Cache Aware Dynamics Data Layout for Efficient Shared Memory Parallelisation of EUROPLEXUS

Marwa Sridi, Bruno Raffin, Vincent Faucher

Cache Aware Dynamics Data Layout for Efficient Shared Memory Parallelisation of EUROPLEXUS

Marwa Sridi[1], Bruno Raffin[2], and Vincent Faucher[3]

[1] CEA, DEN, DANS, DM2S, SEMT, DYN, F-91191 Gif sur Yvette, France.
sridi.marwa@hotmail.fr
[2] Univ. Grenoble Alpes, INRIA, France.
bruno.raffin@inria.fr
[3] CEA, DEN, Cadarache, DTN/Dir, F-13108 St Paul lez Durance, France.
vincent.faucher@cea.fr

Abstract
Parallelizing industrial simulation codes like the EUROPLEXUS software dedicated to the analysis of fast transient phenomena, is challenging. In this paper we focus on the efficient parallelization on a multi-core shared memory node. We propose to have each thread gather the data it needs for processing a given iteration range, before to actually advance the computation by one time step on this range. This lazy cache aware layout construction enables to keep the original data structure and leads to very localised code modifications. We show that this approach can improve the execution time by up to 40% when the task size is set to have the data fit in the L2 cache.

Keywords: EUROPLEXUS; Shared Memory; Cache-aware Data Layout ; Parallel Programming

1 Introduction

EUROPLEXUS (EPX) [1] is an industrial grade simulation software co-owned by the Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA) and the Joint Research Center of the European Commission (EC/JRC). EPX si dedicated to the analysis of fast transient phenomena involving structures and fluids in interaction. It is based on a space discretization by means of Finite Elements, SPH Particles (Smooth Particle Hydrodynamics) or Discrete Elements for structures, or by means of Finite Elements, Finite Volumes or SPH Particles for fluids. Time integration is achieved through a conditionally stable explicit scheme. The solving algorithm is completely non-linear, at both geometric level (large displacements; large rotations) and material level (constitutive laws implementing plasticity or damage for example). The program provides a large number of kinematic links between entities, for instance for boundary conditions, contact between structures or fluid-structure interaction. EPX is characterized by

[1] http ://www epx.cea.fr/

its minimal use of non-physical parameters to enforce these links, such as penalty coefficients. It relies on direct methods to compute the link forces whenever it is possible and otherwise, the links are dualized by means of Lagrange Multipliers, the unknown forces being then deduced from the resolution of an additional linear system.

EPX is parallelized with MPI for clusters. But today supercomputer nodes show complex multi-core shared memory architectures. Relying on MPI for taking advantage of these multiple core lead to extra memory usage (phantom cells) and complex code to support load balancing. Task oriented programming models for shared memory architectures, available through programming environments like OpenMP, Intel TBB, Cilk or XKAAPI , offer an interesting alternative. The user only needs to delimit the potential parallelism through tasks or loop iterations, the runtime taking care of distributing these tasks to cores, relying on dynamics load balancing schedulers.

A large part of the computational cost in EPX classically occurs in the main loop iterating over the simulation elements to update their state (75% of the execution time for the MARA2 dataset introduced in Section 4). The cost of iterations is not uniform, mostly due to the different element formulations and materials co-existing within a single industrial model, but also to changes in the physical state of the dynamic systems modifying the work load required to compute the local equations. This calls for dynamics load balancing capable schedulers. Data access patterns also show some irregularities: updating the state of a given element often requires to access both its connected nodes and neighbor elements. The iterations of this main loop being independent, it can easily be parallelized with a task oriented environment.

The cost of accessing the data can significantly impair the performance. It is important to rely on data structures with a memory layout enabling to efficiently benefit from the cache hierarchy. However, the data layout EPX works on is inherited from upstream software tools used to mesh and partition the simulation domain. These tools can be different depending on the users and the type of simulation targeted. The quality of the data layout regarding the temporal and spatial locality is thus not ensured and can lead to numerous cache misses impairing performance.

