Enabling the UCD-SPH code on the Xeon Phi

Christian Lalanne, Ashkan Rafiee, Denys Dutykh, Michael Lysaght, Frédéric Dias

▶ **To cite this version:**

Christian Lalanne, Ashkan Rafiee, Denys Dutykh, Michael Lysaght, Frédéric Dias. Enabling the UCD-SPH code on the Xeon Phi. 2014. hal-00927227v2

HAL Id: hal-00927227
https://hal.science/hal-00927227v2

Submitted on 22 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Enabling the UCD-SPH code on the Xeon Phi

Christian Lalanne[*c], Ashkan Rafiee[b,†], Denys Dutykh[ba], Michael Lysaght[*],

Frederic Dias[b]

[*]*Irish Center of High-End Computing, Dublin, Ireland*
[b]*University College Dublin, Dublin, Ireland*
[†]*Now at Carnegie Wave Energy Ltd, Perth, WA, Australia*
[a] *LAMA, UMR, 5127 CNRS, Universite de Savoie, Campus Scientifique, France*

Abstract

This white-paper reports on our efforts to enable an SPH-based Fortran code on the Intel Xeon Phi. As a result of the work described here , the two most computationally intensive subroutines (*rates* and *shepard_beta*) of the UCD-SPH code were refactored and parallelised with OpenMP for the first time, enabling the code to be executed on multi-core and many-core shared memory systems. This parallelisation achieved speedups of up to 4.3x for the *rates* subroutine and 6.0x for the *shepard_beta* subroutine resulting in overall speedups of up to 4.2x on a 2 processor Sandy Bridge Xeon E5 machine. The code was subsequently enabled and refactored to execute in different modes on the Intel Xeon Phi co-processor achieving speedups of up to 2.8x for the *rates* subroutine and up to 3.8x for the *shepard_beta* subroutine producing overall speedups of up to 2.7x compared to the original unoptimised code. To explore the capabilities of auto-vectorisation the *shepard_beta* subroutine was refactored which results in speedups of up to 6.4x for the *shepard_beta* subroutine relative to the original unoptimised version of the *shepard_beta* subroutine. The development and testing phases of the project were carried out on the PRACE EURORA machine.

1. Introduction

It has been known that bottom hinged Oscillating Wave Surge Converters (OWSCs) are an efficient way of extracting power from ocean waves. OSWCs are in general large buoyant flaps, hinged at the bottom of the ocean and oscillating back and forth under the action of incoming incident waves. The oscillating motion is converted into energy by pumping high-pressure water to drive a hydro-electric turbine.

The UCD-SPH code utilises the Smoothed Particle Hydrodynamics (SPH) method for modelling wave interaction with an Oscillating Wave Surge Converter (OWSC) device. The SPH scheme used in the UCD-SPH code is based on the SPH-ALE formulation [4][5]. The standard SPH method is a purely Lagrangian technique and the "particles" are moving nodes that are advected with the local velocity and carry field variables such as pressure and density. However, the SPH-ALE formulation is based on the solution of a moving Riemann problem in the Arbitrary Lagrangian-Eulerian context and hence the so-called particles are moving control volumes and not particles. As the fields are only defined at a set of discrete points, smoothing (interpolation) kernels are used to define a continuous field and to ensure differentiability.

The purpose of this project is to first introduce OpenMP parallelisation to the UCD-SPH code, to then enable the UCD-SPH code on the Intel MIC architecture and to then subsequently explore optimisations and modes of execution of the code on that architecture. This project corresponded to 3PMs of effort.

[c]Corresponding author. *E-mail address:* christian.lalanne@ichec.ie

All benchmarks and profiles of the baseline code and optimisations of the code were performed on EURORA, within a 2 processor, 8 core, Intel Xeon E5-2658, 2.10GHz, 16 GB RAM node with an Intel Xeon Phi 5120D, 8GB RAM co-processor attached. The compiler used in this project was the Intel Fortran compiler (ifort) v14.0.1 20131008. All the executables produced were compiled using the *-O3* optimisation level.

The code was developed by Dr. Ashkan Rafiee and has been validated extensively in numerous applications [6][7][8][9]. The version of the code that this project worked on is based upon the three-dimensional SPH approach and has recently been parallelised using an MPI based domain decomposition method.

