# D6.2 – POP Proof-of-Concept Report
# Version 1.0

## Document Information

| | |
|---|---|
| Contract Number | 824080 |
| Project Website | www.pop-coe.eu |
| Contractual Deadline | Month 42, May 2022 |
| Dissemination Level | Public |
| Nature | Report |
| Authors | José Gracia (USTUTT) |
| Contributors | Marta Garcia (BSC), Tomas Panoc (IT4I), Martin Rose (USTUTT), Andres Charif Rubial (UVSQ), Radita Liem (AACHEN), Jonathan Boyle (NAG) |
| Reviewers | Marta Gasulla (BSC) |
| Keywords | Proof-of-Concept |

***Notices:*** *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No n° 824080.*

# Change Log

| Version | Author | Description of Change |
|---|---|---|
| v0.1 | José Gracia | Initial structure of document |
| v0.5 | Marta Gasulla | Internal review |
| v1.0 | José Gracia | Released to the EC |

# Contents

**1 High-level Overview of Proof-of-Concept Activities** **5**

**2 Report on Activity CODE_A** **6**
2.1 Description of the Application . . . 6
2.2 Previous Assessments and Recommendations . . . 6
2.3 PoC activities . . . 7
2.4 Results . . . 12
2.5 Conclusion . . . 14
2.6 Lessons Learned . . . 14

**3 Report on Activity CODE_B** **15**
3.1 Description of the Application . . . 15
3.2 Previous Assessments and Recommendations . . . 15
3.3 PoC activities . . . 17
3.4 Results . . . 20
3.5 Conclusion . . . 22
3.6 Lessons Learned . . . 22

**4 Report on Activity CODE_C** **23**
4.1 Description of the Application . . . 23
4.2 Previous Assessments and Recommendations . . . 23
4.3 PoC activities . . . 23
4.4 Change MPI communication pattern that avoid quadratic growth . . . 24
4.5 Dynamic Load balancing . . . 24
4.6 Results . . . 28
4.7 Conclusion . . . 29
4.8 Lessons Learned . . . 31

**5 Report on Activity CODE_D** **32**
5.1 Description of the Application . . . 32
5.2 Previous Assessments and Recommendations . . . 32
5.3 PoC activities . . . 32
5.4 Results . . . 41
5.5 Conclusion . . . 44
5.6 Lessons Learned . . . 44

**6 Report on Activity CODE_E** **45**
6.1 Description of the Application . . . 45
6.2 Previous Assessments and Recommendations . . . 45
6.3 PoC activities . . . 46
6.4 Porting "calc_ddx_visc_cfd_O6" - kernel_2 . . . 49
6.5 Results . . . 50
6.6 Conclusion . . . 51
6.7 Lessons Learned . . . 52

**7 Report on Activity CODE_F** **53**
7.1 Description of the Application . . . 53
7.2 Previous Assessments and Recommendations . . . 53
7.3 PoC activities . . . 53
7.4 Results . . . 56
7.5 Conclusion . . . 59
7.6 Lessons Learned . . . 59

**8 Report on Activity CODE_H** **60**
8.1 Description of the Application . . . 60
8.2 Previous Assessments and Recommendations . . . 60
8.3 PoC activities . . . 60
8.4 Results . . . 60
8.5 Conclusion and lessons learned . . . 65

**9 Report on Activity CODE_G** **66**
9.1 Description of the Application . . . 66
9.2 Previous Assessments and Recommendations . . . 66
9.3 PoC activities . . . 66
9.4 Results . . . 70
9.5 Conclusion . . . 71
9.6 Lessons Learned . . . 72

**10 Report on Activity CODE_I** **73**
10.1 Description of the Application . . . 73
10.2 Previous Assessments and Recommendations . . . 73
10.3 PoC activities . . . 73
10.4 Results . . . 78
10.5 Conclusion . . . 78
10.6 Lessons Learned . . . 79

**11 Report on Activity CODE_J** **80**
11.1 Description of the Application . . . 80
11.2 Previous Assessments and Recommendations . . . 80
11.3 PoC activities . . . 80
11.4 Results . . . 84
11.5 Conclusion . . . 85
11.6 Lessons Learned . . . 85

**12 Report on Activity Energy-efficiency analyses** **86**
12.1 Description of the Applications . . . 86
12.2 Previous Assessments and Recommendations . . . 86
12.3 Results . . . 90
12.4 Conclusion . . . 91
12.5 Lessons Learned . . . 91

**Acronyms and Abbreviations** **92**

**References** **93**

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

The project POP2 offers three distinct service activities. The first one is the performance analysis service hosted in Work Package 5. Starting with an initial assessment of the applications performance characteristics and identification of possible bottlenecks, it is usually followed by more detailed root cause analysis of bottlenecks. One of the outputs of theses assessments, is concrete recommendations for changes of the application's 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 develop domain science applications codes. Therefore, POP offers a second 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 customer's application or possibly on a simplified skeleton thereof. The aim is to illustrate the necessary changes and to train the customers on their own code.

The procedure of the PoC activities has been defined in the previous project's deliverable D5.1 [1] and remains essentially unchanged. In short, customer and POP staff prepare a PoC Plan. This document defines the specific use-case and lists the particular recommendations that will be addressed during the activity. It also defines a metric to track progress and optionally sets a target to be reached. At the end of the activity, POP staff prepares a PoC Report which is handed over to the customer for approval, thus concluding the activity. Due to their nature, PoC activities take more time than analysis activities. We expect PoC activities to last between 3 and 6 months.

PoC activities are expected to yield best-practice guidelines and input for training material and training events, which will be taken up in Work Packages 6 *Dissemination, Cooperation, and Training*. We also expect some feedback on and suggestions for improvement of practice of performance assessment in Work Package 5 *Performance Assessments*, and further hope to identify common patterns for inclusion in the kernel repository of Work Package 7 *Co-design Activities*.

By the end of the project, POP2 staff has concluded 26 PoC activities. A further 7 activities are still in progress and will be finalised after the formal end of the project. This deliverable presents a selection of 10 PoC activities covering a wide range of performance issues, application areas, and parallelization techniques. In addition, we have also done several studies to assess and optimize the energy efficiency of applications by varying operational parameters, etc. While these are not strictly PoCs is the original sense, no code is refactored, they still demonstrate energy saving techniques and are thus reported here as well in a separate chapter.

For reasons of data protection and confidentiality in ongoing activities, this deliverable will use pseudonyms rather than real code names. After conclusion of PoC activities (and for that matter also assessments), the project actively seeks permission from customers to make reports publicly available. These can be accessed on the project's website.

The remainder of this document consists of a chapter for each of the PoC activities, followed by a chapter on the energy-efficiency studies. We make some final observations in Section **??**.

# 2 Report on Activity CODE_A

## 2.1 Description of the Application

CODE_A is an open source finite element analysis with pthread as its programming model. It solves the computation fluid dynamic problem iteratively until it reaches convergence.

## 2.2 Previous Assessments and Recommendations

In the performance analysis assessment, we selected a function named *dgmres1mt* as our Focus of Analysis. Function *dgmres1mt* accounts for around 75% time inside the application. POP metrics result for our initial assessment is as follows:

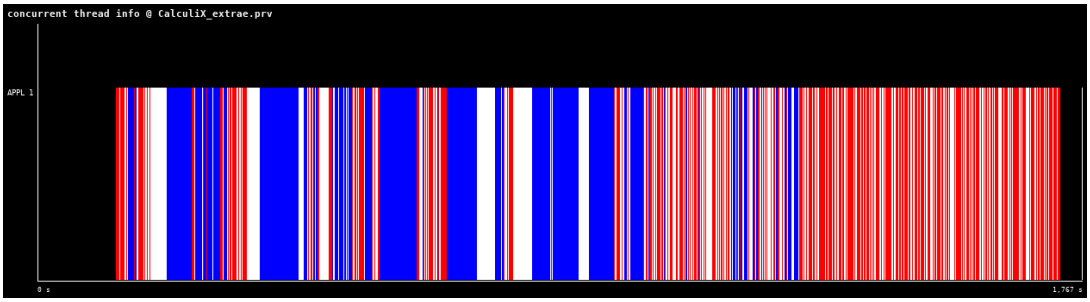
Table 1: Efficiency Metrics of ***dgmres1mt*** on INCF=5 and FREQUENCYF=5

| | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| Global efficiency | 80% | 73% | 61% | 68% |
| Parallel efficiency | 80% | 73% | 72% | 72% |
| Load Balance | 80% | 73% | 72% | 72% |
| Communication efficiency | 100% | 100% | 100% | 100% |
| Computation scalability | 100% | 100% | 85% | 95% |
| IPC scalability | 100% | 103% | 89% | 101% |
| Instruction scalability | 100% | 98% | 95% | 94% |

From table 1, we see that Load Balance is the main issue of the application. Another issue is in the application's I/O performance, where our analysis shows that I/O costs 40% time of the application. Both issues are in our performance improvement plan.

Besides producing a high number of threads, the application also shows several peculiar behaviors. We cannot get a deterministic result if the number of OMP_NUM_THREADS is bigger than one. Nearing the end of the iterations, the number of concurrent threads also increases, as we can see from figure 1. At first, we were suspecting that the application's algorithm is the cause of the problem. However, after a thorough investigation, we found that this behavior is from data race.

Figure 1: This image shows concurrent threads inside the application. Red color shows the part where there are three threads running at the same time




Based on our assessments, there are four action items that we can recommend:

- Fixing the the data race inside the application.
- Refactoring the application using OpenMP. By refactoring the application using OpenMP, the performance analysis tools can work on measurement with high number of threads. This is important to ensure the scalability of the application when working with bigger problem size.
- Improving load balance. As it has been shown by the POP metrics, load balance is one of the performance issues of the application.
- Improvement on the I/O write practice to reduce I/O time of the application.

We have delivered two action items to ensure the application's correctness and enable the performance measurement tools to work properly with this application. In the end of the first part of the PoC where we fixed the data race and conducting a refactoring, we saw that there were increases in the efficiencies according to the POP metrics (table 2)

Table 2: Refactored code efficiency metrics for ***dgmres1mt*** with INCF=5 and FREQUENCYF=5. Green color is for the increase from the original value, red color if there is a decrease, and black for no change. The value inside in the bracket is the number difference

| | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| Global efficiency | 80% | 82% (+8%) | 73% (+12%) | 71% (+3%) |
| Parallel efficiency | 80% | 74% (+1%) | 76% (+4%) | 77% (+5%) |
| Load Balance | 81% (+1%) | 75% (+2%) | 77% (+5%) | 78% (+6%) |
| Communication efficiency | 99% | 99% | 99% | 99% |
| Computational scalability | 100% | 110% (+10%) | 96% (+11%) | 92% (-3%) |
| IPC scalability | 100% | 100% (-3%) | 100% (+11%) | 100% (-1%) |
| Instruction scalability | 100% | 110% (+12%) | 96% (+3%) | 92% (-2%) |

## 2.3 PoC activities

### 2.3.1 Scope

We are splitting the PoC report into two parts: Application's Correctness and Performance Improvement to make the works manageable. This performance plan will focus on the second part of the PoC where we are trying to improve the Application's Performance. Action items that will be covered in this Application Improvement report are improving the load balance and I/O practice of the application.

**Use-case and evaluation metrics** We are validating the performance improvement with the same test case we used for performance audit and the first PoC. It is a data sample that calculates the laminar flow of air through a bent pipe. The cross-section of the pipe is square, and the bend is 90 degrees.

As the basis to measure the performance improvement in this PoC, we were using POP measurement number of the application's OpenMP version from the refactoring result in the first PoC (table 3). The tools to generate these performance numbers are the same performance

measurement tools listed in the performance analysis: Paraver[1], Extrae[2], Score-P[3], Cube[4], Vampir[5], Archer[6].

Table 3: Efficiency Metrics of CODE_A's OpenMP version for ***dgmres1mt*** function using 100 increments

| | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| Global efficiency | 80% | 85% | 88% | 89% |
| Parallel efficiency | 80% | 76% | 75% | 75% |
| Load Balance | 81% | 77% | 76% | 76% |
| Communication efficiency | 99% | 99% | 99% | 99% |
| Serial efficiency | 99% | 99% | 99% | 99% |
| Transfer efficiency | 100% | 100% | 100% | 100% |
| Computation scalability | 100% | 112% | 117% | 118% |
| IPC scalability | 100% | 106% | 108% | 109% |
| Instruction scalability | 100% | 106% | 108% | 108% |

**Target system** We are using the same system we used in performance analysis and the first part of the PoC, RWTH Aachen University Compute Cluster with specification as follows:

- CLAIX-2018 compute cluster. The system has 48 cores Intel Skylake processor, 384 GB memory, and Lustre file system.
- Intel MPI 2018 compiler and Intel version 19.0 compiler.
- Lustre filesystem.

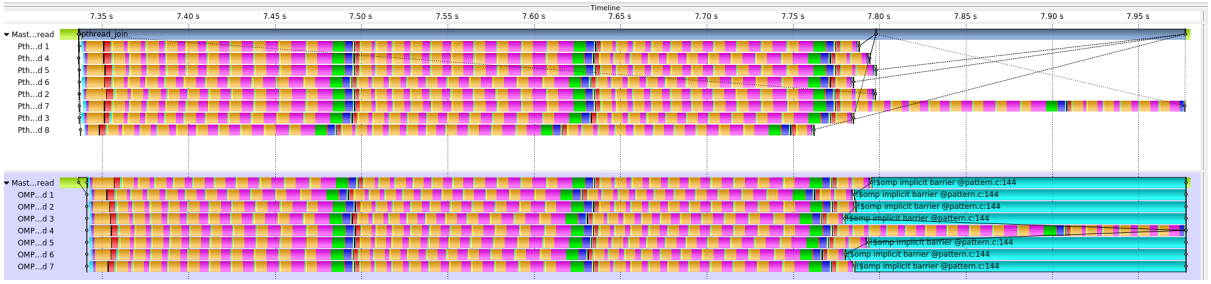
### 2.3.2 Implementation

**Load Balance Optimization** In the performance assessment we found out that the function `dgmres1mt` has load imbalance both in original Pthread code and the OpenMP version. Figure 2 is a kernel program where it contains single call to the function that we can use observe the load imbalance. When executed the original code with 8 threads on 5 increments, we obtained a Load Balance efficiency of 72% and after we did our refactoring, the load balance is now 77%. We believe that this load balance can still be improved further. The load imbalance itself is caused by the GMRES solver that is used to solve a non-symmetric system of linear equations inside this function.

Inside CODE_A, a large system matrix is split into smaller, individual subproblems. The number of subproblems depends on the number of threads used, and each thread gets its own subproblem which is then solved by a serial GMRES solver. Since these systems are independent of each other, their solutions converge with different speed. Here, we noticed that one thread always takes longer to finish the GMRES computation. While all the other threads finish after a similar amount of GMRES solver iterations, this one thread needs to perform more

[1] https://tools.bsc.es/paraver
[2] https://tools.bsc.es/extrae
[3] https://www.vi-hps.org/projects/score-p/
[4] https://www.scalasca.org/scalasca/software/cube-4.x/download.html
[5] https://vampir.eu/
[6] https://pruners.github.io/archer/

Figure 2: Score-P trace comparison of the CODE_A-solver kernel pthread version (top) and the OpenMP version (bottom).




solver iterations until its solution converges. Moreover, we noticed that this scheme continues to happen over the course of the whole simulation run.

In order to improve the load balance we have tried different approaches:

- **Tasking approach:**
  Our initial idea is to create a large number of smaller subproblems instead of just one subproblem for each thread. Each subproblem is expressed as an OpenMP task. If a subproblem takes longer to converge then other threads could execute other subproblems meanwhile, which should increase the Load Balance efficiency.

  The result for this approach shows that the more tasks we create, the better the Load Balance efficiency approaches to 100%. However, the number of instructions executed also increases with the number of tasks. Therefore, there are more computational work is performed than originally required. Overall, this leads to a slowdown of the kernel even though the kernel is now almost perfectly balanced.

Table 4: Results from the tasking approach

| Tasks / thread | Accumulated #iterations | Load Balance Efficiency | #instructions | Runtime accumulated over all threads (sec.) |
|---|---|---|---|---|
| 1 | 266 | 66% | 3.83e10 | 3.73 |
| 5 | 1980 | 89% | 5.63e10 | 6.05 |
| 10 | 4022 | 96% | 5.71e10 | 5.63 |
| 50 | 23040 | 99% | 6.61e10 | 6.59 |
| 100 | 44229 | 99% | 6.41e10 | 6.76 |
| 150 | 63828 | 99% | 6.25e10 | 6.98 |

- **Conditional nested taskloop approach:**
  Our second idea is to introduce a conditional check whether all threads are still computing or not. Furthermore, we identified several subroutines of the serial GMRES solver that could be parallelized. Subroutines that can be parallelized are: ***matvec***, ***daxpy***, ***msolve***, ***drlcal***. If our introduced condition states that only the straggling thread is still computing its solution while all the other threads are already done the straggling thread will split its computation into tasks. The granularity of these tasks is determined by the number of threads that already finished their computation such that each of these threads gets exactly one task. This approach tries to use the idling compute ressources to speed up

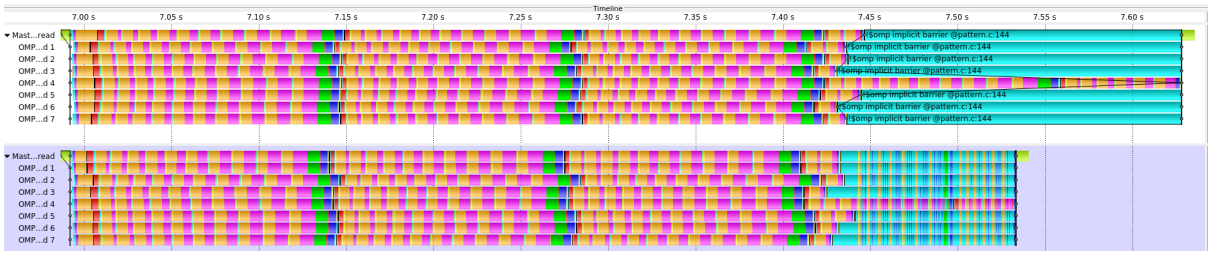
the computation on the straggling thread.

In the code snippet below, we are showing the modification on ***matvec.f***:

```
nThreads = OMP_GET_NUM_THREADS()

!$OMP ATOMIC READ
addThreads = freeThreads

if (addThreads .GE. gmres_task_threshold * nThreads) then
  gs = int((n-1) / (addThreads+1)) + 1
  do k=1,n,gs

!$OMP TASK SHARED(x,y,n,nelt,ia,ja,a,isym) IF(gs.ne.n)
    do i=k,min(k+gs-1,n)
      y(i)=a(ja(i)+1)*x(ia(ja(i)+1))
      do j=ja(i)+2,ja(i+1)
        y(i)=y(i)+a(j)*x(ia(j))
      enddo
    enddo
!$OMP END TASK
  enddo

!$OMP TASKWAIT
else
  do i=1,n
    y(i)=a(ja(i)+1)*x(ia(ja(i)+1))
    do j=ja(i)+2,ja(i+1)
      y(i)=y(i)+a(j)*x(ia(j))
    enddo
  enddo
endif
```

As for the result of this approach, in figure 3, the top trace shows the original pattern of the OpenMP version of the kernel, and the bottom trace shows the implemented best-practice using nested tasks in the OpenMP version of the kernel. One can recognize that as soon as the other threads are finished with their workload the straggling thread can accelerate its computation by splitting it into tasks. These tasks can then be executed by the idling threads as well.

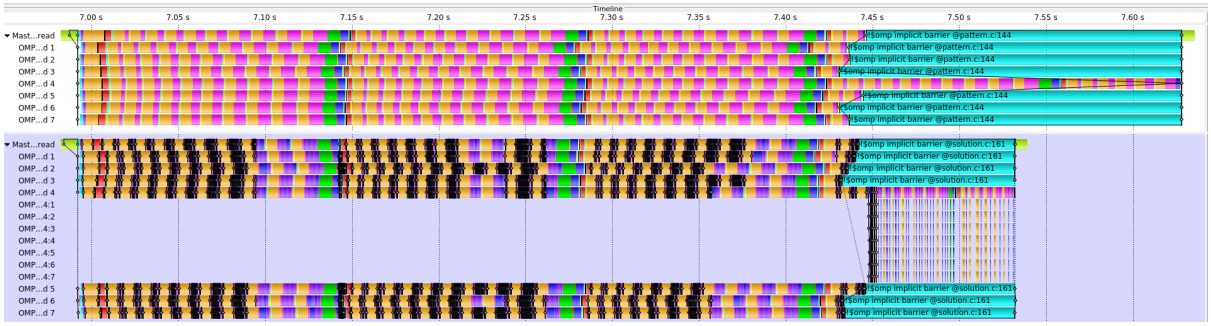
Figure 3: Score-P trace comparison of the CODE_A-solver pattern OpenMP version (top) and the implemented best-practice using nested tasks (bottom).




- **Conditional nested parallel region approach:**
  Our third idea is very similar to the second one. But, instead of conditionally splitting subcomputations into tasks, we use a parallel region that spawns an extra number of threads that corresponds to the number of threads that already finished their computation. In order to make sure that these additional threads will be executed on the idling

cpu cores, we pinned the application to a number of physical cpu cores that corresponds to the number of threads the application was originally executed with. This should simulate the same behavior as if a task would be scheduled on one of the idling threads.

Figure 4: Score-P trace comparison of the CODE_A-solver pattern OpenMP version (top) and the implemented best-practice using nested parallel regions (bottom).




The trace at the top of the figure 4 shows the original pattern of the OpenMP version of the kernel. The bottom trace shows the implemented best-practice using nested parallel regions in the OpenMP version of the kernel. One can recognize that as soon as the other threads are finished with their workload the straggling thread can accelerate its computation by executing the nested parallel regions inside the mentioned subroutines with multiple threads that run on the idling cpu cores.

In addition, we also investigated the performance of several GMRES libraries, which are Intel MKL and the NAG library. Both libraries required the same amount of iterations until convergence as the original CODE_A kernel. With Intel MKL, we got a speedup of only 0.65. Therefore, the execution was actually slowed down. The relative error between the reference solution from the CODE_A application and the computed one by Intel MKL is 6.06e-13. Using the NAG library, we got a speedup of 1.13 compared to the original CacluliX kernel. Compared to our nested parallel region approach, the NAG library is only 5% slower. The computed solution has a relative error of 1.64e-12. The customer does not want to use any commercial software package because the CODE_A code should remain open source. Based on our findings, we can justify that the original solver implementation and our improvements are quite comparable to the commercial, proprietary solutions of Intel and NAG.

Table 5: Results from GMRES comparison

| | Original | Nested | MKL | NAG |
|---|---|---|---|---|
| Time (sec) | 0.646310 | 0.545439 | 0.7191 | 0.572196 |
| Rel. Err. | 3.20868e-16 | 3.2086e-16 | 6.06263e-13 | 1.64144e-12 |

For the load balance improvement, we suggest to use nested taskloop approach. It performs better than the tasking approach and it is relatively easy to use compared to the conditional nested parallel region approach. In this nested taskloop approach, we set the condition that the tasking will start once half of the total threads are idling. The result can be seen in table 6:
The POP metrics result shows that there are increases in Global Efficiency compared to the base numbers in table 3. It is mainly driven by the Load Balance improvement (around 15% increase) even though we see decreases in Communication Efficiency (around 5% decrease)

Table 6: Efficiency metrics of ***dgmres1mt*** with conditional nested taskloop approach using bend dataset on 100 increments

| | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| Global Efficiency | 92% (+12%) | 101% (+16%) | 105% (+17%) | 106% (+17%) |
| Parallel Efficiency | 92% (+12%) | 87% (+11%) | 87% (+12%) | 86% (+10%) |
| Load Balance | 98% (+17%) | 93% (+16%) | 92% (+16%) | 90% (+14%) |
| Communication Efficiency | 94% (-5%) | 94% (-5%) | 95% (-4%) | 95% (-4%) |
| Serial Efficiency | 94% (-5%) | 94% (-5%) | 95% (-4%) | 95% (-4%) |
| Transfer Efficiency | 100% | 100% | 100% | 100% |
| Computational scalability | 100% | 116% (+4%) | 120% (+3%) | 124% (+6%) |
| IPC scalability | 100% | 103% (-3%) | 104% (-5%) | 105% (-4%) |
| Instruction scalability | 100% | 113% (+6%) | 116% (+8%) | 119% (+10%) |

due to the nested taskloop usage. We get high numbers for Instruction Scalability due to the algorithm of the application. Since the large system matrix gets split into smaller independent subproblems (one for each thread) the overall computational work is different for different numbers of threads. The smaller problems are a little easier to solve than the larger ones so they require less instructions.

**I/O Optimization** The application produces output data with a user-defined frequency. Our observation in the application's I/O behavior is that the application writes very small chunks of data (26 bytes) continuously to file. The application can performs more than 1.6 million times of such write activities. It is more efficient to accumulate the data and write larger chunks of data to file by adding buffered parameter in ***openfile.f***:

```
1 open(11,file=fncvg(1:i+4),status='unknown',err=91, buffered='yes')
```

We implemented buffered I/O operations to improve the I/O performance of the CODE_A application. As a result on 8 threads, we achieved an overall speedup of 1.43 for the whole application (table 7). Buffered I/O only affects the runtime improvement, there is no significant change from the base numbers as we can see from table 8

Table 7: Runtime comparison between the base runtime and buffered I/O runtime using INCF=100 and FREQUENCYF=100 settings

| | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| Base Runtime (sec) | 677.60 | 465.72 | 405.63 | 368.47 |
| Buffered I/O Runtime (sec) | 539.93 | 348.44 | 286.11 | 256.91 |
| Speedup | 1.25 | 1.33 | 1.42 | 1.43 |

## 2.4 Results

To evaluate our PoC activity, we generated the POP metrics with our proposed load balance improvement strategy, nested taskloop approach and also buffered I/O. The result can be seen in table 9. From the table, we can see that the Global Efficiency number is higher than the number from the load balance improvement only (table 6) and buffered I/O improvement only (6). The main driver for this Global Efficiency improvement is coming from the Load Balance numbers. This Load Balance improvement is even higher than in the nested taskloop only result.

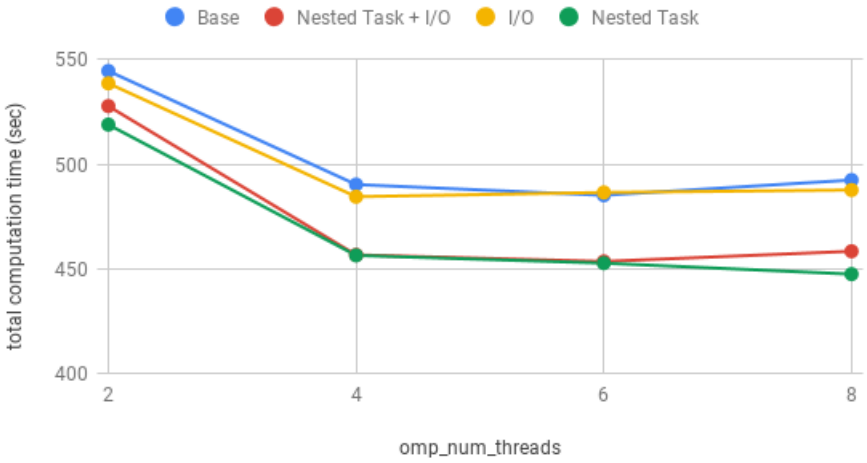
Table 8: Efficiency Metrics of ***dgmres1mt*** with buffered I/O using bend dataset on 100 increments

| | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| Global Efficiency | 81% (+1%) | 87% (+1%) | 87% (-1%) | 86% (-2%) |
| Parallel Efficiency | 81% (+1%) | 76% | 75% | 73% (-2%) |
| Load Balance | 82% (+1%) | 77% | 76% | 74% (+2%) |
| Communication Efficiency | 99% | 99% | 99% | 99% |
| Serial Efficiency | 99% | 99% | 99% | 99% |
| Transfer Efficiency | 100% | 100% | 100% | 100% |
| Computational scalability | 100% | 114% (+2%) | 116% (-1%) | 118% |
| IPC scalability | 100% | 107% (+2%) | 107% (-1%) | 108% |
| Instruction scalability | 100% | 106% | 108% | 109% |

Table 9: Efficiency metrics of ***dgmres1mt*** with conditional nested taskloop approach and buffered I/O using bend dataset on INCF=100 and FREQUENCYF=100

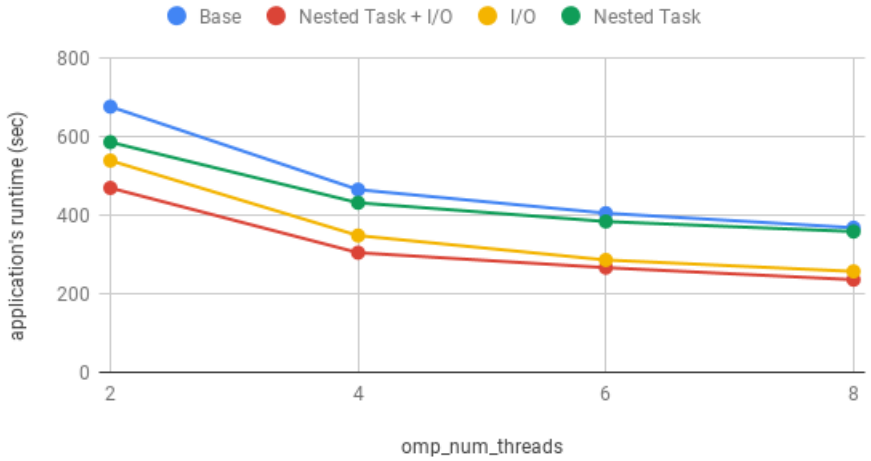
| | 2 | 4 | 6 | 8 |
|---|---|---|---|---|
| Global Efficiency | 91% (+11%) | 103% (+18%) | 111% (+23%) | 112% (+24%) |
| Parallel Efficiency | 91% (+11%) | 87% (+11%) | 88% (+13%) | 90% (+15%) |
| Load Balance | 99% (+18%) | 94% (+17%) | 93% (+17%) | 95% (+19%) |
| Communication Efficiency | 92% (-7%) | 93% (-6%) | 95% (-4%) | 95% (-4%) |
| Serial Efficiency | 92% (-7%) | 93% (-6%) | 95% (-4%) | 95% (-4%) |
| Transfer Efficiency | 100% | 100% | 100% | 100% |
| Computational scalability | 100% | 118% (+6%) | 125% (+9%) | 124% (+6%) |
| IPC scalability | 100% | 105% (-1%) | 106% (-2%) | 105% (-3%) |
| Instruction scalability | 100% | 113% (+6%) | 118% (+10%) | 118% (+9%) |

Figure 5: Comparison of total computation time inside ***dgmres1mt*** function




From figure 5, we see that the total computation time inside ***dgmres1mt*** changes because of the introduction of the nested taskloops. The buffered I/O does not have any impact on decreasing the total computation time even though we have seen that using buffered I/O can improve total runtime of the application. When we combine both improvements, we can see that not

Figure 6: Runtime comparison between improvements




only there is a huge improvement for the Global Efficiency, we also see an even bigger runtime improvement (figure 6). In conclusion, both improvements are necessary for the application's performance.

## 2.5 Conclusion

In a previous performance assessment of CODE_A we identified a significant load imbalance and also an inefficient use of I/O operations in the application code. Using 8 OpenMP threads the load balance was only 72% and I/O operations accounted for 40% of the execution time. The load imbalance was related to the implementation of a parallel GMRES method, where individual subsystems with different convergence are solved by each thread. In this PoC activity we implemented three different approaches in a miniapp, which extracted the parallel GMRES implementation, to improve the load balance of CODE_A. While a plain tasking approach did not result in a performance improvement, conditionally creating nested tasks or nested parallel regions when a load imbalance is detected improved the load balance to 90%. By implementing buffered I/O operations we obtained a speedup of 1.43x for the whole CODE_A application.

