Porting and Verification of ExaFMM Library in MIC Architecture

Valentin Pavlov[1], Nikola Andonov, Georgi Kremenliev

National Center for Supercomputing Applications (NCSA)

Abstract

ExaFMM is a highly scalable implementation of the Fast Multipole Method (FMM) – an O(N) algorithm for solving N-body interaction with applications in gravitational and electrostatic simulations. The authors report scaling on large systems with O(100k) cores with support for MPI, OpenMP and SIMD vectorization. The library also includes GPU kernels capable of running on a multi-GPU system. The objective of the project is to enable the use of the ExaFMM solver in the MIC architecture by performing porting, verification, scalability testing and providing configuration suggestions to its potential users.

1. Introduction

Determining the motions of N bodies that interact via long-distance forces is a classical problem in gravitational and electrostatic calculations. It has a conceptually very simple solution, but computationally a very intensive one – evaluating all pair-wise interactions between interacting bodies, resulting in $O(N^2)$ computational complexity. Using an algorithm with such complexity is not reasonable for large systems comprising millions of bodies. The usual strategy in such cases is to trade precision for performance by approximating the solution with a computationally less complex algorithm. Gravitational and electrostatic interactions follow inverse-square laws, thus contributions from distant bodies quickly diminishes proportional to the square of the spatial separation involved. This allows splitting of the effect into near-field, where interactions are precisely calculated, and far-field, where interactions are approximated. Far-field approximation can be done by performing series expansion (*multipole expansion*) of distant interactions, limiting the number of terms in the series based on the precision required, and summing the expansion coefficients of neighboring sources into a single series, effectively treating a cluster of source particles as a single body. The complexity of such scheme depends on the nature of the expansion, the number of terms used and on the mechanism of clustering the particles, but is in general O(N logN) [1], [2]. By applying the same idea in reverse – clustering *target* particles, applying far-field contributions to the cluster as a whole (*local expansion*), and then distributing the resulting total contributions among the clustered particles, O(N) complexity is achieved [3]. The Fast Multipole Method (FMM) is a family of such O(N) schemes, which may differ in various details such as the force law being approximated, nature of the expansions being used (e.g. Fourier Series, Taylor series in Cartesian coordinates, Spherical harmonics, etc.), criteria for achieving the required precision, etc.

ExaFMM [4], [5], [6] is a highly scalable FMM implementation. The authors report scaling on large systems with O(100k) cores with support for MPI, OpenMP and SIMD vectorization. The library also includes GPU kernels capable of running on a multi-GPU system. It is actively being developed and this research is based on the version of the source code that was current at the start of the project, continuously merging[2] with the upstream source code repository (https://bitbucket.org/rioyokota/exafmm-dev).

The objective of the project is to enable the use of the ExaFMM solver in the MIC architecture by performing porting, verification, scalability testing and providing configuration suggestions to its potential users.

[1] vpavlov@rila.bg

[2] The last such merge was on 11/05/2013.

2. ExaFMM Algorithm Analysis

The algorithm, as described in [3] requires that the spatial domain of the problem is hierarchically split into cells, each cell containing either sub-cells or a cluster of bodies. This is represented by a tree structure, which in 3D is an oct-tree (the original paper treats a problem in 2D and thus a quad-tree is built instead). The algorithm starts by building the oct-tree, then performing an upward, followed by a downward pass. During the upward pass the multipole expansion of each cell is calculated. During the downward pass the multipole expansions are turned into local expansions, which at the leaf level are applied to the target bodies. Finally, near-field interactions are calculated directly. This requires the presence of several routines (kernels). The authors of the ExaFMM library came up with a convenient naming for these kernels, which we will follow throughout the paper. The kernels and their corresponding functionality is as follows:

- **P2P** kernel (particle to particle) – this is the original pair-wise interaction between near-field bodies. It is applied on all particles inside a given cell or between particles in cells that are not well separated (not far enough of each other in order to satisfy the accuracy requirements);
- **P2M** kernel (particle to multipole) – given the collection of bodies inside a cell, calculates the multipole expansion attributed to that cell. This kernel is applied on every leaf cell of the tree during the upward pass.
- **M2M** kernel (multipole to multipole) – applied on non-leaf cells, this kernel calculates the multipole expansion for each cell by considering the multipole expansions of its children (8 in the case of 3D). This kernel is applied during the upward pass.
- **M2L** kernel (multipole to local) – this kernel is applied on a pair of cells and converts the multipole expansion of the source cell to a local expansion of the target cell. In the original paper [3], this is done during the downward pass, by constructing for each visited cell an interaction list of *well separated* cells for which far-field approximation is valid. In the ExaFMM implementation this is done different, in a separate in-between pass that uses dual tree traversal technique which considers each pair of cells in the tree. If they are *well separated*, the M2L kernel is applied; otherwise direct summation (P2P) is performed.
- **L2L** kernel (local to local) – during the downward pass the local expansion of each non-leaf cell is distributed to its child cells (8 in the case of 3D).
- **L2P** kernel (local to particle) – during the downward pass, local expansions at the leaf level are applied to the target particles, forming the sum of the far-field contributions.

An important deviation point between the original algorithm and the ExaFMM implementation is in the way the oct-tree is built. In the original algorithm a balanced oct-tree is used. The depth of tree is an input parameter to the algorithm, usually some function of N. When dealing with balanced oct-tree of known depth, the interaction list of a cell (that is, the list of cells with which M2L needs to be applied) is independent of the actual charge distribution and can be exactly calculated using only arithmetic operations on cell indices in some form of space-filling curve indexing scheme. M2L is then made part of the downward pass, as described in [3]. However, this approach has the disadvantage that with non-uniform charge distributions the leaf-cells end up with greatly varying number of particles which at the end compromises the performance of the algorithm.

To counter this, ExaFMM builds the oct-tree adaptively: starting from the root of the tree (the whole domain), each cell is recursively split into sub-cells only if the number of particles in it is larger than a given threshold (parameter `ncrit` in the code). This leads to a situation in which regions of space containing small number of bodies are represented by larger cells, while densely populated regions are represented by many smaller cells. In this scheme the interaction list for a cell depends on the charge distribution, which means that pairs of cells has to be considered and the possibilities for M2L evaluated based on the spatial separation between corresponding cells. If the two cells are well separated, M2L can be applied; otherwise P2P for each particles in the two cells has to be applied. In ExaFMM this evaluation is performed in an additional in-between pass, implemented as a dual tree traversal.

Thus, while the original algorithm has two passes, the ExaFMM has three:

- Post-order traversal (upward) pass, in which P2M is applied at leaf cells and M2M is applied at non-leaf cells. The post-order ensures that when a given cell is considered for M2M, the multipole expansions of its children cells are already calculated, which is necessary for the correct operation.
- Dual tree traversal in which pairs of cells in the tree is considered. If the cells are well separated, M2L is applied; otherwise P2P is applied.
- Pre-order traversal (downward) pass in which L2L is applied at non-leaf cells and L2P is applied at leaf cells. The pre-order ensures that when a given cell is considered for L2L, the local expansion of its parent is already calculated, which is necessary for the correct operation.

Even without knowing the details of the kernels involved, it can be intuitively inferred that the middle pass would take the majority of time, since it acts on pairs of cells, while the first and third pass consider each cell only once. Indeed, as shown in [6], the two most time-consuming operations are the P2P and M2L kernels, which are exactly the operations that occur in the middle pass. As such, they are identified as the focus of our optimization work.

The algorithm 'skeleton' possesses natural recursive description. It is independent on the actual interaction calculated by the kernels and the expansion method being used, as long as the kernels obey the interface and produce physically meaningful results. The version of the library being investigated contains Laplacian kernels, applicable for gravitational and Coulombic interactions, in two variants: Taylor series expansion in Cartesian coordinates and Spherical Harmonics expansion in spherical coordinates. In this project we consider the Cartesian expansion, while the Spherical expansion can be the subject of a future work.

The parallelization strategy employed by ExaFMM involves three hierarchical levels:

- distributed-memory parallelization using MPI tasks;
- shared-memory multi-threading inside each MPI task;
- SIMD vectorization of some of the kernel operations;

For MPI parallelization, the oct-tree is built in parallel, each rank operating on its local sub-tree during the first and third passes. Body and cell exchanges occur during the dual tree traversal, each rank calculating its locally essential tree (LET) – portions of the global tree that resides in other ranks and which may be involved in M2L and P2P calculations. Analysis of the code shows that the chosen method for calculating the LET requires the domain to be split between 2^k ranks in a recursive bisection fashion. This means that the code can only be correctly executed on number of processes that is power of 2. This is a hard limitation that cannot be overcome without major redesign of the LET calculation.