2. Analysis of the sequential version of the code

The original code at the start of the project was already parallelised using only MPI, where the problem domain is partitioned across MPI processes along the dimension of the wave propagation (leading dimension). The entire domain is divided into cells where these cells contain local particles. These local particles interact with particles located in the same cell and also particles in the nearest neighbour cells. The code that was delivered as the test case for this project, was configured with a smoothing length of $h = 1.5dp$ where dp is the initial particle's spacing. The smoothing length (h) affects the number of interactions that every particle participates in, thus increasing this parameter increases the number of particles that every single particle interacts with, increasing the computation that the algorithm has to perform (size of the problem). It was found that increasing this parameter increases the parallelism available in the algorithm and the size of the problem to solve, thus different values of h were used during this project (*1.5dp, 2.5dp and 3.5dp*).

The code, is composed of a setup phase, where data is read from files and data structures are initialized, a main loop, where every iteration of this loop represents a time step, and a deallocation phase where system resources are released.The first task of the project was to measure the most computationally intensive functions in the code. The most computationally intensive phase is the main loop. Figure 1 shows a profile of the most computationally intensive child subroutines in the main loop of the original unoptimised code running with 1 MPI process.

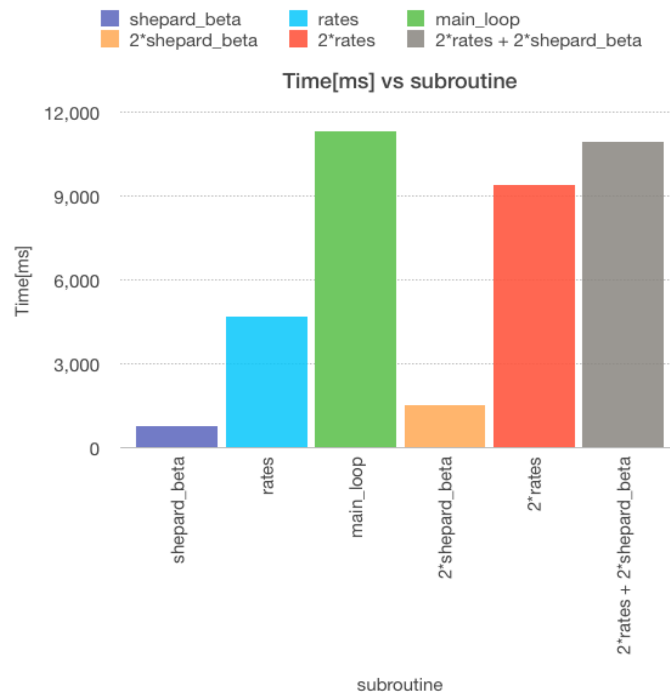

Figure 1. Profile of the original unoptimised code running with 1 MPI process.

The most computationally intensive subroutines of the code are briefly described in Table 1:

Subroutine	***description***
shepard_beta	*Calculation of the renormalisation factors for both shepard correction of the kernel and also MLS correction of the gradients of the kernel. In addition Turkel low mach preconditioning factors are also calculated.*
rates	*Calculation of forces and mass.*

Table 1. Description of the most computationally intensive subroutines.

From Figure 1 we can clearly see that almost 96% of the time of the main loop is spent in the *shepard_beta* and *rates* subroutines. For every iteration of this loop each of these subroutines is called twice as Figure 2 shows.

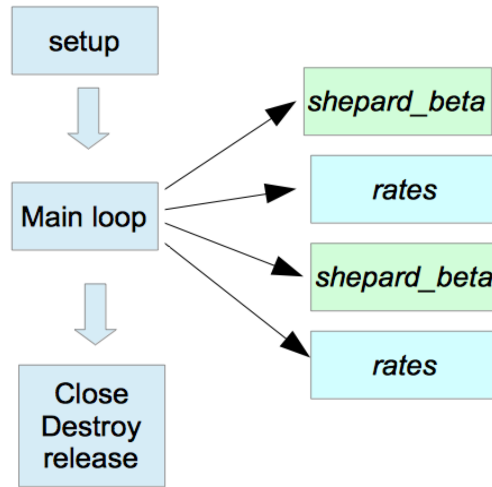

Figure 2. General structure of UCD-SPH program.

As a result it was decided to spend the efforts of this project analysing and optimising these 2 subroutines. The subroutines *rates* and *shepard_beta* have a similar structure, in that they compute physical values such as forces and mass, or renormalisation factors for every pair of interacting particles. This is done by first calculating a possible set of neighbour particles before the update of these values is performed. Both of these subroutines traverse a three-dimensional array of cells, and for each of these iterations, physical values and renormalisation factors of each interacting particle are updated.