## 2.6 Lessons Learned

### 2.6.1 Recommendations for tool developers

### 2.6.2 Recommendations for programming model developers

**Lessons learned** In FORTRAN I/O operations such as read and write are not buffered by default. In order to write simulation results to a file CODE_A iterates over each finite element of the simulation mesh and calls a separate write operation for each physical quantity. As a result data is written in very small chunks that have the size of a one to three floating point values, which is a very inefficient use of the file system. We recommend developers of the FORTRAN programming language to automatically open files in a buffered way.

# 3 Report on Activity CODE_B

## 3.1 Description of the Application

CODE_B is a collection of codes developed at Forschungszentrum Jülich implementing the Korringa-Kohn-Rostoker (KKR) Green's function method to perform density functional theory calculations. In this PoC activity we focus on one of these codes which is the *KKRhost* program.

## 3.2 Previous Assessments and Recommendations

Prior to this PoC activity we conducted a performance assessment of the CODE_B code that can be found under *POP_AR_079*. The application implements a hybrid parallelization using MPI and OpenMP. MPI processes are logically arranged in a two-dimensional grid. One dimension represents different atoms in the input while the other dimension represents different energy discretization points.

| Threads per Process | 1 | 2 | 4 | 8 | 12 |
|---|---|---|---|---|---|
| Global Efficiency | 0.94 | 0.64 | 0.40 | 0.19 | 0.13 |
| ↪ Parallel Efficiency | 0.94 | 0.76 | 0.59 | 0.44 | 0.39 |
| ↪ Process Level Efficiency | 0.94 | 0.93 | 0.91 | 0.88 | 0.94 |
| ↪ Load balance | 0.97 | 0.97 | 0.95 | 0.94 | 0.98 |
| ↪ MPI Communication Efficiency | 0.97 | 0.96 | 0.96 | 0.94 | 0.96 |
| ↪ MPI Transfer Efficiency | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| ↪ MPI Serialisation Efficiency | 0.97 | 0.96 | 0.96 | 0.94 | 0.96 |
| ↪ Thread Level Efficiency | 1.00 | 0.83 | 0.68 | 0.56 | 0.45 |
| ↪ OpenMP Region Efficiency | 1.00 | 0.98 | 0.98 | 0.97 | 0.92 |
| ↪ Serial Region Efficiency | 1.00 | 0.85 | 0.70 | 0.59 | 0.52 |
| ↪ Computational Scaling | 1.00 | 0.84 | 0.67 | 0.44 | 0.34 |
| ↪ Instruction Scaling | 1.00 | 0.97 | 0.94 | 0.90 | 0.86 |
| ↪ IPC Scaling | 1.00 | 0.96 | 0.88 | 0.74 | 0.67 |

Figure 7: POP additive hybrid efficiency metrics for CODE_B

Figure 7 shows the POP efficiency metrics obtained by this assessment using the additive hybrid model. On the process level we do not see any issue as the corresponding child metrics report nearly optimal values. However, on the thread level we see a significant drop in the *Serial Region Efficiency*. This is mostly caused by parts of the application which are not parallelized with OpenMP (yet). Moreover, we see a very drastic decrease of the *Computational Scaling* which is mainly driven by a decreasing IPC.
Hence, we conducted a more detailed analysis of the IPC behavior of the code by collecting data from hardware performance counter related to cache hits and misses. For the most important hotspot (i.e. `main1b`) of the code the results are shown in Figure 8. The top graph shows how the IPC evolves when increasing the OpenMP parallelism inside each MPI rank from one to twelve OpenMP threads. Overall the number of cycles increases faster than the number of instructions so that we see a critical drop of the IPC down to 0.65 when using twelve OpenMP threads. The bottom left graph of Figure 8 shows that the miss ratio in the L3-cache is less than 1 %. However, the bottom right graph shows a miss ratio between roughly 32 % and 38 %

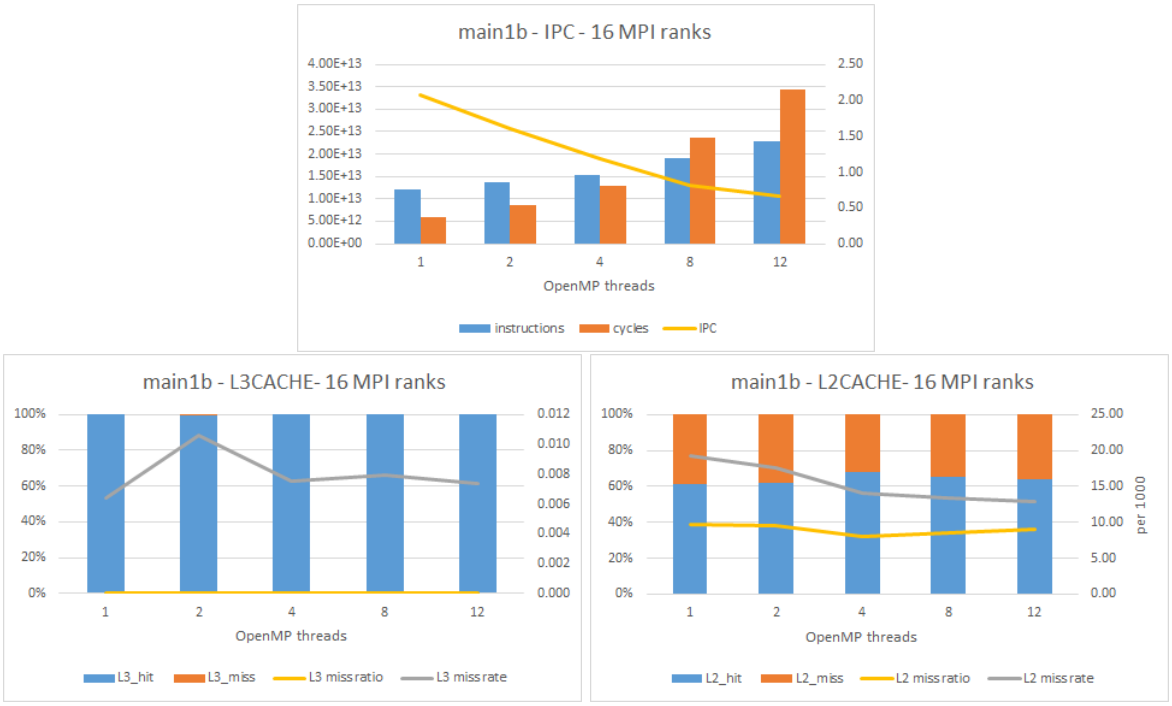



Figure 8: IPC behavior and cache hardware counters for the hotspot code region `main1b`.

in the L2-cache. Thus, we conclude that the low IPC is probably caused by an inefficient use of the caches.
We then went back and thought about the algorithmic structure of the code. In the hotspot region (*main1b*) of the code the algebraic Dyson equation is solved for each energy point $E$. In order to obtain the solution an integration over all *k-points* needs to be performed:

$$\frac{1}{V_{\mathrm{BZ}}} \sum_{\mathbf{k}} w^k \underbrace{\left[ \left(\Delta t^{\mu}(E)\right)^{-1} \delta_{\mu\mu'} - G^{\mathrm{ref},\mu\mu'}(\mathbf{k};E) \right]^{-1}}_{=\tau^{\mu\mu'}(\mathbf{k};E)} e^{-i\mathbf{k}\cdot(\mathbf{R_i}-\mathbf{R_{i'}})} \tag{1}$$

Here $\tau^{\mu\mu'}(\mathbf{k};E)$ is the scattering path operator for k-point $\mathbf{k}$ and energy point $E$, $V_{\mathrm{BZ}}$ is the volume of the cell corresponding to the k-point $\mathbf{k}$, $w^k$ is the integration weight and $e^{-i\mathbf{k}\cdot(\mathbf{R}_i-\mathbf{R}_{i'})}$ is the exponential factor required to Fourier transform the scattering path operator.

**Algorithm 1** k-point integration

1: $s \leftarrow 0$
2: **for** $k = 0, \ldots, kpts$ **do**
3: $\quad c_k \leftarrow \frac{1}{V_{\mathrm{BZ}}} w^k e^{-i\mathbf{k}\cdot(\mathbf{R_i}-\mathbf{R_{i'}})}$
4: $\quad \mathbf{M}_k \leftarrow \left[ \left(\mathbf{\Delta t}^{\mu}(E)\right)^{-1} \delta_{\mu\mu'} - \mathbf{G}^{\mathrm{ref},\mu\mu'}(\mathbf{k};E) \right]$
5: $\quad \tau_k = \mathbf{M}_k^{-1}$
6: $\quad \mathbf{S} \leftarrow \mathbf{S} + c_k \cdot \tau_k$
7: **end for**

The implementation of this k-point integration is schematically shown in Algorithm 1. First the exponential prefactor $c_k$ is constructed. The computation of the matrix $M_k$ involves an inverse Fourier transformation of $G^{ref}$ to get $G^{ref,\mu\mu'}$. Afterwards the inversion of $M_k$ is done

by computing an LU decomposition using the LAPACK routine `zgetrf`. For this purpose the CODE_B code uses the threaded version of Intel's Math Kernel Library (MKL) which implements the LAPACK routines as well.
The reason for the inefficient cache usage is the parallelization of the k-point integration as described above. All the k-points are distributed among the MPI processes. So the for-loop in line 2 of Algorithm 1 gets executed serially by each MPI rank. OpenMP parallelism is only triggered inside each MPI rank to compute the LU factorization of each $M_k$ in parallel. This results in a very long sequence of very fine-grained OpenMP parallel regions. The matrices $M_k$ are usually very small while the number of k-points is very large. In our testcase from production 40 k-points are used per dimension, which results in a total of 64000 k-points for a 3D simulation. The size of the matrices $M_k$ is given by $L_{\max} = s \cdot (l_{\max} + 1)^2$. In our case the typical choices $l_{\max} = 3$ and $s = 2$ are made, so that each matrix $M_k$ is of size $32 \times 32$. Moreover, our testcase uses 24 energy points $E$. We run the testcase on 16 MPI ranks with a varying number of OpenMP threads per rank from 1 to 12 threads. All in all this means that a total of 1 536 000 $M_k$ matrices needs to be processed, which means that each rank has to process 96 000 matrices. This further implies that there are 96 000 OpenMP parallel regions executed one after the other where up to 12 threads work together on the LU decomposition of a $32 \times 32$ matrix. On the one hand this explains why this part of the code does not scale particularly well with an increasing number of threads. With 12 threads we only yield a speedup of 1.36 with an efficiency of 11.33 %. On the other hand it may also explain why the IPC decreases so drastically as many threads on different cores work together on very small problems that would easily fit into the L1-cache of a single core. Based on these results our recommendation is to move the OpenMP parallelism from line 5 to line 2 in Algorithm 1, so that the k-points, which are already distributed on the MPI level, will also be distributed on the OpenMP level.

## 3.3 PoC activities

### 3.3.1 Scope

Our PoC activity will address the current parallelization of the k-point integration in the hotspot region *main1b* of the code. We will move the OpenMP parallelization level from the individual LU decompositions (line 5) up to the k-point loop (line 2) in Algorithm 1.

**Use-case and evaluation metrics** For the purpose of this PoC activity the customer will provide a miniapp which extracts exactly the k-point integration part from the whole application. So we will use this miniapp for the implementation and evaluation of our PoC activity.
We will use the following performance metrics to monitor our progress and possibly success of our PoC activity:

- **wall-time**: without any further optimizations we expect the refactored code to reach a speedup of at least 6 when executed with 12 OpenMP threads (and 16 MPI ranks)

- **Serial Region Efficiency**: we expect the Serial Region Efficiency to improve as more code will be executed in parallel

- **IPC Scaling**: we also expect a more optimal IPC scaling since each thread can reuse the LU decomposition of the matrix $M_k$ when computing the full integrand.

### 3.3.2 Implementation

**OpenMP code refactoring in k-loop miniapp** The k-loop miniapp performs the k-point integration as a standalone application. It contains two loops that each perform a k-point integration. The first one is rather short while the second one is longer. In this report we will focus on the longer loop to describe our modifications but they are very similar to the first loop.

The reference code of the k-point integration loop is shown in Figure 9. First of all the

```
1  call read_input_file('input_kloop_ept_2.txt', nofks, bzkp, volcub)
2
3  allocate(gs_k(lmmaxd*(1+kBdG), lmmaxd*(1+kBdG), nsymat, nshell))
4  allocate(gs(lmmaxd*(1+kBdG), lmmaxd*(1+kBdG), nsymat, nshell))
5  call allocate_BZintegrand_local(1, lly, alm, almgf0, use_virtual_atoms)
6
7  gs = czero
8  lly_grtr = czero
9
10 do kpt = 1, nofks
11   call get_integrand(bzkp(1:3, kpt), volcub(kpt), gs_k, lly_grtr_k)
12   trace = czero
13   do lm1=1, lmmaxd*(1+kBdG)
14     trace = trace + gs_k(lm1, lm1, 1, 1)
15   end do
16   if (.not.noout) write(9999,'(6ES16.7)') bzkp(1:3, kpt), volcub(kpt), trace
17   gs(:,:,:,:) = gs(:,:,:,:) + gs_k(:,:,:,:)
18   if (lly/=0) lly_grtr = lly_grtr + lly_grtr_k
19 end do ! kpt = 1, nofks
```

Figure 9: Reference code of the k-point integration loop

application reads in some input data (line 1) like the mesh of k-points (`bzkp`) and allocates some data structures (lines 3-5) to store the integrand of the current k-point (`gs_k`) and the sum of all integrands (`gs`). The structure of the k-point loop is quite simple (lines 10-18). First it computes the scattering path operator $\tau$ as the integrand described by equation 1 (line 11). Then it computes the trace of the scattering path operator (lines 12-15). Afterwards it writes the trace for each k-point to an output file (line 16) and accumulates the integrand of the current iteration to the total integral sum (line 17).

Recall our recommendation from the previous section. The goal for this PoC acitivity is to move the very fine grained OpenMP parallelism hidden behind the routine `get_integrand` (line 11, Fig 9) to the k-point loop (line 10). Our modifications to the code are highlighted in red in Figure 10. We enclosed the whole k-point integration loop in an OpenMP parallel region. Lines 9 to 15 show the necessary data scoping clauses. When using the OpenMP worksharing construct `!$omp do` for the k-point integration loop (lines 24-35) each thread will compute a set of integrands for some part of the k-point mesh, which need to be accumulated in the end. This would usually require an OpenMP `reduction(+:gs)` clause on the sum of all integrands `gs`. However, the Intel Fortran Compiler version 19.0.1.144 20181018 that we use for this PoC activity does not seem to support array reductions because it crashes with an internal compiler error pointing to the line in the code that contains the reduction clause.

Hence, we used a workaround and implemented a simple array reduction ourselves. Therefor we introduced some `threadprivate` variables for each thread by allocating them inside the parallel region (line 17-19), namely `gs_k` and `gs_priv` and a lot of variables that are hidden behind the routine `allocate_BZintegrand_local`. Similar to the reference version of the code each thread will store the integrand for the current k-point in `gs_k` (line 26) but will then

```
1  call read_input_file('input_kloop_ept_2.txt', nofks, bzkp, volcub)
2
3  allocate(gs(lmmaxd*(1+kBdG), lmmaxd*(1+kBdG), nsymat, nshell))
4  allocate(trace_per_kpt(nofks))
5
6  gs = czero
7  lly_grtr = czero
8
9  !$omp parallel &
10 !$omp shared(lmmaxd, kBdG, nsymat, nshell, nofks) &
11 !$omp shared(bzkp, volcub, lly, gs, trace_per_kpt, noout) &
12 !$omp private(kpt, trace, lm1) &
13 !$omp private(gs_k, gs_priv, lly_grtr_k) &
14 !$omp reduction(+:lly_grtr) &
15 !$omp default(none)
16
17 allocate(gs_k(lmmaxd*(1+kBdG), lmmaxd*(1+kBdG), nsymat, nshell))
18 allocate(gs_priv(lmmaxd*(1+kBdG), lmmaxd*(1+kBdG), nsymat, nshell))
19 call allocate_BZintegrand_local(1, lly, alm, almgf0, use_virtual_atoms)
20
21 gs_k = czero
22 gs_priv = czero
23
24 !$omp do
25 do kpt = 1, nofks
26   call get_integrand(bzkp(1:3, kpt), volcub(kpt), gs_k, lly_grtr_k)
27   trace = czero
28   do lm1=1, lmmaxd*(1+kBdG)
29     trace = trace + gs_k(lm1, lm1, 1, 1)
30   end do
31   trace_per_kpt(kpt) = trace
32   gs_priv(:,:,:,:) = gs_priv(:,:,:,:) + gs_k(:,:,:,:)
33   if (lly/=0) lly_grtr = lly_grtr + lly_grtr_k
34 end do ! kpt = 1, nofks
35 !$omp end do
36
37 !$omp critical
38 gs(:,:,:,:) = gs(:,:,:,:) + gs_priv(:,:,:,:)
39 !$omp end critical
40 !$omp end parallel
41
42 if (.not.noout) then
43   do kpt = 1, nofks
44     write(9999,'(6ES16.7)') bzkp(1:3, kpt), volcub(kpt), trace_per_kpt(kpt)
45   end do
46 endif
```

Figure 10: Modified code of the k-point integration loop

accumulate a partial sum of all its computed integrands in `gs_priv` (line 32). After all k-points have been processed the partial sums of integrands computed by each thread will be accumulated to a global sum using an OpenMP `critical` region (lines 37-39).

Moreover, we made a change to the output writing of the computed traces for each k-point. In the reference version the k-point integration loop was executed serially and so writing the computed trace values to file also happened serially. In order to avoid multiple threads writing to the same file after parallelizing the k-point loop in this PoC activity we introduced a buffer `trace_per_kpt` (line 4) which stores the computed trace per each k-point. After all k-points

have been processed a serial loop iterates over this buffer and writes the computed traces to an output file (lines 42-45).

Finally, in order to verify the correctness of the code after our modifications were made we compared the sum of all integrands stored in `gs` by computing a relative error elementwise. The relative errors range from $6.64 \times 10^{-17}$ to $2.52 \times 10^{-13}$ which are small enough for us to assume that our modified code still produces the correct results.

## 3.4 Results

We will now evaluate the performance improvements yielded by our modifications made to the code. Therefore we investigate the scalability of the code as well as the POP efficiency metrics.

### 3.4.1 Scalability

Figure 11 shows a comparison of the scalability between the reference implementation of the k-point integration and our modified code for an increasing number of OpenMP threads (x-axis). The bars plots show the runtime of the miniapp measured as wall-time in seconds (left y-axis). The line plots show the efficiency computed as speedup over number of OpenMP threads (right y-axis). For the reference code the blue bars and the grey line illustrate that scalability is really bad. The reference implementation scales only up to 8 threads with a very poor efficiency of roughly 15 %. With 12 threads the miniapp gets slower again. The behavior of the miniapp shows exactly the same scalability behavior as the `main1b` part of the whole juKKR-KKRhost application that we investigated prior to this PoC activity.

After our modifications to the OpenMP parallelism we see a much better scalability of the code. The orange bars show that the runtime now decreases significantly with an increasing number of threads. Furthermore the miniapp now scales to 12 OpenMP threads as well and yields an efficiency of 51 %. This means a speedup of more than 6x is achieved which is one of our goals that we defined at the beginning of the PoC activity.

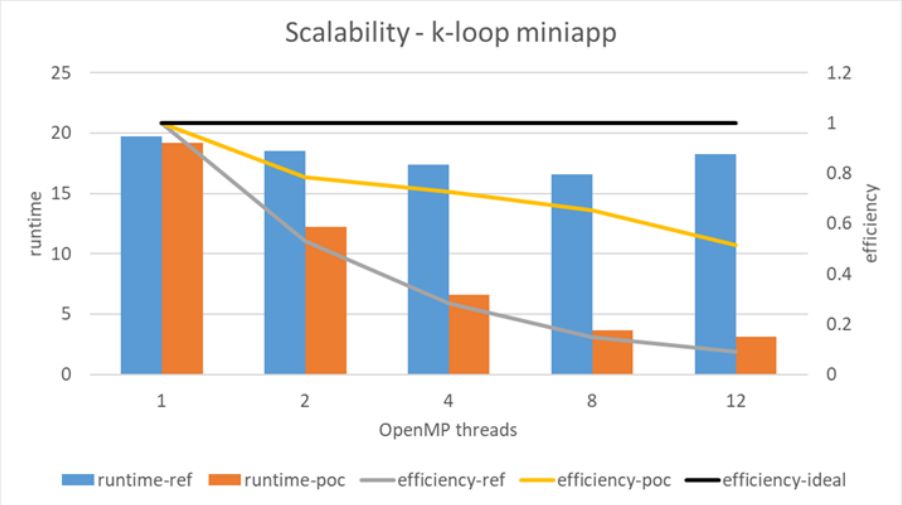



Figure 11: Comparison of the scalability between the reference and our modified code

### 3.4.2 POP Efficiencies

A comparison of the POP efficiency metrics is helpful to understand why the scalability of the code is much better after we implemented our suggested code modifications. Table 10

| Threads per Process | 1 | 2 | 4 | 8 | 12 |
|---|---|---|---|---|---|
| Global Efficiency | 0.99 | 0.53 | 0.28 | 0.14 | 0.09 |
| Parallel Efficiency | 0.99 | 0.62 | 0.39 | 0.26 | 0.21 |
| Process Level Efficiency | 0.99 | 0.98 | 0.98 | 0.98 | 0.98 |
| MPI Load balance | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| MPI Communication Efficiency | 0.99 | 0.98 | 0.98 | 0.98 | 0.98 |
| Thread Level Efficiency | 1.00 | 0.64 | 0.40 | 0.27 | 0.23 |
| OpenMP Region Efficiency | 1.00 | 0.95 | 0.95 | 0.95 | 0.94 |
| Serial Region Efficiency | 1.00 | 0.69 | 0.45 | 0.32 | 0.29 |
| Computational Scaling | 1.00 | 0.85 | 0.72 | 0.57 | 0.42 |
| Instruction Scaling | 1.00 | 0.98 | 0.97 | 0.93 | 0.90 |
| IPC Scaling | 1.00 | 0.93 | 0.86 | 0.76 | 0.65 |
| Frequency Scaling | 1.00 | 0.93 | 0.86 | 0.81 | 0.71 |

Table 10: POP additive hybrid metrics for the reference k-loop miniapp code

shows the additive hybrid POP metrics that we obtained on the reference implementation of the k-point integration. These metrics are very similar to the ones we obtained for the whole juKKR-KKRhost application in the performance assessment prior to this PoC activity. They point to the same performance issues that we identified in our performance assessment, namely a poor *Serial Region Efficiency* as well as significantly decreasing *IPC Scaling*. Again this shows that the k-point integration miniapp represents the original of the hotspot region `main1b` of the whole application very well.

| Threads per Process | 1 | 2 | 4 | 8 | 12 |
|---|---|---|---|---|---|
| Global Efficiency | 0.99 | 0.78 | 0.73 | 0.65 | 0.51 |
| Parallel Efficiency | 0.99 | 0.89 | 0.89 | 0.89 | 0.83 |
| Process Level Efficiency | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 |
| MPI Load balance | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| MPI Communication Efficiency | 0.99 | 0.99 | 0.99 | 0.99 | 0.98 |
| Thread Level Efficiency | 1.00 | 0.89 | 0.89 | 0.89 | 0.83 |
| OpenMP Region Efficiency | 1.00 | 0.99 | 0.99 | 0.99 | 0.99 |
| Serial Region Efficiency | 1.00 | 0.89 | 0.89 | 0.90 | 0.83 |
| Computational Scaling | 1.00 | 0.89 | 0.82 | 0.73 | 0.62 |
| Instruction Scaling | 1.00 | 1.00 | 1.00 | 1.00 | 0.99 |
| IPC Scaling | 1.00 | 1.00 | 1.02 | 1.02 | 1.01 |
| Frequency Scaling | 1.00 | 0.88 | 0.80 | 0.71 | 0.61 |

Table 11: POP additive hybrid metrics for modified k-loop miniapp code

After we implemented our proposed code modifications to the miniapp we obtained the efficiency metrics shown in Table 11. A first visual impression of the data already shows that both the *Serial Region efficiency* and the *IPC scaling* have improved significantly. The *Serial Region Efficiency* decreased down to 29 % in the reference implementation of the k-point integration when scaling up to 12 OpenMP threads. With our modifications to the code we improved the *Serial Region Efficiency* by 54 % yielding an efficiency value of 83 % with 12 OpenMP threads now. Similarly, the *IPC scaling* went down 65 % in the reference code. After we implemented our code modifications we now yield a perfect *IPC scaling* and even a small superlinear scaling of up to 102 % is yielded when using 4 or more threads. The reason for this is probably the fact

that the $32 \times 32$ sized matrices corresponding to the scattering path operators for which an LU decomposition needs to be computed fit into the L1 cache of our machine and are not shared between multiple cores anymore.

The *Global Efficiency* of the miniapp also increased from 9 % to 51 % when using all 12 OpenMP threads. This perfectly matches with the efficiency of 51 in terms of speedup reported earlier.

## 3.5 Conclusion

In the performance assessment *POP_AR_079* of the CODE_B application prior to this PoC activity we identified a very poor scalability caused by significant serial parts in the application and an decreasing *IPC scaling* which could be related to a very fine-grained OpenMP parallelism. The very fine-grained OpenMP parallelism came from a long sequence of small $32 \times 32$ matrices for each of which an LU decomposition was computed in parallel involving all available threads on each MPI rank. In this PoC activity we showed how we improved the OpenMP parallelisation of the hotspot region `main1b` based on a miniapp that resembles the k-point integration loop. We moved the OpenMP parallelism from the sequence of very small parallel LU decompositions to a higher level in the call path and parallelized the loop over all k-points instead. With this modification to the code we improved the scalability of the miniapp significantly. While the reference implementation only yielded a speedup of 1.19x at best our modified version of the code yields a speedup of 6.16x now and scales to one full sub-NUMA domain of our nodes as well. Moreover with our modifications to the code we additionally parallelized more code than before which results in an improvement of the *Serial Region Efficiency* by 54 % from 29 % in the reference code to 83 % in our modified code. Finally the *IPC scaling* is now perfect with a small superlinear scaling of up to 102 % when using 4 or more OpenMP threads because each LU decomposition in the sequence is now done by a single thread and the problem is small enough to fit in the L1 cache on our machine. All in all we successfully reached all goals that were set at the beginning of this PoC activity.

## 3.6 Lessons Learned

### 3.6.1 Recommendations for tool developers

The OpenMP parallelism inside the threaded version of the Intel MKL was not observable at first when using ScoreP to instrument the code. Since the code of the Intel MKL is not available for users, ScoreP can also not instrument it. Thus, we used a development branch of ScoreP in which the OMPT interface is implemented to instrument OpenMP parallelism instead of OPARI. We recommend tool developers to use the OMPT interface because codes using 3rd party libraries where the code is not available is a common use case in practice.

### 3.6.2 Recommendations for programming model developers

We had to implement an array reduction manually as a workaround because the Intel Fortran Compiler version 19.0.1.144 20181018 does not seem to support this operation and crashed with an internal error. According to the OpenMP standard array reductions are valid. Our manual implementation certainly has potential for further optimization. Users should not implement such operations themselves but should be able to rely on optimized versions provided by developers of programming models and the corresponding compilers.

# 4 Report on Activity CODE_C

## 4.1 Description of the Application

CODE_C is an open-source software for parallel mesh adaptation of 3D volume meshes. The parallel mesh adaptation algorithm is based on iterative remeshing and repartitioning of the distributed mesh. It uses the Mmg software to perform the sequential remeshing steps.

## 4.2 Previous Assessments and Recommendations

This Proof of Concept follows the analysis done in the Audit POP2_AR_066. The assessment studied the performance metrics of CODE_C using a single, weak scaling input set, ranging from 2 to 256 MPI ranks. During the performance analysis of CODE_C there were detected two main factors that limited the scalability:

- Load Balance.
- Instruction Scalability.

To a lesser degree, the program was also affected by Serialization.

In particular, the Load Balance issue was caused due to an uneven distribution of computation between each MPI process. The problem starts evenly distributed among the ranks. Still, after every iteration step (partition of the mesh), some ranks become more loaded than others, creating a difference in the number of instructions executed.

As the load distribution changes at each iteration and when scaling the number of processes, we proposed using DLB, a Dynamic Load Balance library, to address this problem. Using DLB, the application will be able to dynamically adjust the number of active threads of each MPI rank at run time. That is, processes will be able to reduce the number of active resources while they are idle. In contrast, busier processes will use more of them to finish their work much earlier, boosting the effective load balance.

Regarding the Instruction Scalability, the problem was localized in the communication phase of the iterations. We called that zone *diagonal*, as it exhibited a diagonal communication pattern in the analyzed traces. Each MPI rank exchanged some point-to-point MPI messages with all the ranks, but in reality, ranks only needed to communicate with a subset of ranks. Consequently, the instructions executed at that part of the program were increasing by a factor of 8.

To solve that, we proposed to develop a new MPI communication pattern. The key point is to avoid strategies that grow quadratically (or higher) with the number of MPI ranks since it is directly related to the increase of instructions.

## 4.3 PoC activities

### 4.3.1 Scope

The following recommendations were addressed:

- MPI communication pattern that avoids quadratic growth
- Use Dynamic Load Balancing library

## 4.4 Change MPI communication pattern that avoid quadratic growth

The developers applied the suggestion of changing the communication pattern to avoid the quadratic growth of instructions and communications.

The new version shows a much better Instruction Scalability and Communication efficiency as can be seen comparing Figure 12 and Figure 13. From now on, we will call this version "Improved."

| MN4 - MPI | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Number of processes** | 2 MPIs | 4 MPIs | 8 MPIs | 16 MPIs | 32 MPIs | 64 MPIs | 128 MPIs | 256 MPIs |
| **Global efficiency** | 96.72% | 89.28% | 81.36% | 73.83% | 59.75% | 48.55% | 37.71% | 22.06% |
| ├─ **Parallel efficiency** | 96.72% | 93.21% | 89.38% | 87.46% | 76.58% | 64.12% | 58.10% | 49.39% |
| │ ├─ Load balance | 98.72% | 95.87% | 95.48% | 90.83% | 83.33% | 69.56% | 68.96% | 64.07% |
| │ └─ Communication eff. | 97.98% | 97.23% | 93.61% | 96.30% | 91.91% | 92.17% | 84.25% | 77.08% |
| │ ├─ Serialization eff. | 97.99% | 97.33% | 94.01% | 96.95% | 93.03% | 94.24% | 87.53% | 81.03% |
| │ └─ Transfer eff. | 99.99% | 99.89% | 99.58% | 99.32% | 98.79% | 97.80% | 96.25% | 95.12% |
| └─ **Comp. scalability** | 100.00% | 95.79% | 91.02% | 84.41% | 78.02% | 75.72% | 64.91% | 44.67% |
| ├─ IPC scalability | 100.00% | 95.55% | 91.43% | 85.00% | 80.15% | 83.15% | 90.61% | 117.54% |
| ├─ Instruction scalability | 100.00% | 100.30% | 99.61% | 99.35% | 97.45% | 91.29% | 71.87% | 38.06% |
| └─ Frequency scalability | 100.00% | 99.95% | 99.94% | 99.96% | 99.90% | 99.77% | 99.68% | 99.86% |

Figure 12: POP efficiency metrics of original code

| MN4 - INRIA Improved | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Number of processes** | 2 MPIs | 4 MPIs | 8 MPIs | 16 MPIs | 32 MPIs | 64 MPIs | 128 MPIs | 256 MPIs |
| **Global efficiency** | 95.67% | 88.19% | 79.14% | 71.31% | 60.51% | 49.60% | 46.49% | 43.12% |
| ├─ **Parallel efficiency** | 95.67% | 91.88% | 87.61% | 86.40% | 80.21% | 66.09% | 65.50% | 61.73% |
| │ ├─ Load balance | 98.36% | 92.97% | 92.40% | 90.98% | 91.39% | 72.78% | 75.63% | 65.76% |
| │ └─ Communication eff. | 97.27% | 98.82% | 94.81% | 94.97% | 87.77% | 90.82% | 86.61% | 93.87% |
| │ ├─ Serialization eff. | 97.30% | 98.89% | nan | nan | 88.10% | 91.28% | 87.06% | 94.70% |
| │ └─ Transfer eff. | 99.97% | 99.94% | nan | nan | 99.62% | 99.49% | 99.48% | 99.13% |
| └─ **Comp. scalability** | 100.00% | 95.99% | 90.33% | 82.35% | 75.44% | 75.04% | 70.98% | 69.86% |
| ├─ IPC scalability | 100.00% | 98.20% | 95.12% | 88.35% | 82.71% | 83.75% | 80.44% | 81.26% |
| ├─ Instruction scalability | 100.00% | 97.75% | 94.99% | 93.40% | 91.25% | 89.82% | 88.48% | 86.41% |
| └─ Frequency scalability | 100.00% | 99.99 | 99.98% | 100.02% | 99.96% | 99.82% | 99.73% | 99.50% |

Figure 13: POP efficiency metrics of Improved version

## 4.5 Dynamic Load balancing

Our goal is to use DLB to lad balance the execution, for that we need a shared memory parallelization of the region that presents load imbalance. We are going to use OmpSs as a shared memory programming model because at the moment of doing this work is the one offering a better integration wit DLB. The first task then, is to annotate the loop with OmpSs

tasks to partition the workload, so DLB can provide additional CPUs from idle processes to help the other ranks with a higher load. To do that, we require that all the iterations are entirely independent.

