Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Xeon Phi Performance for HPC-based Computational Mechanics Codes

M. Vázquez[a,b], F. Rubio[a], G. Houzeaux[a], J. González[a], J. Giménez[a], V. Beltran[a], R. de la Cruz[a], A. Folch[a]

[a]*Barcelona Supercomputing Center, Spain*
[b]*IIIA / CSIC, Spain*

In this paper we describe different applications we have ported to Intel Xeon Phi architectures, analyzing their performance. The applications cover a wide range. Alya, which is an HPC-based multi-physics code for parallel computers capable of solving coupled engineering problems in non-structured meshes. Waris, which is a simulation code for Cartesian meshes, that uses efficiently well-ordered data and well-balanced parallel threads. The last analysis is performed for a cornerstone of several simulation codes, a Cholesky decomposition method. The results are very promising, showing the great flexibility and power of Xeon Phi architectures.

Computational solid mechanics for MICs using Alya

Alya [1,2,3,4] is the HPC-based computational mechanics code developed in Barcelona Supercomputing Center. It is designed from scratch to simulate multi-physics problems with high parallel efficiency in supercomputers up to thousands of cores. Together with Code Saturne, Alya is one of the Computational Fluid Dynamics (CFD) codes of the PRACE benchmark suite. It has been ported to several supercomputers of the PRACE ecosystem and its scalability assessed. Alya simulates fluid mechanics (compressible and incompressible), non linear solid mechanics, combustion and chemical reactions, electromagnetism, etc. Alya parallel architecture is based in an automatic mesh partition (using Metis [5]) and MPI tasks. Additionally it has an inner parallelization layer based on OpenMP threads. Both can be combined in a hybrid scheme.

Intel Xeon Phi represents a very appealing architecture of codes such as ours for several reasons. Firstly, Alya works with non-structured meshes, where connectivity bandwidth is not uniform. These meshes are well suited for complex geometries. On the other hand, data access is more complex than in the case of structured meshes, where GPUs excels. Secondly, due to the Physics that Alya solves, the numerical schemes cannot guarantee that all the threads will have the same amount of work. Finally, coupled multi-physics requires a lot of flexibility to program the different subproblems. All these aspects, together with the development degree of Alya (around a million lines) make Intel Xeon Phis a better option than GPUs (at least a priori). From our point of view, Xeon Phis are small “clusters” on their own.

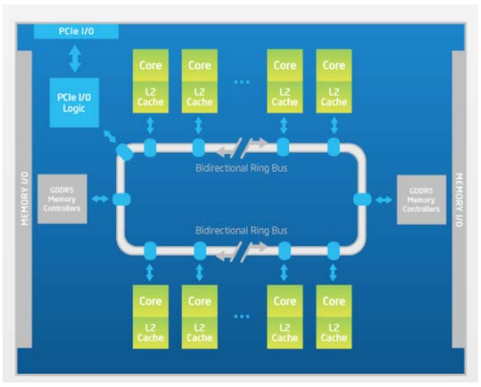

Figure 1. Intel Xeon Phi coprocessor block diagram (from Intel's web page).

After some initial tests checking pure OpenMP, pure MPI and hybrid, we have decided to focus in the pure MPI version in native mode (all compiled and running in the MIC). We have successfully ported Alya to MICs. Porting was not complex, no supplementary programming required. We ran several test cases of progressively more complexity. The largest cases are around 400.000 elements. In Alya, we have programmed both explicit and implicit schemes. At each time step, explicit schemes have an assembly phase and implicit schemes have both an assembly and a solver phase. Both phases have different programming features. The assembly is basically a set of gather/scatter operations and the solver is matrix-vector and vector-vector operations in sparse format. Therefore, Alya allows testing both schemes.

The selected Alya module is “Solidz”, which solves solid mechanics problems. The numerical scheme is FEM-based, in a total-Lagrangian formulation including dynamical terms and large deformations. The case chosen is an arterial aneurism wall deforming under stresses imposed on the inner wall coming from the blood dynamics.

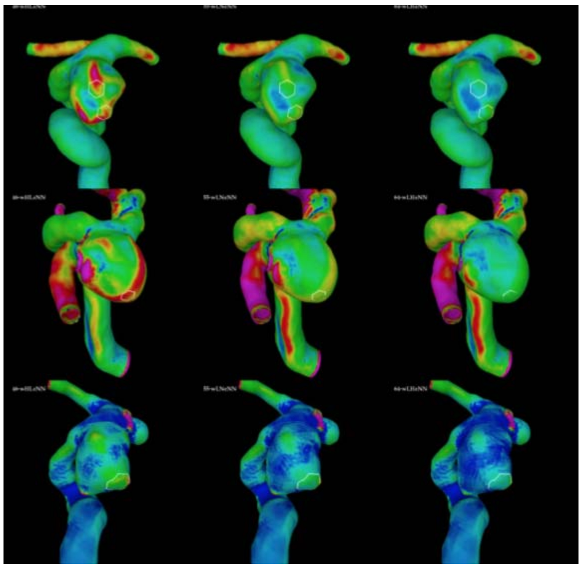

Figure 2. Examples of the aneurisms’ geometries, provided by J.Cebral (GMU, USA).