Table 2 shows that the time spent searching for neighbours relative to the computation of physical values or renormalisation factors is negligible. [1]

task	*shepard_beta[ms]*	*rates[ms]*
neighbour_search	2.07	3.35
particle_interaction	770.58	4674.99
total	773.18	4692.01

Table 2. Profile of particle interactions and neighbour searching in rates and shepard_beta.

3. OpenMP parallelisation

The calculation of physical values and renormalisation factors between particles (between local particles in the current cell, and particles of the current cell with particles in neighbour cells) is implemented as a triangular loop which computes sequentially the contributions of particle A on particle B and particle B on particle A (particle-to-particle interaction symmetrisation[2]). This way of computing these contributions is an important optimisation in the original code as it avoids duplicating the computation of certain parameters (e.g. distance between particles).

There are several problems when trying to parallelise the computation of particle interactions. For example when multithreading the algorithm the possibility of race conditions exist, which only be prevented by introducing locks with high overhead. As well as this the triangular loop is naturally unbalanced in terms of workload. Thus careful analysis was needed to avoid these issues.

Our solution to this was to perform a significant refactoring of the code to break the triangular loop that performs updates of physical values in the case of *rates* and renormalisation parameters in the case of *shepard_beta* only calculating the effects of the particles in one direction [2], whereby read/write accesses of each iteration of the loops are independent of read/write accesses of other iterations. This method breaks the optimisation achieve by the sequential implementation of the original code (particle-to-particle interaction symmetrisation[2]), increasing the time of the sequential version up to 52% in the case of *shepard_beta* and in the case of rates up to 34%, this increment is shown in Figure 3.

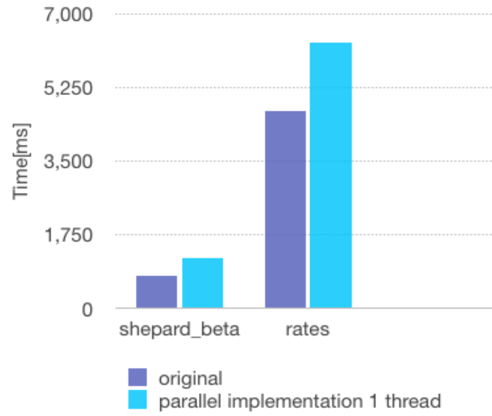

Figure 3. Comparison between the sequential original unoptimised implementation and the parallel implementation with 1 thread.

Figure 4 shows the results of the parallelisation strategy implemented for *rates, shepard_beta* and *main_loop* on the 2 eight-core Intel Xeon Sandy Bridge CPU on a single EURORA node. These figures show the behaviour of the code with dynamic and static scheduling and with compact and scatter affinity.

Table 3 summarises the optimal results of the parallel implementation of the *shepard_beta* and *rates* subroutines.

	*1.5*dp (small size problem)*	*2.5*dp (medium size problem)*	*3.5*dp (large size problem)*
main_loop	3.4x(ds,ca)	3.9x(ss, ca)	4.2x(ss, ca)
rates	3.8x(ds,ca)	4.0x(ss, ca)	4.3x(ss, ca)
shepard_beta	4.0x(ss, ca)	5.2x(ss, ca)	6.0x(ss, ca)

Table 3. Summary of the optimal results obtained for the parallel implementation of rates and shepard_beta (ss: static scheduling, ds: dynamic scheduling, ca: compact affinity, sa: scatter affinity).

The results in Table 3 demonstrate that as is expected, scalability over OpenMP threads improves with larger problem sizes. It is also worth noting that better results are achieved with compact affinity in all cases, and with static scheduling with a bigger smoothing length ($h = hfac*dp$).

4. Enabling the UCD-SPH code on the Intel Xeon Phi.

In this section the two basic modes of execution on the Intel Xeon Phi are explored. Firstly we investigated the performance of the OpenMP version of the code implemented during this project and described in the previous section in native mode. We then subsequently, investigated the code executing in offload mode.