Initially, the application had control of the total memory used in the meshes. That was done using shared variables, which were accessed several times during the same iteration. The program could limit the total amount of memory used, depending on what the user asked via the execution flag *mesh-size*. Since this part of the code did affect the execution flow due to data races but did not take part in the result calculation, the developers modified the code to get rid of this memory control.

Besides, OmpSs (and OpenMP, too) does not accept some directives inside its clauses, e.g., *break* and *continue*. *Continues* were replaced by a proper *if-else* chain that behaves exactly like the previous *if( ) → continue*. *Breaks* were used to get out of the loop when the program suffered an error. We replaced them with error messages since they should never be executed in a correct run.

### 4.5.1 Loop taskification

Finally, we annotate the loop with OmpsSs tasks as shown in Listing 1. Data sharings are explicitly declared as shared, except for the ones inside the *private* clause. Each iteration accesses different elements based on the value of *i*, so there are no data races when accessing the shared data structures.

Additionally, we added priorities to the tasks. Since the tasks' granularity has high variability, tasks with larger granularity are tagged with higher priority to be executed earlier. Reserving tasks with smaller granularity at the end allows DLB to deliver more tasks to inactive threads at the end of the loop, improving the performance.

```
#ifdef USE_OMPSS
  #pragma omp parallel
  #pragma omp single
#endif
  for(i = 0; i<parmesh->ngrp; ++i){
    int prio = (parmesh->listgrp[i].mesh->ne / 100000);
#ifdef USE_OMPSS
    #pragma omp task default(shared) firstprivate(i) \
    private(mesh, met, field, k, facesData, permNodGlob, ier, is, psl, ieresult
    , kgrp, ksol, warnScotch) \
    priority (prio)
    {
#endif

    ...//Loop code

#ifdef USE_OMPSS
    }//End of the task
#endif
  }
```

Listing 1: Task annotation.

Usually, after compilation with the OmpSs compiler, the program should run without any problem. In this case, as the structures used by the threads consume a considerable amount of memory, we need to export an OmpSs variable:

```
export NX_ARGS+=" --stack-size=32M"
```

Figure 14 shows the execution of OmpSs tasks with 64 MPI ranks. The pink color indicates that a task is being executed, and green flags separate the execution of two different tasks. The load balance pattern is the same as in Figure **??** since we are only using one thread per rank.

Each task corresponds to an iteration of the loop shown in Listing 1.

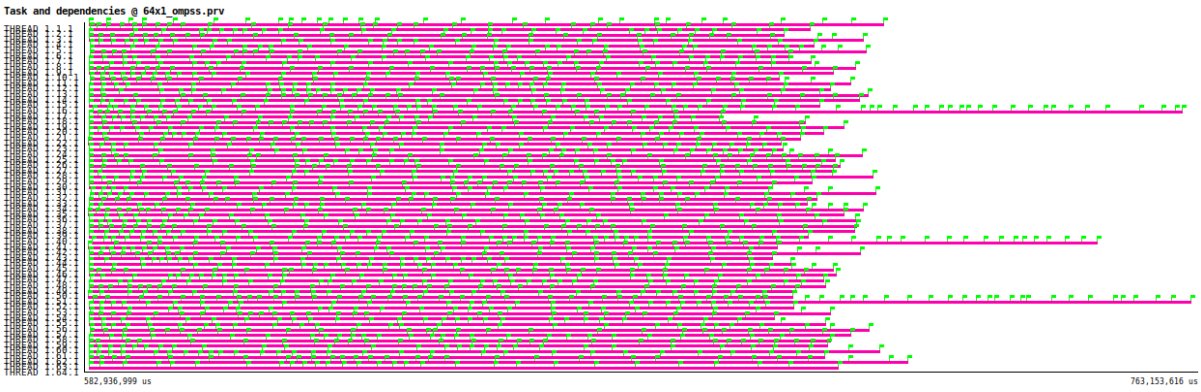



Figure 14: Execution of OmpSs tasks. Iteration 4.

### 4.5.2 Executing with DLB

Now CODE_C can execute OmpSs tasks under the Nanos++ runtime library, so DLB can be incorporated to add malleability to each of the MPI ranks. As DLB is already integrated with Nanos++, we do not need further modifications to the code. The DLB library will be preloaded before executing the application; this way, all the MPI calls will be intercepted by the DLB library and it can modify the number of active threads of the application at any time.

As happened with just OmpSs tasks, there is no need for further modifications of the code to use DLB. However, we must export the following environment variables when launching the application:

```
export LD_PRELOAD+=" $DLB_HOME/lib/libdlb_mpi.so"
export NX_ARGS+=" --stack-size=32M --enable-dlb --enable-block"
export DLB_ARGS+=" --lewi"
export OMP_MAX_TASK_PRIORITY=20
```

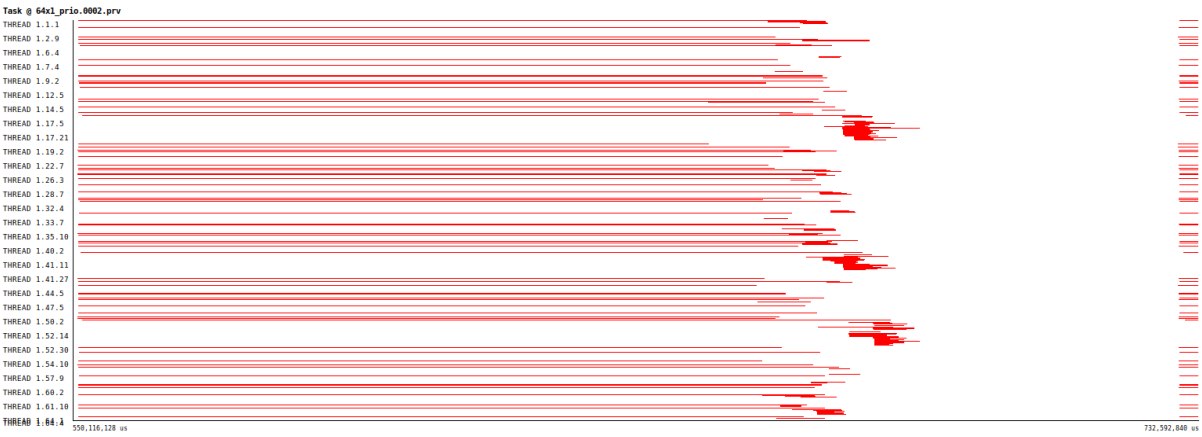



Figure 15: Execution of OmpSs tasks using DLB. Iteration 4.

Figure 15 depicts the execution of OmpSs tasks with DLB for the fourth iteration. In the beginning, there is only one thread active per MPI rank. As some of them reach the barrier, DLB lends their cores to slower processes, so they start using more resources and can finish their execution faster.

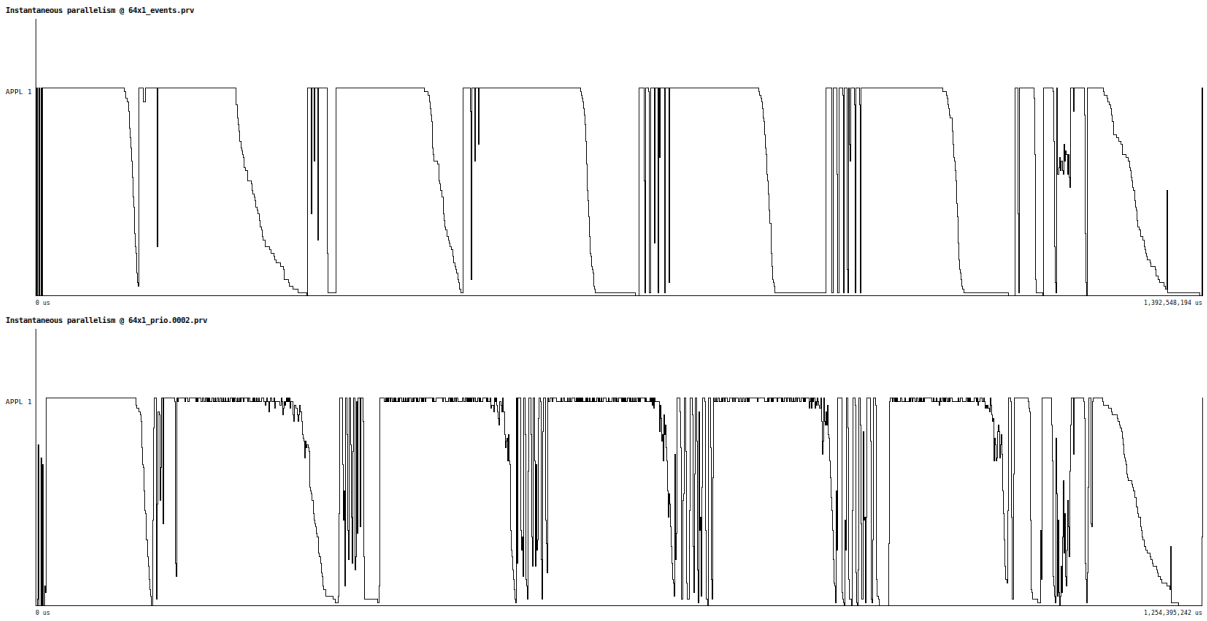



Figure 16: Number of active threads. Top: Original. Bottom: OmpSs with DLB.

Figure 16 shows the instantaneous parallelism for the original code (top) and the taskified code with DLB (bottom). The instantaneous parallelism is defined as the number of active threads doing useful work (not idle or waiting for a message/work) in a given moment of time. The ideal and desirable situation would be to have a constant line at 64 cores, meaning that we are using all the resources during the whole execution. We can observe how the DLB execution can keep the instantaneous parallelism close to 64 during more time than the original version of the code.

### 4.5.3 Fine-tuning the DLB execution

With the proposed changes, we can already run the application with OmpSs and DLB. The application has been annotated with tasks on the PMMG_lib1 subroutine, and the OmpSs runtime is able to increase the number of threads if necessary when DLB detects that other processes are waiting on MPI calls.

DLB intercepts every MPI call in the program in order to manage the number of threads per process. However in our scenario, as CODE_C can only exploit OmpSs parallelism in the taskified loop, intercepting all the MPI calls may only add overhead.

DLB offers an API to fine control were DLB will act, how or give some hints. In our case instead of intercepting all the MPI calls, we will place a simple call to the function DLB_Barrier after the loop shown in Listing 1. This DLB function performs a barrier, similar to an MPI Barrier, but only synchronizing the processes on the node. While these processes are waiting for the others to arrive, DLB can use the CPUs from those to help other MPI processes that have not yet reach the DLB_Barrier. Listing 2 shows the addition of the appropriate call.

```
  ...//Loop code

#ifdef USE_OMPSS
    }//End of the task
#endif
  }//End of the loop
#ifdef USE_DLB
DLB_Barrier();
#endif
MPI_Allreduce(...);
```

Listing 2: DLB_Barrier placement in the code.

As we use the Mercurium compiler for OmpSs in order to compile the code using the DLB API, we just need to add the `--dlb` flag to Mercurium and the library will we properly linked. In case of using an other compiler, the necessary compile and link flags to link with the DLB library should be added.

In order to run with the current version we don't need to preload the DLB library and, we can tell DLB to not intercept MPI calls. In this case, we must export the following environment variables when launching the application:

```
export NX_ARGS+=" --stack-size=32M --enable-dlb --enable-block"
export DLB_ARGS+=" --lewi --lewi-mpi-calls=none"
export OMP_MAX_TASK_PRIORITY=20
```

Figure 17 shows the execution of OmpSs tasks with DLB and the DLB_Barrier for the fourth iteration. Compared to Figure 15, the execution of the next iteration starts soon, as the overhead introduced in the MPI calls between iterations has been removed.

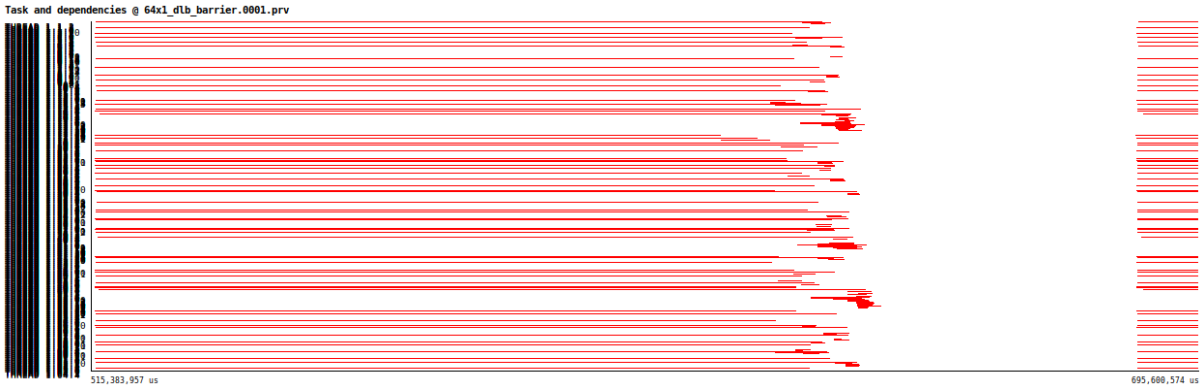



Figure 17: Execution of OmpSs tasks using DLB and DLB_Barrier. Iteration 4.

## 4.6 Results

The taskified loop is invoked six times during the CODE_C execution with the tested input. However, load imbalance appears after the first partition, so only the last five iterations exhibit load balance issues. We have collected the total elapsed time for each MPI rank configuration and performed the arithmetic mean of five independent runs. The evaluation has been performed for the versions discussed previously (Initial and Improved), as well as the Improved compiled with Mercurium (the OmpsSs compiler), and the version with OmpSs' tasks and DLB.

### 4.6.1 Mercurium Compiler Impact

The OmpSs programming model (see Section **??**) is supported by the Mercurium source-to-source compiler. We detected that CODE_C's performance is worse when using Mercurium

instead of the Intel Compilers used before. That occurs even without applying any change to the application, and without any extra compilation flags of Mercurium.

Figure 18 shows the Mercurium's effect in the execution time. For all the different MPI configurations, the application runs slower. It is $1.15x$ times slower at 32 MPI ranks, and $1.05x$ times slower at 256 MPI ranks.

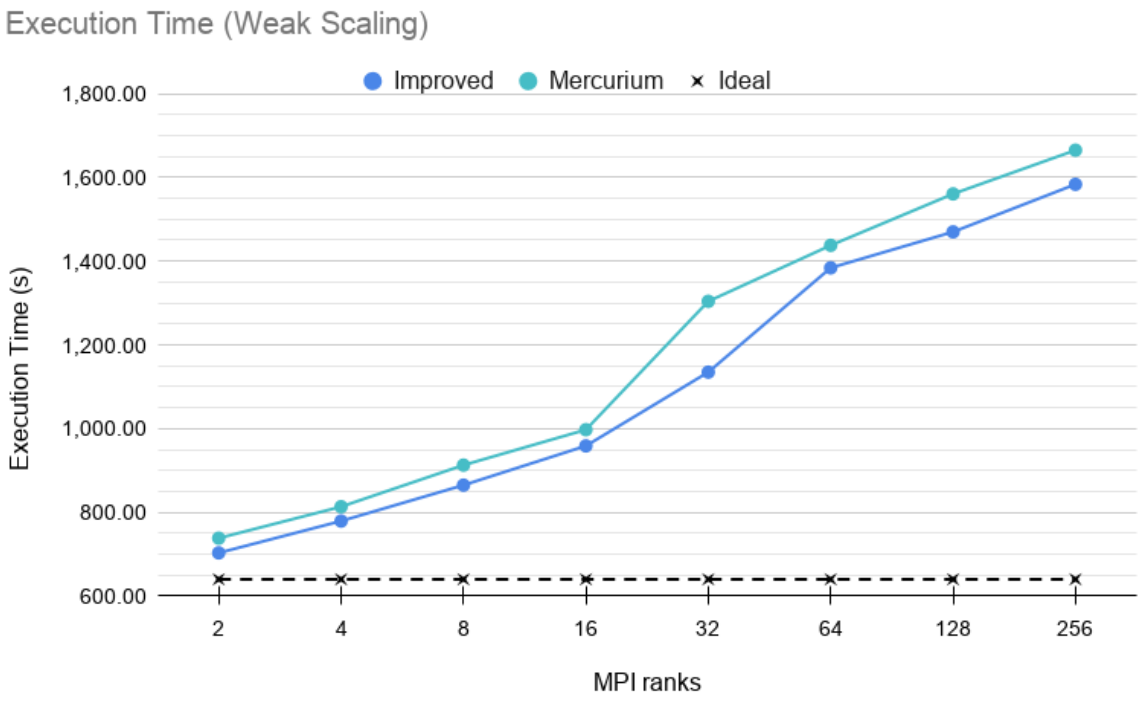



Figure 18: CODE_C elapsed execution time with Mercurium compiler.

### 4.6.2 DLB Evaluation

Figure 19 shows the elapsed time of the different versions of CODE_C, while Figure 20 depicts the speedup. From 2 to 4 MPI ranks, the DLB version runs slightly slower than the Improved, due to the Mercurium slowdown. After that, as the load imbalance is higher, the DLB version outperforms the other ones, achieving $1.15x$ speedup at 256 MPI ranks with respect to the Improved version. After analyzing the program and looking at the chart tendencies, we expect this difference to be even more significant for a higher number of MPI ranks.

## 4.7 Conclusion

During this Proof of Concept CODE_C was modified as necessary to use DLB. These modifications included a parallelization using OmpSs and some code refactoring to make it compliant with the programming model. All the improvements were made inside the PMMG_lib1 subroutine, where the primary source of load imbalance resides. Additionally, we started the Proof of Concept from an improved version of CODE_C, with modifications suggested in the Audit applied by the CODE_C developers.

The code refactor necessary to use DLB within CODE_C includes two `pragmas` to taskify, using OmpSs, the code with load imbalance, and a call to `DLB_Barrier` to enable the resource sharing at that point of the code.

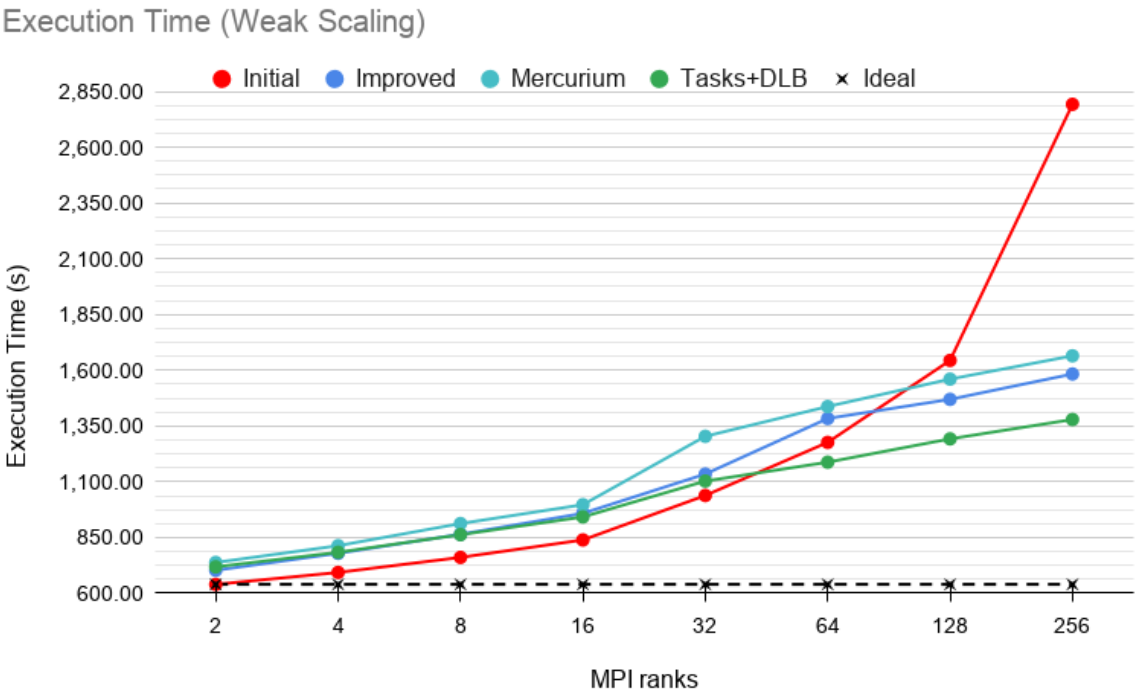



Figure 19: CODE_C elapsed execution time. Weak Scaling from 2 to 256 MPI ranks.

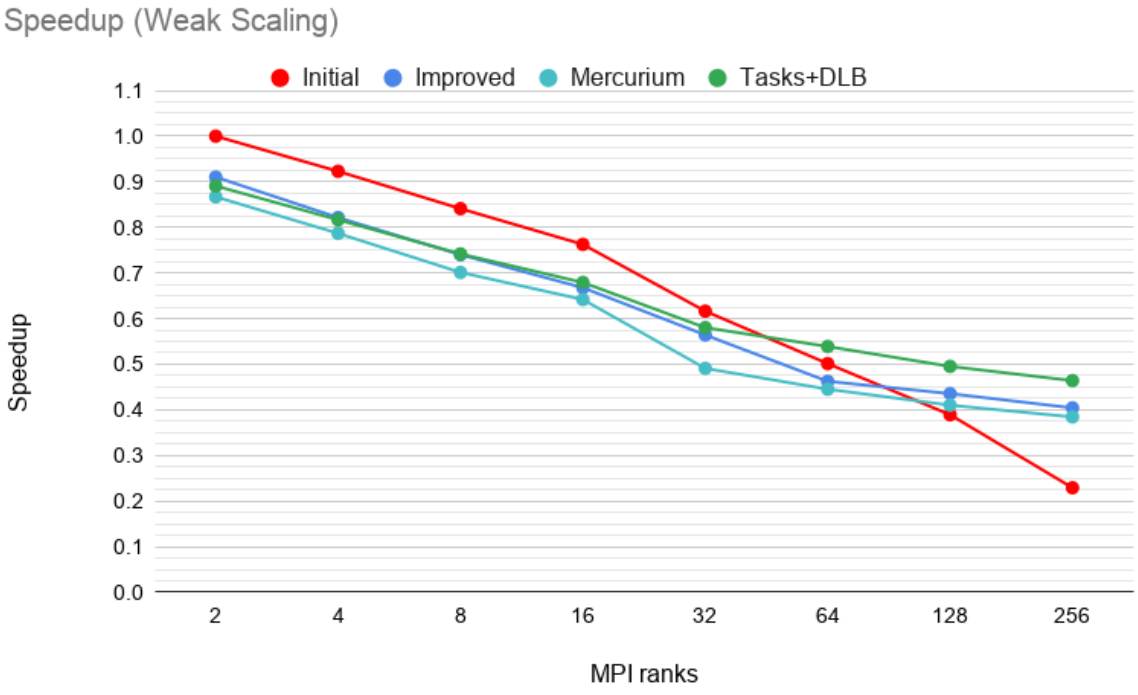



Figure 20: CODE_C speedup. Weak Scaling from 2 to 256 MPI ranks.

When evaluating the *Improved* version with respect to the audited one, we observe that for a low number of MPI processes, the *Improved* version adds an overhead of around 10%. Still, when running with the highest core count, the *Improved* version shows a speedup of $1.76x$.

After applying the changes to the code and before adding DLB, we measure the performance of the code compiled with the Mercurium compiler (necessary to use OmpSs), and we observe an average of 5% overhead added by the compiler.

Finally, we evaluate the performance of the DLB execution; up to 16 threads, the use of DLB has no impact on the performance, as the load imbalance is very low (a 10% in the whole execution as measured by the efficiency metrics). Nevertheless, for more than 32 threads, the use of DLB is able to speedup the execution by almost 20% compared with the execution compiled with Mercurium.

Overall the performance with the highest core count (256) after the changes implemented by the client and the use of DLB is $2x$ faster than the original one.

The conclusion for this PoC is that the use of DLB can help CODE_C address its load imbalance problems when scaling to a high number of cores. However, there is some overhead added by the Mercurium compiler that cannot be avoided when using OmpSs. For this reason, we encourage the developers to port it to an OpenMP+DLB version with LLVM support as soon as it is available.

## 4.8 Lessons Learned

### 4.8.1 Recommendations for tool developers

No special recommendations raised from this PoC for tool developers.

### 4.8.2 Recommendations for programming model developers

In this PoC we used OmpSs because it offers a better integration and thus, performance with DLB. However the use of OmpSs implies the need to compile the code with the Mercurium compiler. From this we observed a reduction in performance. Our suggestion to DLB developers is to improve the integration of DLB with OpenMP, so that the OpenMP standard and the usual compilers can be used (i.e. Intel, GNU or LLVM).

# 5 Report on Activity CODE_D

**Keywords:** OpenMP, Load imbalance, data locality

## 5.1 Description of the Application

CODE_D is an open source, agent-based modeling framework for 3D multicellular simulations widely used in tissue and cancer biology to simulate the effect of genetic and environmental perturbations in cancer progression. It builds upon a multi-substrate diffusion-reactionsolver to link cell phenotype to multiple substrates, such as nutrients and signaling factors. The application is written in C++ and is parallelized with OpenMP using parallel loops with a static schedule. Its computation cost scales linearly with the number of cells.

## 5.2 Previous Assessments and Recommendations

This Proof of Concept follows the analysis done in the Audit POP2_AR_105. The assessment studied the performance metrics of PhysiCell in a range from 1 to 48 cores in a strong scaling scenario.

During the performance analysis of PhysiCell there were detected two main factors that limited scalability:

- Frequency scalability
- IPC scalability
- Load imbalance

Based on them, we recommended the following:

- Change the allocation and deallocation of memory pattern
- Improve data locality to improve IPC
- Address load imbalance

## 5.3 PoC activities

### 5.3.1 Scope

The following recommendations were addressed:

- Use Jemalloc library to show the impact of improving the the allocation and deallocation of memory pattern
- Improve data locality to improve IPC
- Address load imbalance

### 5.3.2 Memory allocation and deallocation

Regarding the issue detected decreasing the frequency we locate a specific region corresponding to the function `update_all_cells` causing scalability problems due to *cycles per µsecond* dropping when application ran with more than 8 threads. We detected this drop was caused by a high number of memory allocations and deallocations performed by the C++ runtime. Our feedback to the developers is that this pattern must be changed, however as a proof-of-concept to show the potential of this change we a address the problem by using a memory management library called Jemalloc, which is intended for efficient memory management in parallel programs.

In figure 21 can be observed the strong scalability comparison of the original version versus the version with Jemalloc integration. From now on, we will call this version "Improved".

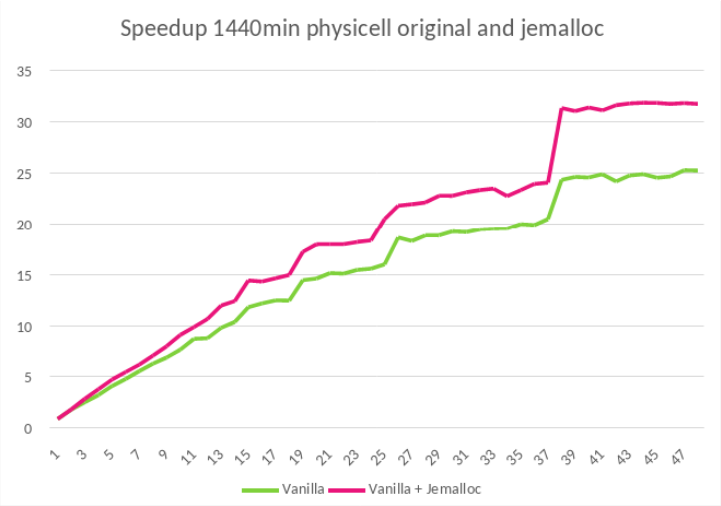



Figure 21: Speedup of original and Improved version, for a 1440 minutes simulation on MareNostrum4 from 1 to 48 threads.

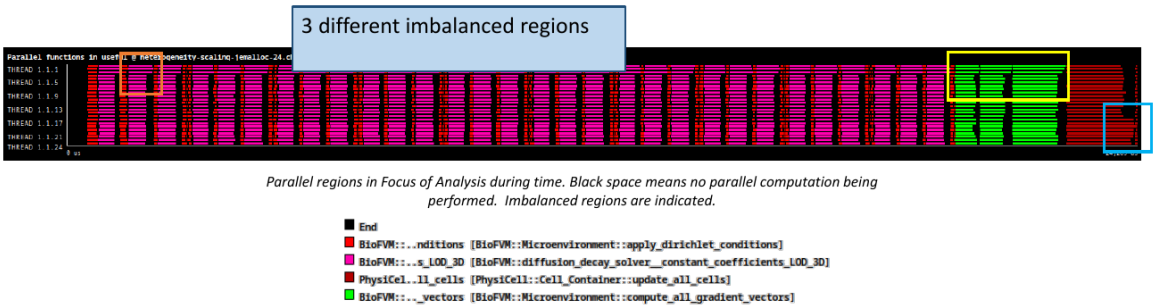
### 5.3.3 Load Imbalance and IPC Scaling

The load imbalance could be identified in 3 different regions, by two different sources. The regions are identified in figure 22. In this figure an execution trace of ourFocus of Analysis can be seen. The colors represent different parallel functions.

Given the analysis, the following observations were outlined regarding the load imbalance regions:

- Imbalance in the solver parallel loop due to the granularity of a voxel.
  - Loop over voxels
  - Other input sets would have higher number of voxels?
- Imbalance in the `update_velocity` parallel loop due to IPC
  - Loop over cells, cells added during the simulation have a lower IPC so they take more time to compute.

To solve/mitigate the previous issues, the following recommendations where proposed:




*Parallel regions in Focus of Analysis during time. Black space means no parallel computation being performed. Imbalanced regions are indicated.*

- *Orange square (left)*: first threads always take longer to complete. Microenvironment solver function.
- *Yellow square (middle)*: first threads always take longer to complete. Compute gradients function.
- *Blue square (right)*: last threads always show larger execution time. Cell's update velocity function.

Figure 22: Execution trace of PhysiCell application with identification of different load imbalance regions.

