Fork-join and data-driven execution models on multi-core architectures: Case study of the FMM

Fork-Join and Data-Driven Execution Models on Multi-Core Architectures: Case Study of the FMM

Abdelhalim Amer[1], Naoya Maruyama[2], Miquel Pericàs[1], Kenjiro Taura[3], Rio Yokota[4], and Satoshi Matsuoka[1]

[1] Tokyo Institute of Technology, Tokyo, Japan
[2] RIKEN, Kobe, Japan
[3] The University of Tokyo, Tokyo, Japan
[4] KAUST, Saudi Arabia

Abstract. Extracting maximum performance of multi-core architectures is a difficult task primarily due to bandwidth limitations of the memory subsystem and its complex hierarchy. In this work, we study the implications of fork-join and data-driven execution models on this type of architecture at the level of task parallelism. For this purpose, we use a highly optimized fork-join based implementation of the FMM and extend it to a data-driven implementation using a distributed task scheduling approach. This study exposes some limitations of the conventional fork-join implementation in terms of synchronization overheads. We find that these are not negligible and their elimination by the data-driven method, with a careful data locality strategy, was beneficial. Experimental evaluation of both methods on state-of-the-art multi-socket multi-core architectures showed up to 22% speed-ups of the data-driven approach compared to the original method. We demonstrate that a data-driven execution of FMM not only improves performance by avoiding global synchronization overheads but also reduces the memory-bandwidth pressure caused by memory-intensive computations.

1 Introduction

Hardware manufacturers now focus on multi-core and many-core technologies as a way to increase performance and make use of the continually increasing number of transistors. The multi-core road-map provides a slowly increasing number of processing units with each new generation, while maintaining high single thread performance thanks to their sophisticated control logic, out-of-order execution, and their complex memory hierarchy. However, this architectural design leads to unprecedented programming difficulties to extract their potential due mainly to its memory subsystem. Non Uniform Memory Access (NUMA), memory bandwidth limitations, and complex memory hierarchies are key properties that hinder programmer productivity.

In order to exploit parallel architectures, different execution models can be adopted. In a fork-join model, independent tasks run concurrently while task

dependencies are ensured by global synchronization barriers. Data-driven (also called data-flow) models have been proposed in order to avoid global synchronizations and improve resources exploitation. However, proponents of the fork-join model argue that data-flow models have worse memory behavior. As a result, fork-join and data-driven methods have their trade-offs, and the achievable performance will depend both on the algorithm and the target architecture.

In the present work, we study these execution models on state-of-the-art multi-core architectures by using the *Fast Multipole Method* (FMM). We choose the FMM, given its wide usage in many scientific domains such as astrophysics[1], electrodynamics[2], and fluid dynamics[3]. Furthermore, FMMs are composed of heterogeneous computations following a complex execution flow. Moreover, we rely on one of the fastest FMM codes and its highly tuned OpenMP fork-join parallel implementation [4][5]. To implement a data-driven execution, there are many runtime schedulers or programming models which have the ability to express task dependencies and perform an asynchronous execution. StarPU[6], OmpSs[7], and Quark[8] are examples of such tools. However, we choose to implement our data-driven FMM using the lightweight thread library MassiveThreads [9]. The low overhead, flexibility, and load-balancing mechanism of this library let us implement an efficient fine-grain data-driven FMM which uses a distributed scheduling approach.

We summarize our contributions and findings as follows:

- We implement a novel thread-based data-driven FMM by using a source centric approach and efficiently managing the task dependencies using a distributed scheduling approach. We further reduce data movements by exploiting the tree nature of the FMM data-structures and using sub-tree based working sets per thread.
- We perform an in-depth analysis of the original implementation which reveals that at large scale the often neglected stages at smaller scale consume more time than the usually compute intensive ones.
- Our evaluation on state-of-the art x86 multi-core architectures, showed that the data locality issues of the data-driven execution are not significant, and when eliminating the synchronization overheads, this method achieved up to 22% speed-ups over the original implementation.
- We also found that the data-driven approach can reduce localized high memory bandwidth stress by spreading the memory traffic along the execution. Moreover, we prove that the memory bound nature of one of the kernels is not only due to its low arithmetic intensity but also bound by remote memory transfers and a non-unit-stride memory access pattern.

The rest of the paper is organized as follows: Section 2 introduces the Kernel Independent FMM and its different computational stages. Then, we discuss our data-driven implementation in Section 3. We describe the configuration of our tests and the details of the target multi-core machines in Section 4. In Section 5, we evaluate and analyze both execution models on the target machines. In Section 6, we discuss related work and we conclude in Section 7.