On the level of multi-threading, the library uses task-based parallelism for building and traversing the tree in the various passes. This is an obvious choice given the recursive nature of the algorithm. The other possibility would be to utilize work-sharing constructs in the actual calculation kernels. This would not be efficient given that each kernel is executed great many times, since thread spawning overhead will be massive. For example, a 5-level oct-tree would have around 35,000 cells; M2L and P2P would be executed several million times and spawning work-sharing threads on each execution would definitely swamp any potential multi-threading performance gain.

The library uses manual SIMD vectorization only for the P2P kernel and the direct summation used only for validation purposes. In all other instances it relies on compiler supplied auto-vectorization.

3. MIC Architecture Specifics

The defining feature of the MIC architecture that sets it apart from other modern HPC architectures is its 512-bit SIMD registers, which, combined with the fused multiply add (FMA), theoretically allows the simultaneous execution of 32 floating point operations. This is clearly stated in [7] where the theoretical peak performance in single precision is shown to be

Clock Frequency x Number of Cores x 16 lanes x 2 (FMA) FLOPs/cycle

While this is certainly beneficial for vector and matrix operations, it is clear that without applying vectorization techniques most of the codes will not benefit from its massive performance. A program that is not vectorized cannot utilize more than 3 – 6% of the massive computational power of a MIC in single precision. Thus, our main optimization effort was focused on vectorizing the most time-consuming calculations.

Vectorization can be performed either automatically by the compiler, or manually, for example by using SIMD instruction intrinsics. While auto-vectorization is very helpful for optimizing simple loops, it does have its limits and programs should be (re-)factored in a way as to allow the compiler to infer possible vectorization. Moreover, it is not applicable in situations where loops are not present. Computationally intensive areas that are not organized as loops cannot be auto-vectorized, as are loops that involve functionally dependent indices. In these cases manual vectorization should be considered.

Another specific of the architecture is that the MIC acts as a co-processor, which means that data exchange between CPU and MIC happens before and after each work unit. The two possible execution modes – offload and native, should be considered when porting an application or library code. Additionally, in native mode, the 60+ cores of the MIC can be organized as a shared-memory (multi-threading), as distributed memory (MPI), or as a hybrid MPI+multi-threading machine. There is also the possibility to treat all cores, CPU and MIC, as equally participating in a heterogeneous MPI cluster.

4. Porting and verification of ExaFMM

Considering the algorithm analysis and architecture specifics outlined above, the following activities were planned and executed:

Compilation of the library code in CPU, native MIC and hybrid setups

In order to experiment with the three different modes of execution (offload, native and hybrid), we compiled the library in all three setups. Compilation was straightforward due to the excellent `icc` compiler that cross-compiles for the MIC with just a simple switch setting. Only the Cartesian Laplace kernel was considered in this project, but compiling the Spherical Laplace kernel should be of no significant effort as well.

OpenMP fix

While the authors claim support for OpenMP, its implementation for spawning tasks was fixed in the development version of the library that we used. In fact, the OpenMP didn't work at all, and turning it on made the library slower. That is also mentioned in the Makefile. However, it turned out that the problem was not in the OpenMP implementation, but in the library code itself. The fix was very simple and boils down to the fact that `#pragma omp task` will ony spawn new thread *only if hit inside a parallel section*, which the authors didn't properly set. After properly setting up a parallel section around the code, OpenMP task-based parallelization became functional.

Offloading mode

The offloading mode was quickly abandoned after confirmation as being non-optimal. For offloading we considered offloading only the computational kernels and offloading the some of the dual traversal section.

Offloading computational kernels is not effective for the same reasons for which work-sharing OpenMP parallelization is not. Each kernel is executed great many times and the overhead of data transfer between CPU and MIC would outweigh the potential performance benefits of executing the code on the MIC. We confirmed that with a simple test of offloading the P2P kernel and achieving much worse performance than in the original code executed on the CPU without offloading.

