Performance Plan, is concrete recommendations for changes of the applications code, algorithm or data structures.

In some cases, the recommendations can be implemented with simple code refactoring by a person with average parallel programming skills. In general, however, implementing the recommendation will require complex code refactoring and expert knowledge of parallel programming. Such specialised skills are not available in most groups that do parallel programming. Therefore, POP offers a third level of service called Proof-of-Concept (PoC). During this activity, POP staff will exemplify some of the recommended code refactoring, either directly on the customers application or on a simplified skeleton thereof. The aim is to illustrate the necessary changes and to train the customer in the usage of parallelisation and optimisation techniques.

At project start, we anticipated to do PoC activities mainly on kernels which had been extracted from application, or on mini-apps and mock-ups, rather than working on the customer's actual code. This practise would require less effort of POP staff. However, from the start customers preferred PoC activities to be conducted directly on their code: first extracting kernels/mock-ups and then reincorporating the optimisations into their code-base was rightly seen as large additional effort. Due to the low number of PoC requests, the POP project could afford to invest more effort per PoC activity (see D1.3); thus during the course of the second year we changed procedure to add benefit to the customers of the service and now work on actual customer code whenever possible.

PoC activities are expected to yield best-practise guidelines and input for training material which will be taken up in Work Packages 3 *Community Development*, and 6 *Training and Documentation*, respectively. We also expect some feedback on and suggestions for improvement of practise of performance assessment in Work Package 4 *Analysis*.

The remainder of this document reports on the PoC activities conducted during the second year of the project. Activities, which have been concluded already are reported individually in Secs 3-9. Ongoing activities are summarised in Sec 2. We make some final observations in section 10.

2 Summary of Ongoing PoC Activities

In the second year, the POP Project has concluded seven PoC activities, which are describe in subsequent sections of this document. In addition there are four further PoC activities under way, specifically on the codes *BAND*, *Kratos*, *MOLCAS*, and *Tangaroa*. Their current state is briefly reported in the following.

BAND The first phase of the BAND PoC work analysed performance of two of Intels MKL BLAS/PBLAS routines, to assess optimal use within BAND's shared memory and distributed memory matrix-matrix multiplications. Currently these use serial dgemm and MPI pzgemm respectively.

The dgemm work investigated the partitioning of the array data when calling dgemm, and also looked for improvements from using multithreaded dgemm or zgemm in place of the current dgemm call. There were no significant improvements identified from using zgemm or multithreaded dgemm, however the efficiency of serial dgemm is significantly improved for data partitionings which give output arrays which are square rather than tall and thin.

The pzgemm work studied the relationship between the BLACS process grid and load balance, and also compared MPI pzgemm with hybrid (MPI+OpenMP) pzgemm. It was possible to improve load balance efficiency, but at the expense of reducing communication efficiency, with no overall improvement. In addition performance was worse when using hybrid pzgemm.

Phase 2 of the PoC will look at assessing the improvements identified in phase 1 and the earlier Performance Plan in the actual BAND computation.

Kratos The Kratos Performance Assessment identified several issues in the OpenMP version of the Kratos code: load imbalances caused by both NUMA architecture; different locality behaviour for the same code in different cores; and issues with atomic reductions with indirections on large arrays. To address then we suggested different refactorings: parallelising data initialisation to leverage the first touch mechanism; look at renumbering mechanisms to uniformize (and improve) locality across threads; and approaches to eliminate atomics by properly scheduling (using commutative and multidependece clauses in OmpSs) a taskified version of the code.

In the PoC we are advising the customer through the implementation of some of these proposals. The work till now has focused on the data initialisation to address the NUMA architecture, using dynamic scheduling (Guided) as a first approach to address locality caused imbalances and partial approaches to address the atomics issue. Addressing the NUMA issues required some changes in the code caused by limitations imposed by the C++ std::vector data types the code was used. The issue is that allocation of this data type happens to sequentially touch the data thus inhibiting the possibility of leveraging the first touch mechanisms.

We also worked a bit on renumbering schemes although the observed gains from this is only about 10%. We need to compare in detailed the impact on IPC of the new numbering compared to the previous one.

The current global result is that an aggregated acceleration above 2x has been obtained till now. We have to do a detailed analysis of the current behaviour and implement more elaborated mechanism to eliminate the atomics overhead.

MOLCAS The MOLCAS Performance Audit identified that the code's CASPT2 module was spending a great deal of time in MPI collective operations such as MPI_Barrier, MPI_Win_create and MPI_Win_free. This PoC will investigate the feasibility in reducing the number of these operations. We are currently in the process of agreeing a PoC plan with the customer."

Tangaroa The POP Performance Audit on the code Tangaroa already identified issues with the MPI communication pattern. The customer requested a PoC to get rid of unnecessary serialisation on MPI operations with the target to achieve scaling significantly beyond the current limit at approximately 384 MPI ranks on Cavium ThunderX systems. Further analysis revealed that in this multi-threaded application, a large number of unmatched communication requests accumulate on the receiving side because of unnecessary, false dependencies between threads in the current code. The approach is to do communication asynchronously and without prescribing any particular order between threads. In addition, MPI ranks on the same node shall exchange data via shared-memory rather than through MPI. Development on a Cray machine is underway with promising first results. These however, need to be confirmed on the target system Cavium ThunderX.

3 Report on Activity DFTB

Keywords: MPI; asynchronous MPI communication; parallel matrix–matrix multiplication

3.1 Description of the Application

Software for Chemistry & Materials (SCM) is an Amsterdam-based computational chemistry software company that supports and develops the ADF Modelling Suite. DFTB is a standalone part of the modelling suite that implements a fast-approximate Density Functional Theory approach for molecules and periodic systems.

3.2 Previous Assessments and Recommendations

In the POP Performance Audit of DFTB we observed that the Parallel Efficiency declined gradually as the number of processes increased, probably because of the use of MPI collective operations in the matrix–matrix multiplications that comprised the region of interest. Trace analysis identified that the periods of computation were longer than the periods of communication which followed them. This suggested that it might be possible to hide the communication by overlapping it with computation. We recommended that a PoC should be conducted to investigate distributed sparse matrix–matrix multiplication approaches that use non-blocking MPI collectives to overlap communication and computation.

3.3 PoC Activities

The PoC investigated the effectiveness of rewriting DFTB's matrix–matrix multiplication routine `AddMatrixProductNN` to use non-blocking MPI routines. The aim was to improve the scalability of the code on distributed-memory machines. The original implementation was a version of the SUMMA algorithm [1] that had been adapted to block-sparse matrices.

We made three main changes to `AddMatrixProductNN`:

1. Separating the broadcast of block size information from the broadcast of matrix elements. The original version of DFTB used two rounds of `MPI_Bcast`s on each iteration to transmit first the dimensions of the matrix blocks and then the actual matrix elements. This made it hard to overlap communication and computation because the first broadcast had to complete before the broadcasts that actually distribute the data could start. We therefore restructured the code to separate these two phases, with the exchange of matrix block sizes being moved into a new subroutine that was called once at the beginning of `AddMatrixProductNN`.

2. Implementing a double-buffering scheme to avoid overwriting matrix elements that are being used in computation with those that are being communicated. This was achieved by using Fortran pointers to point to two `allocatable` arrays, and swapping these pointers over after computation finished rather than copying the contents.