The tables below shows the scalability figures for the MPI strategy when solving the aneurism case using an implicit scheme. The implicit strategy is a conjugate gradient with a diagonal preconditioner. The scalability was measured using 1 MIC and 2 MICs for a 300.000 tetrahedra mesh.

As expected, the behaviour is very similar to that observed in a distributed memory cluster. The first column is the number of MPI tasks (i.e. Metis mesh partitions). The second column is the CPU time per time step (the simulation is for a transient problem). The third column the parallel efficiency, normalized by the first row “time per step” value. The last column shows the mean number of cells per task that comes from Metis partition. Metis partitions the original mesh maximizing load balance and minimizing inter-process communication, meaning that scalability cannot be linear al the way up, because while computing load diminishes, communication increases. For the implicit solid mechanics Alya module, we have observed that the “rule of thumb” is around 5000 cells per-node for this particular material model (see [1,2,3,4] for Alya scalability and performance analysis). Beyond this figure, i.e., with finer partitions, scalability starts to fall below linearity. The tables show the results for two different configurations: 1 MIC and 2 MICs. Although there is an overhead for inter-MICs communication, scalability shows a similar behaviour for both cases. It is observed that in both cases the “rule of thumb” is still valid. It is worth to mention that all efficiency values are computed as mean values over a certain amount of time steps, thus a small amount of statistical dispersion is expected, explaining slightly over-linear behaviour for some table entries.

1 MIC:

TASKS	TIME PER TIME STEP	EFFICIENCY	CELLS PER TASK
1 + 30	44.2366 secs	1.00	11600
1 + 60	21.1368 secs	1.05	5800
1 + 120	14.7152 secs	0.75	2900
1 + 240	28.6468 secs	0.49	1450

2 MICs:

TASKS	TIME PER TIME STEP	EFFICIENCY	CELLS PER TASK
1 + 15 + 15	59.4308 secs	1.00	11600
1 + 30 + 30	30.1814 secs	0.98	5800
1 + 60 + 60	20.3044 secs	0.73	2900
1 +120+120	26.4882 secs	0.28	1450
1 +240+240	89.6468 secs	neg	725

Using a second example of a solid mechanics problem, we analyse Alya performance using a profiler. The second problem is the mechanical deformation of a section of the vacuum vessel of ITER fusion generator (Figure 3). The mesh is made of 400.000 elements, but now it is hybrid, made of tetrahedra, prisms, hexahedra and pyramids.

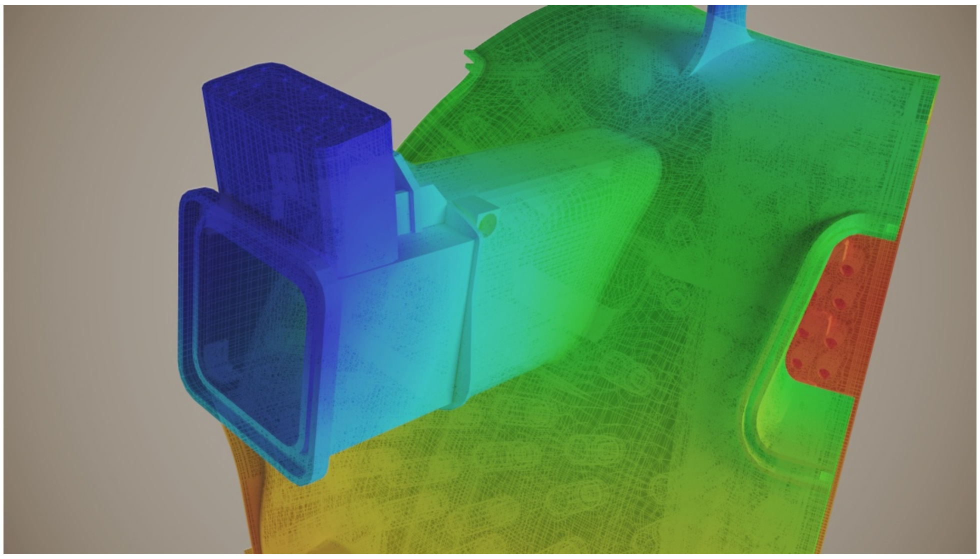

Figure 3. ITER fusion reactor vacuum vessel (close up), mesh courtesy of F4E consortium. Colour scale represent the displacement under its own weight.

In Figure 4 we can observe a Paraver [6] time-line of a single iteration of the main application loop, in an execution with 16 MPI tasks on a Xeon Phi accelerator. In this time-line, the X axis represents time, the Y axis represents the 16 tasks involved in the execution, the colours represent the MPI primitive executed by each tasks at each point of time, and yellow lines represent point-to-point messages. We can observe that the first MPI task, the master task, is mostly executing an `MPI_AllReduce` (in pink) collective operation. This operation waits till the end of the loop where it synchronizes with the worker tasks. On the other hand, these worker tasks are mostly computing (the light blue regions) and the end of the iteration they exchange data using the point-to-point `MPI_SendRecv` primitives (in brown), and finally synchronize with the master. In addition, we can also observe in this time-line that there is a certain load imbalance, as the different worker tasks computations have different durations