We also abandoned offloading the dual tree traversal section because of its recursive nature. We can only offload the *complete* traversal routine; and if we were to offload the piece of code that takes 90% of the whole time, we might as well compile the application in native mode and "offload" the other 10% as well.

Hybrid MPI mode

Hybrid MPI mode was abandoned because the library's MPI parallelization strategy does not allow differentiating of workload between different kinds of cores. In hybrid MPI mode, both CPU and MIC cores are involved on equal basis in the MPI communicator and since their performance is quite different, when hitting collective operations, the MIC cores will have to idle waiting for CPU cores to finish. This could be counteracted by dynamically distributing load between worker nodes or by special domain decomposition techniques that distribute more load to the MICs than to the CPUs, but such techniques are not implemented in the library and their implementation is out of scope of this project because of their complexity.

Thus, native execution mode was selected as the only feasible execution mode of the library in the MIC architecture.

SIMD Vectorization

We reviewed the SIMD vectorization of the two most time-consuming kernels in Cartesian Laplace mode: P2P and M2L.

The P2P kernel had the option of being hand-vectorized or auto-vectorized by the compiler. This option was controlled by a compile-time directive. Starting with the auto-vectorization mode, we found out that the compiler *could not* auto-vectorize the P2P kernel, because the inner loop was coded in a way that prevented the compiler from properly inferring the dependencies between iterations. After refactoring the code, the compiler was able to auto-vectorize the P2P inner loop. We compared the result to the hand-crafted vectorization mode, and found out that the auto-vectorization performs slightly better than the hand-crafted one (in the order of 2-3%).

The M2L kernel does not contain any loops and thus cannot be auto-vectorized by the compiler. Still, the kernel contains many constructs of the form C[10]*M[10] + C[11]*M[11] + C[45]*M[23] + ... in its `getCoeff` routine, which gets recursively called for conversion between multipole and local expansions. The library includes a template class for vectors of 16 floats that uses 512-bit MIC intrinsics and by using this class we were able to vectorize the `getCoeff` functions of the M2L kernel.

Verification of the port

Verification of the port was done by calculating the mean squared error (MSE) of a randomly chosen subset of 1,000 bodies between potentials obtained by running FMM calculations on 1,000,000 bodies and the potentials obtained for the test subset using direct pair-to-pair interaction. For 10-order multipole expansion we got MSE of the order $O(10^{-6})$, which is in accordance with the expectations. The same test was run on non-MIC architecture and showed similar MSE.

Scalability testing

For scalability testing we used the following approach: we testing the code using a fixed number of bodies and time steps on single MIC co-processor using different MPI-OpenMP configurations that result in maximum 240 threads on the MIC. The domain decomposition technique used in the library and the related building of locally essential tree for communication with other ranks restricts the number of MPI ranks to be a power of 2. We used scattering affinity for the threads, which proved to be more efficient than the compact one. The results are shown in Table 1. Maximum performance is achieved when running 32 MPI tasks each spawning 3 threads. We interpret this result as being due to the fact that 1) at least two threads per core are needed to satisfy the MIC scheduling another thread every cycle (as explained in [7], p. 31, last paragraph); and 2) scheduled threads are computationally intensive and saturate their respective cores, so using more than 2 threads per core is not efficient.

Table 2 shows the parallel speedup for the same configuration. Figure 1 displays the thread speedup for different MPI sizes. It is seen that thread speedup drastically falls with increasing the number of threads, which we interpret as being caused by the recursive task-based thread-spawning employed by the code and consequent poor cache hit/miss ratio. Figure 2 shows the same data from perpendicular direction. As seen, when using 1 thread per MPI rank parallel speedup is almost perfect. That means that this FMM implementation is better at scaling on distributed memory architectures than on shared memory ones and MPI parallelism should be preferred to the OpenMP one.

Regarding the scalability with respect to the data size, the algorithm behaves quite well. Increasing the data size 10 times resulted in exactly 10 times increase of the execution times. This shows no major downgrading of parallel speedup due to large data sets.

It should be noted that the `ncrit` parameter plays a very important role for achieving good performance. Setting it too low results in fewer bodies per cell; this defeats the vectorization of the P2P kernel, the most time-consuming computation. Setting it too high results in more bodies per cell and that defeats the main idea of the algorithm to avoid too much P2P. Unfortunately the exact setting of this parameter heavily depends on the size of the system and its charge distribution and cannot be pre-determined. Instead, users are advised to experiment with the system until the optimal value is found and then setting this on a per-system basis.