3. Combining multiple smaller broadcasts of matrix elements into a single larger broadcast. In the original code each process issued multiple `MPI_Bcast`s of matrix elements to each recipient, which had the potential to be dominated by messaging overheads when the amount of data being transmitted was small. We therefore modified the code to transmit all the data in a single larger `MPI_Ibcast`.

3.4 Results

		2 700 atoms		5 000 atoms		10 000 atoms	
Procs.	Nodes	Original	Modified	Original	Modified	Original	Modified
16	1	54.9	56.2	193.6	194.4	–	–
32	2	31.0	31.2	109.2	107.6	–	–
48	3	20.4	21.5	70.4	71.4	473.0	467.6
64	4	16.0	18.0	54.8	56.9	358.4	361.8
128	8	9.6	12.6	31.6	34.9	197.8	204.4

Table 1: Runtimes in seconds for the input test cases on MareNostrum.

Tables 1 shows the runtimes of three input cases on MareNostrum. In general, we did not observe any meaningful performance improvement for the modified code.

The Intel MPI non-blocking communications benchmark (IMB-NBC, [2]) measures the amount of overlap achieved by MPI-3 non-blocking collectives. Running it on MareNostrum revealed that Intel MPI version installed there (5.1.3.21) achieved almost no overlap and its `MPI_Ibcast` was slower than its `MPI_Bcast`. This was consistent with the performance of our modified code.

3.5 Conclusion

We rewrote DFTB's matrix–matrix multiplication kernel to expose the possibility of overlapping communication and computation, but the resulting improvement in runtime was not as large as hoped. We investigated this issue using the Intel MPI Benchmark, which revealed that the limitation came from the performance of the MPI library's implementation of `MPI_Ibcast`.

In the future it would be extremely interesting to run our new code on a distributed-memory machine with the latest version of the Intel MPI library, as our experience on other machines suggests this offers the greatest scope for performance improvement. Even though there we did not explicitly test this in the PoC, the results were encouraging enough that SCM decided to continue development of the improved code in the future. Further investigations will take place to look for performance scalability improvements when there is time.

Lessons Learned

The limited performance of MPI_Ibcast is probably due to insufficient progression of non-blocking operations in this particular MPI implementation. MPI implementors should improve the quality of progress-engines.

4 Report on Activity BPMF OpenMP

Keywords: OpenMP; OpenMP nesting; OpenMP tasking; OpenMP reductions; outer product of vectors; single-node performance

4.1 Description of the Application

Modelling large and complex data sets is a major problem today. Examples therefore are the modelling of movie ratings using the Netflix dataset with more than 100 million entries or the prediction of compound-on-target-activity in chemogenomics from the ChEMBL data set with more than 2 million compound records. The Bayesian Probabilistic Matrix Factorization (BPMF) is an efficient method to solve these complex modeling problems.

4.2 Previous Assessments and Recommendations

The BPMF application has already been analysed in detail in the POP Performance Audit POP_PP_19. The analysis revealed a huge computational imbalance and the origin of this was found in the data distribution of the sparse input matrix that includes rows with different numbers of nonzero elements. In this particular input, one row 1804 (iteration 1803) has 110118 nonzero elements compared to the others, which have in average only 163 nonzero elements. In the analysis run iteration 1803 was executed by OpenMP thread number 17 from 274.004 478 s to 275.038 542 s. A timeline view of this part is shown in Figure 1.

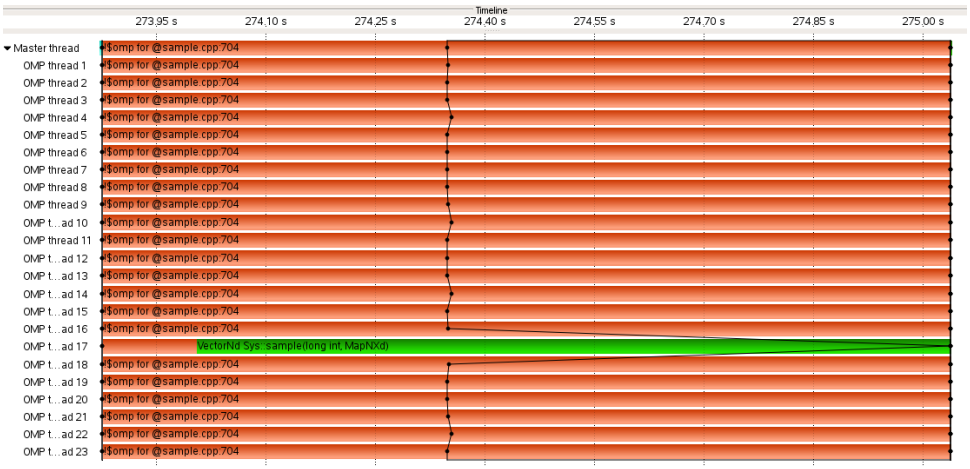


Figure 1: Vampir timeline view of iteration 1803, which runs from 274.004 478 s to 275.038 542 s.

The following recommendations were made in the Performance Plan:

1. **Parallelised reduction**. The serial loop in the `VectorNd sample()` method performs a reduction operation on the matrix `MM` and vector `rr`. This reduction could be parallelised using OpenMP nested parallelism.

2. **Optimisation of the outer product of matrices.** The serial outer product computation inside the `VectorNd sample()` method, which takes close to 50 % of the time in iteration 1803, could be replaced by a parallel version. At the same time other parts inside the `sample` method may be parallelised as well. Especially the `noalias` template should be considered, as it makes up for 20 % of the time here. The last 30 % are other calls

in the sample method which will require an in depth analysis beyond the scope of this Performance Plan.

4.3 PoC Activities

Optimisation of the linear algebra computations. The linear algebra operations inside BPMF make use of the very efficient and well vectorised Eigen Library. But we were able to identify some improvements due to mathematical properties of the matrices saving a good margin of CPU operations in some parts of the code.

Improvement of the selection of the optimised algorithms. BPMF comes already with different flavours of algorithms that are optimised for data element sizes. A crucial point now is the selection process when to use which algorithm. A closer look at the switching points lead here to another improvement.

Load balance issues in the OpenMP parallelisation. The last and most challenging issues nevertheless was load balance. Due to the nature of the problems solved with BPMF, the datasets include very inhomogeneous data, which result in load balance problems in the parallelisation. BPMF therefore comes with a hybrid MPI+OpenMP parallelisation. The still existing load balance problem found was at the lower OpenMP level in the region A. Here a single level OpenMP parallelisation was used on the node level. Now we implemented a second nesting level and made also use of OpenMP tasks, which solved the load balance problem: originally the load balance of the problematic code part was 42.5 % after the modifications 98.9 %.

The traces of region A before and after optimisation are shown in Figure 2. The BPMF application in region A for 24 OpenMP threads spends 57.5 % on the OpenMP implicit barrier (blue color) in original source code and 6.8 % in modified source code.