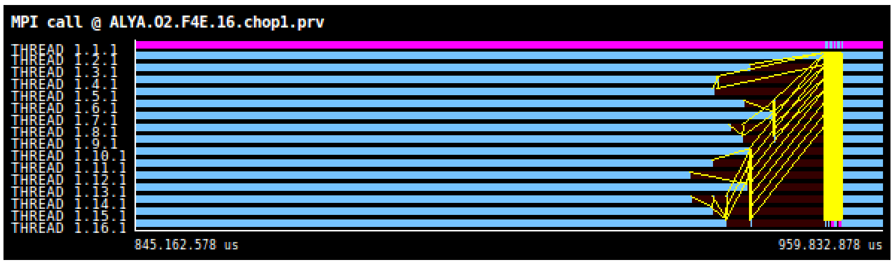

Figure 4 Time-line of MPI calls of ALYA

The scalability study of ALYA we conducted focused a strong scaling scenario. We executed the application using 16, 32, 61 (number of physical cores in the Xeon Phi), 64 and 128 MPI tasks. In these two last scales, we use more MPI tasks than physical cores, but in the Xeon Phi, each core is able to execute up to four threads, sharing internal resources. As we can see in Figure 5, the execution time of the different runs regularly decreased up to 61 tasks scale, and then it has an inflexion, when using more tasks than physical cores available.

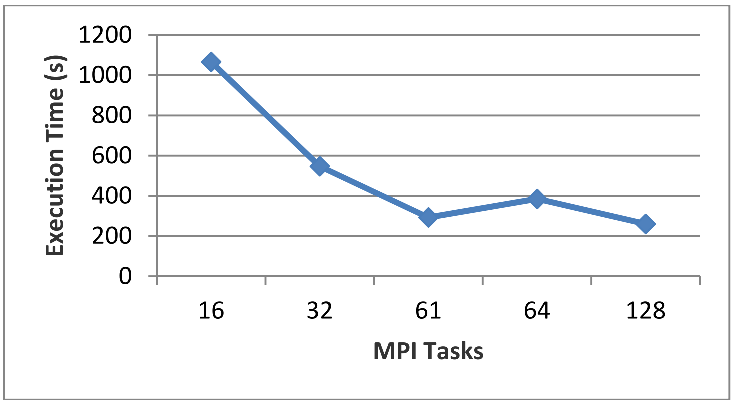

Figure 5 Execution time of a single ALYA iteration at different scales

So as to understand the scalability issues, we studied the application efficiency using the BSC model described in [7]. This model provides different performance factors like load balance (LB) or transfer efficiency (Tx) as a number between 0 (really bad) and 1 (perfect). Figure 6 contains a plot where the X axis represents the number of MPI tasks of the different executions, while the series are different factors of the model measured in ALYA. In this plot we can observe that the Parallelization Efficiency remains steady around 0.9 up to 61 MPI tasks scale, where it decreases to 0.75 (64 tasks) and 0.7 (128 tasks). This decrease is directly caused by the load balance (LB) degradation. It is interesting to note that the load imbalance observed in the time-line is not a critical issue when the number of tasks is less or equal to the number of physical cores, as it scores 0.9 at the three initial scales. Regarding the rest of the factors of the model, we can conclude that there is a minor code replication in the application, as the Instruction Balance (Inst BAL) is always close to 0.9, and also that the communications efficiency is really good, as it always reaches values close to 1. It is worth to remark that in both examples, the aneurism and the ITER vessel, we obtained very similar scalability figures, showing that they are independent of the problem.