In this paper we propose to have each task build its own local data structure gathering the data needed for processing the iteration range associated with this task. This lazy and parallel layout construction leads to very localized code modifications, and neither affects the global EPX data structure nor imposes constraints on the upstream tools for the simulation preparation. Experiments show that the cost of constructing these task local data structures is small compared to the compute cost and enables to significantly decrease the amount of cache misses. Globally our solution can lead up to a 40% performance gain when the task size is set to have the data fit in the L2 cache, compared to the simple loop parallelization on the original data structure. Without code modification we can benefit from the scheduler's dynamics load balancing capabilities. We show that our approach increases the benefits of load balancing by ensuring a strong locality of computation.

The remainder of the paper is organized as follows. In the second section we briefly present the EPX data structure. Then, we introduce our GROUP approach for both sequential and parallel implementation in section 3. In section 4 we provide some numerical experiments designed to evaluate the GROUP approach performance and we discuss the results. Section 5 concludes the paper and explores future research directions to further improve our implementation.

2 Dynamic Data Grouping

The EXP main loop iterates on the geometric elements that describe the simulation domain, to advance the simulation by one time step. Focusing on one element *ei* (Figure 1), carrying out an elementary computation from start to finish requires various data:

- **ELEMENTARY** data includes information allowing the definition of the mesh element *ei* through its intrinsic characteristics (deformation, internal variables...).
- **NODAL** data includes information on the nodes of each mesh element (displacement, velocity, acceleration, forces...).
- **NEIGHBORHOOD** data are variables that describe fluxes between the element and its neighbor cells (exchange between Ngi and ei in Figure 1), whether in form of energy or mass (for Eulerian or *Arbitrary Lagrange Euler* formulations).

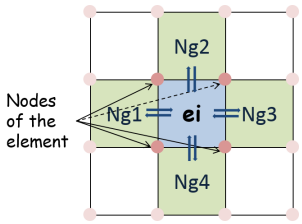

Figure 1: Representation of a mesh element **ei** and its neighbors in 2D

These concepts are valid in either structure only, fluid only or fluid/structure. These different data are stored in separate arrays. The order of the elements in these array is directly inherited from the files EPX loads for initializing the simulation. We have no guarantee that this layout will actually lead to an efficient use of the caching mechanism. The problem is amplified in a parallel context where false sharing for instance, i.e. the concurrent write/write or read/write of distinct data addresses but that belong to the same cache line, can lead to extra cache line invalidations.

To improve data locality and thus on cache efficiency, we propose to rewrite the main loop in 2 nested loops (Algo. 1). The outer loop iterates on GROUPS of elements (`DO LOOP_GR...`) , while the inner loop (`DO IELOOP...`) iterates on each element of the processed loop. Each GROUP consists in a fixed number of elements.

This technique is thus based on a switch from the global structure of arrays layout (SOA) [9] to a local one. Therefore, an hybrid model, balancing the former layout at the scale of the different GROUPS and the array of structures layout at the scale of a single GROUP of elements, has been used. This hybrid SOAOS [15] model promotes the memory locality since the EPX code is based on a loop iterating over all the mesh elements.

To ensure that the inner loop can process the GROUP's elements with a high data locality, each outer loop iteration starts by filling a group data structure that consists in all elements belonging to these GROUPS (ELEMENTARY data) as well as all NODAL and NEIGHBORHOOD data that are required to update these elements (`GROUP_CONSTR` routine). We consider that two elements are neighbors if they share at least one common side on the grid mesh. We determine the list of the cell neighbors by checking all the common sides with the adjacent cells. Then, we check if the global indexes belong to the list of the elements of the current GROUP. If this is not the case, it corresponds to extra data that does not belong to any other element of the GROUP and thus needs to be explicitly added to the GROUP. However, there is no need to copy the complete package for this added cell. We simply copy information ensuring its neighbor role (information coming from the NEIGHBORHOOD data), because it is already completely defined as a local element in another GROUP and therefore it is computed there.