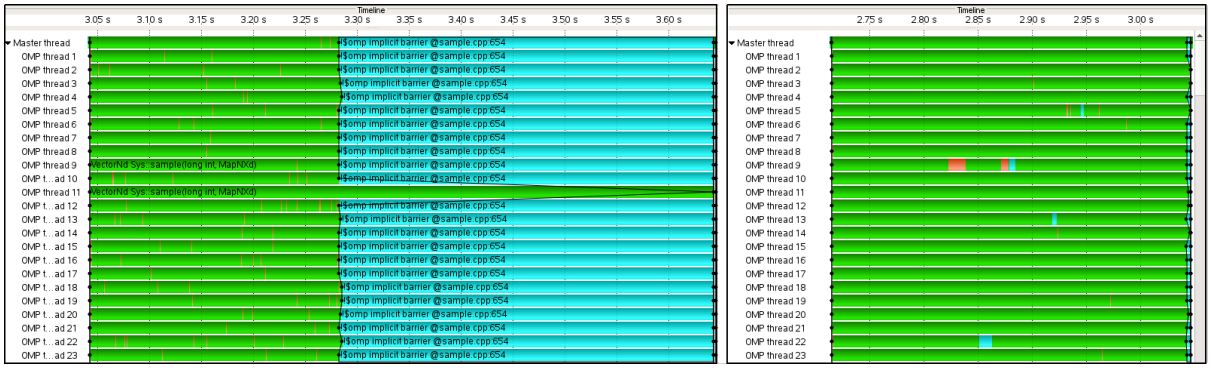


Figure 2: Traces of region A before and after optimisation.

4.4 Conclusions

This PoC report focused on the improvement of load imbalance. The analysis revealed the origin of the computational imbalance to come from the data distribution in the sparse input matrix, which includes rows with different numbers of nonzero elements. In this particular input, one row has a very high number of nonzero compared to the others. The improvements made in this POP Proof-of-Concept were evaluated with three different datasets achieving speedups between 1.6 and 1.8 that correspond to runtime reductions between 38 % and 44 %.

5 Report on Activity COOLFluiD

Keywords: OpenMP; CUDA; CUDA data transfer; single-node performance; kernel parameter optimisation

5.1 Description of the Application

COOLFluiD (Computational Object-Oriented Libraries for Fluid Dynamics) is a component based framework for scientific high-performance computing, CFD and multi-physics applications, originally developed at the Von Karman Institute for Fluid Dynamics. COOLFluiD provides a powerful infrastructure for enabling concurrent multi-physics simulations, possibly exploiting multiple MPI communicators, heterogeneous CPU/GPU computing, massively parallel I/O capabilities and loosely coupled multi-domain simulations. This wide range of supported domains lead to a large and complex C++ code base.

5.2 Previous Assessments and Recommendations

The parallel radiation code base within the COOLFluiD framework, that utilise the closed source PARADE solver, has already been analysed in detail in the POP Performance Audit POP_AR_48. The analysis revealed some problems related to the COOLFluiD framework, that enable the usage of the PARADE solver. The PARADE solver itself has shown satisfying results. Since the customer has had also an open source solver replacement for the PARADE framework, that utilise CPUs as well as GPUs, he was interested in an analysis and optimisation for the cases, where PARADE is not applicable.

The customer specified the following objectives for the PoC

1. Determine and solve a problem with the data transfer of the final results from the GPU back to the CPU within the CUDA solver
2. Implementation of a beneficial OpenMP parallelisation for the serial CPU solver
3. Provide a performance comparison between the GPU and CPU implementations of the custom radiation solver

5.3 PoC Activities

Optimisation of the CUDA Code. Our first task was the analysis of the extremely low achieved bandwidth for the data transfer of the final results of the CUDA implementation scheme `loopOverBins` from the GPU back to the CPU. That made at the end the whole solver non-beneficial compared to the serial CPU solver. During our analysis we were able to determine a problem within the time measurement scheme of the CUDA code as the root cause for the described problem. This problem lead at the end to wrong results for the required execution time of the kernel itself and the corresponding data transfers between the GPU and CPU. After we fixed the time measurement, we were able to determine, that the fraction of time required for the data transfer within the solver is negligible. So in contrast to the customers view, no optimisation of the data transfer was necessary. In contrast, we further analysed the kernel and were able to determine, that the applied parallelisation scheme of the CUDA code lead to a low initial parallelism in addition with a high serialisation potential during the execution. We were able to solve both problems through optimisations of the kernel launch parameters in line with minor modifications of the used CUDA kernel. The customer was able to adopt

easily the presented optimisations to the in total two schemes of the CUDA enabled solver - `loopOverBins` and `loopOverDirs`. Table 5.3 shows the finally achieved execution times and speedups for both schemes of the optimised CUDA solver.

Setting	LoopOverX	Data copy	Sum	Speedup
original Bins	0.01498	467.902	467.922	-
original Bins (fixed timing)	467.893	0.035008	467.933	-
optimised Bins	121.992	0.00030416	121.998	2.2
original Dirs	213.828	-	213.832	-
optimised Dirs	27.3425	0.000263648	27.3641	7.8

Table 2: Execution times and achieved speedups for the two schemes loopOverBins and loopOverDirs of the optimised CUDA solver.

OpenMP parallelisation of the serial CPU solver. Our second task was the development of a beneficial OpenMP parallelised version of the serial CPU solver, that utilises the `loopOverDirs` scheme. It was, because of the very complex code base, at the end necessary to develop the OpenMP parallelised version during an effective face to face meeting with the customer. The resulting OpenMP version of the `loopOverDirs` scheme showed even without further optimisations of the OpenMP environment a good scaling efficiency of 88%, respectively a speedup factor of 10.6, for a full node with two six core Intel(R) Xeon(R) E5-2640 CPUs.

Figure 3 shows the achieved execution time for different OpenMP thread pinning approaches with respect to the used thread counts compared to the execution times of the two serial CPU schemes. While the `OMP_w_proc_bind_close` scheme shows the highest performance up to twelve OpenMP threads, the `OMP_w_cpu_affinity_dense_socket_first` approach shows the highest overall performance with 24 threads as well as the highest scaling efficiency of all approaches. The difference between this two approaches is, that the first one first utilises the physical cores, and therefore uses the second socket for more than six threads, while the second one utilises always the logical core of each physical core, before it uses another physical core, so that it utilises with the same amount of twelve threads only a single socket with six physical cores. We will discuss in the further text only the `OMP_w_cpu_affinity_dense_socket_first` scheme, that we will call from up now the `OpenMP optimised` scheme.

Figure 4 a) shows the achieved speedups for the default and optimised thread placement scheme with respect to the serial execution time of the `loopOverDirs` scheme. It can be seen, that both OpenMP versions show below 24 threads a bad speedup factor. Figure 4 b) visualises, especial for the optimised thread placement, that a low core utilisation per thread is the root cause of the low speedup. All efficiencies above 100% indicate, that the execution of a second thread per physical core decreases the overall execution time by more than the number of used physical cores with respect to the serial performance.