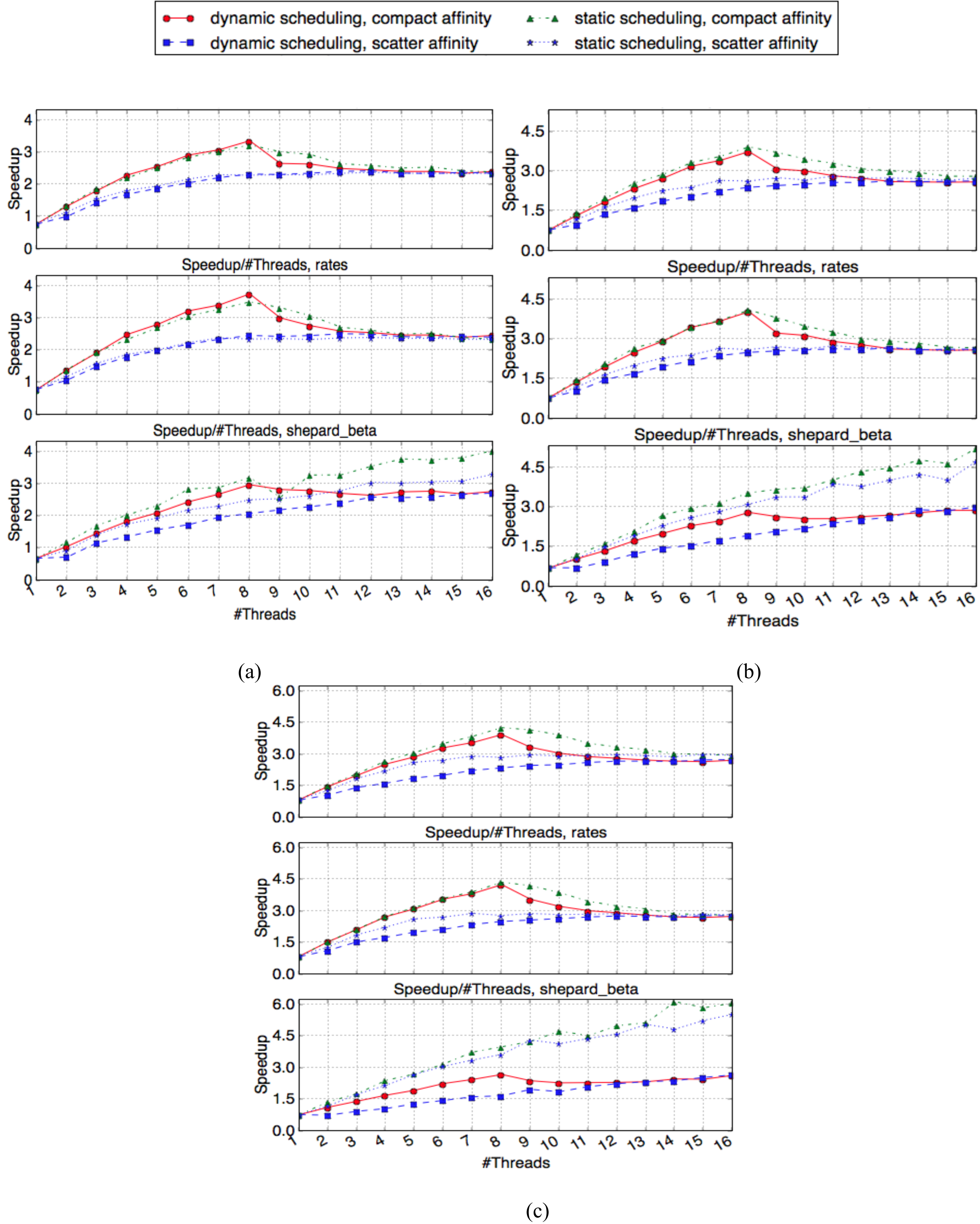

Figure 4. Speedup of the particle parallelisation approach. The figure shows the speedup relative to the original MPI only version of the code (1 MPI process). The top panel shows results for a single time step (main loop). The middle panel shows the results for the rates subroutine and the bottom panel shows the results for the shepard_beta subroutine. Plot (a) shows the results for h = 1.5dp (small size problem), (b) shows the results for h = 2.5dp (medium size problem) and (c) shows the results for h = 3.5dp (large size problem).