1. Make number of voxels multiple of number of threads, so iterations distribute evenly and there is no imbalance; or
2. Decrease granularity of parallelism, by parallelising inside the voxel.
3. Use a dynamic scheduling so "difficult" cells get distributed along all threads
4. Distribute workload for memory-aware execution
5. Rearrange cells to solve memory locality problems

In this Proof of Concept we will apply recommendations 2 and 4.

there are two regions where granularity needs to be decreased in order to better distribute workload and reduce load imbalance due to number of instructions. If we have a look at the original code in listing 3, we see that we can still parallelize inside the loop. Indeed, there is a nested loop (line 4) right after the outter loop that iterates through another axis of the domain.

```
1    #pragma omp parallel for
2    for( unsigned int k=0; k < M.mesh.z_coordinates.size() ; k++ )
3    {
4        for( unsigned int j=0; j < M.mesh.y_coordinates.size() ; j++ )
5        {
6            // Thomas solver, x-direction
7
8            // remaining part of forward elimination,
9            // using pre-computed quantities
10           int n = M.voxel_index(0,j,k);
11           (*M.p_density_vectors)[n] /= M.thomas_denomx[0];
12
13           for( unsigned int i=1; i < M.mesh.x_coordinates.size() ; i++ )
14           {
15               n = M.voxel_index(i,j,k);
16               axpy( &(*M.p_density_vectors)[n] , M.thomas_constant1 , (*M.
   p_density_vectors)[n-M.thomas_i_jump] );
17               (*M.p_density_vectors)[n] /= M.thomas_denomx[i];
18           }
19
```

```
            for( int i = M.mesh.x_coordinates.size()-2 ; i >= 0 ; i-- )
            {
                n = M.voxel_index(i,j,k);
                naxpy( &(*M.p_density_vectors)[n] , M.thomas_cx[i] , (*M.
    p_density_vectors)[n+M.thomas_i_jump] );
            }

        }
    }

```

Listing 3: Fragment of the parallel region for the solver implementation. Original code.

Our implementation of the Proof of Concept consists of collapsing the two outter loops, in order to increase the workload from `M.mesh.z_coordinates.size()` to `M.mesh.z_coordinates.size() * M.mesh.y_coordinates.size()`. Grain size is going to be reduced and a higher number of finer-grain chunks are going to be distributed achieving a more balanced workload. To do this, **OpenMP specification provides the `collapse` clause, which allows to parallelize multiple loops in a nest without introducing nested parallelism**.

In listing 4 you can see the final implementation for one of the parallel regions that we can find in the solver function. The inner code has been omitted because it is the same as in the original code and it is not relevant for the optimization. The parameter in the `collapse` clause indicates the level of nested loops that we want to collapse, which is 2 in our case.

```
    #pragma omp parallel for collapse(2)
    for( unsigned int k=0; k < M.mesh.z_coordinates.size() ; k++ )
    {
        for( unsigned int j=0; j < M.mesh.y_coordinates.size() ; j++ )
        {
            // ...
            // Same code as in the original implementation.
            // ...
        }
    }

```

Listing 4: Fragment of the parallel region for the solver implementation. Optimized code.

In figure 23 we show a trace visualization with Paraver of the execution with our implementation, compared with the trace of the original execution.

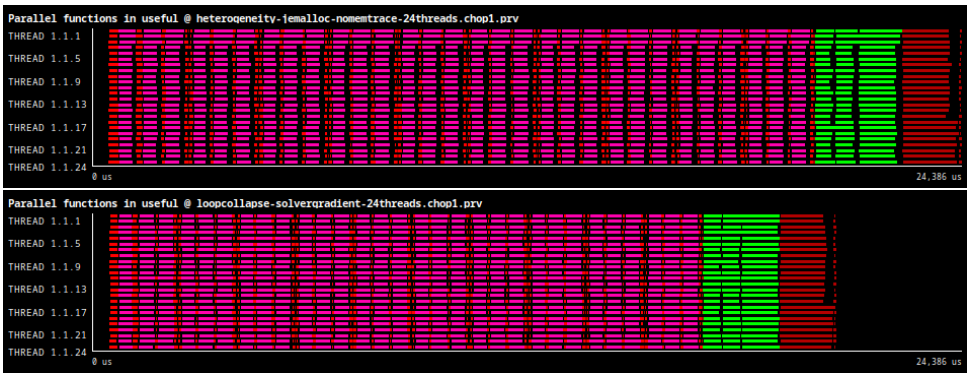



Figure 23: **Execution trace comparison between implementation for optimization 1 and original.** Showing Focus of Analysis, colors represent parallel functions. The top trace corresponds to execution without optimization, bottom trace corresponds to execution with the new implementation.

If we study specifically the load imbalance, which was the cause of the scalability problem, we also see that it has been totally restored. See Figure 24. The displacement on the first three threads does not appear anymore in the bottom histogram.

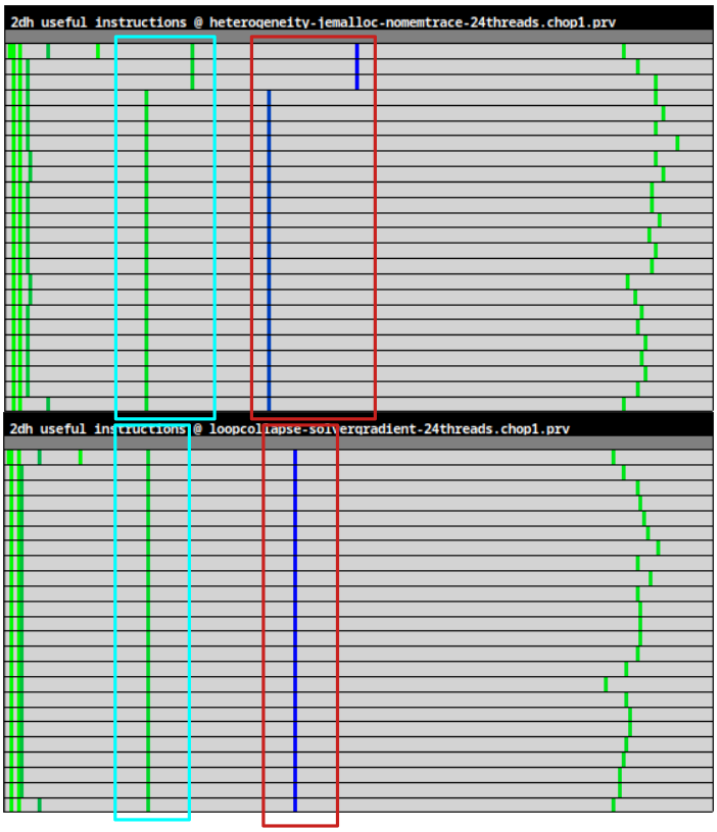



Figure 24: **Histogram of useful instructions per thread comparison between implementation recommendation 2 and original.** Top histogram corresponds to original execution, bottom histogram corresponds to execution with optimization 1. The X-axis represents the number of instructions grouped in intervals, the Y-axis represents the threads, and the color gradient indicates the number of computation bursts that report that number of instructions. Blue means higher, green means lower. The two regions that originally showed load imbalance are marked with a square in both histograms, with the same color.

To improve the IPC of the function that computes the cell's velocities, we propose to divide the workload into voxels, which is the existing subdivision of the 3-D space. Then cells will be grouped and computed by voxels, ensuring that neighboring cells are computed contiguously by the same thread. This parallelization strategy allows the threads to exploit the temporal locality of the cell's data, producing more cache hits and faster accesses

Following the guidelines explained before, the first approach proposed for the implementation consists of distributing workload in tasks, where each task computes one voxel. This implementation has two requirements:

1. Change the parallelized loop of the computation in order to iterate through voxels instead of cells. Then, inside this loop, iterate through the cells that the voxel contains.

2. Create an auxiliary list of voxels that contain cells. This list is required because, otherwise, many empty voxels would be added in the parallelization, causing useless overhead time.

We implemented this approach and studied its peformance using Paraver traces. The study of the traces suggests that an implementation based on a task paradigm is not a good approach for this optimization for the following reasons:

1. Creating a task for only one voxel does not exploit the temporal locality as expected because we do not see an improvement on IPC;
2. The task paradigm adds too much overhead;
3. Creating the list of not empty voxels before the parallel region adds a computation cost that increases execution time.

Taking into account these conclusions, we designed a second approach. We proposed an implementation based on the worksharing approach, using `parallel for` clauses, as in the original PhysiCell code. This implementation also computes the cells grouped by voxels but creates chunks of many voxels that the runtime would assign dynamically to the threads. This implementation would not previously create a list for the voxels containing cells but instead, check on each iteration if it is empty. This approach reduces the overhead of creating this list, and the overhead of task creation and scheduling, which is always higher compared to worksharing paradigms in the GNU OpenMP runtime. We expected to exploit temporal locality better, as voxels contiguous in the 3-D mesh would be placed on the same thread.

After implementing this aproach, and evaluating the executions with 50, 75, and 100 for the grain size value, we got to the following conclusions:

1. Voxels (and, consequently, cells) should be computed in the same order as found in the physical mesh. A coarser grain exploits the temporal locality much better;
2. A worksharing paradigm with dynamic scheduling shows better results;
3. The computation of empty voxels is useless and adds overhead, and this should be avoided. A list of voxels that contain cells should be created as efficiently as possible.

### 5.3.4 Implementation

From the conclusions of the previous experiments, we propose an implementation that distributes the cells in voxels with a worksharing paradigm. Only voxels that contain cells will be treated.

The list of not empty voxels will be created during one of the solver processes, which already iterates through the 3-D mesh. This function is parallelized, so every thread will collect a list of its voxels that contain cells, and then these partial lists will have to be reduced to a single list. Specifically, we will modify the first loop from the solver function `diffusion_decay_solver__constant_coefficients_LOD_3D`.

**Data structures** For each of the threads, we need an intermediate list of integers to store the IDs of the voxels that are not empty. These lists will be vectors with a pre-reserved capacity. A pointer to each of these vectors will be stored in an array. We will also need another array of integers for the final list of not empty voxels. These lists can be allocated with a fixed size at program startup because voxels do not change during simulation. For the intermediate vectors, we set the capacity for $N/P$ integers, where $N$ is the number of total voxels and $P$ is the number of threads; for the final array, we allocate memory to store a maximum of the total number of voxels. Therefore, this implementation sacrifices memory usage to maximize

performance because we will not have to worry about resizing the vectors during the simulation. The declaration of these structures is done in the class `BioFVM::Agent_Container` and can be seen in Listing 5.

```
std::vector<int> * ids_per_reduction;   // Array of intermediate vectors to
    store IDs
                        // Stores a pointer for each thread.
int size_max_vector;                    // Max number of elements of arrays
int *voxels_no_buits;                   // Final array of all IDs of not emptyu
     voxels
int voxel_no_buits_size;                // Actual size of the voxels_no_buits
    array
```

Listing 5: Data structures of the final implementation for optimization 2

**Initialization** These data structures are initialized (and reset) at the solver function. The memory allocation and vector creation works are done only once. Thus, in each iteration, we clear the intermediate vectors and reset their internal iterators with the function `clear`. The capacity of the intermediate vectors is only set at the initialization and then never changed, with the function `reserve`. The initialization and reset code can be seen in Listing 6.

```

// At initialization
M.agent_container->ids_per_reduction = (std::vector<int> * ) calloc(PhysiCell::
    PhysiCell_settings.omp_num_threads, sizeof(std::vector<int>));
M.agent_container->size_max_vector = (M.mesh.z_coordinates.size() * M.mesh.
    y_coordinates.size() * M.mesh.x_coordinates.size()) / PhysiCell::
    PhysiCell_settings.omp_num_threads;
M.agent_container->voxels_no_buits = (int *) calloc(M.mesh.z_coordinates.size()
     * M.mesh.y_coordinates.size() * M.mesh.x_coordinates.size(), sizeof(int));

for (int tid = 0; tid < PhysiCell::PhysiCell_settings.omp_num_threads; tid++)
{
  M.agent_container->ids_per_reduction[tid].reserve(M.agent_container->
    size_max_vector);
}

 // ...
 // When starting each iteration
 // ...
double time_since_last_mechanics= PhysiCell::PhysiCell_globals.current_time -
    ((PhysiCell::Cell_Container *)M.agent_container)->last_mechanics_time;
static double mechanics_dt_tolerance = 0.001 * PhysiCell::mechanics_dt;

bool mechanics_dt_check = fabs(time_since_last_mechanics - PhysiCell::
    mechanics_dt) <
  mechanics_dt_tolerance ||
  !((PhysiCell::Cell_Container *) microenvironment.agent_container)->
    get_initialized();

if( mechanics_dt_check ) {
  for (int tid = 0; tid < PhysiCell::PhysiCell_settings.omp_num_threads; tid++)
  {
    // Clear the vectors and reset internal iterators
    M.agent_container->ids_per_reduction[tid].clear();
  }
}
```

Listing 6: Initialization code of the final implementation for optimization 2

The reset part (as all the other processes related to the optimization that we will see later) is only executed when the condition

```
fabs(time_since_last_mechanics - PhysiCell::mechanics_dt) <
   mechanics_dt_tolerance ||
   ! ((PhysiCell::Cell_Container *) microenvironment.agent_container)
    -> get_initialized()
```

is given. This resolves true in two situations whether the `Cell_Container` is not yet initialized, or this iteration is past the $\Delta t_{mech}$ time for mechanic processes. This way we ensure that the extra computation work needed for the optimization is only done when needed.

Finally, to know if the `Cell_Container` is initialized, we need to add a new public method to this class to get the `initialized` variable, which is private:

```
bool Cell_Container::get_initialized()
{
  return this->initialzed;
}
```

Listing 7: Method added to check condition from the solver processes

**New logic** It consists of three significant code changes:

1. When iterating the mesh in the solver function, each thread checks the voxels and adds those that are not empty to its corresponding intermediate `ids_per_reduction` vector. The corresponding code is shown in Listing 8.

2. The reduction of the intermediate vectors to one final array. The corresponding code is shown in Listing 9

3. In the update velocity region, iterate through the list of not empty voxels instead of the whole grid, and for each voxel, compute the velocity of its cells. Parallelize the outer loop with dynamic scheduling. Inside the loop, obtain the actual voxel ID and call `Cell_Functions::update_velocity` for each cell of the voxel. The corresponding code is shown in Listing 10.

```
// Two levels inside the loop
// First voxel is computed outside the innermost loop

if( mechanics_dt_check ) {
  if(((PhysiCell::Cell_Container *)M.agent_container)->agent_grid[n].size() >
    0) {
    M.agent_container->ids_per_reduction[omp_get_thread_num()].push_back(n);
  }
}

for( unsigned int i=1; i < M.mesh.x_coordinates.size() ; i++ )
{
  n = M.voxel_index(i,j,k);
  axpy( &(*M.p_density_vectors)[n] , M.thomas_constant1 , (*M.p_density_vectors
    )[n-M.thomas_i_jump] );
  (*M.p_density_vectors)[n] /= M.thomas_denomx[i];

  // Rest of the iterations for this (y,z) combination
  if( mechanics_dt_check ) {
    if(((PhysiCell::Cell_Container *)M.agent_container)->agent_grid[n].size() >
     0) {
      M.agent_container->ids_per_reduction[omp_get_thread_num()].push_back(n);
    }
  }
```

```
}
```

Listing 8: New logic in the solver function for the final implementation of optimization 2

```
if( mechanics_dt_check ) {
  M.agent_container->voxel_no_buits_size = 0;
  int vnbs = 0;
  #pragma omp parallel reduction(+: vnbs)
  {

    int myid = omp_get_thread_num();
    std::vector<int> &v = M.agent_container->ids_per_reduction[
    omp_get_thread_num()];

    assert(v.size()<=M.agent_container->size_max_vector);

    vnbs += v.size();
    // Calculate start position a of the copy (depen del TID)
    int tid, prev_positions = 0;
    for (tid = 0; tid < myid; tid++)
    {
      prev_positions += M.agent_container->ids_per_reduction[tid].size();
    }

    std::vector<int>::iterator it;
    // Fem la copia
    int index = 0;
    for (it = v.begin(); it < v.end(); it++)
    {
      M.agent_container->voxels_no_buits[prev_positions+index] = *it;
      index++;
    }

  }

  assert(vnbs<(M.mesh.z_coordinates.size() * M.mesh.y_coordinates.size() * M.
    mesh.x_coordinates.size()));
  M.agent_container->voxel_no_buits_size = vnbs;
}
```

Listing 9: New logic for the reduction of the intermediate vectors for the final implementation of optimization 2

```
#pragma omp parallel for schedule(dynamic, GS)
for (int it_vector_nobuits = 0; it_vector_nobuits < voxel_no_buits_size;
    it_vector_nobuits++)
{
  // Obtain voxel actual ID
  int voxel_id = voxels_no_buits[it_vector_nobuits];
  std::vector<Cell*>::iterator neighbor;
  std::vector<Cell*>::iterator end = agent_grid[voxel_id].end();

  for(neighbor = agent_grid[voxel_id].begin(); neighbor != end; ++neighbor)
  {
    if(!(*neighbor)->is_out_of_domain && (*neighbor)->is_movable && (*neighbor)
    ->functions.update_velocity )
    {
      (*neighbor)->functions.update_velocity( (*neighbor), (*neighbor)->
```

```
    phenotype, time_since_last_mechanics);
14   }
15
16   if( (*neighbor)->functions.custom_cell_rule )
17   {
18     (*neighbor)->functions.custom_cell_rule((*neighbor), (*neighbor)->
    phenotype, time_since_last_mechanics);
19   }
20  }
21 }
```

Listing 10: New logic in the update velocity region for the final implementation of optimization 2

The results of this implementation correspond to the previous observations and hypothesis. Figure 25 shows a comparison of executions of the final implementation with different values for the grain size and the original code. Without the overhead of the empty voxels, and the creation of the vector of not empty voxels integrated in already existing processes, all experiments show an improved execution time respect of the original version. As expected, bigger grain sizes perform better (see table 12), thus the execution of trace a), which corresponds to a grain size of 20, is the one with shortest runtime. However, it also shows the highest load imbalance.

| **Parallelization paradigm** | **Version** | **Average** | **Max** | **Min** | **StDev** |
|---|---|---|---|---|---|
| Worksharing (voxels) with list of not empty voxels | Dynamic, GS=5 | 2.02 | 2.03 | 1.99 | 0.01 |
| | Dynamic, GS=10 | 2.03 | 2.08 | 1.90 | 0.03 |
| | Dynamic, GS=20 | 2.08 | 2.13 | 2.02 | 0.02 |

Table 12: Values of IPC on the update velocity region across diferent executions of the final implementation for optimization 2

## 5.4 Results

Figure 26 shows the strong scalability of all versions implemented. We observe that the optimization for suggestion 2 scales better than the jemalloc-only version. Furthermore, we do not see the stairs effect on the scalability, so the performance now is more independent of the resource configuration, which is positive. With the best optimization case we reach a 36.9x speedup with 48 threads or a 21.3x speedup with 24 threads.

However, the optimization for suggestion 4 does not show the performance improvement that we were expecting. For a low number of threads, it scales the same as the optimization 1 version and for a higher number of threads we don't get any speedup. In the previous section, we observed that the region we were trying to optimize actually showed an improvement in execution time, but the extra code added in the solver process adds execution time. Overall, we can say that with the best optimization version we are closer to the ideal scalability compared to the original version.

If we now look at the speedup obtained for each of the optimizations in Figure 27, we see that for 24 and 48 threads we achieved outstanding speedups with respect to the original version: we achieve a 1.59x of speedup with respect to the original code for an execution with 24 threads, and a 1.7x of speedup with respect to the original code for an execution with 48 threads. That means that for 24 threads and up PhysiCell shows a reduction of the execution time of more than 30%.

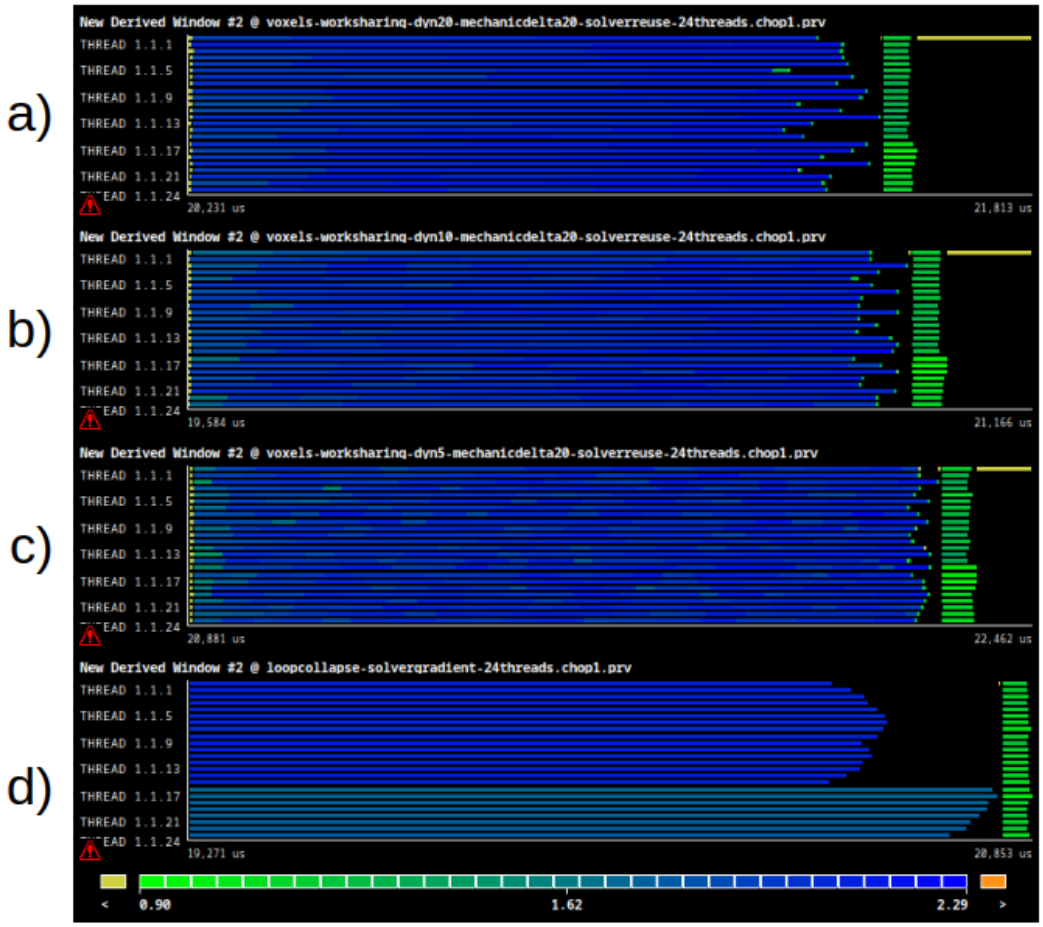



Figure 25: **Execution traces comparison for the final implementation of optimization 2.** Zoomed in the update velocity region. Color gradient represents the IPC value of the computation bursts; green means lower, blue means higher. From top to bottom: a) optimization with GS = 20, b) optimization with GS = 10, c) optimization with GS = 5, and d) original. Own elaboration.

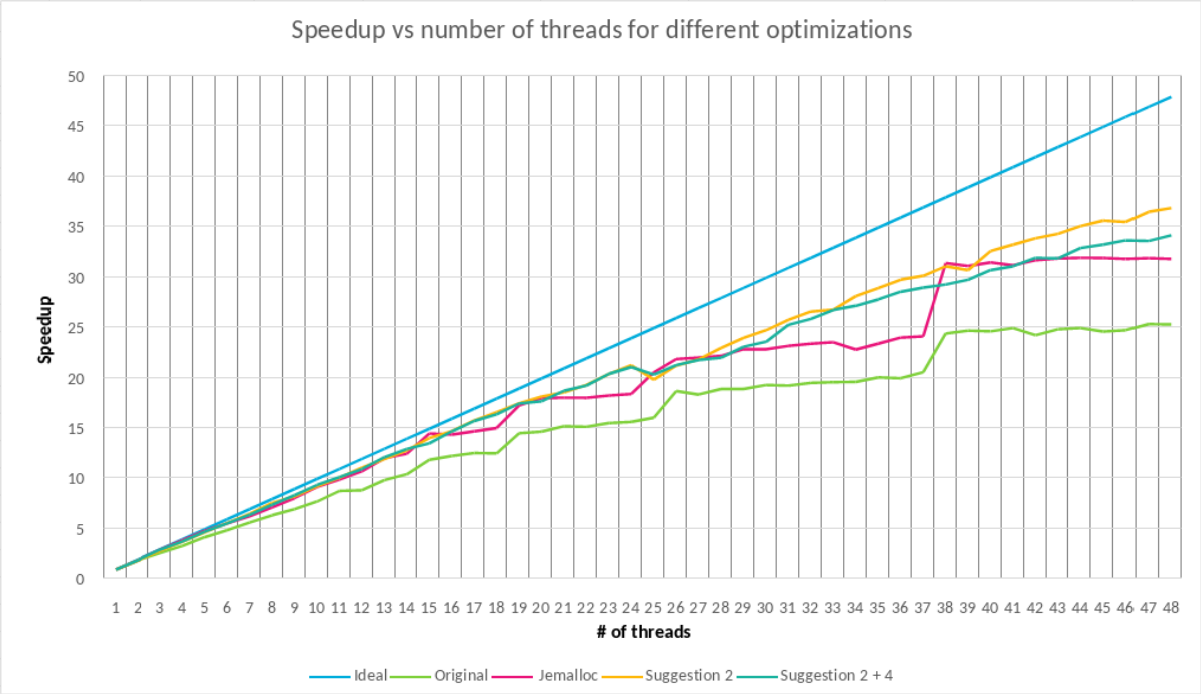



Figure 26: **Strong scalability plot for 1-48 threads, original and different optimizations.** Speedup values calculated for own series' base case. The X-axis represents the number of threads; the Y-axis is the speedup obtained with that number of threads.

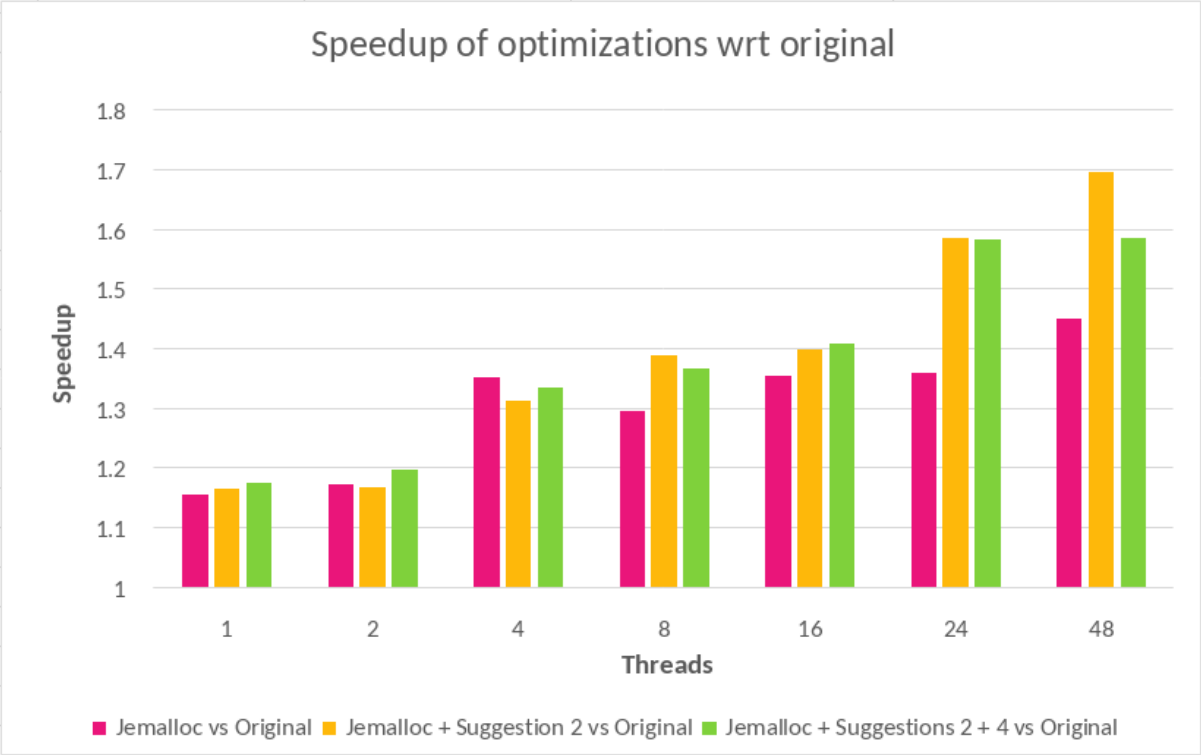



Figure 27: **Speedup of different optimizations respect original.** This bar plot shows the speedup obtained with the jemalloc integration, the optimization for suggestion 2, and the optimization for suggestion 4, wrt the original version of PhysiCell for different resource configurations. Own elaboration.

In the specific case of both optimizations, the execution with 16 threads surpasses the optimization of the solver part alone. This is a big hint that our implementation for suggestion 4 is very dependent on the number of threads or the problem size.

## 5.5 Conclusion

With the performance analysis of PhysiCell we detected a load imbalance problem coming from two different sources. The first load imbalance problem came from worload imbalance. Specifically, a too much coarse division of the workload introduced a bottleneck in the solver phase and in the computation of the gradients. This problem could be solved by tuning the parallelization and creating more and smaller chunks of work. This was our suggestion number 2 for the developers, and this proof of concept shows the benefits of optimizing the code.

On the other hand, the second load imbalance problem came from an imbalance in IPC for the computation of the velocities of the cells. A deep study of the behavior of IPC during the simulation lead us to interesting discoveries about memory locality that affect all agent-based modeling applications like PhysiCell.

With the best optimization case, we achieved a speedup of 1.59x for exectuions with 24 threads and a speedup of 1.7x for executions with 48 threads, using a full node of MN4.

The deep analysis of the load imbalance during this research project also offers another interesting technical outcome apart from the speedup achieved. We conclude that for applications that simulate cellular systems, processes that involve computing a value for the cells and need data from their neighboring cells should iterate through the cells as they physically appear in the 3-D environment. We extract this good practice from the analysis and the experiments performed.

## 5.6 Lessons Learned

### 5.6.1 Recommendations for tool developers

### 5.6.2 Recommendations for programming model developers

# 6 Report on Activity CODE_E

**Keywords:** Navier-Stokes, GPU, OpenCL

## 6.1 Description of the Application

CODE_E implements direct numerical simulation to solve the Navier-Stokes equations.

## 6.2 Previous Assessments and Recommendations

In the earlier assessment POP2_AR_074, we had the following observations:

- The code shows a computational pattern in which a fixed number of MPI ranks (e.g. 32) computes boundary conditions while the remaining (e.g. 480) ranks are waiting for that computation to finish.
- A closer look at the trace revealed, that three code blocks packages are processed in serial.
- An analysis of the source code showed that the three code blocks use a single instance of a random number generator (RNG).
- Another finding was the low IPC of 0.62. An analysis of the binary revealed the usage of AVX2 instructions in combination with 256 bit registers. From this observation we concluded that the code is memory bound.

Based on them, we recommended the following:

- The first recommendation was to distribute MPI ranks on the available nodes in a way that mixes idle and busy ranks within a node.
- The second recommendation was to process all three packages in parallel by one OpenMP thread each, where each thread uses its private state for the RNG.
- The third recommendation was to process the work packages serially using multiple OpenMP threads. Again, each OpenMP thread would use its private state for the RNG.
- The fourth recommendation was to introduce additional MPI ranks that are dedicated to the computation of the boundary conditions. The number of additional MPI ranks has to be sufficiently high so that the result of the computation is available when it is required by the other MPI ranks.
- The fifth recommendation was to explore the performance of GPU kernels that perform computations of some of the most time consuming code parts.

We discussed our recommendations with the customer and talked about the code in general. The customer was interested in our recommendation and he thought about some of them in the past already. The code does not have conditional statements inside the loops performing the computational work and the memory access pattern is sequential. Additionally, the code is readable and understandable as it is written in Fortran. The user provided a list of routines that cover about 70% of the computation time and the purpose of this PoC is to show that these routines run efficiently on GPUs.

## 6.3 PoC activities

### 6.3.1 Scope

In this PoC, three routines or code blocks will be ported to OpenCL. The performance of the OpenCL kernels running on a GPU will be measured and compared to that of the CPU kernels. The performance will be measured in processed cells per second. The following code parts will be ported to OpenCL:

- sd_compact_finite_differences_O6.F90:2606
- matrix_solver.F90:432
- ns_ddt_trafo_xy.F90:404 "ENERGY EQUATION"

A host application will be written that handles the OpenCL framework, performs the performance measurements and validates the results of the OpenCL kernels by comparing to results obtained from the corresponding CPU kernels.