Comparison of the performance of GPU and CPU implementations. Figure 5 shows, that the OpenMP parallelised CPU code can outperform both GPU schemes with just two used physical cores. It even shows, that the serial implementation of the `loopOverBins` scheme can outperform both GPU implementations. We want to mention at this point, that we suppose, that the GPU implementation can't gain a significant benefit from newer GPUs

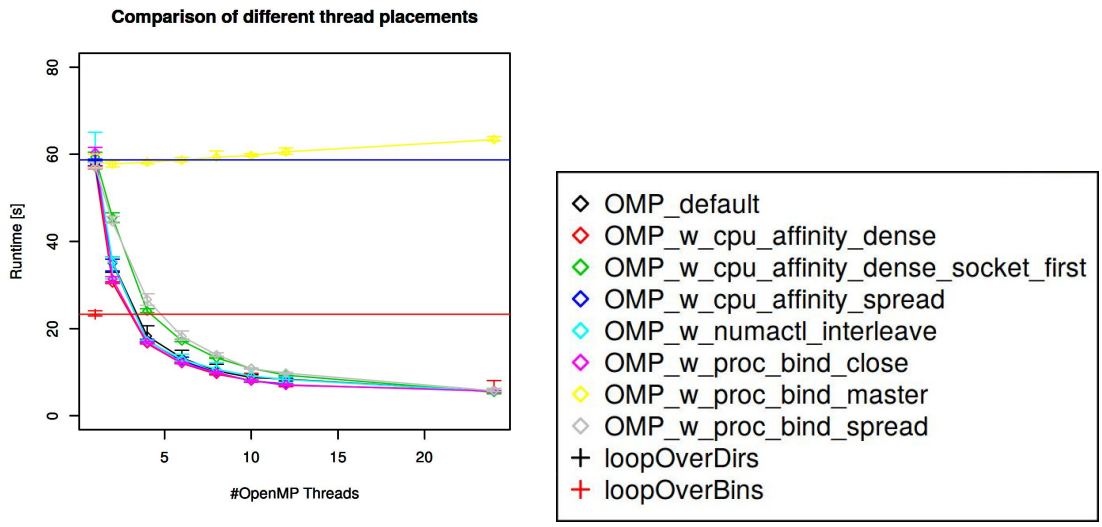
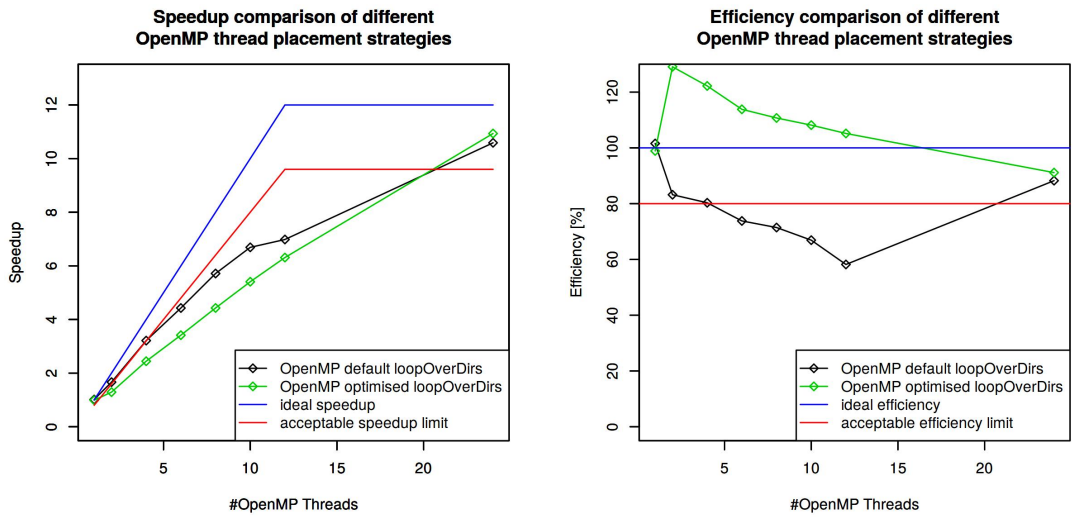


Figure 3: The horizontal blue and red lines highlight the execution time of the two serial CPU schemes, while the lines with the diamonds show the required execution time of the OpenMP parallelised loopOverDirs scheme for different thread placement strategies. We have to note that the number of used OpenMP threads doesn't correspond with the number of used physical cores.



(a) (b)

Figure 4: The green and blue lines correspond to 100% and 80% speedup (a) respectively efficiency (b) with respect to the serial loopOverDirs execution. An efficiency above 100% means, that the performance per physical core is higher than in the serial case.

because of the limited concurrency of just O(100). We suppose therefore, that the OpenMP code can gain a significant benefit from newer CPUs up to dozens of cores. So that the advantage of the OpenMP implementation would increase even further.

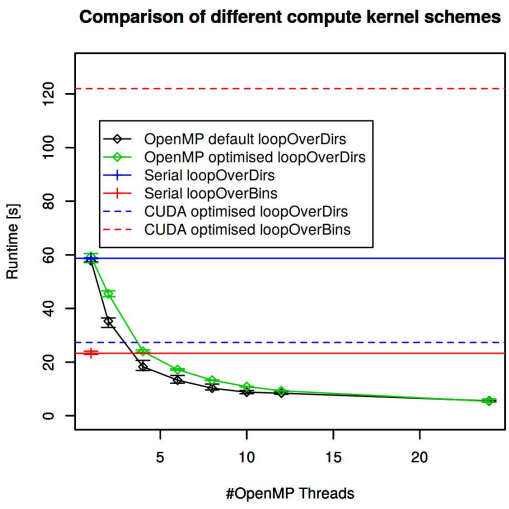


Figure 5: Performance of the radiation kernel for: CPU serial - cross with horizontal vertical line; CPU OpenMP - lines with diamonds; GPU CUDA - dashed horizontal lines

5.4 Conclusions

This PoC report focused on the improvement of the execution time for the CPU and GPU versions of the `Finite Volume Radiation Transfer` solver of the `COOLFluiD` framework. Our analysis revealed an error of the time measurement within the GPU version as well as an under-utilisation of the available compute resources. We were also able to parallelise the serial CPU version with OpenMP in a way, that we achieved for a full node with twelve physical cores a speedup of nearly 11. In addition to this, we were also able to show a scaling efficiency of more than 100% up to a single socket. This is an outstanding result and very useful for a potential MPI+OpenMP solution. In closing, we were able to show, that our developed OpenMP version outperforms even on a moderate CPU the best optimised CUDA implementation easily. Our implementation also promises a much higher performance benefit on newer CPU hardware.

Lessons Learned

This PoC showed nicely, that well vectorised CPU code can outperform CUDA implementations, particularly in those situations, which are characterised by non-trivial memory access patterns (as those encountered in typical stencil codes). In these cases, GPUs cannot efficiently exploit their nominally high memory bandwith to stream computations through the large number of cores available.

6 Report on Activity EPW

Keywords: MPI; parallel I/O

6.1 Description of the Application

EPW (Electron-Phonon simulation using Wannier interpolation) is a materials science DFT code distributed in the popular open-source Quantum ESPRESSO suite. It is developed by the University of Oxford and written using Fortran parallelised with MPI.

6.2 Previous Assessments and Recommendations

A variety of load balance issues were identified in an initial performance audit, as well as excessive time in the simulation phase, of EPW executions on several 24-core compute nodes of the ARCHER Cray XC30. The simulation phase became the focus of a subsequent Performance Plan, where specialised routines were provided which avoided unnecessary calculation and optimised vector summations, running 60% faster.