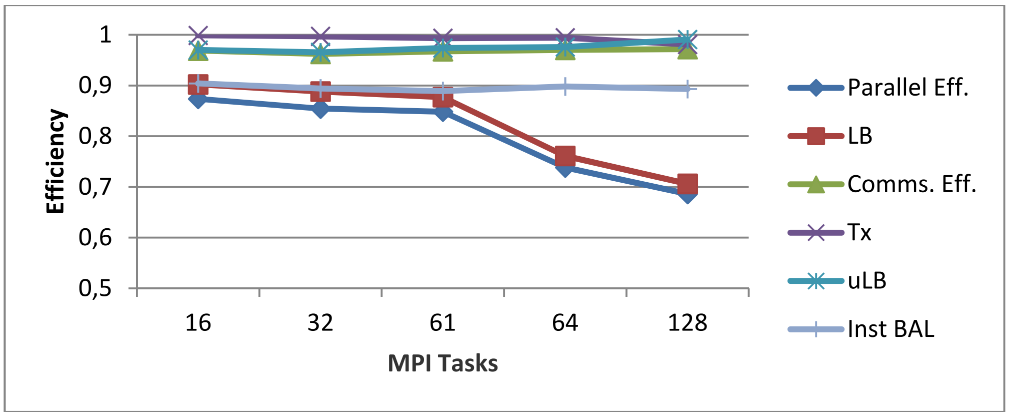

Figure 6 Efficiency model factors of ALYA

To evaluate the causes of the detected load imbalance, we analysed the application traces extracted on the different runs. In Figure 4 we can observe the time-line of two iterations of the application, executed with 128 tasks. The colour indicates the duration of the longest computation regions, using a gradient from light green (short computations) to blue (long computations). Black regions indicate where the application is communicating or executing short computations. In this time-line we can distinguish a set of 18 MPI tasks at the top whose durations are larger than the rest, causing the imbalance. Figure 8 contains a time-line of the same iterations where the colour gradient indicates the value of the Completed Instructions of each computation region (light green small amount of completed instructions, dark blue large amount of completed instructions). Comparing this time-line with the previous one, we cannot establish a correlation between the number of instructions completed in a region and the duration of such a region. In addition, this time-line points that almost all MPI tasks have a similar blue colour, which confirms the Instruction Balance measure by the efficiency model depicted in Figure 6. Finally, Figure 9 contains the time-line of the same iterations of the previous ones, where the colour indicates the value of the Millions of Instructions executed per Second (MIPS) on each computation region. In this case, we can establish a clear correlation with the time-line of Figure 9, as the initial 18 tasks, whose durations are larger, are the ones whose MIPS are smaller (in light green). With these three time-lines we can confirm that the instruction balance of the application is good, being the tasks pinning is the cause of the imbalances. Considering a round-robin based pinning of application tasks, in this execution we observe 6 cores executing 3 MPI tasks each (the initial 18 tasks), while the remaining 55 cores execute just two MPI tasks. As the initial 6 cores and 18 tasks share more resource than the rest (for example, the local caches or the Floating Point Units), their performance is lower.

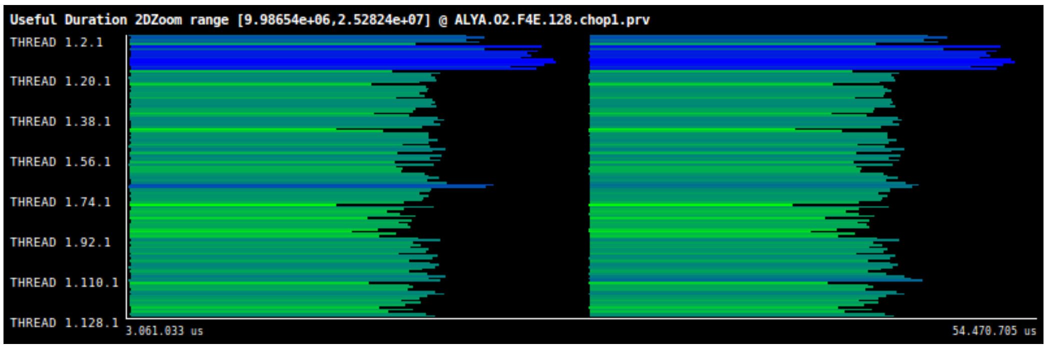

Figure 7 Time-line of computation regions duration

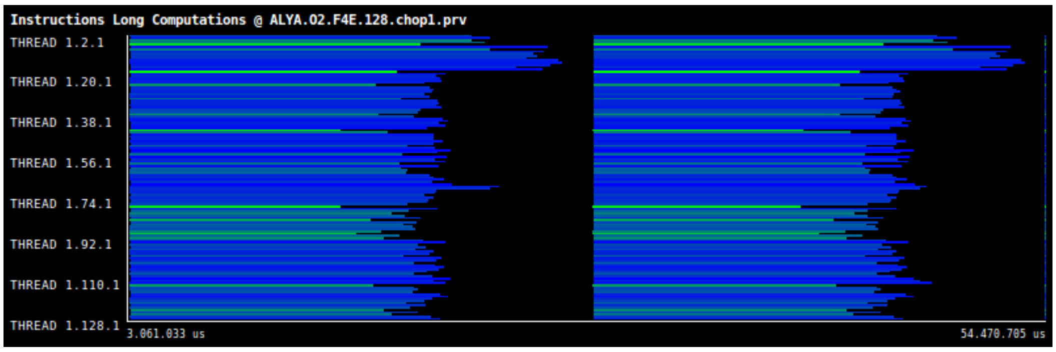

Figure 8 Time-line of Instructions Completed

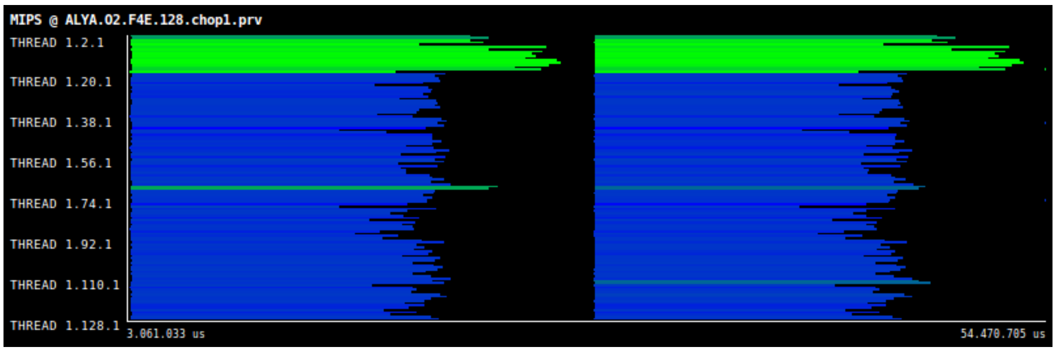

Figure 9 Time-line of MIPS

As a final comment of the study presented, we can affirm that, ranging in the number of physical cores the Xeon Phi provides, ALYA scales well when using its MPI capabilities, showing a similar scalability whether the MPI tasks are on a distributed memory machine or on Xeon Phis. The next steps are to further test pure OpenMPI and hybrid cases. We have not observed good scalabilities for pure OpenMPI further than a few tens of threads. The cause for that must be analysed. One possibility could be the implementation of the threads in Alya, in particular, in node renumbering. The hybrid case remains to be tested. Finally, performance on more than two MICs accelerators, located across multiple nodes, should also be tested.