Once the inner loop finished, the updated data are written back to the EPX global data

structure (`UPDATE` routine). The elements in each GROUP correspond to consecutive indexes in the EPX global data structure. A local index is used to indicate the rank of each element relative to the other elements of the same GROUP. Beside reorganizing the main loop, we thus also need to change the arguments of the subroutines called in the inner loop to access to the GROUP's data instead of the global data structure. Elementary operations inside subroutines are kept intact.

Algorithm 1: The main iteration loop rewritten with 2 nested loops working on GROUPS

```
1  Subroutine COMPUTE(A)
2  !      Initialization step
3    NGR = NELEM/NB_ELT_GR
4    IF (NGR × NB_ELT_GR < NELEM)
   THEN
5      NGR = NGR+ 1
6    ENDIF
7    DO LOOP_GR = 1, NGR
1
8      ISTART=1 + NB_ELT_GR
9        ×(LOOP_GR − 1)
10     IEND = MIN(ISTART+NB_ELT_GR-1,
   NELEM)
11     CALL GROUP_CONSTR (A → A′)
12     DO IELOOP = ISTART, IEND
13       CALL LOOPELM (A')
14     ENDDO
15     CALL UPDATE (A′ ← A)
16   ENDDO
17 END SUBROUTINE COMPUTE
```

Algorithm 2: Skeleton of the XKAAPI code used to parallelize the outer loop on GROUPS (Algo. 1)

```
1  Subroutine COMPUTE(A)
2  !      Initialization step
3    NGR = NELEM/NB_ELT_GR
4    IF (NGR × NB_ELT_GR <
   NELEM) THEN
5      NGR = NGR+ 1
6    ENDIF
7    Err =
   KAAPIF_SET_GRAIN(...)
8    Err =
   KAAPIF_SET_POLICY(...)
9    Err =
   KAAPIF_SET_DISTIBUTION(...)
10   Err = KAAPIF_FOREACH(...,
11 &          KAAPI_GROUPS, ...)
12 END SUBROUTINE
   COMPUTE
```

3 Parallel Implementation

Our implementation relied on the XKAAPI task oriented programming environment[2]. XKAAPI is a C++ library providing a low overhead work stealing scheduler. Parallelizing a loop with independent iterations simply require to rely on the `KAAPI_FOREACH` callback. The `KAAPIF_SET_POLICY` function enables to control the scheduling algorithm used for parallelizing the loop. The scheduler can be set to be static. In this case the user specifies an initial distribution pattern of the loop iterations through `KAAPIF_SET_DISTIBUTION`. A possibility is to split the iteration range in p sub ranges, where p is the number of cores involved. When starting the execution all cores are responsible to process one range. If the work stealing scheduler is used, the actual parallelism is extracted on demand according to the core availability, leading to a dynamics work load balancing. The scheduling algorithm works as follow: The first core initially gets the full iteration range and starts processing the loop sequentially. When a core becomes idle, it randomly selects a victim core. If this victim core is in charge of executing an

[2]https://gforge.inria.fr/projects/kaapi/

iteration range, the idle core steals half of this iteration range. If the victim has no work to steal, an other victim is randomly selected. A grain parameter (`KAAPIF_SET_GRAIN`) prevents that an iteration range is split in two if too small, to avoid a performance degradation: the cost of stealing would not be compensated by the extracted parallelism.

The work stealing scheduling algorithm has proven performance [3]. Today several parallel programming environments like Cilk [2], TBB [12] or XKAAPI [8] rely on low overhead work stealing schedulers. The decentralized model of work stealing enable to ensure a good scalability at high core count in opposite to dynamics scheduler that rely on a centralized list cores refer to when they need more work. We made the choice to rely on XKAAPI, but the implementation could have been equally done with OpenMP. OpenMP offers all necessary constructs and several OpenMP runtimes rely on work stealing schedulers. In particular the XKAAPI scheduler can be used as runtime for OpenMP [4].