Although file I/O had been identified as a source of run-to-run execution variability, at this smaller scale it was negligible. Unfortunately, larger scale executions were dominated by the final simulation timestep where the simulation results were output to disk, and it was recommended to pursue this (and potential remedies) in a proof-of-concept investigation.

6.3 PoC Activities

The last simulation timestep differs from the others in that it also writes the final state to a single 50 MB formatted text file on disk (along with 100 MB of streaming output to stdout). This writing was done concurrently by all MPI ranks, resulting in redundant writing and massive contention for the file on disk.

A straightforward remedy involved making writing (and associated file open and close) conditional, such that only a single MPI process (master rank 0) output the entire data. Due to the relatively small amount of formatted data to be written, and the already effective performance of this simple technique, alternate approaches for optimised parallel writing (via MPI File I/O or external pHDF5, pNetCDF or SIONlib libraries) were not considered necessary.

6.4 Conclusions

Remedying the critical bottleneck when writing the final simulation state to disk reduced this from over seven hours to under one minute (over 450-fold improvement) with 480 MPI processes. Along with the prior optimisation of the simulation computation, this provided a 10-fold scalability improvement with 85% parallel efficiency to 960 processes, enabling previously impractical large scale simulations.

Lessons Learned

While the result is very satisfying, certain tools challenges needed to be surmounted.

Instrumenting EPW and its requisite libraries with Score-P (version 3.0) was rather complex, as the executable is only one of over 30 built as part of Quantum ESPRESSO and the Score-P instrumenter was unable to accurately determine which of these executables should be linked with its MPI measurement libraries (or not) as required. Since some executables use MPI and

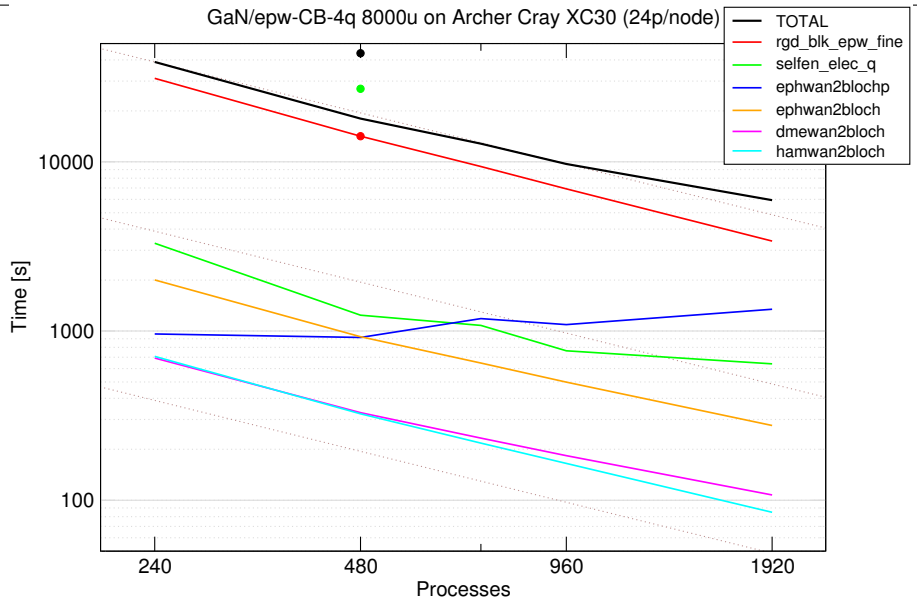


Figure 6: EPW testcase scalability on Archer Cray XC30 with breakdown of most significant simulation routines. (Oblique dotted lines represent perfect scaling from 240 processes.) For comparison, previous times with 480 processes concurrently writing output in selfen_elec_q (green) as large dots.

others do not, yet in the Makefiles all are linked using the same linker (i.e., ftn), it was also not possible to manually specify distinct Score-P instrumenter configurations for each set.

Identifying that file writing was the likely origin of the bottleneck required intuition and corresponding examination of the source code, to later be confirmed via manual source instrumentation. While other tools are able to automatically distinguish I/O costs, Score-P lacks the ability to distinguish file operations (other than those of MPI File I/O). While introducing manual instrumentation is relatively straightforward, even for Fortran sources which subsequently need to use the C preprocessor to expand Score-P API macros, it is rather cumbersome.

Score-P also doesn't yet provide description of application execution configurations required for topological presentation by the CUBE analysis report explorer, such that only the system-tree list is available which shows at most around 60 processes on a single screen. While imbalance is evident from the partial tree presentation and the distribution summarised in the associated box-plot, the precise nature of the imbalance (and correlations between call-paths) is much clearer in the TAU ParaProf bar-chart. A bug identified in the library that ParaProf uses for reading CUBE analysis reports has now been fixed.

7 Report on Activity ArgoDSM

Keywords: distributed shared-memory; MPI; MPI one-sided communication

7.1 Description of the application

ArgoDSM is a parallel programming model developed by Uppsala University, Sweden. It introduces a new PGAS design that reduces long-latency communications between sockets/nodes required in both coherence and critical section execution. Through that, execution of applications will be accelerated.

7.2 Previous assessments and recommendations

In previous observations, POP Audit POP_AR_42, we ported several benchmarks such as NAS EP and CG in order to utilize ArgoDSM APIs, namely made them execute on distributed systems. Then we investigated executions and identified several performance issues: performance drops with increasing number of threads and high time consumption of *pthread_mutex_lock* and *pthread_wait_barrier* functions. At this POP proof-of-concept activity, we looked at performance scalability in more detail based upon a real application, the FIRE application which retrieves a set of given images from a database.

7.3 PoC activities

In general, our work comprises two tasks: design and implement an algorithm to port the FIRE application and investigate execution performance with different number of threads. Our algorithmic design aims at improving load balance and eliminating data synchronisation that will reduce overhead introduced by ArgoDSM. According to our design, the given set of images to be retrieved is shared among threads. Each thread has a separate chunk of the database. Chunks have the same size. For any image, each thread compares it first to images in the local database chunk. After that, results are synchronised using ArgoDSM library calls and post-processed.

Processing a query image comprises 6 steps: retrieving, allocating global memory for final results, coping local results into global memory, calling a barrier to synchronise data, postprocessing results and finalising (deallocating global memory). Except the first step, any other steps call ArgoDSM functions. Each of the 6 steps is investigated separately through Score-P by instrumenting the source code manually.

7.4 Results

Performance scalability of the entire FIRE application is illustrated in Fig. 7. With up to 4 threads, the application scales well, the achieved speedup is almost linear to the number of employed threads. However, with many threads the observed speedup is much lower than the optimal value: with 128 threads, the speedup is only 3.6.

Looking at the execution with 128 threads, the retrieving step (first step) takes a small portion of the entire execution time, about 1 %. Time is mainly elapsed in ArgoDSM environment initialisation and function calls.

After tracing and analysing, we determined the following performance issues:

- ArgoDSM needs a long time to initialise the runtime environment.