Cholesky performance evaluation (OmpSs vs MKL)

Many OmpSs (OpenMP Super Scalar, developed at BSC) applications, such as the QR, QRCA, Cholesky, Hydro (MPI+OmpSs), …, have been ported and tested on the Xeon Phi platform without any change of the source code, showing reasonable performance on most cases. However, the Xeon Phi processor requires large amounts of parallelism to make the most out of all the cores available. A well-known technique to provide this extra level of parallelism is task nesting, which dramatically increases the available number of tasks, but requires efficient management of fine-grained tasks to get the best performance. This technique has been applied to the Cholesky decomposition to study its performance and scalability.

Figure 1, compares the scalability of three different configurations of the Cholesky decomposition. The X axis shows the number of hardware threads used and the Y axis the GFlop/s obtained. The problem size is set to 8192x8192 elements. The first configuration evaluated (dark blue) is the sequential version of Cholesky compiled and linked with the parallel version of the Intel MKL library, the second one (purple) is the OmpSs version of Cholesky linked with the sequential version of the MKL library, finally the last configuration (light blue) is the OmpSs version augmented with task nesting to generate more task level parallelism. As we can see, the OmpSs version with nesting is the best performing up to 64 hardware threads. With more than 64 hardware threads the MKL obtains the best performance. Finally, the OmpSs version without nesting only scales reasonably up to 32 cores, starting at that point the task parallelism available is insufficient to make the most of more hardware threads.

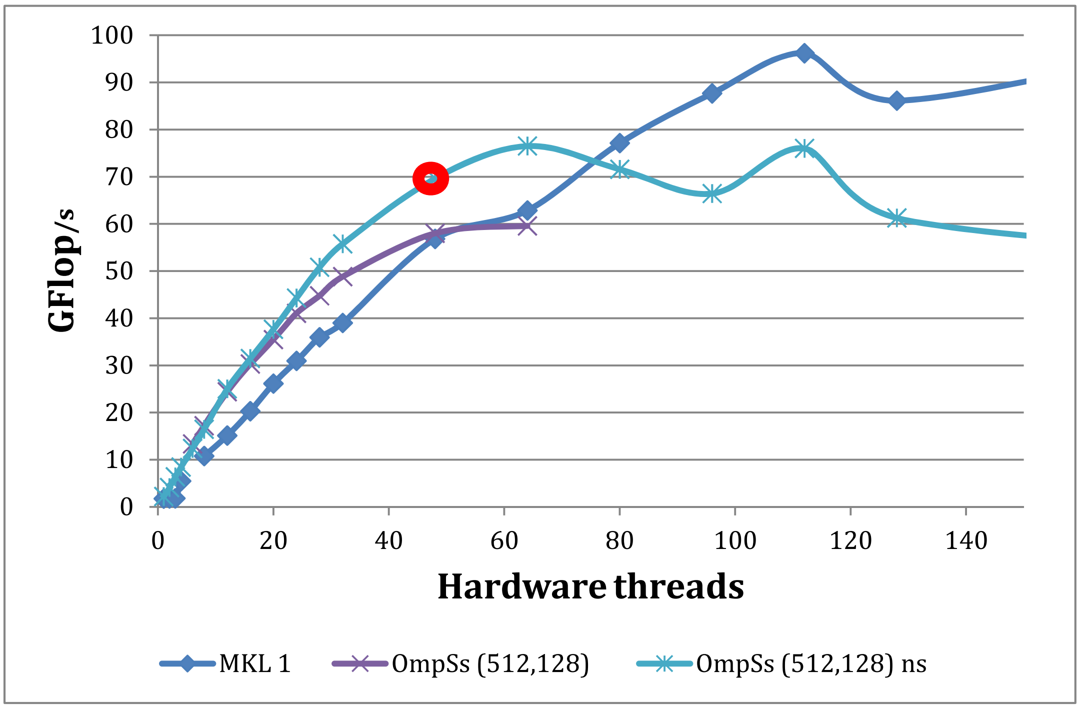

Figure 10: Cholesky scalability on KNF (OmpSs vs MKL)

With the Extrae instrumentation library and the Paraver visualization tool the different runs have been analysed to better understand what is going on. Figure 11 shows part of the timeline of the execution circled in red on Figure 10.