We first parallelized the original code (without group construction) with `KAAPI_FOREACH` on the loop on the elements. For the version with group construction (Algo. 1), the loop on the groups (line 7) was parallelized with a `KAAPI_FOREACH` (Algo. 2), the work inside a group being sequentially executed by the core in charge of that group (Algo. 2). Thus in parallel cores construct groups, compute the new states of the elements of their group accessing only data local to their group and write back the updated new element values to the global data structure. Because we expect that the work related to one group be sufficiently large, a grain of one can be used: a core can steal an iteration loop to a victim if this victim has at least 2 iterations left (the one currently processed and one waiting to be executed).

4 Experiments

4.1 Environment and Use Cases

All experiments are performed on a machine with 4 eight-cores CPU Intel Xeon clocked at 2.2 Ghz (hyperthreading disabled). Each core has its own L1 and L2 cache (256 KB) . The L3 cache (16MB) is shared among the eight cores of each CPU. We use XKAAPI version 3.0.6 and EUROPLEXUS development version. For all experiments, each thread is pinned on a different core of the machine. For this reason, in the following the term core is also used to designate the thread running on this core. All reported measures are the average over 5 runs to obtain significantly stable values.

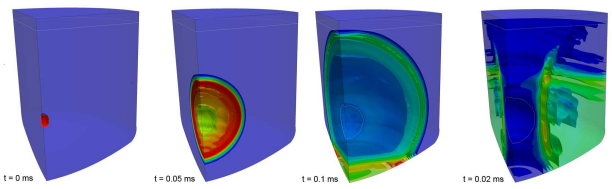

Figure 2: Gas bubble expansion and pressure waves in the early moment of the MARA 2 simulation

Experiments are based on the so called MARA2 simulation (Figure 2) named after an experiment in the field of nuclear safety and consisting in the relaxation of an explosive in a closed vessel. The fluid in the vessel is composed of a liquid covered with a gas, with a compressed bubble of another explosive gas located at the center of the vessel. The pressure

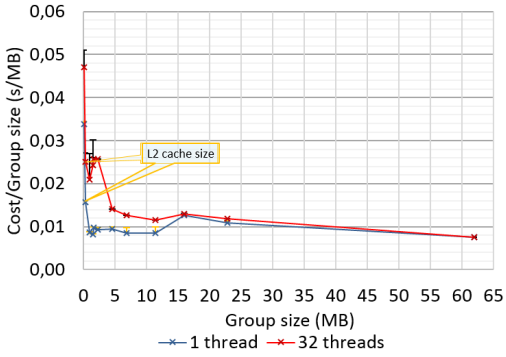

Figure 3: GROUP construction costs for different sizes for sequential and 32 threads runs for Mara_mdm

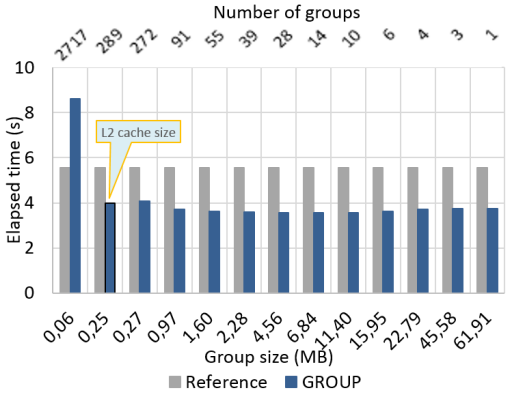

Figure 4: Elapsed time for different group sizes for a sequential execution (Reference: single loop, no group; GROUP: 2 nested loops with GROUPS) for Mara_mdm

in the elements is computed via an iterative system adjusting the volume fractions of each component to balance the partial pressures. The cost is thus irregular and evolves during the simulation as it depends on whether the cells are filled with a single fluid material or a mixture of different fluids. We tested two variations of MARA2, Mara_mdm composed of 271.615 elements (about 61 MB), and a simulation with a finer meshing, Mara_big, composed of 2.172.920 mesh elements (about 495 MB).