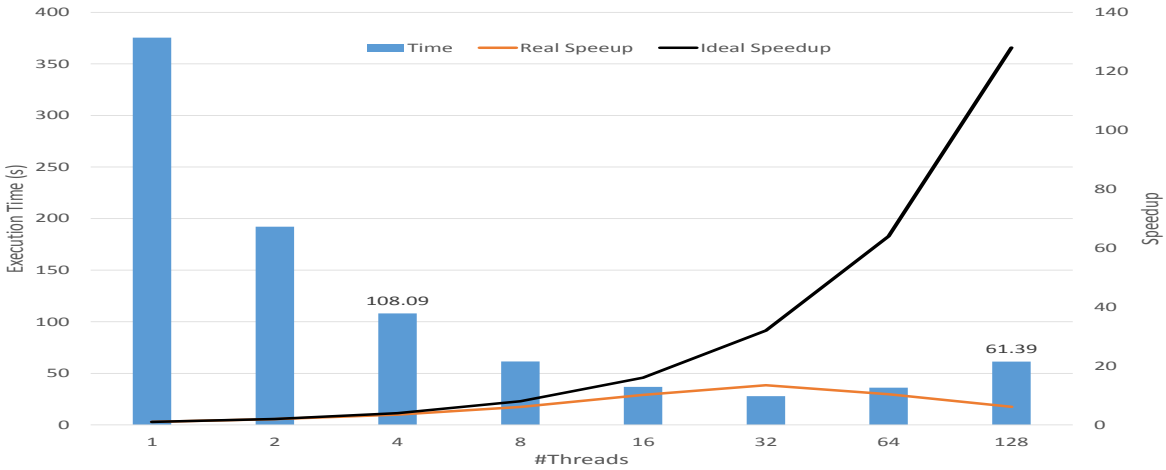


Figure 7: Performance scalability of FIRE after porting to ArgoDSM

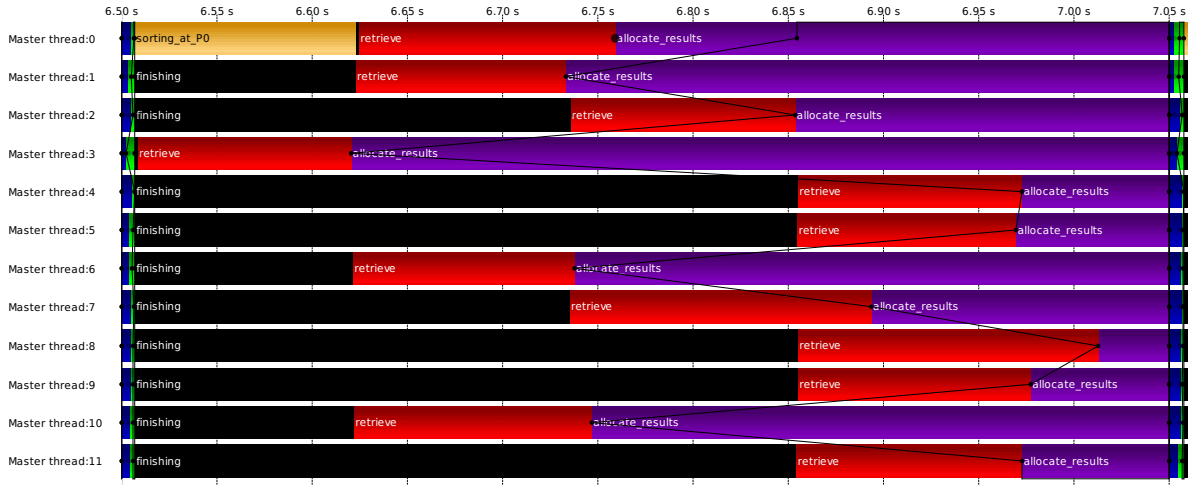


Figure 8: Serialising parallel execution through MPI one-side communications.

- ArgoDSM needs a long time to finish the first two image retrieving iterations, due to establishing communications.

- Executions are partly serialised, since MPI one-side communication calls wait for other MPI events. Fig. 8 illustrates this issue: the retrieving step of threads 1, 6 and 10 can continue only after thread 3 finished retrieving and called *MPI_Barrier*.

- Data communication is asymmetric. Data is collected by a few threads and distributed to other threads. Network bandwith to such threads can become a bottleneck.

- Some *MPI_get* functions are called redundantly, since no data is communicated in such calls.

Performance executing with many threads will be improved, if issues stated above are resolved. In particular, improvements on MPI implementations for one-side communication are necessary in order to avoid serialising parallel executions.

7.5 Conclusions

ArgoDSM is applicable for parallel executions with few threads / processes. The gained performance nears optimal values.

Running with many threads the achieved speedup can be lower than expected. The low scalability is due to increasing overhead introduced by ArgoDSM for runtime-environment initialisation and data synchronisation. In particular, since ArgoDSM is implemented based on MPI one-side communication, parallel execution can be serialised due to current MPI implementations.

Lessons Learned

Our analysis shows that performance of innovative programming model depends on its algorithm design as well as platform where it is implemented, in this case MPI. To improve performance, both aspects have to be considered.

8 Report on Activity Quantum ESPRESSO/FFTlib

Keywords: OmpSs; OmpSs tasking; overlap communication and computation; asynchronous task scheduling

8.1 Description of the Application

FFTXlib is the stand-alone miniapp that represents the Fast Fourier Transformation (FFT) kernel of Quantum ESPRESSO, one of the most used plane-wave DFT codes in the community of material science. The miniapp allows analysing the impact of the parallelisation parameters and their performance and is an easy-to-use tool for co-design and benchmarking of novel architectures like the KNL. The FFT kernel implements a layered MPI communication with FFT task groups to split the cost of collective communication operations to balance the impact on the performance. However, the complexity in the many parallelisation layers can be hard to manage, which was one reason for the development of the miniapp.

8.2 Previous Assessments and Recommendations

We analysed the FFTXlib with the performance tools Extrae and Paraver to understand the behaviour of the FFTXlib on the KNL architecture and identified two main performance issues that arise when scaling to a full KNL node. The first issue is the increasing communication cost for the use of the collective communication operations. The second issue is the decreasing computation efficiency caused by a decreasing IPC, where we identified resource contention as the main contributor.

8.3 PoC activities

We developed and implemented two approaches that use tasks with the OmpSs parallel programming model. The first approach targets the increasing communication costs by overlapping computation and communication. The second approach targets the decreasing computation efficiency by softening resource contention, which is done by asynchronously scheduled tasks, so, compute phases of high resource requirements can be overlapped with phases of low resource requirements.

8.4 Results

We implemented both approaches in the FFTXlib and evaluate the results of the second approach on the Knights Landing test system. We choose the second version since the test node contains only 68 cores and the clock frequency of 1.4GHz is lower than on standard CPUs, i.e. issues in computation are more critical than issues in communication. The task-based optimisation reduces resource contention and increases the IPC about 10% in the main compute phase. As a result, the FFT phase shows up to 10% runtime reduction on the already highly optimised version. An increased performance in the FFTXlib will likewise increase the performance of the entire Quantum ESPRESSO suite. Furthermore, since the task scheduling is done dynamically during execution, not statically within the code, it relieves some of the complexity of the two-layered MPI communication from the user and shifts it to the parallel runtime.