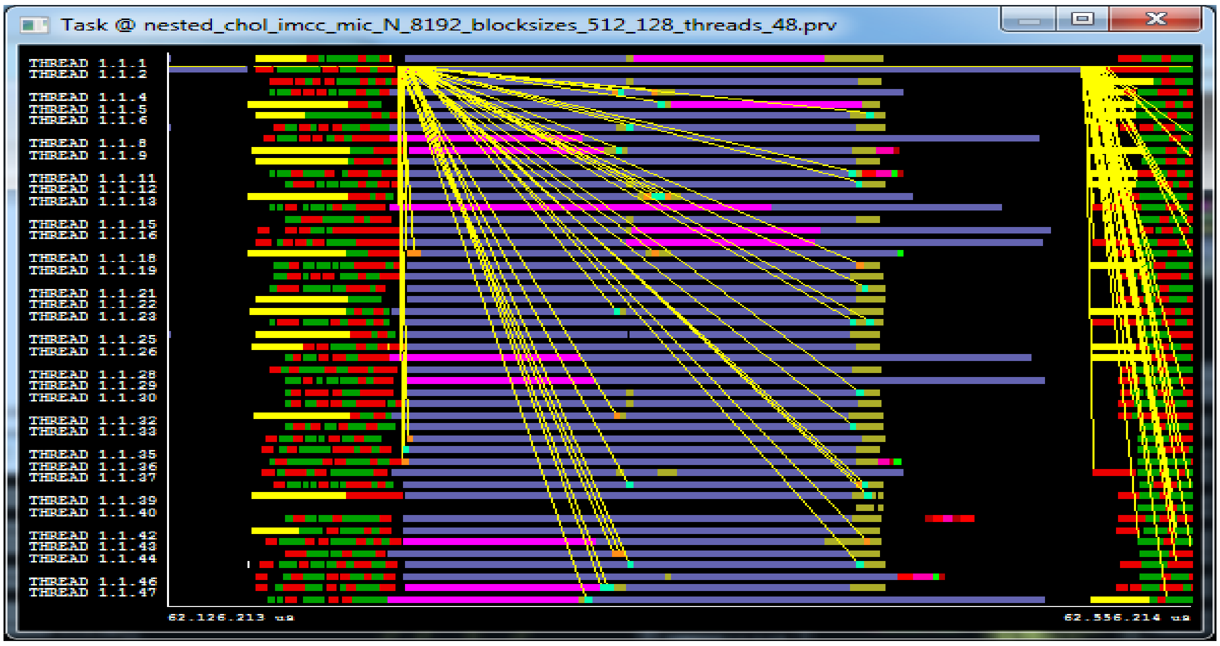

Figure 11. Tracefile of a Cholesky execution using OmpSs

As we can see there are still many opportunities to impove the performance of this algorithm on the KNC, because there are still many black zones (which mean that nothing is being executed on that hardware thread). The yellow lines mean tasks creations and the blue ones task executions. Hence on that trace we can see how thread 1.1.2 (the second one) is generating most of the tasks, which are then executed by the other threads. This same thread, at the end of the trace, executes a task that is on the critical path, so the other threads are blocked until this task is finished and new tasks are generated again.