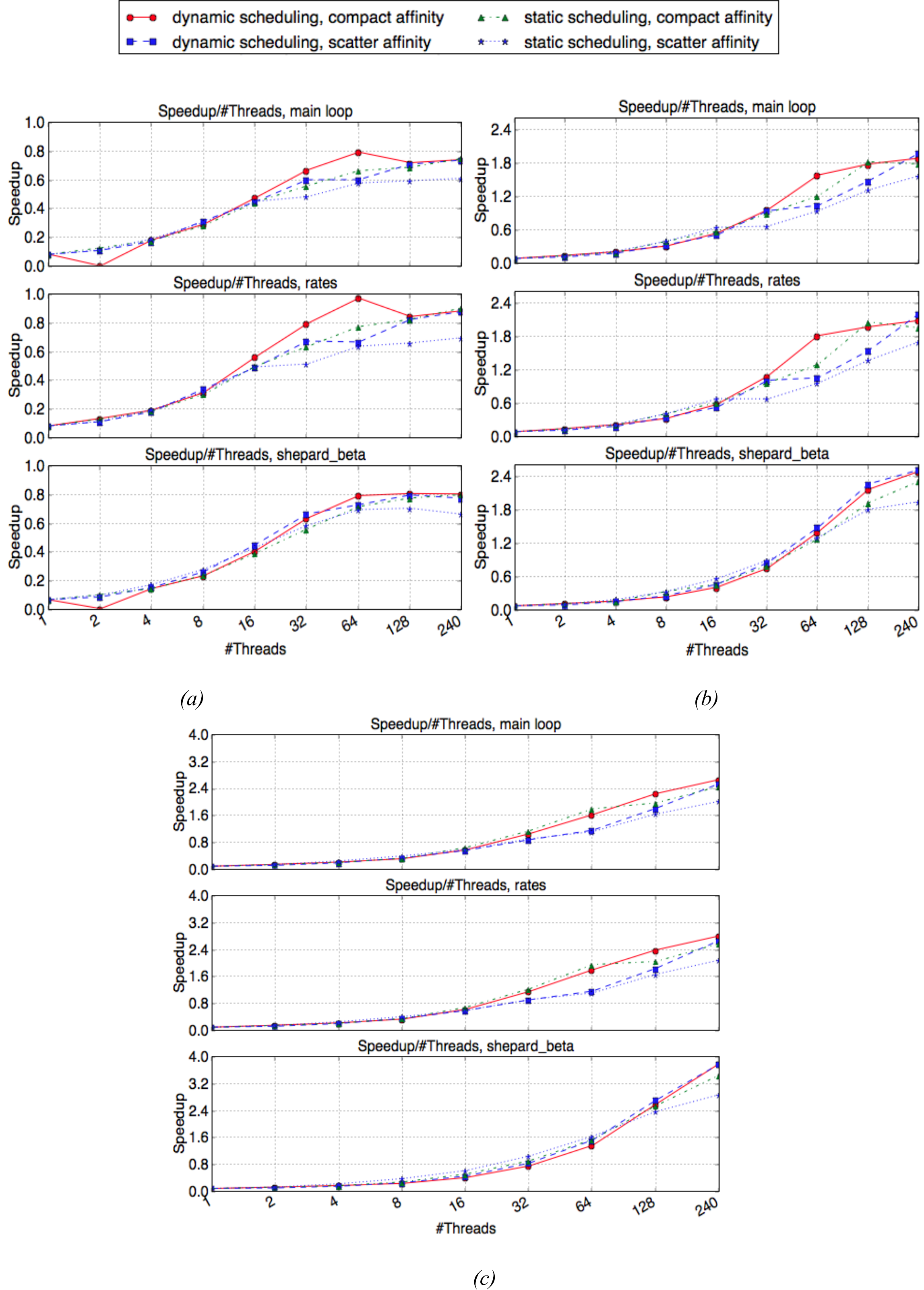

Figure 5. Speedup of the particle parallelisation approach in native mode. The figure shows the speedup relative to the original MPI only version of the code (1 MPI process) running on the host (2 eight-core Intel Xeon Sandy Bridge CPU). The top panel shows results for a single time step (main loop). The middle panel shows the results for rates subroutine and the bottom panel shows the results for the shepard_beta subroutine. Plot (a) shows the results for h = 1.5dp (small size problem), (b) shows the results for h = 2.5dp (medium size problem) and (c) shows the results for h = 3.5dp (large size problem).

4.1 Native execution.

To execute the parallel version of the code in native mode it had to be recompiled with the ***-mmic*** flag. This allows the compiler to create a binary that can be executed natively on the co-processor. The results of the parallel version of the program executed in native mode on the co-processor can be seen in Figure 5.