Figure 9 shows the runtime of the FFT phase with an increasing number of MPI ranks for the second optimisation strategy with OmpSs vs. the original version (the last two entries use 2 and 4 hyper-threads per core, respectively). Thereby, the original version uses Nx8 MPI

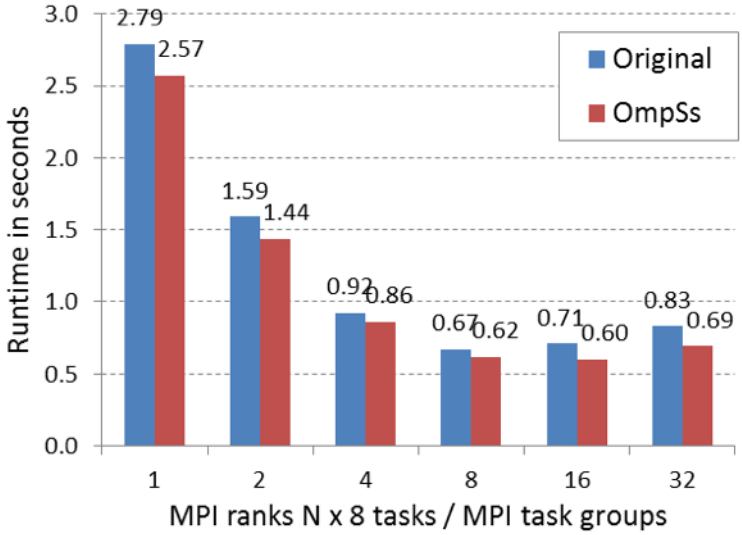


Figure 9: Runtime of the FFT phase with increasing number of MPI ranks with the following parameters: Plane wave energy cut off: 80, lattice parameter: 20, number of bands: 128, number of task groups: 8 (original), 1 (OmpSs).

ranks, i.e. N ranks for the first MPI layer and 8 FFT task groups. The OmpSs version uses N MPI ranks and 8 threads that replace the FFT task groups. From the figure it can be seen that the version using OmpSs performs the FFT phase about 7-10% faster (not counting hyper-threading), in particular, the fastest version with OmpSs (16x8) is about 11% faster as the fastest original version (8x8).

The ideas and results of the proof-of-concept report were compiled into a paper that was accepted and published [3]. This further underlines the success of the analysis and proof-of-concept study performed within the POP project and the fruitful collaboration with the customer.

8.5 Conclusion

In the proof-of-concept study for the FFTXlib, which implements the FFT kernel of the Quantum ESPRESSO suite, we achieved an improvement of about 10% on the already highly optimised version. The benefits rely entirely on the use of a task-based programming model that supports task dependencies. For that we use the OmpSs programming model.

Lessons Learned

While the OpenMP standard includes task dependencies since version 4.0, still, many compilers do not support this standard completely. Our recommendation would be to urge compiler developers to fully comply with the OpenMP standard to allow a wider usage of applications using task-based programming models

9 Report on Activity BPMF MPI

Keywords: MPI; MPI collective operations; MPI shared-memory model

9.1 Description of the Application

The BPMF application is a Bayesian Probabilistic Matrix Factorization code for the simulation of recommender system that allows to predict compound-on-protein activity in chemogenomics from the chembl database. BPMF is developed and distributed at the ExaScience Life Lab in Belgium. This implementation uses Markov Chain Monte Carlo (MCMC) method. The code is written in C++, parallelised with MPI and run in a hybrid distributed-shared memory system. Below this version of BPMF application is called as original broadcast version.

9.2 Previous Assessment and Recommendation

In the previous POP performance audit POP_AR_19 the application's communication behaviour gets analysed by profiling the number of invocations to the MPI communication-related routines. The statistics demonstrated that a high number of invocations to MPI non-blocking point-to-point routines and broadcast routine occurred, which would potentially lead to low Communication Efficiency (CE). To precisely pinpoint the cause of bad CE, the audit strongly recommended further run of BPMF application using a larger number of processes.

Region A+B	MPI_Isend	MPI_Irecv	MPI_Test	MPI_Wait	MPI_Bcast
1st Measurement	7.01%	1.33%	0.48%	1.99%	89.19%
2nd Measurement	3.89%	0.36%	1.06%	0.62%	94.07%

Table 3: MPI message passing statistics for one iteration of the first and second measurements. Values are the average time percentage of MPI function in the total communication time across all processes.

Accordingly, two runs of the BPMF application with two measurement setups on Cray XC40 system were performed, from which we can correctly find the performance bottleneck. The first measurement launches 120 cores (5 nodes) while the second measurement uses 360 cores (15 nodes). Table 3 shows the role of each of the above MPI communication routines from the perspective of time percentage. Compared to *MPI_Bcast*, the point-to-point communication routines bring insignificant overhead. Clearly, the calls to *MPI_Bcast* is the major contributor to the bad CE and liable to become the performance bottleneck.

The recommendations were to replace the broadcast routines with the more appropriate collective communication pattern (i.e., all-to-all gather routines) and optimise the native all-gather algorithm based on the MPI Shared Memory (SHM) model to reduce the occurrence of inter-node data exchanges. The goal of this PoC study is to improve the MPI implementation of the BPMF application for achieving scalable performance on hybrid distributed-shared memory system.

9.3 PoC Activities

Allgather + Iallgather

Our first step was the substitution of MPI broadcast calls with three all-to-all gather communication operations, with which the dependence of the number of invocations to collective

communication operations on the process count got removed. The CE and Parallel Efficiency (PE) were both increased by 21% after the substitution. This optimised version is below named as the native allgather version.

Allgatherv + MPI SHM model

Our second step was the appliance of a hybrid communication scheme of combining MPI and MPI SHM model [4] to the rewriting of the native allgather algorithm. In this hybrid approach the processes within one node do not need to collect the data because the on-node data can be shared. In this regard, a representative for each node needs to be chosen and then they constitute a node communicator. The following all-to-all gather operation only happen on this node communicator. This will lead to a tremendous reduction in the participants of each allgather communication. Because the sent message size associated with each node varies, we replace the *MPI_Allgather* and *MPI_Iallgather* with *MPI_Allgatherv* routines [5] since this special variant routine allows the programmer to specify a different number of elements for each of the involved processes. In the following, this optimised version is called the hybrid allgather version.

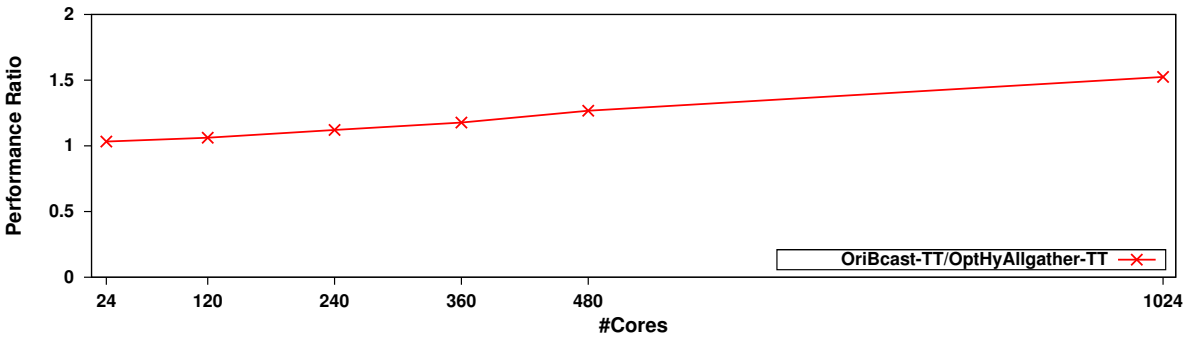


Figure 10: Performance ratios of the total time as the number of the MPI processes is increased from 24 to 1024, for BPMF application. *OriBcast* signifies the original broadcast version. *OptHyAllgather* signifies the hybrid allgather version. *TT* means the total time of the tested 20 iterations in BPMF application.