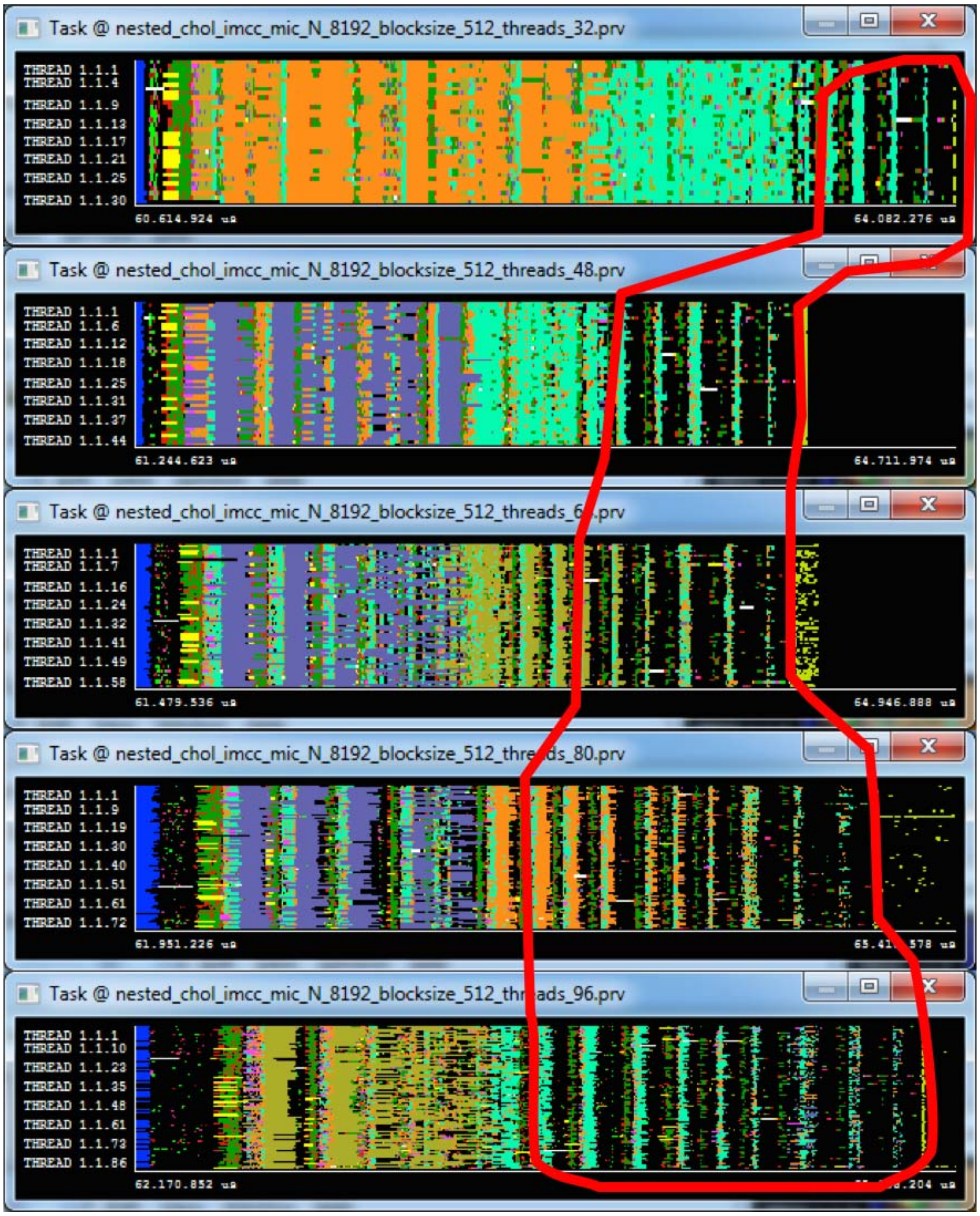

Figure 12. Trace file of Cholesky (OmpSs with nesting) with 32, 48, 64, 80 and 96 hardware threads

Figure 12 compares in a qualitative way five traces of the OmpSs version with nesting running with 32, 48, 64, 80 and 96 hardware threads respectively (from top to bottom). The timeline corresponds to the whole execution of the application. We can clearly see that the initial part of the trace scales well with the number of hardware threads. But the execution time of last part of the trace, marked in red, proportionally increases with the number of threads and it is the main cause of the limited scalability once the hardware threads are more than 64.

Performance evaluation of WARIS-Transport module on MIC architecture

WARIS [8] is an in-house multi-purpose framework focused on solving scientific problems using Finite Difference Methods as numerical scheme. Its framework was designed from scratch to solve in a parallel and efficient way Earth Science and Computational Fluid Dynamic problems on a wide variety of architectures. WARIS uses structured meshes to discretize the problem domains, as these are better suited for optimization in accelerator-based architectures. To succeed in such challenge, WARIS framework was initially designed to be modular in order to ease development cycles, portability, reusability and future extensions of the framework. In order to assess its performance on Intel Xeon Phi architectures, a code that solves the vectorial Advection-Diffusion-Sedimentation (ADS) equation has been ported to the WARIS framework. This problem appears in many geophysical applications, including atmospheric transport of passive substances.

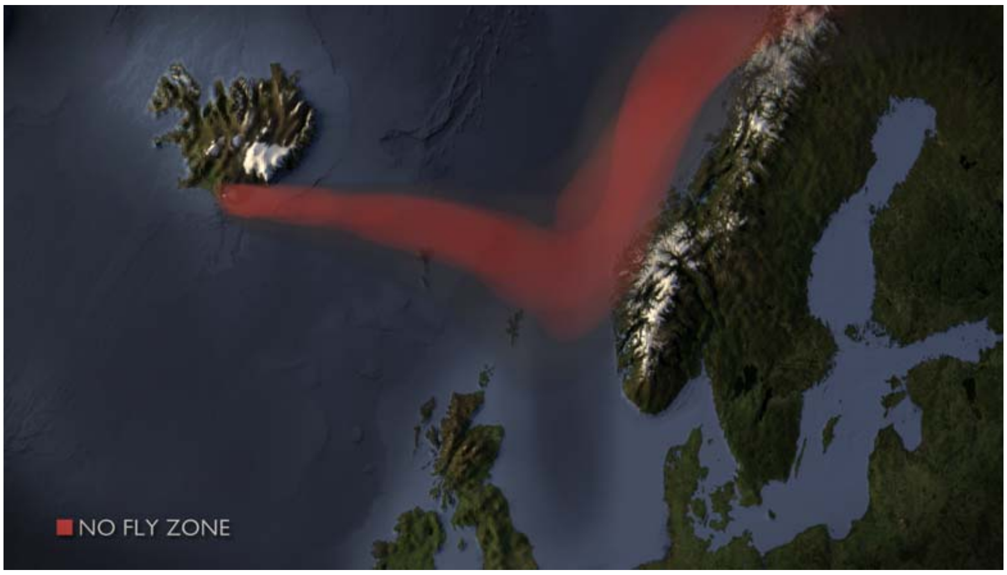

Figure 12: Output example of WARIS-Transport module

As an application example, the FALL3D model has been ported to WARIS framework. FALL3D [9] is a multi-scale parallel Eulerian transport model coupled with several mesoscale and global meteorological models, including most re-analyses datasets. Although FALL3D can be applied to simulate transport of any substance, the model is particularly tailored to simulate volcanic ash dispersal and has a worldwide community of users and applications, including operational forecast, modeling of past events or hazard assessment. Here, we investigate to which extent WARIS-Transport can accelerate model forecasts and could be used for ensemble forecasting. We focus on a paradigmatic case occurred during April-May 2010, when ash clouds from the *Eyjafjallajökull* volcano in Iceland disrupted the European airspace for almost one week, resulting in thousands of flight cancellations and millionaire economic loss. The following results include several performance techniques carried out in the WARIS-Transport module. These techniques are: SIMDization, blocking and pipeline optimizations of the explicit kernels. Furthermore, the Parallel I/O operations have been dramatically improved by implementing an active buffering strategy [10] with two-phase collective I/O calls.