	Number of OpenMP threads per MPI rank							
MPI ranks	**1**	**3**	**7**	**15**	**30**	**60**	**120**	**240**
1	375.581494	143.347597	75.3670261	51.9568591	44.3464229	42.023947	46.34463	62.369422
2	193.641072	74.3009682	40.4539099	30.7897701	27.4091301	30.0856111	39.7556181	
4	95.805001	38.223681	22.9141769	18.5868499	19.2526939	25.8137479		
8	50.334491	20.816978	13.426976	13.27795	16.746649			
16	23.3726511	10.257725	8.4499819	10.245769				
32	11.798099	**6.9972742**	7.5422869					
64	25.4168686	29.7153339						
128	35.8164809							

Table 1. Execution time (in seconds) for a system of 1,000,000 uniformly distributed bodies on a single MIC in native execution mode, using scattering thread affinity, for different MPI and thread counts. Only setups that do not exceed a total of 240 threads are considered.

	Number of OpenMP threads per MPI rank							
MPI ranks	**1**	**3**	**7**	**15**	**30**	**60**	**120**	**240**
1	1	0.87335842	0.71190946	0.48191455	0.28230875	0.14895535	0.06753416	0.02509119
2	0.96978779	0.84247779	0.66315591	0.40660853	0.2283798	0.10403132	0.03936357	
4	0.98006756	0.81882375	0.58538541	0.33678066	0.16256664	0.06062362		
8	0.93271405	0.75175316	0.49950282	0.23571755	0.09344693			
16	1.00432952	0.76280213	0.39685365	0.15273845				
32	0.99481465	0.55911875	0.22230699					
64	0.23088843	0.06582977						
128	0.08192403							

Table 2. Parallel speedup for a system of 1,000,000 uniformly distributed bodies on a single MIC in native execution mode, using scattering thread affinity, for different MPI and thread counts. Only setups that do not exceed a total of 240 threads are considered.

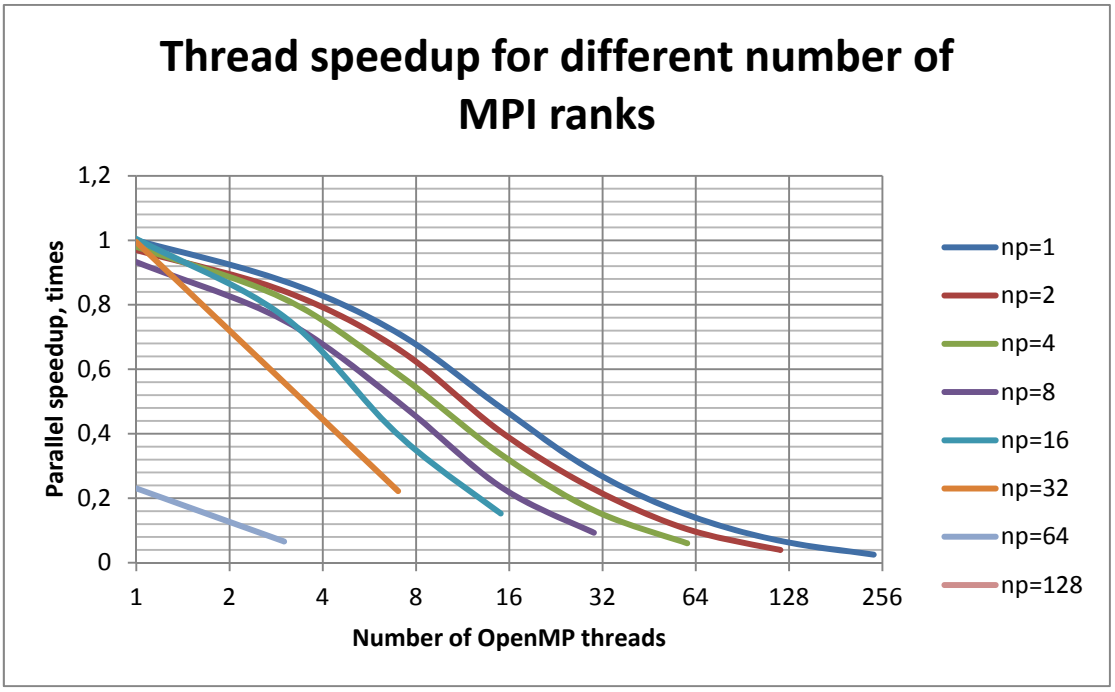

Figure 1. Thread speedup for different number of MPI ranks. The speedup drastically falls, which shows poor thread scalability. Moreover, the more MPI ranks there are, the less is the speedup.