The PE and CE are essentially improved in a manner similar to the native allgather version, although the improvement is greater with an increase of around 25% over the original broadcast version. The performance ratios of the BPMF application time in the hybrid version relative to that in the original broadcast version are illustrated in Figure 10. The ratio curve is on a slow rise as the increasing number of cores. It can be deduced that, the profit from the adoption of hybrid approach tends to grow steadily as the process population increases. Such incremental benefit can bring better scalability. Specifically, the ratio slightly increases (by 18%) with a total of 1024 cores.

9.4 Conclusions

During this PoC study the BPMF code was optimised to obtain better scalability across multiple nodes and all goals were fulfilled in terms of the given evaluation metrics. This PoC report focused on the improvement of the applied MPI collective communication behaviour, which was proved to be the major performance bottleneck. The substitution of the original broadcast communication with the all-to-all gather communication operations led to an improvement of up to 21% in both CE and PE for this BPMF application. The hybrid all-to-all gather scheme

based on MPI SHM model was further applied which greatly reduced the occurrence of inter-process data transfers. The communication overhead thus fell by up to 89%.

Lessons Learned

Our experiences with the above improvement of the BPMF application indicate the scalability issue of MPI collective communication operations and the powerful role that the MPI-3 SHM model plays in large-scale application and system.

We strongly recommend the usage of MPI-3 SHM model when possible since applications need to be always memory efficient. This can also fix the memory consumption issue on the large scale systems by only maintaining one copy of data shared by all processes within one node instead of making local copies of data for all processes. The MPI collective communication operation, especially all-to-all, however, is not a scalable communication pattern. Taking the on-node SHM scheme into consideration the scalability issue of MPI collective communication operation itself will most likely get alleviated. Therefore, we sincerely hope that MPI standard could add some new interfaces to accommodate the collective functionalities to the on-node SHM model. In detail, a pre-defined communicator split type for forming the node communicator can be included for facilitating programmers and improving the application productivity as well. Providing a relative cheap mechanism other than the calls to barrier operation is preferable to synchronise the accesses to the shared data in the middle of exchanging them between different nodes.

It is widely believed that large scale systems will require support for hybrid programming. The HPC-related training curricula should be aware that, apart from the traditional combination of MPI and threading programming model (OpenMP or Pthreads), the innovative combination of MPI and MPI SHM model is worth encouraging and enhancing. The latter hybrid style perfectly matches the existing applications that run in process-based model and then facilitates the MPI application porting. Besides, the mock-up tests using the MPI SHM model offered in the curricula should emphasise not only the examples dealing with the simple on-node halo exchanges, but also the approaches of performing multiple across-node data exchanges.

10 Conclusions

In this document we have reported the current state of Proof-of-Concept activities, which is the main service activity in Work Package 5. PoC is the highest level of service of the POP Project aiming at assisting customers to implement recommendations of previous POP Project performance assessments.

So far, the POP Project has concluded 10 PoC activities, with a further 4 in progress. These activities have in most cases led to significant improvement of the respective code's performance under the conditions define at the beginning of the activity. More importantly however, follow-up communication done under the Work Package 2 shows, that significant improvements are also realised under production conditions on arbitrary data sets.

While the lowish number of completed PoC does not allow us to draw any robust conclusion, it is worth to noting, that a large part of the codes struggle with single-core performance, not only parallel performance. In the second year, we continue to observe the trend that a significant number of PoC applications suffer from some kind of load imbalance. However, we have observed various reasons for load imbalance, as for instance data-locality issues which lead to different execution times depending where the task is allocated, but also classical imbalance in the amount of computation or communication volume.

In this document, we have also shown that exploitation of modern parallel programming techniques, such as task nesting and user-provided reductions in OpenMP (PoC BPMF OpenMP), combination of collective operations and shared-memory operations in MPI (PoC BPMF MPI), and usage of data-dependencies in OmpSs (PoC Quantum ESPRESSO), can significantly improve application performance. On the other side, we have also shown that supposedly beneficial parallel programming techniques do not lead to the expected performance increase. One such example is the PoC activity for DFTB, where the usage of non-blocking MPI collectives unexpectedly did not result in a performance benefit from overlapping communication and computation. In this particular case one might suspect, that the specific MPI implementation on the system does not properly support progression on asynchronous MPI communication in the background.

Acronyms and Abbreviations

- BSC: Barcelona Supercomputing Center
- CA: Consortium Agreement
- CAdv: Customer Advocate
- DoA Description of Action (Annex 1 of the Grant Agreement)
- D: deliverable
- EC European Commission
- FFT: Fast-Fourier-Transform
- GA: General Assembly / Grant Agreement
- HLRS: High Performance Computing Centre (University of Stuttgart)
- HPC: High Performance Computing
- IPR Intellectual Property Right
- Juelich: Forschungszentrum Juelich GmbH
- KPI: Key Performance Indicator
- NUMA: non-uniform memory access
- M: Month
- MPI: Message-Passing Interface, a shared-memory programming model
- MS: Milestones
- OpenMP: a shared-memory programming model
- PEB: Project Executive Board
- PM: Person month / Project manager
- PoC: Proof-of-Concept
- POP: Performance Optimization and Productivity
- R: Risk
- RV: Review
- RWTH Aachen: Rheinisch-Westfaelische Technische Hochschule Aachen
- USTUTT (HLRS): University of Stuttgart
- WP: Work Package
- WPL: Work Package Leader

References

[1] R. van de Geijn, J. Watts, SUMMA: Scalable universal matrix multiplication algorithm, http://www.netlib.org/lapack/lawnspdf/lawn96.pdf, accessed: 2017-09-30.

[2] Intel MPI non-blocking communications benchmark, https://software.intel.com/en-us/node/561946, accessed: 2017-09-30.

[3] M. Wagner, V. López, J. Morillo, C. Cavazzoni, F. Affinito, J. Giménez, J. Labarta, Performance analysis and optimization of the fftxlib on the intel knights landing architecture, in: 46th International Conference on Parallel Processing Workshops, ICPP Workshops 2017, Bristol, United Kingdom, August 14-17, 2017, IEEE Computer Society, 2017, pp. 243–250. doi:10.1109/ICPPW.2017.44.
URL https://doi.org/10.1109/ICPPW.2017.44

[4] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, MPI+MPI: A new hybrid approach to parallel programming with MPI plus shared memory. (2013) 1121–1136.

[5] MPI Forum, MPI: A Message-Passing Interface Standard. Version 3.0, Tech. rep., available at: http://www.mpi-forum.org (Sep. 2012).