Table 4 shows a summary of the results of the OpenMP parallelisation of the code executed in native mode on the Intel Xeon Phi and, also shows as in the previous section that when increasing the smoothing length (*h=hfac*dp*) better scalability over threads is achieved.

	*1.5*dp (small size problem)*	*2.5*dp (medium size problem)*	*3.5*dp (large size problem)*
main_loop	0.8x(ds, ca)	1.96x(ds, sa)	2.7x(ds, ca)
rates	0.97x(ds, ca)	2.19x(ds, sa)	2.8x(ds, ca)
shepard_beta	0.8x(ds, ca)	2.5x(ds, sa)	3.8x(ds, ca)

Table 4. Summary of the optimal results obtained for the parallel implementation of rates and shepard_beta executed in native mode on the coprocessor (ss: static scheduling, ds: dynamic scheduling, ca: compact affinity, sa: scatter affinity).

4.2 Offload execution

The other mode of execution that we investigated is the so-called offload mode. In this mode only the sections of code that can exploit the features of the Intel Xeon Phi architecture are executed on the co-processor. For this mode it is necessary to transfer relevant data for these kernels from the host to the co-processor. The main overhead associated with this mode of execution is the transfer of data from the host to the co-processor [3] and vice-versa.

In order to allow for offloading of data, the code was refactored where the majority of refactoring was focused on moving subroutines inside Fortran modules to allow the compiler to offload these subroutines. Also Intel Language Extensions for offload (LEO) pragmas were used to inform the compiler of the variables and subroutines used on the co-processor.

Figure 6, shows the results of the offload version of the code relative to the native version of the code, for compact affinity and static scheduling and a smoothing length of *2.5dp (medium problem size)*.

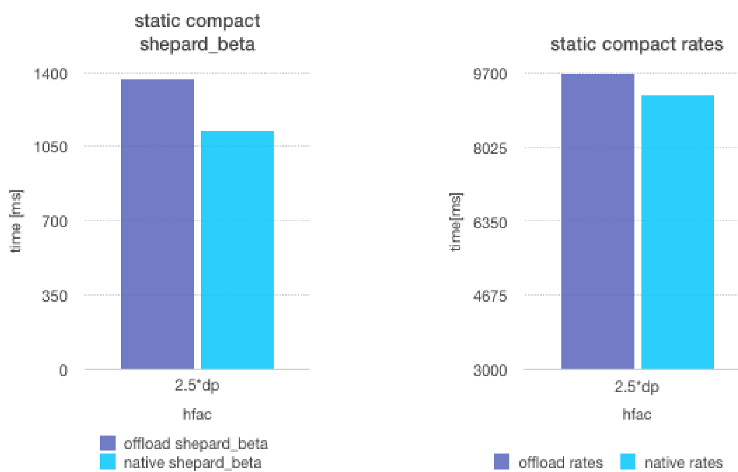

Figure 6. Difference between offloading execution and native execution with an h of 2.5dp.

One can clearly see from Figure 6 that performance is poorer when the code executes in offload mode compared to the code running in native mode which can be put down to the overhead associated with data transfers over

the PCIe bus. The subroutines *shepard_beta* and *rates* are on average 21% and *5.6*% slower in offload mode than in native mode. This difference is because *rates* has to transfer much more data to the co-processor for every offload section, where the amount of data transferred is shown in Table 5.

The overhead of offloading could possibly be hidden using asynchronous data transfers but in this case the current structure of the code does not expose opportunities to transfer data and perform heavy computation on the host at the same time, making the gains of asynchronous offloading marginal. The results with smoothing length equal to *1.5dp* and *3.5dp* are similar to the ones in Figure 6.

Subroutine	***Direction of transfer***	***Size of Data [MB]***
shepard_beta	*CPU->MIC*	160.67
shepard_beta	*MIC->CPU*	15.12
rates	*CPU->MIC*	254.23
rates	*MIC->CPU*	39.69

Table 5. Data that is transferred for every offload section.

5. Refactoring for vectorization

In this section we describe how we have investigated the possibilities of exploiting the 512-bit SIMD VPU capabilities on the Xeon Phi for the UCD-SPH code. The basic structure of the OpenMP code is represented by the pseudo-code shown in Figure 7.