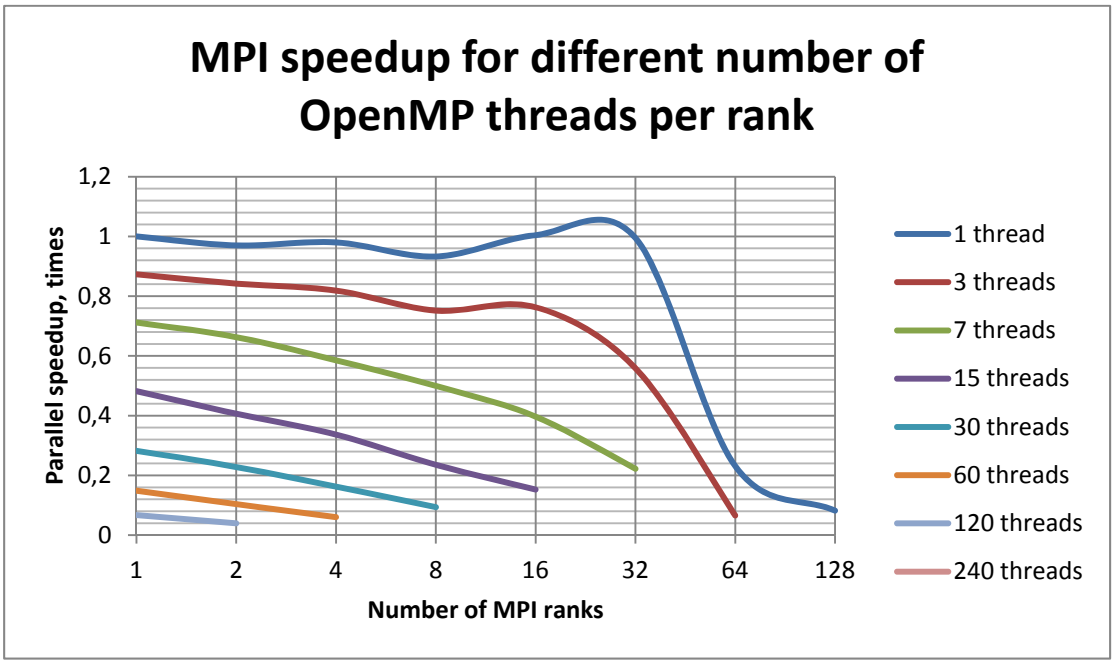

Figure 2. MPI speedup for different number of OpenMP threads per rank. Scalability is almost perfect for 1 thread per rank and gradually falls with increasing the thread number.

References

[1] Andrew W. Appel. An efficient program for many-body simulation. SIAM J. Sci. Stat. Comput., 6(1):85–103, 1985.
[2] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. Nature, 324:446–449, December 1986.
[3] L. Greengard, V. Rokhlin. A Fast Algorithm for Particle Simulations. J. Comput. Phys. 73 (1987) 325

[4] R. Yokota, L.A. Barba, “A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems”, *Int. J. High-perf. Comput*

[5] R. Yokota, J.P. Bardhan, M.G. Knepley, L.A. Barba, T. Hamada “Biomolecular electrostatics using a fast multipole BEM on up to 512 GPUs and a billion unknowns”, *Comput. Phys. Commun.*,182(6):1271–1283 (2011)

[6] R. Yokota, T. Narumi, L. Barba, K. Yasuoka. “Scaling Fast Multipole Methods up to 4000 GPUs”, In proceedings ATIP '12 of the ATIP/A*CRC Workshop on Accelerator Technologies for High-Performance Computing, Article No. 9, A*STAR Computational Resource Center Singapore, Singapore 2012

[7] J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kaufmann, 2013 ISBN 978-0-12-410414-3

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.