The following table shows some preliminary results of the strong scalability obtained running the WARIS-Transport module on a single MIC. All those results were gathered with pure-MPI executions, no OpenMP was used for this preliminary results. In order to enable a large execution with many MPI processes, a large mesh was chosen for this test (32x32x2048 points for 1 MIC). Then, this domain is equally decomposed among all spawned MPI processes by cutting only through the least-unit dimension of the cartesian mesh (*Y* dimension of the *Z*x*X*x*Y* grid). Furthermore, *I_MPI_PIN_DOMAIN=auto:compact* environment variable was used in Intel MPI implementation to enforce a good balance among MPI processes and MICs cores.

Tasks (MIC nodes - MPI processes)	**Time per step (sec)**	**Efficiency**	**Mesh size per task**
1 – 1 *(mic0)*	520.89 *(261.19)*	1.00 *(1.00)*	32x32x2048
1 – 15 *(mic0)*	35.61 *(18.04)*	0.97 *(0.91)*	32x32x136
1 – 30 *(mic0)*	20.96 *(9.97)*	0.82 *(0.87)*	32x32x68
1 – 60 *(mic0)*	11.57 *(4.69)*	0.75 *(0.92)*	32x32x34
1 – 120 *(mic0)*	9.92 *(3.12)*	0.43 *(0.70)*	32x32x18
1 – 240 *(mic0)*	13.35 *(2.14)*	0.16 *(0.51)*	32x32x9

The information presented in the table shown above are: the number of MICs and MPI processes used, the total time per iteration, the normalized efficiency of the execution in terms of scalability and the average mesh size per MPI task respectively. Two different execution times and efficiency results are presented; including and not including (between parenthesis) initialization and parallel I/O costs. The former results belong to the initial iterations of the simulation where many I/O operations are performed in order to read meteorological data and write initial grid data. However, the latter results are related with the transient iterations where the execution time is dominated mainly by the computation of the explicit ADS solver and the computational part of the pre- and post-processing.
As can be seen, initialization and the parallel I/O costs degrades performance in a significant way for initial steps when many MPI tasks are launched. Besides, scalability results for transient iterations are fairly good up to 120 MPI tasks. However, given that large scale simulations require many steps, transient iterations will dominate over I/O-related iterations. Therefore, the bad I/O performance on many MPI tasks will be diminished by the remaining ones. As a conclusion, the preliminary results obtained in one MIC interface are very promising despite the low parallel I/O performance. Good scalability results have been obtained for transient iterations up to 120 MPI tasks opening new perspectives for operational setups in ash dispersion simulations which may include efficient ensemble forecast

References

[1] G. Houzeaux, M. Vázquez, R. Aubry and J.M. Cela. A massively parallel fractional step solverfor incompressible flows, Journal of Computational Physics 228: 6316–6332. 2009.

[2] G. Houzeaux, M. Vázquez, X. Saez and J.M. Cela Hybrid MPI-OpenMP performance in massively parallel computational fluid dynamics, Presented at PARCFD 2008 Internacional Conference on Parallel Computacional Fluid Dynamics Lyon, Francia. 2008

[3] P. Lafortune, R. Arís, M. Vázquez, G. Houzeaux. Coupled electromechanical model of the heart: Parallel finite element formulation Int. J. Numer. Meth. Biomed. Engng., 28: 72-86. doi: 10.1002/cnm.1494

[4] E. Casoni, A. Jérusalem, C. Samaniego, B. Eguzkitza, P. Lafortune, D. D. Tjahjanto, X. Sáez, G. Houzeaux and M. Vázquez. Alya: computational solid mechanics for supercomputers. Submitted to Archives of Computational Methods in Engineering. 2014.

[5] Metis Web site. http://glaros.dtc.umn.edu/gkhome/views/metis

[6] Performance Tools Website@BSC. url: http://www.bsc.es/paraver

[7] M. Casas, R.M. Badia and J.Labarta. *Automatic Analysis of Speedup of MPI Applications.* ICS'08

[8] Raúl de la Cruz, Mauricio Hanzich, Arnau Folch, Guillaume Houzeaux and José Maria Cela. *Unveiling WARIS code, a parallel and multi-purpose FDM framework.* ENUMATH 2013, Lausanne, Switzerland.

[9] A. Folch, A. Costa, and G. Macedonio, *Fall3d: A computational model for transport and deposition of volcanic ash,* Comp. Geosci. 35:6 (2009), 1334–1342.

[10] X. Ma, M. Winslett, J. Lee, and S. Yu, *Improving MPI-IO output performance with active buffering plus threads*, IPDPS '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 68.2–.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-261557.