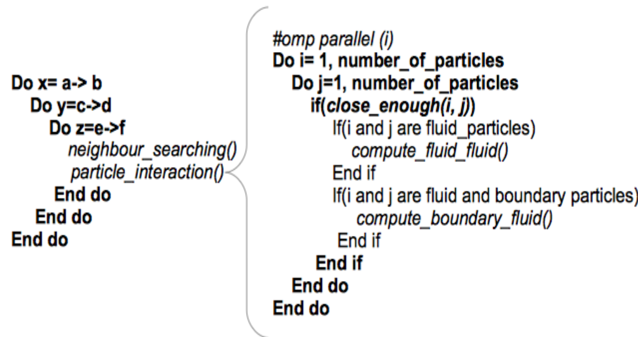

Figure 7. Parallelisation, basic structure

The loop in Figure 7 is not vectorised by the compiler, thus changes were performed to simplify the loop in Figure 7 and expose explicitly to the compiler vectorisation opportunities, where this refactoring is shown as pseudo-code in Figure 8. As a result of these changes loop (1) and (2) in Figure 8 are automatically vectorised by the compiler.

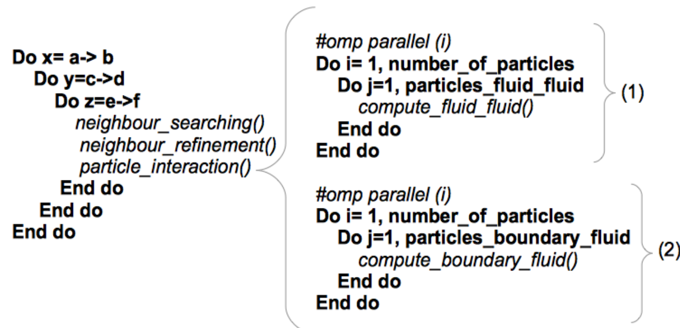

Figure 8. Refactoring of particle_interaction and the extraction of neighbour_refinement.

One significant change to the code that allowed for the simplification and vectorisation of loop (1) and (2) in Figure 8 was the moving of the calculation of distances between particles from *particle_interaction* to *neighbour_refinement*, creating new vectorisation possibilities for the loops created in *neighbour_refinement* as shown in Figure 9.

```
#omp parallel (i)
Do i=1, number_of_particles
   Do j=1, number_of_particles          (1)
      dx = calculation_of_x()
   End do

   Do j=1, number_of_particles
      dy = calculation_of_y()
      dz = calculation_of_z()
      r = calculation_of_distance()
      if(r is close enough)
         if(fluid fluid interaction)
            set_fluid_fluid_interaction()
         End if                          (2)

         if(boundary fluid interaction)
            set_boundary_fluid_interaction()
         Endif
      End if
   End do
End do
```

Figure 9. Structure of the neighbour_refinement subroutine.

It is important to notice that in Figure 9, only loop (1) is vectorised by the compiler on the host and the MIC, and loop (2) is vectorised only on the MIC, which is the main source of the performance improvements shown in Figure 10. It is also important to mention that to perform these changes in the code, data computed inside of these loops had to be stored and communicated to other loops that require the data, thus there is a trade off between performance and memory footprint which is relevant due the limited memory RAM available on the Xeon Phi. No special pragmas were used to force the compiler to vectorise the loops, in an effort to maintain code readability.

The refactoring for vectorization was implemented for the *shepard_beta* subroutine only. Figure 10 summarises the results of this refactoring showing the performance of the code on the host with 8 and 16 threads and comparing with the performance of this refactoring on the coprocessor executing natively.

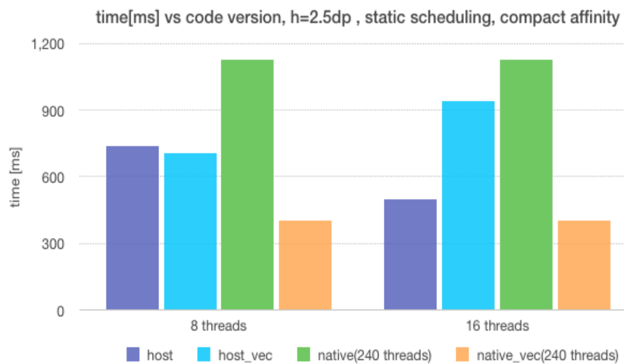

Figure 10. Performance comparison of shepard_beta between basic parallelisation on the host and coprocessor (host, native) and parallelisation that exposes vectorisation to the compiler (host_vec, native_vec) with an h=2.5dp, static scheduling and compact affinity was used.