The benchmarks do not require a physically meaningful data set as input. The number of cells will be varied between 28 and 220. The performance measurements will provide the number of processed cells per second for the GPU kernels and the CPU routines, this is the first metric used to measure success. The CPU routines will be executed on an AMD EPYC 7742 (HAWK). All CPU cores will be occupied with the same benchmark in order to make a fair comparison chip to chip. The second metric is the correctness of the results. The difference between results obtained on CPU and GPU should be tiny and only resulting from a different order of execution of floating point computations. The metrics to measure success are:

- number of processed cells per second, success: $GPU \geq 2 * CPU$
- results GPU = results CPU, success: tiny difference due to floating point rounding, relative error $\leq 1e - 10$

The CPU routines will be compiled with the GNU C compiler and benchmarked on a single node of HAWK. The GPU kernels will be benchmarked on a NVIDIA V100 GPU.

### 6.3.2 Implementation

**Software environment** The host program including the CPU routines that are ported in this PoC were compiled using the GNU compiler version 9.2.0. The GPU driver version is 440.95.01. The OpenCL kernels created here require only OpenCL 1.0 features.

**Benchmarking CPU routines** The CPU benchmarks were performed on HAWK using a single socket, i.e. on a single AMD EPYC 7742. The program measuring the performance of the CPU routine repeatedly executes the computation. The number of repetitions was chosen high enough for the performance to saturate. The program used four OpenMP threads and the program was pinned to cores 0 to 3 using numactl. Additionally, local allocation was enforced using numactl. The number of OpenMP threads matches that of the original program. The remaining cores of the CPU are running the same program with the same problem size but much larger number of iterations in order to simulate additional MPI ranks of the original application. Keeping cores 4 to 63 busy reduces the performance obtained from the benchmark while it reflects realistic operation conditions.

**Benchmarking GPU kernels** The GPU kernels are benchmarked by running a single program on a single NVIDIA V100 GPU. The structure of the program is rather simple. First of all, everything related to OpenCL is set up, including the compilation of the OpenCL kernel. Before compiling the kernel, the parameters defining the problem size, i.e. nx, ny and nz are appended to the source code. The compiler can therefore optimize some computations which improves the performance of the kernel. The arrays storing input and output are initialized. Second, the kernel is executed repeatedly and the performance is measured. The time measurement includes the time taken for enqueuing the kernel and the time waiting for the completion of the queue, i.e. all the overhead involved in using the GPU. The time taken for memory transfers is not included in the measurement as the transfers occur only during initialization and finalization of the program. Finally, the results obtained from the GPU kernel are compared to those obtained from the CPU routine. The results are considered equal when the relative difference is less than 1e-10.

**Porting "ENERGY EQUATION" - kernel_0** The code block at line 404 in file ns_ddt_trafo_xy.F90 is referred to as kernel_0 from here on. This kernel consists of a loop over all elements. In each iteration, 58 values are loaded from 58 different input arrays and after computing about 80 intermediate values, a single output is computed. As the memory access is sequential for all input arrays and the output array, this kernel translates directly to an OpenCL kernel where each work-item (i.e. thread) computes exactly one output value. The index of computed value is equal to the global ID of the work-item. The memory access pattern is perfect for a GPU. The code of this kernel is outlined below.

```
__kernel void ns_ddt_trafo_xy_V_original( /* input and output arrays*/ )
{
    __private size_t index=get_global_id(0);
    // load data from arrays
    __private double t1=u[index];
    ... many more
    // compute
    __private double t25=t2*heat_flux_factor;
    ... many more
    // store result
    output[index]=result;
}
```

Figure 28 shows the performance of kernel_0 running on CPU and GPU. The performance of the GPU kernel increases linearly with the size of the data set and reaches a constant value for data sets larger than 4e4 elements. In that range, the GPU outperforms the CPU by at least a factor of 4. The performance of the CPU routine decreases with increasing size of the data set due to the well known cache effects. CPU and GPU show a different behavior: CPUs achieve highest performance for rather small data sets while GPUs require a certain amount of work to achieve peak performance. There is no reason to expect a decrease in performance of the GPU kernel for larger data sets.

**Porting "solve_tridiag_system_x" - kernel_1** The routine solve_tridiag_system_x at line 432 in file matrix_solver.F90 is referred to as kernel_1 from here on. This kernel uses a cubic domain with nx=ny=nz elements in each direction. The computation in this kernel has a data dependency in the x-direction. For illustration, a 3D Cartesian grid can be used. An element at

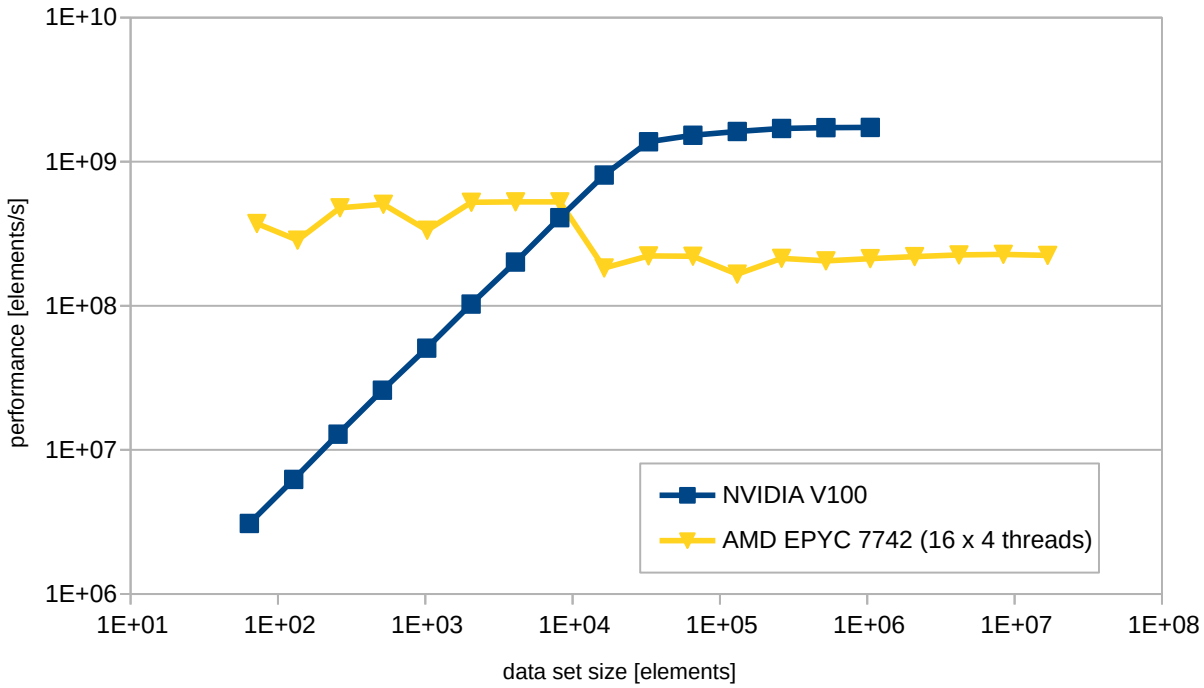



Figure 28: Performance of kernel_0 running on CPU and GPU.

coordinate (x/y/z) can only be computed after the element at ((x-1/y/z) has been computed. The computation consists of the following code parts:

- loop over the elements at x=0 which is a 2D loop processing ny*nz elements
- loop over the interior elements which is a 3D loop processing (nx-1)*ny*nz elements
- loop over the elements at x=nx which is a 2D loop processing ny*nz elements
- loop over the elements at x=nx which is a 2D loop processing ny*nz elements
- loop over the interior elements which is a 3D loop processing (nx-1)*ny*nz elements

The two 3D loops contribute most to the required execution time due to the larger number of processed elements. The CPU version handles the data dependency by switching the loop over elements in x-direction with the loop over elements in y-direction. This allows an efficient parallelisation with OpenMP.

The routine was ported to OpenCL by creating a separate kernel for each one of the loops above. The OpenCL kernel corresponding to the first 3D loop consumes the most time due to the unfavorable memory access pattern. In order to respect the data dependency in the x-direction, each work-item computes all values in x-direction for a given pair of y and z coordinates. See

the code below for clarification.

```
__kernel void solve_tridiag_system_x_b_V_original( /* input and output */ )
{
//  k=1..nz     j=1..ny     i=2..nx

    __private size_t global_id=get_global_id(0);
    __private int k=global_id/(ny+1);
    __private int j=global_id-k*(ny+1);

    if ((k>0)&&(k<=nz)&&(j>0)&&(j<=ny))     /* ignore boundary elements */
    {
        for (int i=2;i<=nx;i++)
        {
            __private int e=k*(nx+1)*(ny+1)+j*(nx+1)+i;
            sol_vec[e]=(rhs[e]-a[i]-sol_vec[e-1])*q[i];
        }
    }
}
```

While sequential memory access of a single thread performs well on CPUs, it does not on GPUs. Figure 29 shows the performance of kernel_1 running on CPU and GPU. Like with kernel_0, the performance of the GPU kernel increases with increasing size of the data set until a size of about 3e5 elements. Beyond that size, the performance gradually decreases, probably due to competition for cache.

At a data set size of about 3e6 elements, the performance of the GPU kernel and the CPU routine are about equal. However, in order to achieve the performance of the CPU routine, 16 MPI ranks need to be used while only a single MPI rank is required for the GPU kernel. For this kernel, the increase in performance will result only from the reduced number of MPI ranks.

## 6.4 Porting "calc_ddx_visc_cfd_O6" - kernel_2

The code block at line 2748 in file sd_compact_finite_differences_O6.F90 is referred to as kernel_2 from here on. Together with four 2D loops, kernel_2 forms a routine called calc_ddx_visc_cfd_-O6. The 2D loops are not ported here as their contribution to the runtime is very small, like in kernel_0.

kernel_2 is similar to kernel_0 as it computes a single output per iteration. Therefore, the loop is translated directly into an OpenCL kernel where each thread computes a single value of the output. The input for each thread consists of the corresponding element in an array and it two neighbors in positive and negative x-direction. As the elements in x-direction are stored sequentially, the memory access pattern is sequential for neighboring threads which in turn is perfect for a GPU. The access to the neighboring elements is also fast, as the data is in the

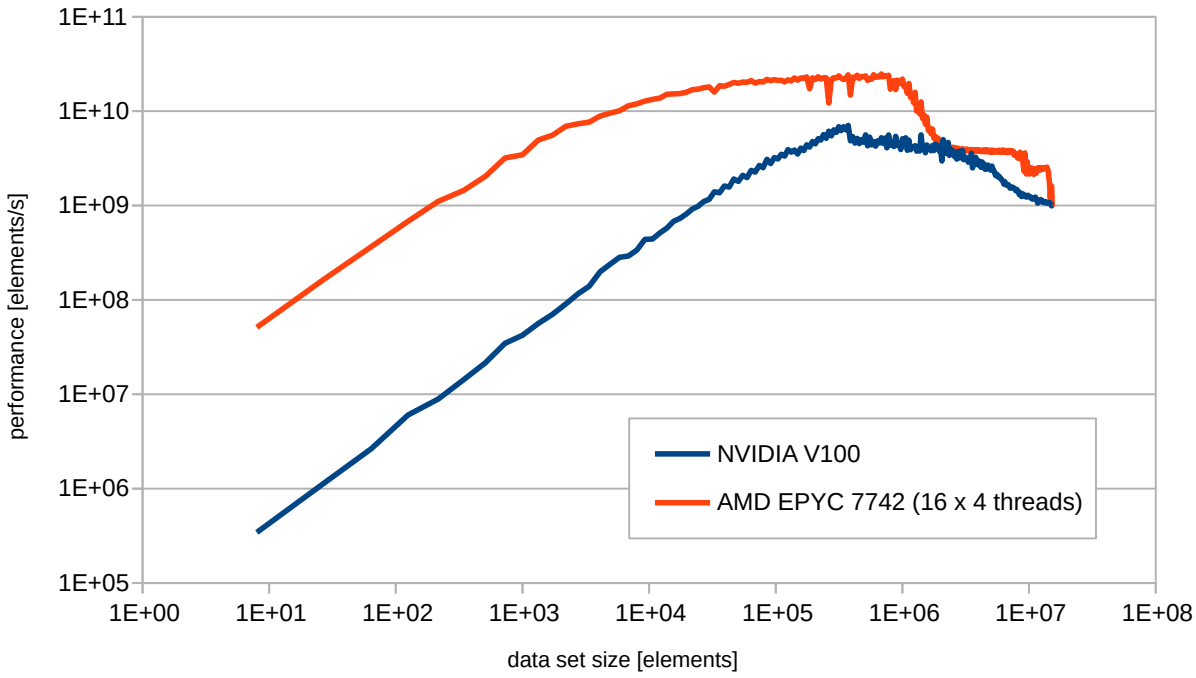



Figure 29: Performance of kernel_1 running on CPU and GPU.

cache already. The OpenCL kernel reads as follows.

```
__kernel void calc_ddx_visc_cfd_06_a_V_original( /* input and output*/ )
{
    __private size_t global_id=get_global_id(0);
    __private int k=global_id/((nx+1)*(ny+1));
    __private int j=(global_id-k*(nx+1)*(ny+1))/(nx+1);
    __private int i=global_id-k*(nx+1)*(ny+1)-j*(nx+1);

    if ((k>0)&&(k<=nz)&&(j>0)&&(j<=ny)&&(i>2)&&(i<=nx-2)) /* no boundary */
    {
        __private int e=k*(nx+1)*(ny+1)+j*(ny+1)+i;
        rhs[e]=c[0]*F[e-2]+c[1]*F[e-1]+c[2]*F[e]+c[3]*F[e+1]+c[4]*F[e+2];
    }
}
```

Figure 30 shows the performance of kernel_2 running on CPU and GPU. Again, the performance of the GPU kernel increases with increasing data set size and reaches a plateau for data set sizes larger than 1e6 elements. The maximum performance exceeds that of kernel_0 by a factor of 20 as the computation is relatively simple and the kernel performs only two memory accesses per element. The performance of the CPU routine decreases for data set sizes above 1.1e6 elements as the data set does not fit into the cache anymore.

## 6.5 Results

Three code blocks or routines have been ported to OpenCL kernels. The kernels have in common that they all compute a single output per thread but the number of memory access,

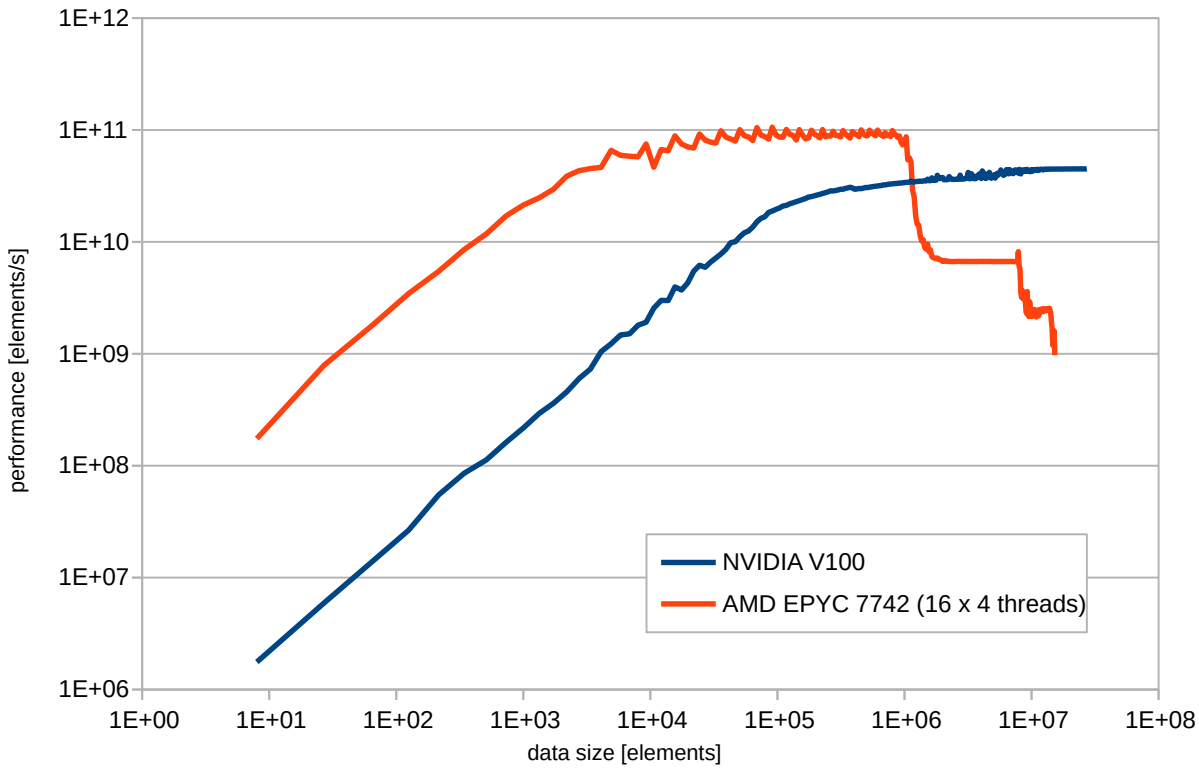



Figure 30: Performance of kernel_2 running on CPU and GPU.

the memory access pattern and the complexity of the computation vary. Benchmarks were performed to compare the performance of the OpenCL kernels to the original CPU routines. The benchmarks revealed that the performance of the OpenCL kernels increases with the size of the data set before a plateau is reached. Above a certain data set size, the two kernels with sequential memory access and access to neighboring memory addresses achieve significantly higher performance than the CPU routines. The kernel with the unfavorable memory access pattern (kernel_1) achieves a performance that is close to the CPU routines only for larger data set sizes.

Figure 31 shows the ratio between the OpenCL kernel running on the GPU and the CPU routine for various data set sizes. It should be noted that the GPU kernel requires only a single MPI rank to execute while sixteen MPI ranks need to execute in parallel on the CPU in order to achieve maximum cumulative performance. This means that the GPU version of the code would suffer less from inefficiencies related to the parallelisation as these POP metrics usually decrease with the number of MPI ranks.

Using the metrics defined above, it can be concluded that two kernels, i.e. kernel_0 and kernel_2, pass both metrics while kernel_1 fails the first metric.

## 6.6 Conclusion

The user was highly interested in this PoC because it is completely different compared to the other recommendation. The user had already explored some of the other recommendations.

This PoC showed that the selected parts of the CPU code can be implemented easily in OpenCL. The performance of the GPU kernels significantly exceeds that of the CPU code when the memory access pattern in the CPU version is favorable, i.e. sequential. For CPU code

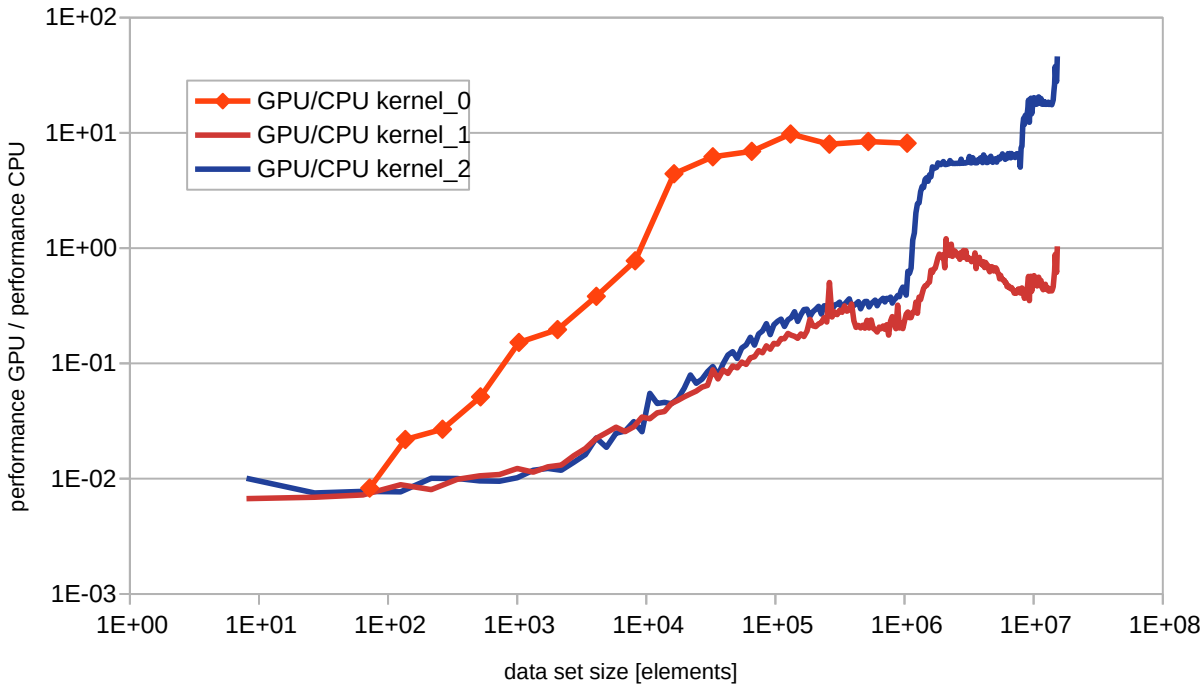



Figure 31: Performance of GPU kernel in relation to the performance of the CPU routine for kernel_0, kernel_1 and kernel_2.

where the memory access pattern is random in the CPU code, the GPU version achieves an equal performance when the size of the data set is big enough.

In this PoC, the performance of one CPU is compared to that of one GPU. It is important to notice that the GPU code uses one MPI rank per GPU while the CPU code requires 16 MPI ranks and 4 OpenMP threads per rank to achieve maximum performance. This means that the GPU code achieves the same performance as the CPU code with less MPI ranks. A reduction in the number of MPI ranks usually corresponds to an improvements in POP metrics.

The parts of the CPU code ported here are OpenMP loops. This fact should not lead to the conclusion that the porting effort ends with porting only these regions. Instead, the complete code has to be ported to GPU in order to achieve best performance.

## 6.7 Lessons Learned

### 6.7.1 Recommendations for tool developers

The performance analysis revealed a low IPC of 0.62. To identify the reason for this low value, the executable was disassembled and vector instructions were found. The size of the registers used in the vector instructions allowed to determine the vector instruction set. This process could be automated and provided as output of the tools commonly used for performance analysis. The output could contain the number of vector instructions, the vector instruction set and the number of code regions that contain vector instructions. This information could first of all explain low IPC but also help to identify codes that are suitable for porting to GPU.

### 6.7.2 Recommendations for programming model developers

# 7 Report on Activity CODE_F

**Keywords: OpenFOAM, I/O, Asynchronous I/O, C++Thread**

## 7.1 Description of the Application

*Openfoam/CODE_F* is a mini-app which reproduces some of the behaviour of the application *CODE_F*. Both are OpenFOAM applications. The main difference is that the mini-app solves a simpler system of equations with less physical variables and no particles. This proof-of-concept uses the mini-app openfoam/CODE_F. For simplicity, most of this document will refer to the mini-app as *CODE_F*. Where necessary, we will use the terms 'openfoam/CODE_F mini-app' and 'full CODE_F code' for distinction.

## 7.2 Previous Assessments and Recommendations

Both, the full CODE_F code (POP2_AR_039) and the mini-app (POP2_AR_057) have been analysed by POP previously.

In the particular analysed use-cased, the full CODE_F code was found to spend significant amount of time exchanging a large number of small messages as well as doing collective operations, which results in lowish parallel efficiency at large number of nodes. Further, some parts of the code showed low IPC values, which however increased with smaller local problem sizes (as cache utilisation increases).

Since the size of traces for the full CODE_F code was very large, the authors of the code suggested to use the mini-app for subsequent follow-up assessments. The assessment POP2_AR_057 concentrated on the single-core performance, in particular the identification of source code regions with low IPC and its behaviour under scaling.

While setting up the use-cases for these assessments, POP staff noticed that I/O for check-pointing may take significant time. However, since I/O was expected to happen only infrequently, it was not investigated further. We recommended in private communication that I/O should be done asynchronously to overlap with computation if possible. Recently, the code owners have a use for a higher check-pointing frequency (visualisation) and thus suffer from high I/O cost.

## 7.3 PoC activities

### 7.3.1 Scope

This activity aims at implementing in the mini-app openfoam/CODE_F an asynchronous scheme which overlaps I/O with computation as a demonstration for the full CODE_F code. The scheme should not increase the memory footprint of the application nor change the structure and order of solving physical quantities. At the code owners request, the implementation should use standard C++ features without requiring further programming models such a OpenMP. This will increase the ease of porting the proof-of-concept to the full code.

The code owners have selected a volcanic plume eruption as use-case. The use-case has 32 million cells distributed over 512 MPI processes. The number of MPI ranks can be changed to study (strong) scalability if necessary.

We have selected the suitably averaged execution time per timestep as the main evaluation metric. If possible and necessary, we foresee the I/O overlap ratio as a secondary evaluation metric. The use-case is set up to do check-pointing at every time-step. This is mainly in order

to increase the time spent in I/O for more accurate comparison of the optimised version and the baseline.

Any reduction of the mean execution time is a success.

This proof-of-concept will be executed mainly on the system Hawk at HLRS. Hawk consist of roughly 6000 nodes, each with two sockets of AMD Epyc Rome processors. Each processor has 64 cores, thus 128 cores to the node.

If necessary, benchmarks may be done also on MareNostrum at BSC.

### 7.3.2 Implementation

**Basic concept** The idea of having an asynchronous I/O is to hide the I/O latency of the code by spawning a service thread that runs in the background, in charge of writing data asynchronously while the main thread executes the rest of the code in parallel. The proposed idea is based on concepts presented in [2].

The code consists of a main time-stepping (or iteration) loop. Every $n$ time-steps, the code "dumps" or writes all the data, i.e all the values of the fields involved, probes, and other variables. In the initial version, I/O is done right at the end of the time-stepping loop, just before the next iteration starts. The I/O routine is blocking, meaning that execution is stopped to perform the output writing operation and not resumed until I/O is finished.

As a first optimisation step, the code was modified such that every time a field or any other variable is ready to be written, it is sent for writing to a dedicated thread (referred to as background thread), while the main application thread keeps executing the next batch of instructions. However, at the end of each time-step, the application thread needs to wait for pending I/O operations of the background thread. This synchronisation is required to safe-guard against data races between the background thread (still reading data) and application thread (already overwriting with new data). This is illustrated in Figure 32 below.

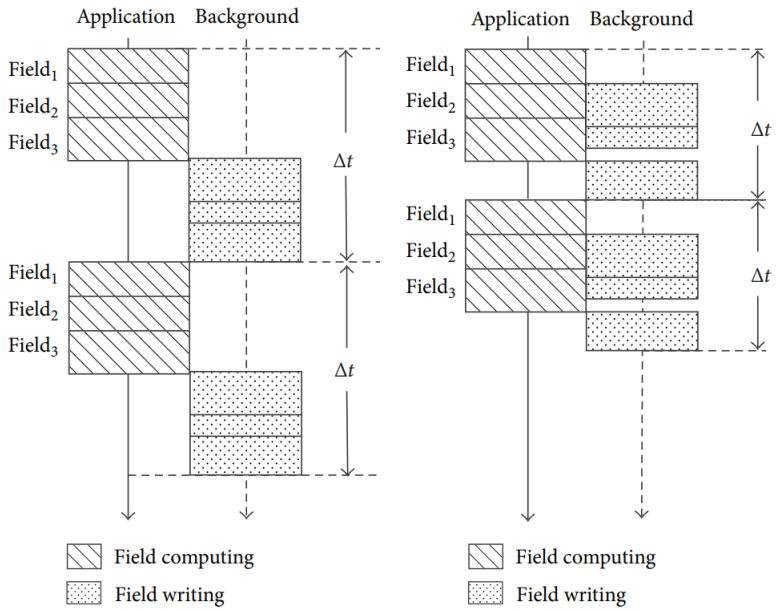



Figure 32: **Left**: Original execution scheme. Writing of data is done at the end of each iteration with blocking I/O routine while the main thread is waiting. **Right**: Asynchronous I/O scheme. Each data field is handed over to a dedicated background thread for asynchronous writing as soon as ready, The main thread continues regular execution. Extracted from [2].

At the request of the code owners, we have not implemented an alternative approach which is take a copy of the data and write to disk from such a copy. While this will have solved the issue of data races, it would have increased the required amount of memory (doubled for each field which needs to be dumped). Also, copy operations can be quite expensive and will need to be executed on the critical path of the application. Finally, deep copies of C++objects are not automatically requiring each future extension of the data structures to account for the seemingly unrelated concern of deep copies for I/O.

With our initial asynchronous I/O scheme we're able to partially hide the latency of I/O operations behind field-computation instructions on the application thread. Partially, because there is a limit to how much of the I/O workload we're able to hide and thus overlap with computational operations. One limitation is that I/O operations can only be fully overlapped, if the remaining duration of computation (i.e. before the synchronisation point at the end of the time-step loop) is longer than the duration of the I/O operation. We refer to this as Potential Shield Time (PST). Note, that for the field which is computed last in the iteration loop, the PST is negligible and the application thread will always have to wait for at least the time to write this particular data.

As a further optimisation step, we have removed the global synchronisation point at the end of the time step loop and replaced it with fine-granular synchronisation points for each individual data field. Conceptually, this is possible, because not every field is a required input for all computational steps, nor are all fields updated at each computational step (see right panel of Figure 32. A careful analysis of the data dependencies of the code has allowed us to determine for each field where it is updated (output) and where it is used (input). Note, that this analysis will need to be repeated when inserting, modifying, or even reordering computational steps. Also note, that this is essentially the same as applying OpenMP tasks with dependency clauses; in particular, it will require the same data dependency analysis and familiarity with the computational scheme.

With this second optimisation, the PST is the span of time between the last update to a field and the first usage of the field data in the next iteration. It is thus the longest duration of I/O operations which still can be fully overlapped with computation. Figure 33 illustrates qualitatively the PST of all field variables, and the actual duration of writing fields to disk. Replacing the global synchronisation point at the end of the time-step loop increases the PST for all fields. Note, that writing the field $U$ takes by far the most time, but has the shortest PST. Thus, writing of the field $U$ will have the biggest impact on the performance improvement, at least for similar use-cases.

**Realisation with C++** We decided to use POSIX threads through the C++ `pthread` library, which allow for a fine-grained control of threads and are perfectly suited for simple problems with a small number of them. Unlike other shared memory, parallel programming frameworks like OpenMP, POSIX threads offer a low-level tuning that grants total control over the scheduling of threads and the synchronisation between them.

The parallel scheme is pretty simple: At the beginning of the simulation, the background thread is spawned, which does nothing other than taking writing jobs from a queue and executing them one after the other. In order to exploit the ability of most current processors to accommodate multiple threads (SMT), the background thread is mapped and pinned to the spare logical CPU in, hoping to better spread the workload. This is done by setting the CPU affinity of the background thread with `pthread_setaffinity_np()`. The background thread will in principle compete for CPU resources with the main application thread. However, since the heavy part of all I/O operations is taken care of by the I/O controller in the background, this does not have a large impact on performance.

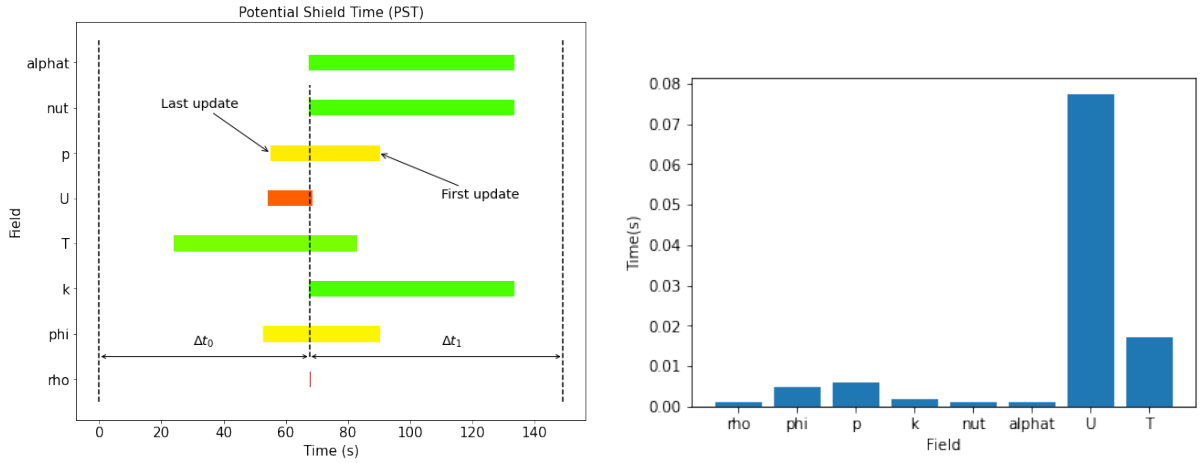



Figure 33: PST for each field computed for the CODE_F code (left panel) and I/O times (right panel) for every field for a use-case with 32 million cells on a single node of MareNostrum (48 cores). Please note, that the time values on the two panels cannot be compared as they originate from different problem sizes.