4.2 Results

We first evaluate the cost of building a group for different group sizes. Figure 3 plots the average time spent to build one group divided by the group size (Algo 1 routine `GROUP_CONSTR`). These performance results are obtained for both, a sequential run (single thread) and one run parallelised on 32 threads. The results of both versions shows that the sequential version prevails slightly the 32 threads one. This is due to the intensification of the memory traffic when a large number of threads accesses simultaneously the shared memory. Notice that for the largest group size the performance for the sequential an parallel run are identical. In this case we only have one single group, and thus the parallel execution is reduced to a sequential one. This plot mainly highlights that the cost of building very small groups is high will it stabilises around 10ms/MB for larger groups.

We now compare the execution time of the `Reference` execution that corresponds to the original implementation using a single loop and no group, and the group execution that corresponds to the 2 nested loops with the group construction (Figure 4). In this last case the execution time includes the cost of the group construction as well as the update of the elements in the global EPX data structure. For group sizes above 0,25 MB, the `GROUP` version is about 35% faster than the reference one. The data rearrangement into each group leads to improved performance, mainly due to a more efficient cache usage. This also occurs for the largest group size where only one groupe is created. The variation of the `GROUP` version is closely related to the cost of group construction (Figure 3). Overall, the overheads of group creation and update

of EPX main data structures stay small enough to keep significant performance gains.

We now run tests with 32 threads using both Mara_mdm (Figure 5) and Mara_big dataset (Figure 6). In each case we compare four different parallelizations: the `Reference` code (resp. `GROUP` code) with a static data partitioning (`Reference-static`, resp `Group-static`) and with work stealing enabled dynamic load balancing (`Reference-steal`, resp. `Group-steal`).

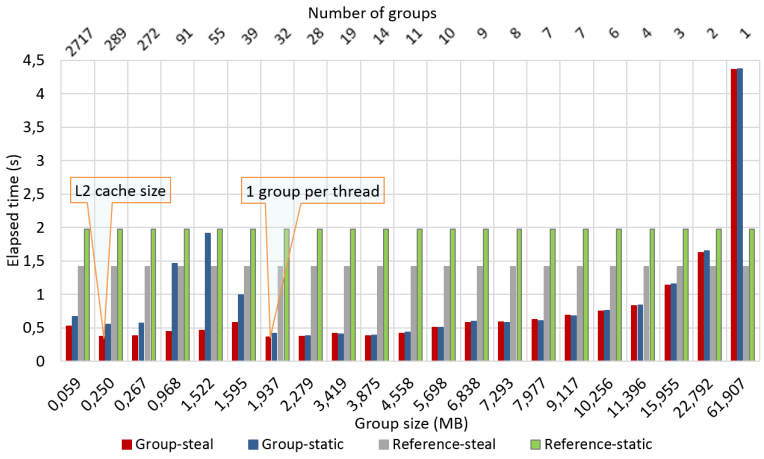

Figure 5: The elapsed time in the elementary loop with different GROUP sizes for a 32 threads run using the Mara_mdm dataset

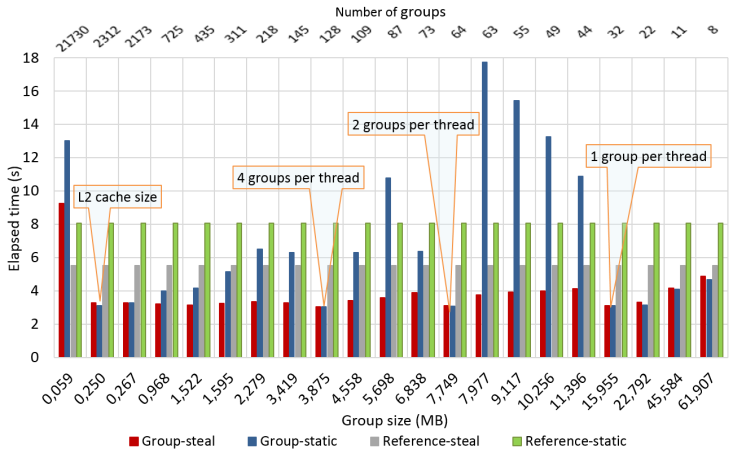

Figure 6: The elapsed time in the elementary loop with different GROUP sizes for a 32 threads run using the Mara_big dataset