Auto-vectorisation of the *shepard_beta* subroutine after the refactoring described in this section running in native mode on the co-processor is 1.75x faster than the fastest version running on the host (same version of the code) with 8 threads and is 1.23x faster than the fastest version of the code running with 16 threads (basic parallelisation).

6. Performance Summary

Table 6 shows a summary of the performance achieved for each mode of execution and optimisation performed in this project.

	base[ms]	*OpenMP host[ms]*	*OpenMP MIC native[ms]*	*OpenMP + vect host[ms]*	*OpenMP + vect MIC native[ms]*	*Offload[ms]*
main_loop	41165.58	10594.67	20988.8	-	-	21047.84
rates	17800.06	4377.46	8103.85	-	-	9091.82
shepard_beta	2588.76	498.4	1033.31	706.9	404.58	1244.86

Table 6. Summary of optimal results obtained with an smoothing length of 2.5dp.

7. Conclusions

To conclude, we have profiled the UCD-SPH code and identified the most computationally intensive sections of the code. We subsequently refactored the original version of the code to allow for OpenMP parallelisation for the first time so that the code could exploit multi-core and many-core shared memory architectures. We have provided results for the code running on the co-processor in two different modes of execution, namely native mode and offloading mode where no significant overheads were found in the code running in offload mode. Our performance analysis of the code running in native mode has demonstrated relatively poor scalability across OpenMP threads. Better performance on the MIC was achieved only after investigations were made into refactoring computationally intensive loops in order to allow for auto-vectorisation. These better results for the *shepard_beta* subroutine indicate that further investigations into the vectorisation of the code would be worthwhile.

In summary, initial results indicate that the UCD-SPH code is currently not well suited to the Intel MIC architecture, but we feel that further investigations of the code running in MPI symmetric mode as well as further investigations into possibilities for vectorisation may lead to improved performance.

References

[1] Markus Ihmsen, Nadir Akinci, Markus Becker, Matthias Teschner, "A parallel SPH implementation on multi-core CPUs", Computer Graphics Forum, Vol. 30, pp. 99-112, 2011.
[2] G. Oger, E. Jacquin, M. Doring, P.M. Guilcher, R. Dolbeau, P.L. Cabelguen, L. Bertaux, D. Le Touze, B. Alessandrini, "Hybrid CPU-GPU acceleration of 3-D parallel code SPH-Flow", Proceedings of the 5th Spheric Workshop, Manchester, U.K, 2010.
[3] Chris J. Newburn, Rajiv Deodhar, Serguei Dimitriev, Ravi Murty, Ravi Narayanaswamy, John Wiegert, Francisco Chinchilla, Russ McGuire, "Offload Compiler Runtime for the Intel Xeon Phi Coprocessor", Supercomputing Lectures Notes in Computer Science, Vol. 7905, pp. 239-254, 2013.
[4] J.P. Vila, "On particle weighted methods and smooth particle hydrodynamics", Mathematical Models and Methods in Applied Sciences, 09(02): 161-209, Mar. 1999.
[5] J.P. Vila, "SPH Renormalized Hybrid Methods for Conservation Laws: Applications to free surface flows", Meshfree Methods for Partial Differential Equations II, Lecture Notes in Computational Science and Engineering, Vol. 43, 2005, pp. 207-229.
[6] A. Rafiee, S. Cummins, M. Rudman, K. Thiagarajan, "Comparative study on the accuracy and stability of SPH schemes in simulating energetic free-surface flows", European Journal of Mechanics-B/Fluids, Vol.36, November-December 2012, pp. 1-16.
[7] A. Rafiee, N. Repalle, F. Dias, "Numerical Simulations of 2D Liquid Impact Benchmark Problem Using Two-Phase Compressible and Incompressible Methods", In Proceedings of 23rd International Offshore and Polar Engineering Conference (ISOPE), Alaska, USA, 2013.
[8] A. Rafiee, D. Dutykh, F. Dias, "Numerical simulation of wave impact on a rigid wall using a two--phase compressible SPH method", In Proceedings of IUTAM symposium of Particle Methods in Fluid Mechanics, under review.
[9] A. Rafiee, B. Elsaesser, F. Dias, "Numerical simulation of wave interaction with an oscillating wave surge converter", In Proceedings of OMAE 2011, ASME 31th International Conference on Ocean, Offshore and Arctic Engineering (OMAE), Nantes, France, June 2013.

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. The work was achieved using the PRACE Research Infrastructure resources at Cineca and Ichec.