For the synchronisation between threads, condition variables were used. A condition variable is an object that can be used to block or resume the execution of a thread through a system of locks and notifications. So instead of polling for a given result to be available, a thread can be notified ("woken up") by another thread when it should resume.

In our case, one condition variable was used for the background thread to wait whenever the job queue is empty and be notified whenever a job is added to it (see Figure 34 Right). In addition, we use a condition variable for every field, which sends the main thread to sleep (`cv.wait()`) when the field is still being written and gets notified (`cv.notify_one()`) when the output operation is over and the field can be updated again (see Figure 34 Left). This requires the use of two separate `mutex` or mutual exclusion objects: One is used to lock the job queue to the writing thread when this one is empty through the condition variable and the other locks the field variables to the main thread to prevent it from updating them while they're being written to disk.

## 7.4 Results

Contrary to the original plan, the results of the code modifications have been benchmarked on MareNostrum rather than Hawk. The reason is that filesystem operations on Hawk were suffering from severe performance variations at the time. Due to the limited amount of CPU time on MareNostrum, the benchmarks used lower number of MPI ranks than planned. This PoC has been the basis for a Masters thesis project. Here, we present only the most relevant benchmarking results. Further data and results can be found at [4].

In order to improve the statistics of benchmarking, we have setup the code to dump results to disk at every time-step. The optimisations have been evaluated in terms of two metrics: 1) total execution time of the application, and 2) I/O overlap ratio. The latter is defined as one minus the ratio of time spent waiting for completion of I/O operations measured on the main application thread ($t_{\mathtt{W}}$), over the duration of I/O operations in the original baseline version ($t_{\mathtt{IO}}$), i.e.

$$1 - \frac{t_{\mathtt{W}}}{t_{\mathtt{IO}}}.$$

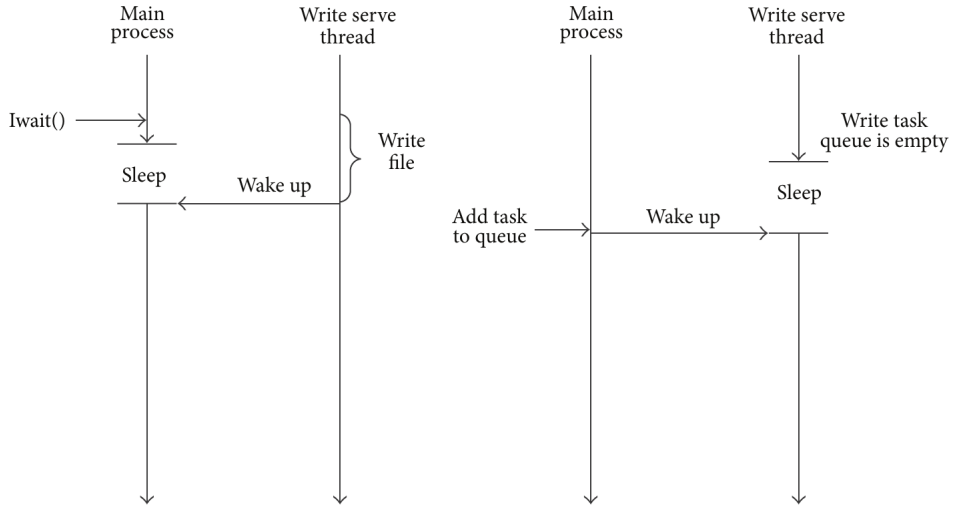



Figure 34: **Left**: The background thread has finished writing the field and notifies the main application thread to resume execution and update the field. **Right**: Application thread notifies the "sleeping" (empty queue) background thread that a new field is ready to be written. Extracted from [2].

The baseline version and the optimised version are run in the same job to get comparable values for both quantities and avoid environmental influence between jobs. For each run, we have multiple independent repetitions of I/O operations and can calculate statistically sound averages and standard deviations. For each setup, we have done multiple runs and show them side by side to show run-to-run variations due to the influence of the changing environment.

Figures 35 and 36 show results for single-node runs (48 MPI ranks). The application as a whole runs roughly 5% faster. On a single node, I/O operations are almost fully overlapped with computation; the overlap ratio is 98%.

When increasing the node count, the overlap ratio decreases to values between 50% and 60%, as shown in Figure 37 for 2 nodes (96 MPI ranks). Similar values are observed for runs with 3 nodes, which is the largest we could do. Also, the overlap ratio does not seem to depend much on the problem size.

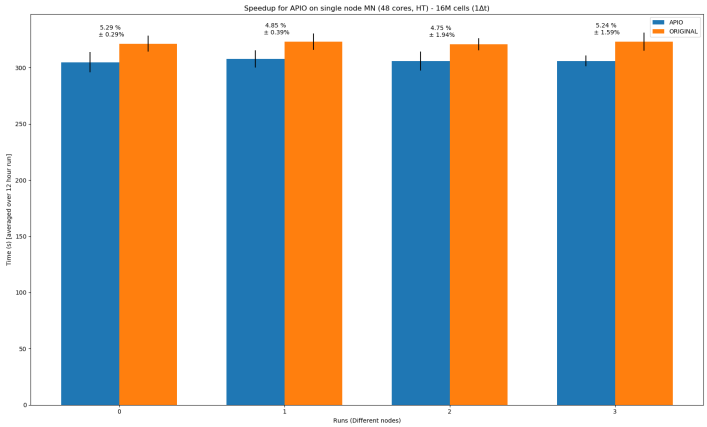



Figure 35: Comparison of total execution time of the original and the optimised version for single-node runs (48 MPI ranks) using 16 million cells. The plot shows 4 different runs.

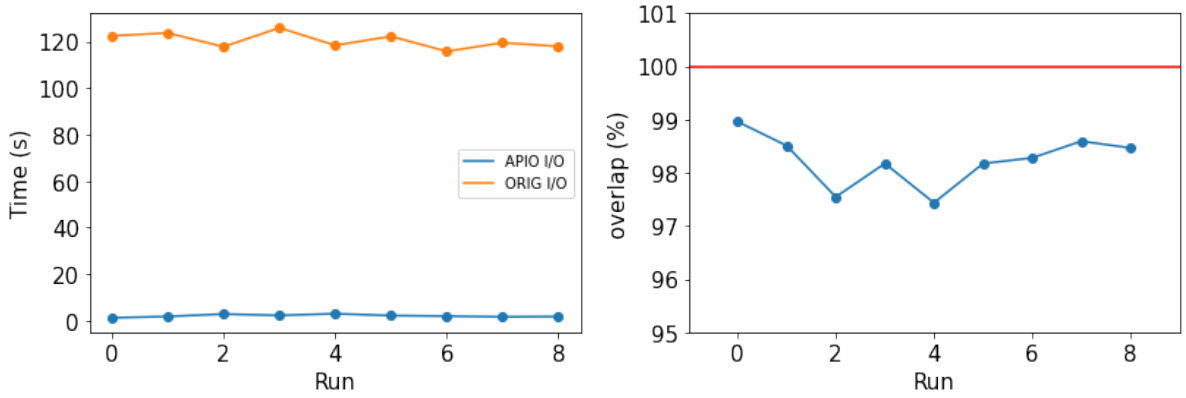



Figure 36: Left: Comparison of I/O times of the original and the optimised version for various single-node runs (48 MPI ranks) using 32 million cells. Right: Overlap ratio for the same runs.

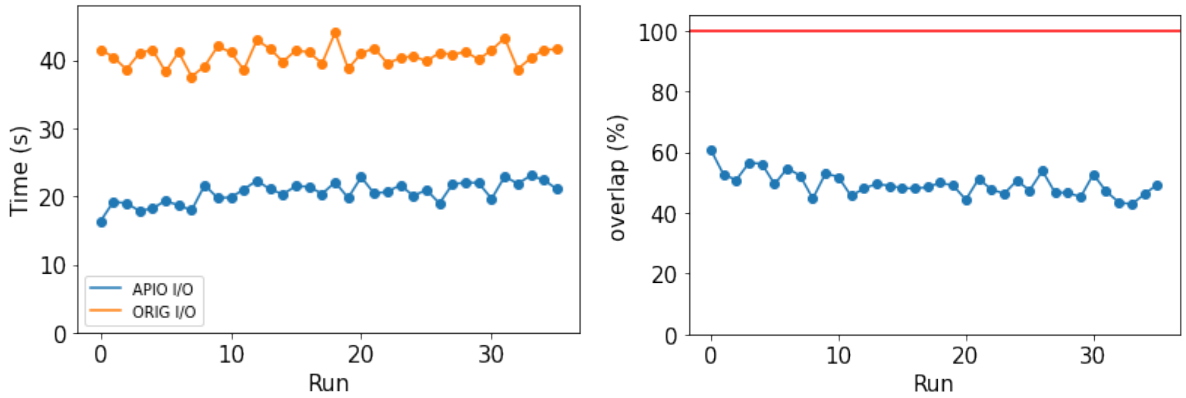



Figure 37: Left: Comparison of I/O times of the original and the optimised version for various multi-node runs (96 MPI ranks) using 32 million cells. Right: Overlap ratio for the same runs.

Lastly, the few (and noisy) runs that we have for Hawk show similar overlap ratios can be reached even for runs at larger scale (up to 4096 MPI ranks) and with larger problems sizes (up to 256 million cells).

## 7.5 Conclusion

This PoC implemented a scheme to do I/O asynchronously in the mini-app openfoam/CODE_F. The design consists of a background thread which handles I/O operations while the main application thread continues calculating. It takes data dependencies between the background and the application thread into account. The design was implemented with C++threading using condition variables to model data dependencies. We expects, that this design, and in fact the implementation, can be easily applied to the full CODE_F code.

We have evaluated the implementation on various node counts and for various problem sizes on MareNostrum and on Hawk. Experiments on MareNostrum, show with high statistical significance, that $40\% - 60\%$ of the I/O time can be overlapped with computation leading to substantial reduction of the total execution time. Beyond a single node (where we observe nearly complete overlap), this result depend only weakly on the number of nodes or the problem size. Due to ongoing issues with the filesystem at the time of evaluation, the data on Hawk is very noisy and incomplete. However, when available, it is consistent with findings on MareNostrum.

## 7.6 Lessons Learned

### 7.6.1 Recommendations for programming model developers

A similar asynchronous I/O scheme could have been implemented based on OpenMP's task construct with data dependencies. However, the code owners shied away from introducing an additional programming model without evidence for the efficiency of the approach.

# 8 Report on Activity CODE_H

**Keywords: Python, Multiprocessing, Numba**

## 8.1 Description of the Application

CODE_H is a probabilistic model used to date paleoclimatic records. It is currently hosted on GitHub.[7] Parallelism is via parallel BLAS, for example when installing via Anaconda or Conda, multithreaded BLAS are provided via MKL.

The code developer reported the code scales poorly, e.g. only 2 times faster on 8 cores.

## 8.2 Previous Assessments and Recommendations

There was no POP assessment on Paleochrono before this work, hence the first phase of this Proof of Concept was to calculate the POP metrics and analyse performance. The main observation of this assessment are:

- There is no particularly good place, i.e. hot spot, to add parallelisation for the particular dataset.
- The time in `residuals_jacobian`, `residuals_jacobian1` and `residuals_jacobian2` is small, and so any parallelism in the loops that call these functions will have small impact.
- The time in the loops within `corrected_jacobian` is small, and so are unlikely to parallelise well.
- The best place to add parallelism is the loop over `corrected_jacobian`, which has only 7 iterations and considerable imbalance.

## 8.3 PoC activities

### 8.3.1 Scope

Based on observations during the initial assessment, we recommended parallelising the loop over `corrected_jacobian`. The speedup will be modest for this dataset, but better speedup may be achieved using datasets with more sites and/or better load balance.

Two strategies were tried as a way of parallelising the most expensive loop, firstly using Python's multiprocessing, and secondly using a Numba `prange` loop.

## 8.4 Results

### 8.4.1 Parallelisation using Python multiprocessing

Python includes a `multiprocessing` package which can be used to add parallelism relatively easily by creating a multiprocessing pool of processes to distribute work over. There are multiple ways to submit work to the pool, for this study the `apply_async` function was used, as this is quick to implement. `apply_async` executes functions asynchronously using processes from the pool.

The first step is to create a pool, e.g. replace the call to `least_squares` with the code in Figure 38, where `n_proc` is the number of processes in the pool.

[7] https://github.com/parrenin/paleochrono

```
   with multiprocessing.Pool(n_proc) as p:
      OptimizeResult = least_squares(residuals,
                                     VARIABLES,
                                     method=pccfg.opt_method,
                                     jac=jac,
                                     tr_solver=pccfg.tr_solver,
                                     xtol=pccfg.tol,
                                     ftol=pccfg.tol,
                                     gtol=pccfg.tol,
                                     verbose=2)
```

Figure 38

```
def nag_corrected_jacobian(site):
   # Call site corrected_jacobian
   site.corrected_jacobian()

   # Return a tuple containing the values
   if site.archive == 'icecore':
      return (site.age_jac,
              site.airage_jac,
              site.delta_depth_jac)
   else:
      return (site.age_jac,)
```

Figure 39

```
l = len(pccfg.list_sites)
result = [None for i in range(l)]

# Loop to submit corrected_jacobian in parallel using apply_async
for i, dlab in enumerate(pccfg.list_sites):
   result[i] = p.apply_async(nag_corrected_jacobian,(D[dlab],))

# Loop to get results of calls to nag_corrected_jacobian
for i, dlab in enumerate(pccfg.list_sites):
   if D[dlab].archive == 'icecore':
      (D[dlab].age_jac,
       D[dlab].airage_jac,
       D[dlab].delta_depth_jac) = result[i].get()
   else:
      D[dlab].age_jac, = result[i].get()

# Now loop over residuals_jacobian, residuals_jacobian1 and
# residuals_jacobian2
for i, dlab in enumerate(pccfg.list_sites):
   for j, dlab2 in enumerate(pccfg.list_sites):
      if j == i:
         jac[i,i] = D[dlab].residuals_jacobian()
      if j < i:
         jac[j,i] = DC[dlab2+'-'+dlab].residuals_jacobian2()
         jac[i,j] = DC[dlab2+'-'+dlab].residuals_jacobian1()
```

Figure 40

Secondly, a new function needs to be written that can be passed to `apply_async` that will call the `corrected_jacobian` functions for each `Site` object, e.g. see Figure 39. This function receives a `Site` object, calls the `Site.corrected_jacobian` function, and then returns the results. This function is necessary to copy result data from the processes in the pool, which have their own memory space, and the main Python process.

Finally, the code can be written as shown in Figure 40, where there are now 3 loops over `pccfg.list_sites`.

- The first loop uses `apply_async` to submit calls to `nag_corrected_jacobian` to the pool be executed asynchronously, i.e. in parallel. The input arguments are the function to be executed (`nag_corrected_jacobian`) and the input data required, in this case the site `D[dlab]`.
- The second loop gets the return values of `nag_corrected_jacobian` and updates the necessary values in the site object.
- The third loop calls `residuals_jacobian`, `residuals_jacobian1` and `residuals_jacobian2` in serial, as in the original code.

Figure 41 shows the time spent in the first two loops in Figure 40, i.e. excluding time in `residuals_jacobian`, `residuals_jacobian1` and `residuals_jacobian2`. The equivalent time for the original code on one OpenMP thread was around 50s (see Figure 42).

The speedup is 1.7 times, which is optimal based on the data in Figure, and is achieved on 2 processes. This is expected as the computation for the EDC site is the first to be submitted

| #processes | Time / s |
|---|---|
| 1 | 91.9 |
| 2 | 55.1 |
| 4 | 54.6 |
| 6 | 53.5 |
| 7 | 54.7 |
| 8 | 55.6 |

Figure 41

to the pool, with two processes the first process can execute this computation while the other sites are processed by the second process.

However, it is clear the overheads of Python's multiprocessing are considerable, the time on 1 process is over 40s longer than the original code on 1 thread. Hence, further attempts to use multiprocessing were abandoned at this point. With a larger, more balanced dataset, this route could still show some value.

### 8.4.2 Parallelisation using Numba prange loops

Numba is a just-in-time (jit) compiler for Python, designed to accelerate Python execution. It converts Python code to optimized machine code at runtime, which often runs significantly faster than the original code. Numba can also be used to parallelise execution.

The simplest way to use Numba is to insert the decorator `@jit(nopython=True)` or the equivalent `@njit` on the line before the function to be jit compiled. The `nopython=True` mode tells Numba to compile a function such that it will bypass the Python interpreter. The alternative `@jit(nopython=False)` or `@jit` can be significantly slower.

By default, Numba compiles each decorated function each time a Python program is run, and the time in jit compilation can add significantly to the run time of a code. The argument `cache=True` tells Numba to cache the compiled code for reuse, which offers best performance. In this case the compilation occurs only once, unless the code is modified.

Numba also implements a parallel `prange` loop, which can be used in place of Python's range loop, and it is this functionality which is of interest here. To use Numba's parallelism the argument `parallel=True` is used, which tells Numba to automatically parallelise execution when it can, including any `prange` loops.

There are considerable restrictions on what Numba supports, and it often isn't obvious from the documentation what will work and what won't. This made it challenging to convert Paleochrono class functions to work with Numba. In order to test the concept as quickly as possible, the approach adopted was to create a standalone version of the `Site.corrected_jacobian` computation, with a function to copy the necessary data from the `Site` objects into Numba `Lists()` for the part of the code to be jit compiled.

We have written a module nag.py which includes all the necessary transformation to adapt

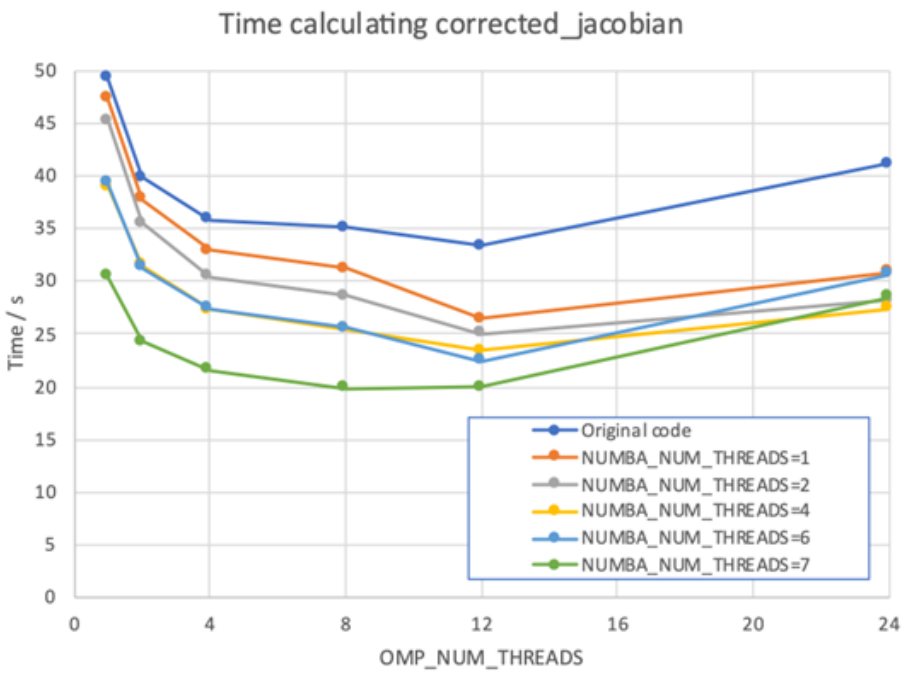

Time calculating corrected_jacobian
Time / s
OMP_NUM_THREADS
0
5
10
15
20
25
30
35
40
45
50
0
4
8
12
16
20
24
Original code
NUMBA_NUM_THREADS=1
NUMBA_NUM_THREADS=2
NUMBA_NUM_THREADS=4
NUMBA_NUM_THREADS=6
NUMBA_NUM_THREADS=7

Figure 42

the code to Numba's limitations.

By default Numba uses Intel's Threading Building Blocks (TBB) threading runtime, and provides methods to control the number of the TBB threads, e.g. the environment variable `NUMBA_NUM_THREADS`. Hence it is possible to independently select the number of TBB threads used by Numba in the parallel loop, and also the number of OpenMP threads used by MKL.

The time spent in the `corrected_jacobian` computation for different thread counts is shown in Figure 42, which shows the expected optimal speedup of around 1.6 times when using 7 Numba threads. Unlike the Python multiprocessing data in Figure 10, maximum speedup is not seen on 2 threads, presumably because of the way the work is distributed over two TBB threads.

It also shows the Numba version of the code with one TBB thread is faster than the original code. Additional speed up is achieved when adding OpenMP threads, with the best time obtained with 7 Numba threads and 8 OpenMP threads.

Unfortunately, the POP tracing tools do not support TBB, hence a deeper analysis of the performance of the modified code was not possible.

## 8.5 Conclusion and lessons learned

A performance analysis of the Paleochrono code using the AICC2012-Hulu dataset identified the main performance bottleneck was a lack of parallelism; there is too much time in serial computation. Further analysis using the Python profiler identified limited scope for improvement as the computation is split over several loops, and the most expensive loop has considerable load imbalance.

Speed up via parallelism of this loop using Python's multiprocessing was unsuccessful. The loop could be parallelised easily enough, but the overheads were greater than the benefits.

Speed up using Numba's `prange` parallelism was demonstrated. This used a standalone version of the `Site.corrected_jacobian` computation, and functions to copy the necessary data from the site objects into Numba `List()` objects. The speedup was shown to be optimal for this dataset, and better speedup may be seen for larger datasets. The best speedup is seen for a combination of TBB and OpenMP threads.

Now that the concept has been proven, it is worth revisiting the design choices made when writing Paleochrono, to see if the code can be rewritten to make more of it compatible with Numba. Alternatively, the objective function could be written in C with OpenMP parallelism, and then called from Python.

# 9 Report on Activity CODE_G

**Keywords:** domain decomposition, Schur complement, matrix factorization, Pardiso library, Math Kernel Library

## 9.1 Description of the Application

CODE_G is a finite element code written in C++ using MPI for parallel and distributed computing.

## 9.2 Previous Assessments and Recommendations

CODE_G was inspected within a performance audit (POP2_AR_104). The audit revealed a load imbalance issue caused by a different size of computational meshes assigned to individual processes. The iterative solver used within the application requires more iterations with an increasing number of subdomains/processes. Therefore, running the application with more than 100 MPI processes does not pay off. Applying a preconditioner could possibly overcome this aspect. Transfer efficiency decreases with an increasing number of processes due to the increasing number of messages to and from the master process. Using MPI collective operations such as Gather and Scatter instead of sequential Send and Receive calls in Round-Robin fashion may mitigate this aspect. The application consists of the factorization phase and iterative phase. The factorization phase is the most time-consuming one, taking more than 7% of the total runtime. Since the factorization is a standard math operation that is implemented naively with no optimization in this application, we advised to make use of some classical math libraries (e.g. MKL).

## 9.3 PoC activities

### 9.3.1 Scope

As the CODE_G application solves mechanical problems using domain decomposition and Schur complement approach, a given problem is transformed into a global linear system

$$Ku = f \tag{2}$$

with symmetric positive definite matrix K. After the domain decomposition, the matrix is arranged such that local interior degrees of freedom are in the first block (denoted by $o$) and interface degrees of freedom are placed at the end (denoted by $r$):

$$K = \begin{pmatrix} K_{oo} & K_{or} \\ K_{ro} & K_{rr} \end{pmatrix}, u = \begin{pmatrix} u_o \\ u_r \end{pmatrix}, f = \begin{pmatrix} f_o \\ f_r \end{pmatrix}. \tag{3}$$

Then the matrix can be decomposed into the following $LDL^\top$ factors,

$$K = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} L_{11}^\top & L_{21}^\top \\ 0 & I \end{pmatrix}, \tag{4}$$

where

$$K_{oo} = L_{11}L_{11}^\top; \quad K_{ro} = L_{21}L_{11}^\top; \quad S = K_{rr} - K_{ro}K_{oo}^{-1}K_{or}. \tag{5}$$

Applying decomposition (3), the original linear system (2) can be solved by Algorithm 2 called Schur complement method.

**Algorithm 2** Schur complement method

```
1: procedure SchurMethod(K, f)
2:     L, L^⊤, S ← K                ▷ compute local factors and Schur complement
3:     L f̂ = f                      ▷ modify right hand side (forward substitution)
4:     S u_r = f̂_r                  ▷ solve global problem with dense matrix
5:     f̂_r = u_r                    ▷ use the global solution for the global part of rhs
6:     L^⊤ u = f̂                    ▷ solve local problems (backward substitution)
7:     return u
8: end procedure
```

In this Proof-of-Concept, we focus on the last part. As the naive implementation of a matrix factorization is not anyhow manually optimized in terms of the mathematical procedure itself nor its implementation (only automatic compiler optimization using `-O3` flag is applied) and since the respective wall-time for this part of the application is dominant, we propose to apply the Pardiso library that is a part of the MKL (Math Kernel Library) library and call the factorization implemented there.

In the application, a system matrix $K$ is stored in sparse skyline format. This matrix is factorized and further condensed into the Schur complement form with matrix $S$ that is stored in the dense format by the `skyline::ldlkon_sky()` function. We plan to transfer the input matrix into the compressed sparse row (CSR) format and call the PARDISO library for which we will prepare an appropriate interface. All these operations are performed on a single core for the original and proposed implementation.

**Use-case and evaluation metrics:** For this Proof-of-Concept we use one of input data files tested in the previous performance audit. We choose the test case with 36 MPI processes with no other parallelization, which fits within one compute node on the Barbora cluster at IT4I. An efficiency of our proposed modification may be measured in terms of the wall-time for the factorization phase. We expect to reduce the wall-time to only 10% – 5% of the original time.

**Target system:** The experiments will be performed on a single node of Barbora cluster at IT4Innovations, which is also one of the target systems for the application. A computational node consists of two sockets Intel Cascade Lake 6240 equipped by 18 cores each running with nominal frequency 2.6 GHz, maximal turbo frequency is 3.9 GHz. Caches are organized as follows:

- L1I Cache: 576 KiB, 18x32 KiB, 8-way set associative,
- L1D Cache: 576 KiB, 18x32 KiB, 8-way set associative, write-back,
- L2 Cache: 18 MiB, 18x1 MiB, 16-way set associative, write-back,
- L3 Cache: 24.75 MiB, 18x1.375 MiB, 11-way set associative, write-back.

Available instructions are:

- x86-64, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA3, F16C, BMI, BMI2, VT-x, VT-d, TXT, TSX, RDSEED, ADCX, PREFETCHW, CLFLUSHOPT, XSAVE, SGX, MPX, AVX-512.

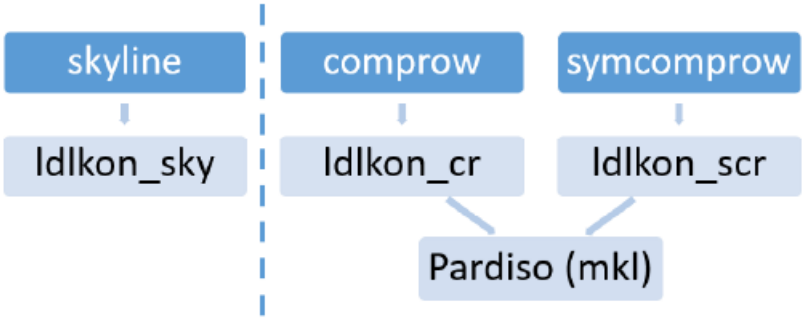



Figure 43: Class diagram with the original code structure on the left, and new implemented functions calling MKL on the right.

The original application is compiled with Intel MPI version 2018.4, Binutils version 2.31.1, and GCC version 8.2.0. For the PoC implementation, we use also MKL library in version 2020.0.166.

| | |
|---|---|
| GCC | 8.2.0 |
| MPI | 2018.04 |
| Binutils | 2.31.1 |
| MKL | 2020.0.166 |

Table 13: Environment setting

### 9.3.2 Implementation

For the Proof-of-Concept, we replace the original implementation of the Schur coplement decomposition that corresponds to line 2 in Algorithm 2 by calling Pardiso function from Math Kernel Library (MKL). Originally, system matrix is assembled and stored in symmetric `skyline` format which is provided by skyline class. The factorization is implemented in `skyline::ldlkon_sky()` function. Since the Pardiso requires sparse matrix in CSR format, we implemented this new version into existing comprow (compress row) and symcomprow (symmetric compress row) classes, described by files `cr.h` and `cr.cpp`, and `scr.h` and `scr.cpp`. We created `comprow::ldlkon_cr()` and `symcomprow::ldlkon_scr()` functions, where we call an interface into the Pardiso that is implemented in `cr_MKL` class described by `cr_MKL.h` and `cr_MKL.cpp` files. The respective class diagram is depicted on Figure 43 with the original skyline class on the left, and newly added functions for the comprow and symcomprow on the right.

A matrix in CSR format is described by three arrays and two additional numbers:

- `a`: array matrix entries are stored,
- `ci`: array where column indices of stored non-zero matrix entries are stored,
- `adr`: array where the first row entries are stored,
- `n`: number stands for number of rows (columns) of matrix,
- `negm`: number stands for number of entries of matrix,

for which the following relations hold,

- array `a` has `negm` components,
- array `ci` has `negm` components,

- array `adr` has `n+1` components,
- `adr[n]` is equal to `negm`.

We made use of the already implemented comprow and symcomprow classes corresponding to the CSR and symmetric CSR format. A user of the CODE_G application may change the matrix storage format by modifying the input files. The ten's line in the input file `*.in` specifies a storage format for the system matrix. The skyline format is denoted by value 2. For the compressed rows set value 10 and for the symmetric compressed row set 11.

We created `cr_MKL` class as the interface for the Pardiso library. Before the pardiso function is called, appropriate control parameters have to be set. It is done by the following code.

```
/* ------------------------------------------------------------------- */
/* .. Setup Pardiso control parameters. */
/* ------------------------------------------------------------------- */
for (i = 0; i < 64; i++) {
iparm[i] = 0;
}

/* ------------------------------------------------------------------- */
/* .. Initialize the internal solver memory pointer. This is only */
/* necessary for the FIRST call of the PARDISO solver. */
/* ------------------------------------------------------------------- */
for (i = 0; i < 64; i++) {
pt[i] = 0;
}

mtype = 11;         /* general sparse real matrix */
//mtype = -2;       /* general sparse symmetric real matrix */

iparm[1-1] = 1;     /* No solver default */
iparm[2-1] = 2;     /* Fill-in reordering from METIS */
iparm[10-1] = 8;    /* Perturb the pivot elements with 1E-13 */
iparm[18-1] = -1;   /* Output: Number of nonzeros in the factor LU */
iparm[19-1] = -1;   /* Output: Mflops for LU factorization */
//iparm[28-1] = 1;  /* Matrix checker */
iparm[35-1] = 1;    /* Zero-based indexing */
iparm[36-1] = 2;    /* Use Schur complement */
maxfct = 1;         /* Maximum number of numerical factorizations. */
mnum = 1;           /* Which factorization to use. */
error = 0;          /* Initialize error flag */
msglvl = 0;         /* Suppress printing statistical information */
```

Regarding the input matrix form and quality, it is important to set the appropriate Pardiso matrix type by `mtype` variable. We use 11 for general sparse real matrix (class `comprow`) and -2 for its symmetric form (class `symcomprow`).