The work stealing enabled `Reference` version outperforms the static one by about 20% for both, the Mara_mdm and Mara_big datasets. The `GROUP` code brings in most cases significant performance gains over the `Reference` code, the work stealing version often being the best

one. Notice in particular that the performance of the static version can be affected by a load imbalance. For instance when there is only 63 groups (Figure 6) one thread has only one group while all the other have 2 groups each, causing a major performance drop. Work stealing enables to smooth this type of effect. It enables to take advantage of the uneven group processing time by allocating unprocessed groups to the fastest cores.

The acceleration with 32 threads is about 21 if we compare the performance for the Mara_big dataset. It is 3 times greater than the reference acceleration. However, the speedup of the Mara_mdm (Figure 4 and figure 5) simulation is 2 times lower than the Mara_big one. This performance gap is due to the limited size of this dataset which reduces the parallelism that could be extracted in this case.

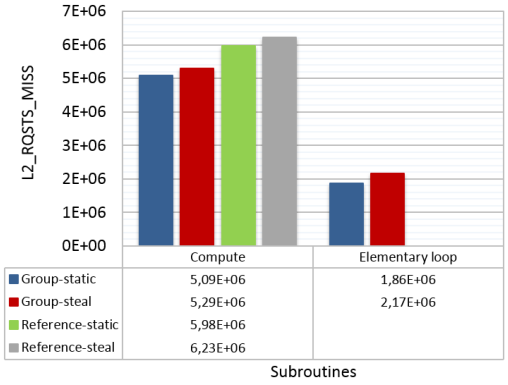

Figure 7: The L2 cache misses measurement within the EPX subroutines using the Mara_mdm dataset

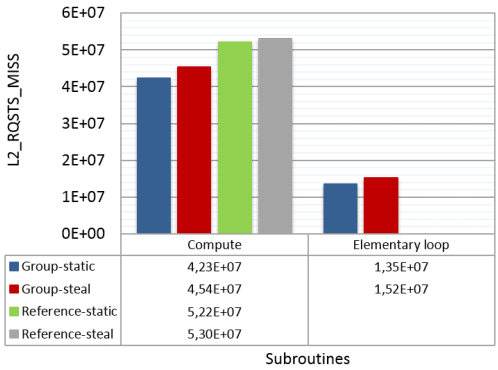

Figure 8: The L2 cache misses measurement within the EPX subroutines using the Mara_big dataset

We also looked at the average number of the L2 misses per core for the execution with 32 threads and groups of size 0.25 MB (L2 cache size) (Figure 7 and Figure 8). These L2 caches misses numbers were obtained using LIKWID[3] [14]. For each figure we provide the number for the execution of the full `COMPUTE` routine (Algo. 1) as well as the execution of the inner loop for the GROUP code (Algo. 1 `DO IELOOP`). The GROUP based approach enables to improve the number of L2 cache misses by about 15%, with about 35% of the remaining cache misses occurring in the inner loop, the others being spent when creating the groups (`GROUP_CONSTR` routine) and updating the main data structure. (`UPDATE` routine). The work stealing based GROUP code generates slightly more cache misses than the static one. We attribute this to the coherency management over the L2 private cache and the L3 cache caused by the data migration during the steal operation.

Globally these results show that it pays off to dynamically build groups. The best results are obtained when the groups can tightly fit into the L2 cache. This lead to many small groups, enabling a fine grain load balancing but without the extra overheads that smaller groups lead to. The performance with larger group sizes is less stable and more dependent on the data set.

[3] https://code.google.com/p/likwid/

5 Related work

M. Wolf [16] worked on algorithms to reorder nested loop to improve the cache efficiency by improving data locality. This approach requires changes in the code while working on the bounds of the loops as well as on their bodies. It is not always applicable especially in the case of codes with complex dependencies. M.Kandemir et al. [10] favored memory data reorganisation using linear algebra techniques to perform transformations on the matrices of data. P. Clauss [5] proposed a more general method based on polynomial transformations applied to the data access. This requires the parameterisation of the loop bounds and the array sizes. The advantage of these approaches based on the reorganization of data tables is that the code modifications are restricted to the accessed data tables and do not affect the internal routines. However, they require an a priori knowledge on the order of data accesses.