For zero-based indexing used in C codes, set `iparm[35-1]=1`. The list of all available `iparm` parameters is available at Pardiso C reference manual. As we want to create a matrix factorization with the Schur complement, we set `iparm[36-1]=2`. Then the Schur complement is computed based on the permutation vector specifying indices related to the condensed part.

```
MKL_INT *perm = new MKL_INT[rows];
for (i = 0; i < rows - SCsize; i++)
  perm[i] = 0;
for (i = rows - SCsize; i < rows; i++)
  perm[i] = 1;
```

Individual actions within the Pardiso are driven by the `phase` variable. For the initial analysis and the numerical factorization, we set its value to 12. Then we call the pardiso library with appropriate input parameters, where `rows` specifies the size of the input matrix, `a`, `adr`, and `ci` specify the input matrix in CSR format, and `SCmat` is a vector, where the output Schur complement matrix is stored in dense format.

```
/* ------------------------------------------------------------------- */
/* .. Numerical factorization. */
/* ------------------------------------------------------------------- */
phase = 12; /* Analysis, numerical factorization */
PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
  &rows, a, adr, ci,
  &perm[0], &nrhs,
  iparm, &msglvl, &ddum, SCmat, &error);
```

After this step, we may use internally computed matrix factors to solve a system with the input matrix. At the end of the application, we need to free allocated data. To release the Pardiso internal memory, we set the `phase` parameter to -1 and call the Pardiso again.

```
/* ------------------------------------------------------------------- */
/* .. Termination and release of memory. */
/* ------------------------------------------------------------------- */
phase = -1; /* Release internal memory. */
PARDISO (pt, &maxfct, &mnum, &mtype, &phase,
  &rows, &ddum, adr, ci, &idum, &nrhs,
  iparm, &msglvl, &ddum, &ddum, &error);
```

For more comprehensive details about Pardiso configuration, we refer the reader to Pardiso documentation page and tips for using Pardiso. It is also very useful to take a look on examples available in the MKL installation directory.

## 9.4 Results

Measured statistics comparing the original factorization and proposed MKL Pardiso factorization are presented in Table 14. The measurement was obtained using Extrae and Paraver tools. A direct comparison is shown also on Figure 44 where a significant speedup is evident.

In Figure 44, we present three traces. The runtime of the application is along the x-axis, while individual processes are along the y-axis. Traces show states of the application, where black color corresponds to an ended application, the blue color is computation state, red color stands for communication or synchronization, and by yellow lines, the communication flow is depicted.

The runtime of the original factorization measured with the Extrae instrumentation is over 33 seconds. An equivalent factorization computed by the Pardiso for the general and symmetric matrix forms take only 1.1 and 1.2 seconds which corresponds to a speedup of 27.81 and 29.65, respectively. From the number of total instructions, it is clear that the main reason for the

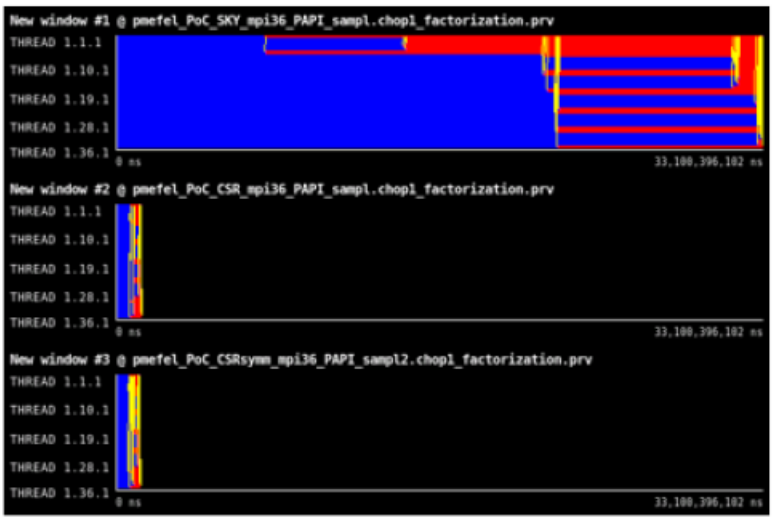



Figure 44: Traces of the orginal factorization with symmetric skyline matrix format, MKL factorization with general compressed row matrix format, and MKL factorization with symmetric compressed row matrix format.

presented improvement is a lowering the number of instructions by a factor of 21 and 23 for the general and symmetric matrix form, respectively.

Useful duration (total) is computed as a sum of individual useful durations for all processes. It contains only the computation. Measured speedup numbers 26.47 and 28.14 can be obtained by multiplying the other presented statistics, namely the total instructions, average IPC, and average CPU frequency.

Frequency is most likely decreasing because of the higher ratio of vector instructions that makes the base CPU frequency lower.

## 9.5 Conclusion

In the presented Proof-of-Concept, we propose to apply the Pardiso library, which is part of the math kernel library, for the computation of a matrix factorization and Schur complement.

| | original | MKL - CSR | MKL - sym. CSR |
|---|---|---|---|
| runtime [s] | 33.10 | 1.19 | 1.11 |
| (speedup) | (1.00) | (27.81) | (29.65) |
| useful duration (total) [$\mu$s] | 9.46e+09 | 3.57e+08 | 3.36e+08 |
| | (1.00) | (26.47) | (28.14) |
| useful instructions (total) | 3.75e+12 | 1.77e+11 | 1.60e+11 |
| | (1.00) | (21.15) | (23.44) |
| average IPC | 1.21 | 2.06 | 2.13 |
| | (1.00) | (1.69) | (1.75) |
| average frequency [GHz] | 3.26 | 2.40 | 2.23 |
| | (1.00) | (0.74) | (0.69) |

Table 14: Overview of the runtime, total useful duration, total useful instructions, IPC, and frequency with the relative ratio to the original code stated in brackets.

While the original implementation was using the symmetric skyline matrix form, we prepared the appropriate interface for the Pardiso library that is applied for both compressed row matrix formats, general and symmetric.

We have compared original and Pardiso implementations within the CODE_G application on a single node with 36 MPI processes where each process performs the factorization on its own local matrix. Tested input matrices had approximately 45000 rows with 800000 nonzero elements, and the size of the Schur complement matrix varies from 600 to 1200. For such a setting, we observe a speedup of factor 27.81 for general CSR matrix form and 29.65 for symmetric CSR matrix form. The speedup is highly dependent on the input matrix size and size of the Schur complement matrix, and thus it may vary for different settings. We suppose that it will grow for larger matrices and will drop for smaller matrices. The obtained speedup is even better than expected. We highly recommend the customer to apply the proposed Pardiso library for the matrix factorization and condensation (computation of the Schur complement matrix). After this significant speedup of the application, the customer may focus on some other parts of the code for optimization, for example, the solver part, if needed.

## 9.6 Lessons Learned

### 9.6.1 Recommendations for tool developers

There were no particular issues or missing features specific only to this PoC in the performance tools, still, some of the common obstacles hold. E.g., the communication modelling with the Dimemas tool is very sensitive to the consistency of process states. Often it is difficult to ensure the same state for all processes on the borders of the trace, typically due to cutting a subtrace within skewed iterations in Paraver. The inconsistency then leads to problems with obtaining the Serialization and Transfer efficiencies, possibly in the advanced stages of analysis or in a different part of the toolchain, e.g. in the Basic Analysis tool. Together with manipulation with large traces, which is inherently slow, the described issue may significantly prolong the analysis. Hence, if technically possible, it should be addressed at some point of the tools development.

### 9.6.2 Recommendations for programming model developers

In this PoC, we utilize the Intel MKL library, which is popular in the HPC community for solving many types of math problems. The PoC primarily demonstrates the benefits of using an optimized library instead of implementing own routines. But it also indicates that application developers do not use the proposed approach by default. There might be different reasons, e.g. the developers are not aware of the existence of the libraries or their dramatic performance benefit, or they consider their usage too complex. Therefore, we would encourage the (vendor) library developers to focus on dissemination activities, user-friendly API development and documentation presentation.

# 10 Report on Activity CODE_I

**Keywords:** machine learning, image processing, worksharing

## 10.1 Description of the Application

CODE_I is a machine learning application written in C++ and MPI+OpenMP paradigms to distribute and parallelize computations. The goal of the application is to find points of interest in images such that these can be characterized in a vector form. Given an input image, it is then possible to find similar and even identical pre-processed images.

## 10.2 Previous Assessments and Recommendations

In the earlier assessment POP2_AR_036, we reported that parallel performance had a very good shape.

We had the following observations:

- MPI3 (RMA) + OpenMP implementation working very well ensuring a very good scalability
- Substantial amount of I/O detected but well handled (not a bottleneck)
- Most of issues are located at the computation level. We identified a very poor vectorization level (globally)

Based on them, we recommended the following:

- Investigate vectorization issues
- Target the four main hotspots (1 function, 2 loops and 1 external function call) with optimisation opportunities

## 10.3 PoC activities

### 10.3.1 Scope

Our recommendations were oriented towards core level parallelism. The following recommendations were addressed:

- Main function hotspot: BoxIntegral function, described in listing 11 is the main hotspot of the program. Its body is very short and composed of many comparisons (min, max, bounds check). We checked if somehow we could simplify or group operations (i.e. applying vectorization) with no success. So we decided to try a different approach (algorithm).

```
1 float BoxIntegral( int row, int col, int rows, int cols,
2                    const int nbc, const int nbl, const float *pp  )
3 {
4     const int c1 = std::min( col, nbc ) - 1;
5     const int c2 = std::min( col + cols, nbc ) - 1;
6     const int r1 = std::min( row, nbl ) -1;
7     const int r2 = std::min( row + rows, nbl ) -1;
8
9     float A(0.0f), B(0.0f), C(0.0f), D(0.0f);
```

```
10
11      if (r1 >= 0 && c1 >= 0) A = pp[ r1*nbc +c1];
12      if (r1 >= 0 && c2 >= 0) B = pp[ r1*nbc +c2];
13      if (r2 >= 0 && c1 >= 0) C = pp[ r2*nbc +c1];
14      if (r2 >= 0 && c2 >= 0) D = pp[ r2*nbc +c2];
15      return std::max(0.f, A - B - C + D);
16  }
```

Listing 11: BoxIntegral function in mkII.cpp lines 66-81

- External exponential call: The second hotspot function is a huge amount of calls to the exponential routine from *Surf::gaussian()* in mkII.cpp at line 1055. We have to check if the values (parameter) are similar (very few) to try to setup a memorizing mechanism and also check if we can use less precision (does not change the app results)

- Grouping calls to main hotspot function: The main loop hotspot in *getDescriptors()* function contains a lot of calls to the main program hostspot (*BoxIntegral()*) as shown below in listing 11. We have to try to specialize these calls to *BoxIntegral* into one function to expose to the compiler optimizations opportunities (redundant calls, intermediate values, group operation like FMA and more generally vectorization).

```
1  Dxx = BoxIntegral(r-l+1, c-b, 2*l-1, w, nbc, nbl, pp)
2        - BoxIntegral(r-l+1, c-l/2, 2*l-1, l, nbc, nbl, pp)*3;
3  Dyy = BoxIntegral(r-b, c-l+1, w, 2*l - 1, nbc, nbl, pp)
4        - BoxIntegral( r-l/2, c-l+1, l, 2*l-1, nbc, nbl, pp)*3;
5  Dxy = BoxIntegral( r-l, c+1, l, l, nbc, nbl, pp)
6        + BoxIntegral( r+1, c-l, l, l, nbc, nbl, pp)
7        - BoxIntegral( r-l, c-l, l, l, nbc, nbl, pp)
8        - BoxIntegral( r+1, c+1, l, l, nbc, nbl, pp);
9
```

Listing 12: BoxIntegral Calls

- Accessors not being inlined and costly: A smaller hotspot but still significant is the calls to *GetDimX/GetDimY* functions that returns the dimensions of a given image. After checking the source code it appears that these accessors could be called earlier in the call chain. The main idea here is to perform these calls at a higher level (in the surrounding loop or function) to drastically decrease the total number of calls. Hence just passing values with an additional parameter. See section for the implementation details.

- Computing greyscale: another hotspot even if less important (around 3%) was interesting to investigate. It's one logical operation but actually split into smaller operations. This idea here is to take advantage of a subtlety of the algorithm. Only the red component of the RGB image is used to compute they geyscale representation once for all.

- Enabling AVX512: by default even if avx512-core instructions are used on the AVX512 capable processors the compilers do not enable the full vector length. The vectorizer static cost model rarely considers it is a good idea ...

**Use-case and evaluation metrics:** We will be using a 50 tars (each containing 10K images) dataset on 4 nodes. We will use the standard configuration: 2 MPI ranks per node and 36 OpenMP threads per rank The target metric is walltime. It corresponds to an image throughput for the user.

**Target system:** We will use the user's system with the same environment:

- Platform: User's cluster (9x Skylake Xeon Gold 6154 3GHz 36c/72hc)
- Compilers: GCC9.1 / OpenMPI 4.0.1

### 10.3.2 Implementation

**BoxIntegral - Dealing with border image conditions (FILTER_IPS)** As seen in section 10.3.1 / *BoxIntegral()* function, it was not possible to find a better way to perform the costly comparisons in the function.

However after studying the rational behind the function we decided to propose a different solution. The idea behind this function is a moving window moving around a picture to compute a sum. The problem is that when the window is close to the borders, some part of it may be outside the image boundaries. Thus, causing an invalid array access.

We proposed to the user creating a halo (cloning the border pixels) around the image in order to avoid invalid accesses and remove any conditions. While the idea is considered as valid, the implementation is not straightforward because images are stored in a linear way. By linear we mean flattening a two dimensional array into a one contiguous dimensional array (hence pp[ X*nbc + Y] access in the user's code). The consequence is that the halo would be part of that linear representation of the picture. All the code parts dealing with picture traversal would have to be halo-aware to only deal with non-halo pixels. This transformation would require changing a lot of code everywhere in the application. The user proposed another equivalent solution that was easier to implement. The box is centered on points of interests. The idea is to remove the points of interest (IPS) that are too close to the border in order to avoid verifying if the box is outside the image limits.

Listing 13 depicts the transformation.

```
1    const int dims=signature.size();
2    int const nbc=int_img.GetDimX();
3    int const nbl=int_img.GetDimY();
4    unsigned char remv[dims];
5    memset( remv, 0, dims );
6    for( int rm=0; rm<dims; rm++)
7    {
8      if( signature[rm].x <VALBORD  || signature[rm].x > (nbc-VALBORD)
   )
9          remv[rm]=1;
10     if( signature[rm].y <VALBORD  || signature[rm].y > (nbl-VALBORD)
   )
11         remv[rm]=1;
12   }
13   for( int rm=0; rm<dims; rm ++)
14     if( remv[rm] )  { signature.erase(signature.begin()+rm); kill++;
   }
```

Listing 13: original code dealing with greyscale conversion

**Grouping calls to main hotspot function BoxIntegral (GBOX)** The main loop hotspot in *buildDet()* function contains a lot of calls to the main program hostspot (*BoxIntegral()*) depicted in listing 12. We have to try to specialize these groups (Dxx, Dyy and Dxy) of calls into one function to expose to the compiler optimizations opportunities (redundant calls, intermediate values, group operation like FMA and more generally vectorization).

We can see in listing 12 that many parameters are the same across all these 8 calls to *BoxIntegral()* (e.g. r – l +1, 2 * l -1, etc.). Grouping these calls naturally factorizes the common operations. Merging the bodies of these functions will eliminate redundancies (computations based on the same parameters) and enable grouping the same computations to leverage vectorization.

**Costly exponential call (FASTMATH)** Based on the profiling data, the second hotspot function consists in a huge amount of calls to the exponential routine (17%).

The first step was to collect all the (exponential) values for each MPI rank and check the number of distinct values. We detected 10K values out of $2.64 * 10^9$ calls to *exp()*.

We decided to implement a memorizing mechanism to cache the requested values and serve them back without calling *exp()*.

Listing 14 shows the code we introduced to replace the original code shown in listing 15.

```
1     int rank;
2     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3     float keyval = -(x*x+y*y)/(2.0f*sig*sig);
4     gfloat *key;
5     gfloat *val;
6     #pragma omp critical
7     val = (gfloat *)g_hash_table_lookup(rankexp[rank].exptable,&keyval)
      ;
8     if(val == NULL) {
9       key = (gfloat *)g_malloc0 (sizeof (gfloat));
10      val = (gfloat *)g_malloc0 (sizeof (gfloat));
11      *key = keyval;
12      *val = exp( -(x*x+y*y)/(2.0f*sig*sig));
13      #pragma omp critical
14      g_hash_table_insert (rankexp[rank].exptable, key, val);
15    } else {
16      rankexp[rank].exptable_hit++;
17    }
18
19    rankexp[rank].exptable_total++;
20    return 1.0f/(M_2PI*sig*sig) * *val;
```

Listing 14: Memoizing mechanism implementation (exp values cache)

```
1
2 return 1.0f/(M_2PI*sig*sig) * exp( -(x*x+y*y)/(2.0f*sig*sig));
```

Listing 15: Original exponential hotspot code

This transformation did not provide any performance gain. Our guess is that since the program is multithreaded and each rank has its own table we suffer for the mandatory critical section we have to add to access (read/write) the hashtable.

Another option was to use the *-fastmath* flag (GCC) in order to use a quicker *exp()* function at the cost of losing some precision. The user confirmed that in his case it was not a problem, simple precision is sufficient. Note that the intel (libimf.so) version of *exp()* does not seem to outperform GCC's, we saw no significant difference.

**Accessors not being inlined and costly (DIMXY)** A smaller hotspot but still significant is the call to *GetDimX()/GetDimY()* functions that returns the dimensions of a given image. We will try avoiding these calls by passing the values by parameters (caller function) and hoisting calls when in a loop.

Surprisingly the compiler does not inline these functions.

We changed all the function definitions that were using an image pointer (Picture *). Listing 16 shows an example of function signature modification while listing 17 shows how call sites are prepared.

```
#ifdef DIMXY
float BoxIntegral( int row, int col, int rows, int cols, const int nbc,
    const int nbl, const float *pp  )
#else
float BoxIntegral( Picture *int_img, int row, int col, int rows, int
   cols)
#endif
```

Listing 16: BoxIntegral new signature

```
#ifdef DIMXY
    const float *pp=img->GetMemoryPointer();
    const int    nbc=img->GetDimX();
    const int    nbl=img->GetDimY();
#endif
```

Listing 17: DIMXY values definition

**Greyscale transformation (GREY)** In the application's workflow, before processing an image, it must be converted to greyscale. The original implementation (listing 18) first converts an RGB image into to a Greyscale (*getGrey()* function). The resulting image is not meant for display but for processing (find point of interest). So it is then converted to a 0.0 to 1.0 range (from 0 to 255) by the divide function.

```
    Picture grey_255;
    Cropper.getGrey( Cropper, &grey_255 );
    grey_255.divide( grey_255, 255.f, &grey );
```

Listing 18: original code dealing with greyscale conversion

Looking at the source code we noticed that only one component (Red from RGB) was used in the computations. It is actually coherent because the value is the same in the three components and only one component is needed to perform points of interest computations. This is very important because it means that we can have a contiguous data structure because we only have to represent one component of the image (here arbitrarily Red). As shown in listing 19, the first function *CopyFrom()* we implemented creates a Picture object with only one component that will hold the grey data once computed. Then everything happens in *GreyNormalize()* described in figure 45 were we implemented AVX512 and AVX (256) variants to be compatible with recent and old x86 architectures.

```
Picture grey
grey.CopyFrom( Cropper );
Cropper.GreyNormalize( grey );
```

Listing 19: Optimized greyscale conversion

The main idea in this function is to apply greyscale conversion and division by 255 at the same time using data level parallelism. This is achieved by dividing the original values used for greyscale conversion by 255. Then we only need to perform multiplications and additions.

**Enabling AVX512** By default, compilers do not enable full width vector code when told to ena-ble AVX512. We have to force full width vectorization using the following flags:

```
int Picture::GreyNormalize( Picture &isGrey ) const
{
  assert( nbplane==3 ); // KO
  assert( val!=NULL ); // KO

  const size_t surf = nbcol*nbligne;
  const size_t surf2 = surf<<1;
  float *ppix = this->GetMemoryPointer();

  if( isGrey.val!=NULL )  free( isGrey.val );
  isGrey.nbcol   = nbcol;
  isGrey.nbligne = nbligne;
  isGrey.nbplane = 3;
  isGrey.val = (float*)malloc( surf*3*SIZEFLOAT );
  float *pgreypix = isGrey.val; // isGrey.GetMemoryPointer();


#if __AVX512F__

  const __m512  rdr=_mm512_set_ps( 0.0011722f, 0.0011721f, 0.0011721f, 0.0011721f, 0.0011721f, 0.0011721f, 0.0011721f, 0.0011721f,
                                   0.0011722f, 0.0011721f, 0.0011721f, 0.0011721f, 0.0011721f, 0.0011721f, 0.0011721f, 0.0011721f);
  const __m512  rdg=_mm512_set_ps( 0.0023005f, 0.0023004f, 0.0023004f, 0.0023004f, 0.0023004f, 0.0023004f, 0.0023004f, 0.0023004f,
                                   0.0023005f, 0.0023004f, 0.0023004f, 0.0023004f, 0.0023004f, 0.0023004f, 0.0023004f, 0.0023004f);
  const __m512  rdb=_mm512_set_ps( 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f,
                                   0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f, 0.0004491f);

  // => 2 puissance 4 = 16 (16 floats in 512 bits registers
  for( size_t linear=0; linear<(surf>>4); linear++, ppix+=16, pgreypix+=16 )
    {
      __m512 rp = _mm512_mul_ps( rdr, _mm512_loadu_ps( (float const *)ppix ) );
      __m512 gp = _mm512_mul_ps( rdg, _mm512_loadu_ps( (float const *)ppix+surf ) );
      __m512 bp = _mm512_mul_ps( rdb, _mm512_loadu_ps( (float const *)ppix+surf2 ) );
      __m512 rs = _mm512_add_ps( bp, _mm512_add_ps( rp, gp ));

      _mm512_storeu_ps( pgreypix, rs );
      _mm512_storeu_ps( pgreypix+surf, rs );
      _mm512_storeu_ps( pgreypix+surf2, rs );
    }

#elif __AVX2__
```

Figure 45: Greyscale optimized code.

```
GCC: -march=skylake-avx512 -mprefer-vector-width=512
ICC : -xCORE-AVX512 -qopt-zmm-usage=high
```

Listing 20: AVX512 Flags

## 10.4 Results

We implemented each transformation using "#ifdef" pragmas to be able to enable them incrementally and observe the benefit of each of it.

Figure 46 provides the timings obtained when applying our transformations. The table is built in an incremental fashion so that we can observe the impact of each transformation.

Each value of the Wall Time column is obtained by taking the median over 11 executions of the application incrementally activating each optimization (lines in the table). We use this protocol to avoid any strange execution behavior leading to a biased timing (slower execution).

## 10.5 Conclusion

After selecting the most relevent (highest ROI potential) code transformations to address, we applied six code optimizations based on the performance audit analyses. The most of the speedup came from new ideas that we had while doing this work, at the algorithmic level. Each of these was validated by the user.
Thanks to all these transformations we achieved a 3.2X speedup.

| Optimization name | Wall Time (s) | Speedup |
|---|---|---|
| Reference value without any optimization | 94 | 1 |
| AVX512 | 84 | 1,12 |
| FASTMATH | 75 | 1,25 |
| DIMXY | 62 | 1,5 |
| GBOX | 59 | 1,6 |
| GREY | 57 | 1,65 |
| FILTER_IPS | 30 | 3,2 |

Figure 46: Speedup summary table.

## 10.6 Lessons Learned

### 10.6.1 Recommendations for tool developers

From the parallel performance perspective tools were very effective in showing how well the code was behaving. From the computation point of view, even if MAQAO helped us understand the main bottlenecks of this code most of the transformations were algorithmic.

# 11 Report on Activity CODE_J

**Keywords: CFD, turbulence based on higher-order discretization**

## 11.1 Description of the Application

CODE_J is a computational fluid dynamics application written in Fortran langage and MPI paradigm to distribute and parallelize computations. The application uses the 2DECOMP&FFT library as an efficient interface to perform three-dimensional distributed FFTs. The goal of the application is to develop a parallel code for turbulence based on higher-order discretization.

## 11.2 Previous Assessments and Recommendations

In the earlier assessment POP2_AR_077, we had the following observations:

- Parallel efficiency is quite good overall around 70% to 80%
- MPI has a non negligible impact (mainly due to 2DECOMP&FFT library)
- Hopefully code scaling is close to perfect
- The main issue that we found is around memory consumption located in initialization part. User is allocating the same arrays for each process while his algorithm allows splitting the arrays. This prevents users dealing with problems larger than 256x256x512 3D meshs while the target is 512x512x4096. We also detected many memory leaks

Based on them, we recommended to mainly target memory allocation issues and look at vectorization issues.

- Memory allocation issues:
  We have to find a way to split the global memory allocation throughout the processes.
  We also have to track and fix memory allocation leaks
- Vectorization issues It would be interesting to investigate (look into code with user) why most of user's loops are only SSE vectorized (128bits) and not using full vector width

## 11.3 PoC activities

### 11.3.1 Scope

In the context of this PoC we address the both of memory allocation and vectorization issues. The most important issue, by far, is specifically the data-structure size limitation. Listing 21

```
1 allocate(  this%px(ni,ni,nk) , this%pxm1(ni,ni,nk) , this%vpx(ni,nk) ,
     this%flx(ni,2,nk) , this%fx(ni,2,nk) )
2 allocate(  this%py(nj,nj   ) , this%pym1(nj,nj   ) , this%vpy(nj   ) ,
     this%fly(nj,2   ) , this%fy(nj,2   ) )
3 allocate(  this%vpz(nk) )
```

Listing 21: m_solver_diag_iih_cyl.f90:329

This code is common to all MPI ranks. We clearly see that the same data structures (arrays) are duplicated over processes. Analyzing how the data structures are further used helps understanding why the user made this choice. There are three axes (cylindrical coordinates). On the X axis there is a square matrix coupled to a Z point resulting in a 3-dimentional array. Y is independent. Nothing prevents cutting the Z axis in subgroups.

As consequence, the memory issue (problem size that can be computed) which the application suffers from could be fixed by implementing this split along Z-Axis.

Note that fixing this issue should also provide some speedup because be-cause many computation (diagonalization phase) can be avoided.

**Use-case and evaluation metrics:** We will first use a 20 iterations 128x128x512 mesh on 40 cores. Then we will try a 512x512x2048 mesh on 200 cores.

The main metric will be the size of the problem that can be launched. It corresponds to the total memory footprint. We will also consider walltime for performance speedup evaluation.

**Target system:** We will use the user's system with the same environment :

- Machines : User's cluster (Skylake Xeon Gold 40 cores nodes)
- Compiler : Intel ifort 19 / IntelMPI 19

### 11.3.2 Implementation

We will now detail how we fixed the memory allocation issue and how we enhanced vectorization efficiency.

**Memory allocation issue** As explained in the previous section we found a way to fix the memory size issue by splitting one of the data structures (Z axis).
Listing 22 3 shows the new allocation scheme. This idea is to consider the X axis (radial) as the main axis and then split each part of Z dimension across the available ranks. So, the chunks are dynamically computed to fit the available MPI ranks.

```
allocate( self%pz_  (ph%zst(3):ph%zen(3),ph%zst(3):ph%zen(3), sp%zst(1)
    :sp%zen(1) ) )
allocate (self%pzm1_(ph%zst(3):ph%zen(3),ph%zst(3):ph%zen(3), sp%zst(1)
    :sp%zen(1) ) )
allocate (self%vpz_ (ph%zst(3):ph%zen(3), sp%zst(1):sp%zen(1) ) )
allocate (self%flz(ph%zst(3):ph%zen(3),2),self%fz(ph%zst(3):ph%zen(3)
    ,2))
```

Listing 22: m_solver_diag_hii_cyl.f90:578

As a consequence, the global memory footprint will decrease as the number of cores increase. In the previous version all the MPI ranks were requesting the total memory footprint. We then understand easily why the higher use cases quickly saturated the available memory.
Now not only we can split the total memory footprint but each MPI rank only allocates memory for one part of the computations.

```
    do i = sp%zst(1) , sp%zen(1)
       if (i.ne.sp%xen(1)) then
          tmp = -nu*d2
          do j=zstart(3),zend(3)
             tmp(j,j) = tmp(j,j) + real(self%wave(i))*dg_rm1(j,j)**2
```

```
6           end do
7           if (present(opt)) then
8              if (opt) then
9                 do j=zstart(3),zend(3)
10                   tmp(j,j) = tmp(j,j) - dg_rm1(j,j)**2*(-nu)
11                end do
12             end if
13          end if
14          call reduction_not_periodic( staggered, zsize(3), cs_gz(3),
    tmp, b1, bn, p, pm1, vpr,vpi, fl, f, cl )
15          self%pz_(:,:,i) = cmplx(p(:,:),0.0)
16          self%pzm1_(:,:,i) = cmplx(pm1(:,:),0.0)
17          self%vpz_(:,i) = cmplx(vpr,vpi)
18          !> comment flz ne varie pas avec k
19          self%flz = fl
20          self%fz = f
21          self%clz = cl
22          deallocate(p, pm1, vpr,vpi, fl, f)
23       end if
24    end do
```

Listing 23: m_solver_diag_hii_cyl.f90:584

Listing 23 depicts the rest of the processing in which we iterate over the Z portion for the current MPI rank and deal with its coupled matrix on the X axis.
From the computation performance perspective, this means that we have as many Z-groups times less computations (matrix diagonalization) to perform in this initialization phase. Whereas before, each rank had to perform the same computations. Figure 47 exhibits the memory allocation profile obtained with the new version (128x128x512 use case). The memory footprint is 38GB (182 – 144) / 40 = 950MB.

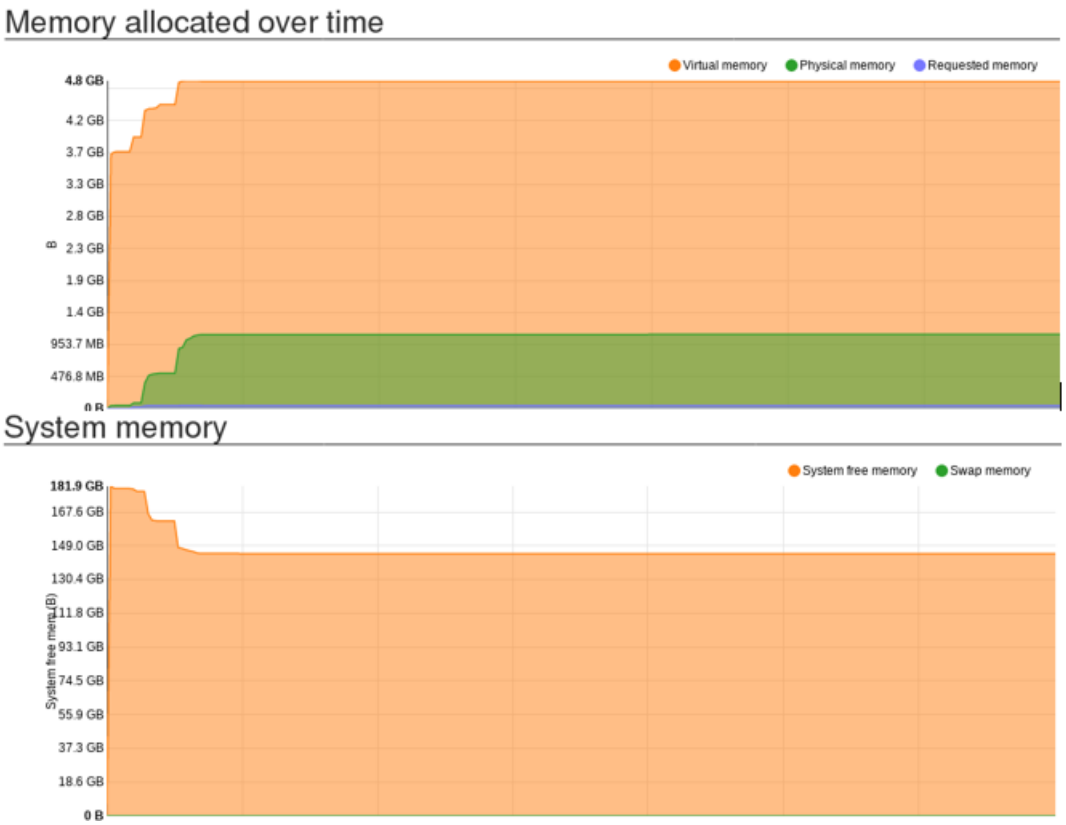



Figure 47: Memory consumption using the new version - 40 cores.
128x128x512 use case. Cores are mapped in 8x 5y 1z fashion.
Memory is spread across 8 groups (x axis).

The memory footprint is drastically reduced. Figure 48 shows a much bigger use case, 512x512x2048. The user was able to run the target 512x512x4096 on 8 nodes. We did not

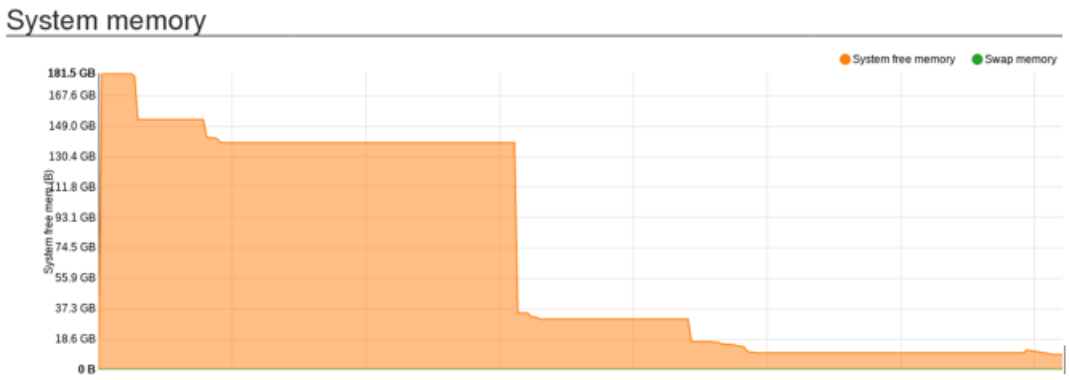
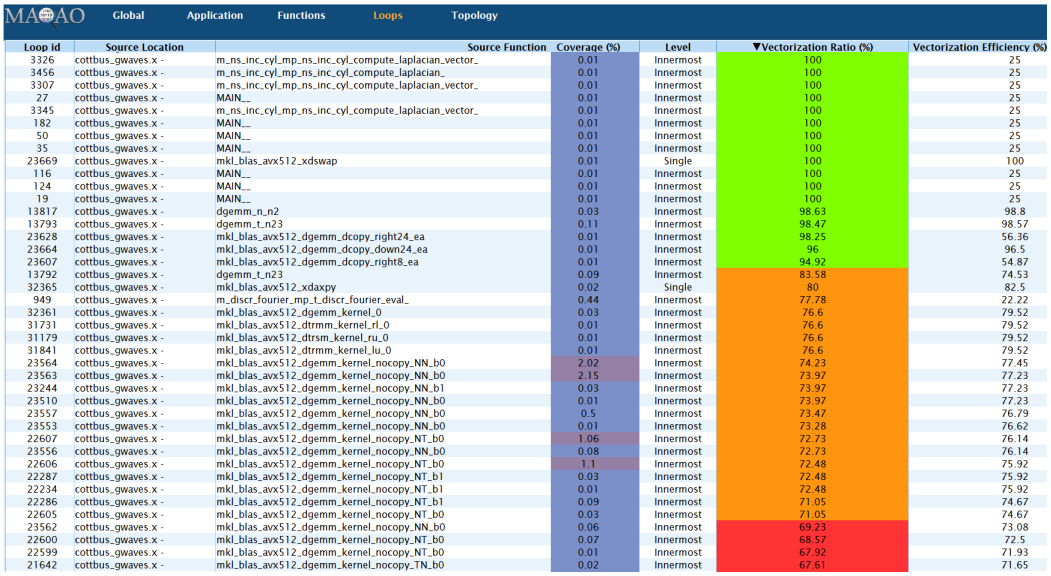



Figure 48: Memory consumption using the new version – 200 cores.
512x512x2048 use case. Cores are mapped in 20x 10y 1z fashion.
Memory is spread across 20 groups (x axis).

include here profiling information because the user performed a production run lasting many hours.

**Vectorization opportunities** Many loops in the user's code were vectorized but no fully efficient as shown in Figure 49. While vectorized (100% ratio), the efficiency is only 25%.



MAQAO Global Application Functions Loops Topology

| Loop id | Source Location | Source Function | Coverage (%) | Level | ▼Vectorization Ratio (%) | Vectorization Efficiency (%) |
|---|---|---|---|---|---|---|
| 3326 | cottbus_gwaves.x - | m_ns_inc_cyl_mp_ns_inc_cyl_compute_laplacian_vector_ | 0.01 | Innermost | 100 | 25 |
| 3456 | cottbus_gwaves.x - | m_ns_inc_cyl_mp_ns_inc_cyl_compute_laplacian_ | 0.01 | Innermost | 100 | 25 |
| 3307 | cottbus_gwaves.x - | m_ns_inc_cyl_mp_ns_inc_cyl_compute_laplacian_vector_ | 0.01 | Innermost | 100 | 25 |
| 27 | cottbus_gwaves.x - | MAIN__ | 0.01 | Innermost | 100 | 25 |
| 3345 | cottbus_gwaves.x - | m_ns_inc_cyl_mp_ns_inc_cyl_compute_laplacian_vector_ | 0.01 | Innermost | 100 | 25 |
| 182 | cottbus_gwaves.x - | MAIN__ | 0.01 | Innermost | 100 | 25 |
| 50 | cottbus_gwaves.x - | MAIN__ | 0.01 | Innermost | 100 | 25 |
| 35 | cottbus_gwaves.x - | MAIN__ | 0.01 | Innermost | 100 | 25 |
| 23669 | cottbus_gwaves.x - | mkl_blas_avx512_xdswap | 0.01 | Single | 100 | 100 |
| 116 | cottbus_gwaves.x - | MAIN__ | 0.01 | Innermost | 100 | 25 |
| 124 | cottbus_gwaves.x - | MAIN__ | 0.01 | Innermost | 100 | 25 |
| 19 | cottbus_gwaves.x - | MAIN__ | 0.01 | Innermost | 100 | 25 |
| 13817 | cottbus_gwaves.x - | dgemm_n_n2 | 0.03 | Innermost | 98.63 | 98.8 |
| 13793 | cottbus_gwaves.x - | dgemm_t_n23 | 0.11 | Innermost | 98.47 | 98.57 |
| 23628 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_dcopy_right24_ea | 0.01 | Innermost | 98.25 | 56.36 |
| 23664 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_dcopy_down24_ea | 0.01 | Innermost | 96 | 96.5 |
| 23607 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_dcopy_right8_ea | 0.01 | Innermost | 94.92 | 54.87 |
| 13792 | cottbus_gwaves.x - | dgemm_t_n23 | 0.09 | Innermost | 83.58 | 74.53 |
| 32365 | cottbus_gwaves.x - | mkl_blas_avx512_xdaxpy | 0.02 | Single | 80 | 82.5 |
| 949 | cottbus_gwaves.x - | m_discr_fourier_mp_t_discr_fourier_eval_ | 0.44 | Innermost | 77.78 | 22.22 |
| 32361 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_0 | 0.03 | Innermost | 76.6 | 79.52 |
| 31731 | cottbus_gwaves.x - | mkl_blas_avx512_dtrmm_kernel_rl_0 | 0.01 | Innermost | 76.6 | 79.52 |
| 31179 | cottbus_gwaves.x - | mkl_blas_avx512_dtrsm_kernel_ru_0 | 0.01 | Innermost | 76.6 | 79.52 |
| 31841 | cottbus_gwaves.x - | mkl_blas_avx512_dtrmm_kernel_lu_0 | 0.01 | Innermost | 76.6 | 79.52 |
| 23564 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0 | 2.02 | Innermost | 74.23 | 77.45 |
| 23563 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0 | 2.15 | Innermost | 73.97 | 77.23 |
| 23244 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NN_b1 | 0.03 | Innermost | 73.97 | 77.23 |
| 23510 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0 | 0.01 | Innermost | 73.97 | 77.23 |
| 23557 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0 | 0.5 | Innermost | 73.47 | 76.79 |
| 23553 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0 | 0.01 | Innermost | 73.28 | 76.62 |
| 22607 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0 | 1.06 | Innermost | 72.73 | 76.14 |
| 23556 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0 | 0.08 | Innermost | 72.73 | 76.14 |
| 22606 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0 | 1.1 | Innermost | 72.48 | 75.92 |
| 22287 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NT_b1 | 0.03 | Innermost | 72.48 | 75.92 |
| 22234 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NT_b1 | 0.01 | Innermost | 72.48 | 75.92 |
| 22286 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NT_b1 | 0.09 | Innermost | 71.05 | 74.67 |
| 22605 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0 | 0.03 | Innermost | 71.05 | 74.67 |
| 23562 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0 | 0.06 | Innermost | 69.23 | 73.08 |
| 22600 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0 | 0.07 | Innermost | 68.57 | 72.5 |
| 22599 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_NT_b0 | 0.01 | Innermost | 67.92 | 71.93 |
| 21642 | cottbus_gwaves.x - | mkl_blas_avx512_dgemm_kernel_nocopy_TN_b0 | 0.02 | Innermost | 67.61 | 71.65 |

Figure 49: Loop profile using MAQAO – Vectorization metrics before fixing vectorization efficiency.

We have to tell the compiler to force full width vectorization using the following flags (ICC in out case).

```
GCC: -march=skylake-avx512 -mprefer-vector-width=512
ICC : -xCORE-AVX512 -qopt-zmm-usage=high
```

Listing 24: AVX512 Flags

## 11.4 Results

After having fixed the memory allocation issue it is now possible to launch much bigger case. The problem we found not only forbade using cases bigger than 256x256x512 but also the number of nodes that could be used. Indeed, the memory footprint could not be spread among nodes.
The user was able to launch the target 512x512x4096 case.
The solution to fix the memory allocation issue also provided a significant performance gain of 2X on the 128x128x512 case we used on 1 node.
In a nutshell, not only the memory footprint can be spread among processes but computations within this phase also.

We also succeeded in enabling the highest vectorization efficiency but it did not provide significant performance enhancement. Most of the time spent in computation is relate to the MKL library which does automatically detect the processor's capabilities. The time spent in the user's loops is actually negligible when compared to the MKL. So even if the user's loops vectorization efficiency was enhanced there is no significant performance gain overall.

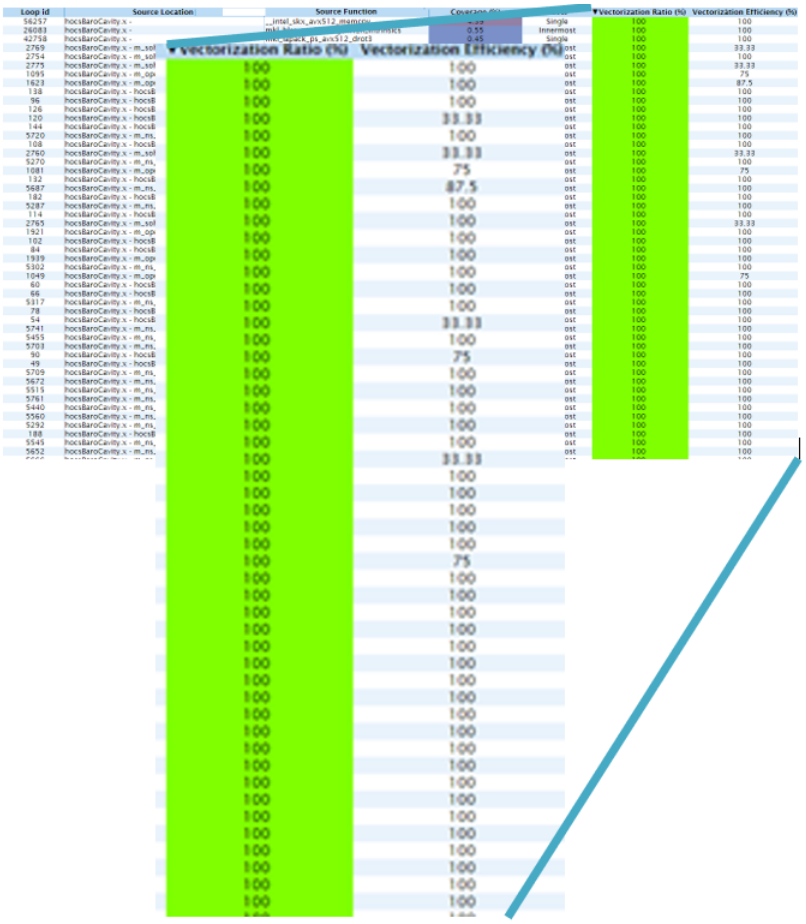


Figure 50: Loop profile using MAQAO – Vectorization metrics after fixing vectorization width (512 bits).

## 11.5 Conclusion

We found a problem with respect to memory allocation and fixed it. The user is now able to launch the target test case he needed. This is a major breakthrough for the user. Fixing the memory allocation issue also leveraged a 2X speedup on the 128x128x512 case we used on 1 node. Enhancing vectorization efficiency did not provide significant performance gain because the MKL, which is intensively used, automatically detects the architecture capabilities and use them (already efficient).

## 11.6 Lessons Learned

### 11.6.1 Recommendations for tool developers

### 11.6.2 Recommendations for programming model developers

# 12 Report on Activity Energy-efficiency analyses

**Keywords: Energy efficiency, green computing, READEX, MERIC, COUNTDOWN**

## 12.1 Description of the Applications

In this report we have analysed energy consumption of three different applications – BDDCML, SIFEL, and Blender Cycles. In each case we have tuned the underlying hardware to improve energy-to-solution, by reducing the computational resource, that is not the limiting factor for the application performance.

First of the applications is BDDCML, that is a Multilevel Balancing Domain Decomposition by Constraints solver, in which we analysed usecase of solving Poisson partial differential equation with unit right-hand side and homogeneous Dirichlet boundary conditions on a cubic domain with a unit edge.

Another application, which we have analysed the energy efficiency is a GPU accelerated rendering engine of the Blender.

This report also presents activity on the SIFEL application, that has been presented into details in the Section 9.

## 12.2 Previous Assessments and Recommendations

This energy efficiency analysis follows the previous assessments of the BDDCML (POP2_AR_071), SIFEL (POP2_AR_104), and Blender (POP2_AR_131). These applications were analysed for the energy efficiency because the customers asked for it. The energy efficiency activities are present in this report, because for each of them we were able to use a different approach.

The application is the most energy efficient if the computational hardware is fully utilized, and reduction of the resources would lead to runtime penalty, that would result in increase in energy consumption. If the underlying hardware is not fully utilized, it gives an opportunity to limit the resources to reach energy savings.

Several various energy-efficient approaches have been used to present possible that major energy savings are possible by tuning of the hardware parameters during a parallel application run to fit the application needs, that changes according the workload of the application phase.

The non-accelerated BDDCML applications has been analysed for its dynamic behavior and later optimised by dynamic switching of CPU core and uncore [8] frequencies using MERIC runtime system, which implements the dynamic tuning of the parameters for each region (in general any parf of the code, that is long enough to perform configuration change and reliable energy consumption measurement; usualy a function) originally presented in the Horizon 2020 READEX project.

Another MPI-only application was tuned using COUNTDOWN runtime system, that underscales CPU core frequency during specific MPI communications, to reduce power consumption during waiting time. This approach does not have any impact on performance, if the frequency is scaled back early enough not to influence the following computation.

Last but not least GPU-accelerated rendering using Blender Cycles path-tracing engine of the Blender application was optimised for energy-to-solution by tuning a frequency of the

[8] Intel uses ‘uncore‘ to refer to the subsystems in the physical processor package that is shared by multiple processor cores e.g., L3 cache or on-chip ring interconnect

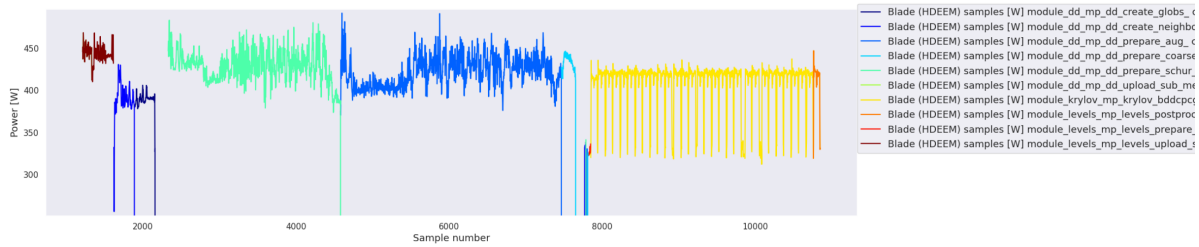



Figure 51: BDDCML power consumption with color identification of the instrumented regions. 1000 power samples corresponds to one second of the application runtime.

streaming multiprocessors of Nvidia A100 GPUs. The rendering engine and respective kernels usually takes almost 100 % of the runtime, so we have focused on this single region of the application.

### 12.2.1 BDDCML

BDDCML was analysed on a dual-socket Intel Broadwell processor (Intel Xeon E5-2697v4, 18 cores, 145 W TDP) powered cluster, so we have scaled the CPU core frequency in the range 1.2–3.6 GHz (2.6 GHz nominal frequency), and uncore frequency in range 1.2–2.8 GHz both with 0.1 GHz step. Intel RAPL (sum of PKG + DRAM power domains) was used for energy consumption measurement. For this test we focused on an application configuration when using 6 nodes.

In the application 10 regions were identified, while the shortest region lasts 1 % of the application runtime. Figure 51 shows power consumption of each of the regions. During runtime the requirements to the computational resources changes, which is visible from the changes in power consumption. In the optimal scenario we should see similar (stable) power consumption level during a single region.

If a static optimisation would be applied (selected hardware configuration is applied at the beginning of the application execution and remains unchained for the whole runtime) it is possible to reduce RAPL energy consumption about significant 25.36 % RAPL energy, while extend the runtime of the application about 4.15 %. When dynamically switching the CPU frequencies to the configuration that suits each region of the code best, the energy savings reach 30.99 % RAPL energy, and the runtime is extended about 0.7 % only.

The RAPL energy measurement includes power consumption of the CPU only, while the remaining parts of the computational board are excluded. The overall power consumption (excluding the additional system infrastructure) can be monitored from IPMI, which is a system, that takes one power sample per second, which is not enough for reliable energy measurement of an application, however it can be used to estimate power consumption of the blade, which is not covered by the RAPL. For the dual-socket Broadwell powered system the power so-called baseline is approximately 80 W. When the power baseline is included, the static energy savings are 18.87 % and the dynamic 22.37 %. These results are worse in compare to presented RAPL savings, however it corresponds to reality much better.

### 12.2.2 SIFEL

SIFEL was analysed on a dual-socket Intel Cascade lake (Intel Xeon 6240, 18 cores, 150 W TDP), which allows to scale the CPU core frequency in range 1.0–3.9 GHz (2.6 GHz nominal frequency) and uncore frequency in range 1.2–2.4 GHz. In this case we have performed week scaling test using 36 to 400 MPI processes. Also in this case we used Intel RAPL (sum of PKG

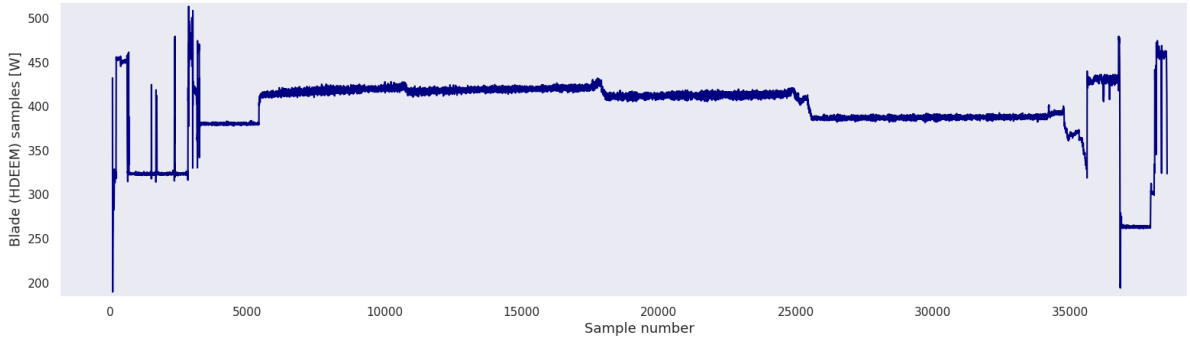



Figure 52: Power consumption of the SIFEL runtime

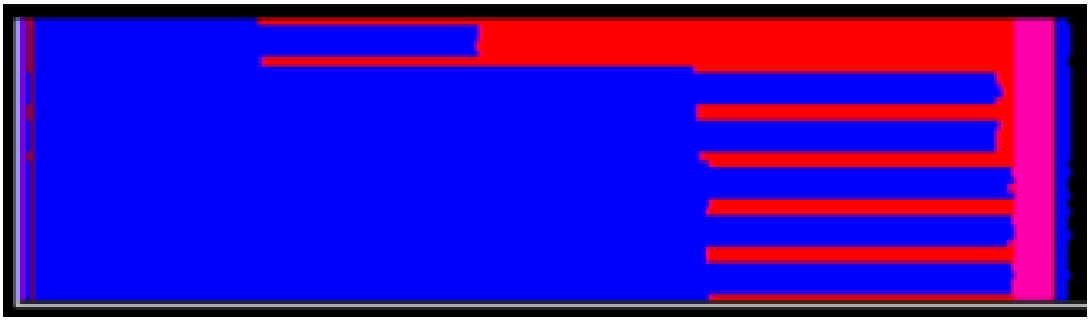


Figure 53: SIFEL single node execution trace.

+ DRAM power domains) for energy consumption measurement, no power baseline was applied in this case.

We have automatically identified the regions of the application and performed static binary instrumentation to analyse the application using MERIC runtime system. The analysis with the MERIC is possible, however inside the application is one region, that takes major amount of execution time, which computes local Schur complements of each subdomain, that is not possible to divide into smaller regions, which limits possible impact of the MERIC tuning. Moreover the Schur region complements computation shows major inter-process imbalance, which has impact also on power consumption, see Figure 52 and Figure 53. The imbalance is so significant, that the SIFEL is considered as a perfect usecase for an energy-efficient runtime system, that can exploit such behavior. Due to these factors we have selected the COUNTDOWN runtime system, that optimise the MPI communication phases, in which the SIFEL spent over 24 % of execution runtime.

On a single compute node when using MERIC we can save 13.8 % RAPL energy, while extending the runtime about 7 %. In the same single node configuration the COUNTDOWN brings 12.8 % RAPL energy savings, but with performance penalty 1.1 % only. When scaling the application the COUNTDOWN CPU core frequency tuning leads at least to 11.9 % energy savings, and the maximum savings reached 23.9 %.

### 12.2.3 Blender

The last evaluated application is a rendering application Blender Cycles, that was set to render Classroom Blender standard benchmark on a single computational node using GPUs to render the scene. For the accelerated applications no energy-efficient runtime systems exists. The MERIC supports to modify the GPU configuration based on CPU runtime, but not accord-

| #nodes | # processes | runtime penalty [%] | energy savings [%] |
|---|---|---|---|
| 1 | 36 | 1.1 | 12.8 |
| 2 | 64 | 1.2 | 11.9 |
| 3 | 100 | 2.5 | 13.8 |
| 7 | 225 | 3.9 | 18.5 |
| 16 | 400 | 0.9 | 23.9 |

Table 15: SIFEL application runtime extension and energy savings when scaling the problem size together with number of MPI processes.

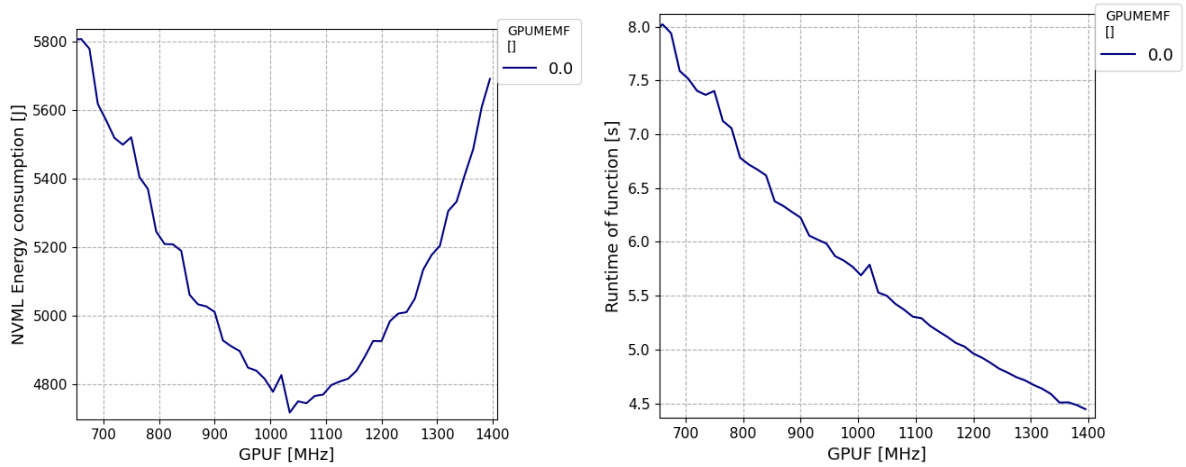



Figure 54: Overall energy consumption of 8 GPUs and execution time according specified GPU streaming multiprocessor frequency when executing Blender render.

ing the GPU runtime. In this case we have focused on the rendering itself, that is a single CUDA kernel with unified behavior, which eliminates the limitations of the current MERIC implementation.

The test were performed on IT4Innovations EuroHPC system Karolina which accelerated partition has two AMD EPYC Zen3 7763 (64 cores, 280 W TDP) and eight Nvidia Ampere GPUs (A100-SXM4, 400 W TDP) per computational node. In this case only frequency of the GPU Streaming Multiprocessors (SM) was scaled [9], while the CPU frequencies were controlled by the operating system. The GPU SM frequency reaches 1 410 MHz, but it can be reduced to 210 MHz with another 79 possible frequencies in between.

AMD CPU as well as Nvidia GPU has integrate system that measure energy consumption of the respective hardware. AMD CPUs implement similar RAPL interface for energy consumption monitoring as Intel, just the power domains are different. AMD Package power domain represents overall CPU energy consumption. In case of Nvidia the system does not have any specific name, but the measurement can be accessed using Nvidia Management Library (NVML), so we identify the measurement as *NVML energy consumption*.

Figure 54 shows almost linear performance degradation of the renderer when the SM frequency is down-scaled, on the other hand the same Figure present major energy consumption that can be reached when scaling the frequency from the maximum frequency of 1 410 MHz to the 1 035 MHz. Further frequency reduction leads to reduced power consumption, however the performance penalty impact cause increase in energy consumption of the GPUs. In the con-

[9] Nvidia GPUs with DRAM memory allows to control frequency of the memory, however the HPC GPUs that use HBM memory, do not have this possibility.

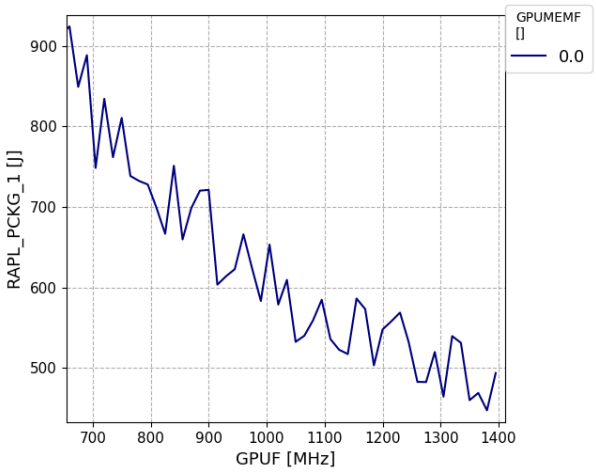



Figure 55: AMD RAPL Package energy consumption of one of the CPUs, only 4 of 64 cores are active, each communicates with assigned GPU, without additional workload for the CPU cores.

figuration, that brings the maximum energy savings (17.1 %), the performance penalty can be considered to be too harmful, since this configuration leads to 24.55 % runtime extension. The MERIC allows to specify a performance penalty limit, that is acceptable for the user. Table 16 present NVML energy savings for various performance penalties.

| **penalty limit** | **SM frequency [MHz]** | **runtime penalty [%]** | **energy savings [%]** |
|---:|:---:|:---:|:---:|
| 5 % | 1320 | 4.50 | 6.77 |
| 10 % | 1230 | 9.91 | 12.03 |
| 15 % | 1155 | 15.00 | 14.96 |
| 20 % | 1095 | 19.40 | 16.18 |
| no limit | 1035 | 24.55 | 17.10 |

Table 16: Blender Cycles NVML energy savings for various performance penalty limits.

Power consumption of each CPU fluctuates because of very low load on the CPU cores, however it keeps around 110 W. The AMD RAPL Package energy consumption mostly increases according the runtime extension caused by lower performance of the GPU when the frequency is down-scaled, as presented in the Figure 55.

## 12.3 Results

Three different HPC applications were analysed for energy consumption and available energy savings when parameters of the underlying hardware are tuned during the applications execution. For each of the usecases we have used a different approach, that fits the usecase the best.

Every usecase allows to save significant amount of energy. In case of the CPU-only applications the savings were possible with negligible performance penalty, which is not possible for the GPU-accelerated Blender application. In this case we have provided several configurations each extending the runtime less than a specified limit of 5, 10, 15, or 20 % respectively.

## 12.4 Conclusion

Three different usecases were presented, each analysed for the energy efficiency a different way. CPU application BDDCML and GPU application Blender were analysed using MERIC runtime system, but the SIFEL application was tuned using COUNTDOWN runtime system, developed under Horizon 2020 ANTAREX.

Under the Horizon 2020 project EUPEX the developers of the runtime systems MERIC (IT4Innovations) and COUNTDOWN (University of Bologna and CINECA) will work on co-tuning, so a user will have a opportunity to tune communication phases using COUNTDOWN, while the rest of the application runtime using MERIC.

Development of a runtime system for dynamic tuning of GPUs based on GPU workload, which is a modification of the Horizon 2020 READEX approach for GPU applications, is currently under development at IT4Innovations.

## 12.5 Lessons Learned

High energy savings can be reached by tuning selected hardware parameters. Limitation of the hardware resource may have major impact on the execution time of the application. User of the application should specify performance penalty limit, that is acceptable for the user. The trade-off ratio depends on kind of the underlying hardware and the application itself.

### 12.5.1 Recommendations for tool developers

The developers were informed how to use their applications with the runtime system and execute the application in the optimal configuration, that brings the requested energy savings.

# 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 Center (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] Niethammer, Ch. (2015) *Proof-of Concept work set-up.* Deliverable D5.1 of the POP project funded under the European Union's H2020 research and innovation programme GA No: 676553. Available at https://pop-coe.eu/further-information/deliverables

[2] Xiaoguang, R. and Xinhai, X. *AP-IO: asynchronous pipeline I/O for hiding periodic output cost in CFD simulation.* The Scientific World Journal, 2014:2356–6140, Hindawi (2014)

[3] Cerminara, M. et al. *ASHEE: a compressible, equilibrium-Eulerian model for volcanic ash plumes.* Geoscientific Model Development, 9:697–730 (2015)

[4] Castellano, M. *AP-IO: An Asynchronous I/O Pipeline for CFD code ASHEE.* Master thesis presented at SISSA: Master in High Performance Computing 2020

[5] Garcia, M. et al. *DLB: Dynamic Load Balancing.* Available at https://www.bsc.es/research-development/research-areas/programming-models/dlb-dynamic-load-balancing

[6] Garcia-Gasulla, M. et al. *Runtime mechanisms to survive new HPC architectures: a use case in human respiratory simulations.* The International Journal of High Performance Computing Applications, 34:42–56, SAGE Publications (2019)

[7] Garcia, M. et al. *Hints to improve automatic load balancing with LeWI for hybrid applications.* Journal of Parallel and Distributed Computing, 74, 2781–2794, Elsevier (2014)

[8] Hackenberg, D. et al. *HDEEM: High Definition Energy Efficiency Monitoring.* In Energy Efficient Supercomputing Workshop (E2SC), 1-–10, 11 2014.