Other researchers have worked on the data reordering in the memory to promote the temporal and the spatial locality. They apply specific algorithms of mesh elements renumbering like the Cuthill McKee [11, 6, 7] algorithm used in some implementations of the mesh creator tool CAST3M. This algorithm permutes the rows of a sparse matrix to reduce the required bandwidth when accessing the data. The space filling curves (SFC) [13] are another way to improve the cache use. They consist in projecting the multidimensional physical space in a one-dimensional space, while maintaining as much as possible the spatial locality of elements. SFCs require to reorganize the global data structure with a priori knowledge on the access pattern. These methods are cache oblivious as the data layout is built independently of any specific cache parameter. Cache oblivious algorithms can be very efficient, but often outperformed by cache-aware approaches where the data layout is built knowing the cache parameters [17].

6 Conclusion

In this paper we propose to dynamically build groups gathering all the data needed for processing a given iteration sub-range. This approach enables to significantly improve performance while imposing only very local code modifications. We did not need to revisit the main data structures of the simulation software. Experiments show that our approach can lead to 40% performance increase on 32 cores when choosing groups that tightly fit in the L2 cache. We expect that this approach can be used in other simulation software than EPX on loops whose performance is impacted by excessive cache misses. Futur work will investigate the benefits of this approach for architectures with scratch pad. Next generation of Xeon Phi processor will have a NVRAM (the main memory) and a smaller high bandwidth RAM (the scratch pad). In this case the user needs to explicitly control the data to be stored in the RAM [1]. We expect that with minimal code modifications we can have our groups stored in the RAM while the globale data structure is stored in NVRAM, enabling to efficiently take advantage of the higher speed of the RAM.

References

[1] Michael A. Bender, Jonathan W. Berry, Simon D. Hammond, K. Scott Hemmert, Samuel McCauley, Branden Moore, Benjamin Moseley, Cynthia A. Phillips, David S. Resnick, and Arun Rodrigues. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. In *IPDPS*, pages 835–846. IEEE Computer Society, 2015.

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[4] François Broquedis, Thierry Gautier, and Vincent Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Philippe Clauss and Vincent Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI signal processing systems for signal, image and video technology*, 19(2):179–194, 1998.

[6] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[7] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In DonaldJ. Rose and RalphA. Willoughby, editors, *Sparse Matrices and their Applications*, The IBM Research Symposia Series, pages 157–166. Springer US, 1972.

[8] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors. In *Parallel Symbolic Computation'07 (PASCO'07)*, number 15–23, London, Ontario, Canada, 2007. ACM.

[9] Nicolin Govender, Daniel N. Wilke, Schalk Kok, and Rosanne Els. Development of a convex polyhedral discrete element simulation framework for {NVIDIA} kepler based {GPUs}. *Journal of Computational and Applied Mathematics*, 270:386 – 400, 2014. Fourth International Conference on Finite Element Methods in Engineering and Sciences (FEMTEC 2013).

[10] M Kandemir, A Choudhary, J Ramanujam, and P Banerjee. Optimizing spatial locality in loop nests using linear algebra. In *Proc. 7th Workshop Compilers for Parallel Computers*, volume 426, page 430. Citeseer, 1998.

[11] Wai-Hung Liu and Andrew H. Sherman. Comparative analysis of the cuthillmckee and the reverse cuthillmckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2):198–213, 1976.

[12] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[13] Hans Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[14] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 207–216, Washington, DC, USA, 2010. IEEE Computer Society.

[15] I. Wald. Fast construction of sah bvhs on the intel many integrated core (mic) architecture. *Visualization and Computer Graphics, IEEE Transactions on*, 18(1):47–57, Jan 2012.

[16] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[17] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 93–104. ACM, 2007.