2 The Fast Multipole Method

Nbody problems can be encountered in many disciplines such as mathematical physics, machine learning, approximation theory, etc. The problem is how to efficiently evaluate pairwise interactions between N bodies. It can be formally described as follows:

$$f(x_i) = \sum_{j=1}^{N} K(x_i, y_j)s(y_j), i = [1..N] \quad (1)$$

where $f(x_i)$ is the potential at the target x_i resulting from the sources y_j, s the source density, and K the interaction kernel. A direct computation results in a $O(N^2)$ complexity which makes it very expensive for large problem sizes. First attempts towards a faster method brings the complexity to $O(NLogN)$ like the Barnes-Hut method[10]. The FMM was proposed as an even faster solution that uses a rapidly convergent method achieving a $O(N)$ complexity [11].

Most of FMMs rely on analytic expansions to evaluate pairwise interactions. Analytical expansions are problem dependent, not always available, and difficult to build. In this work we use the Kernel-Independent FMM (KIFMM) developed by Ying et al. which relies only on kernel evaluations, thus enabling FMMs to a wider range of engineering and scientific problems [12][13]. In KIFMM, the domain is represented by an octree of cells, where interaction lists are built for each cell following Greengard notation [14], namely: U-list, V-list, W-list, and X-list. The KIFMM implements the force evaluation through the following large stages: U-list, Upward, V-list, X-list, W-list, and Downward. These stages are synchronized by global barriers, are embarrassingly parallel, and traverse the tree cells independently (for the list computations) or level-by-level (Upward and Downward). We distinguish two independent flows of computation: the near field *direct evaluation* represented by the U-list computation, and the *far-field approximation* starting from the Upward stage, computing V-list, W-list, and X-list stages and finishing by the Downward stage. We note that the W-list and X-list computations are negligible for a uniform distribution of bodies.

3 Data-driven implementation

In this section we discuss the implementation of our data-driven solution. That is, the flow of execution goes from the sources to the targets where the far-field and direct evaluation computations are merged into a single flow by starting the Upward and the direct evaluation at the same time.

3.1 From target centric to source centric

In KIFMM, the data structures are built from a target centric point of view, thus these data structures need to be rebuilt from a source centric view to enable a data-driven execution. Although in theory if a cell A interacts with cell B, B

will interact with A whether symmetrically (U and V lists) or dually (W and X lists), in practice it depends on how a cell's neighbors are determined. Indeed, a cell's interaction lists are only built around a neighborhood, and in KIFMM this neighborhood does not ensure bidirectional interactions between two cells. In order to maintain a correct behavior of the algorithm in a data-driven execution, we compute these lists from a source point of view.

3.2 Thread-based data-driven implementation

The dependencies in the data-driven execution can be seen as a producer-consumer synchronization problem as shown in the simplified FMM far-field computation task dependency graph in Figure 1.(a). In our implementation, each task is aware of the tasks that depend on it and may trigger their execution upon termination. Moreover, the task dependencies are satisfied using a combination of *recursive calls* and *atomic counters*. For instance, an atomic counter is used at the `Down` task dependency in Figure 1.(a) which is updated by other `Down` tasks or `V` tasks. In the following we give an example on how a V-list task is executed for a source cell (`src`) after it was called by an `Up` task:

```
void* V (src){
  for(trg in Vlist(src))              //Compute the contribution of src
  {                                     to all the target cells that
     compute_V(trg,src);                depend on it
     trg.down_counter++;              //Atomic incrementation of
                                        the synchronization counter
     /* Test if all dependencies are satisfied */
     if(trg.down_counter = nb_input_depend(trg))
           create_task(Down, trg);    //Start Down computation.
     }
  }
}
```

This pseudo-code shows the V-list and Downward computation tasks (`V` and `Down` resp.) and the target's synchronization counter (`trg.down_counter`) between them. In the Massivethreads library, tasks are embedded in lightweight threads scheduled to be executed by *workers*. Each *worker* is an OS-thread and has a private queue of ready tasks which is managed by a LIFO (Last In First Out) scheduler and a FIFO (First In First Out) work stealing policy between *workers* is adopted. As a result, the `Down` task will be executed first and the `V` task goes at the front of the worker's ready queue. We note that the creation of tasks is incremental and done at the worker level, while the first created tasks, which are situated at the back of the ready queue, may be stolen by other workers ensuring good load-balancing. Since each *worker* is scheduling the tasks to be executed independently from the others and uses a private task queue, this method results in a *distributed scheduling* scheme avoiding a *centralized scheduler* that will constitute a potential bottleneck. We note that, being oblivious of which stage in KIFMM, contributions from many cells may be reduced at a target cell. While this is naturally serialized in the original target approach, in

our source approach, we serialize these updates by using the *locks* provided by the lightweight thread library.

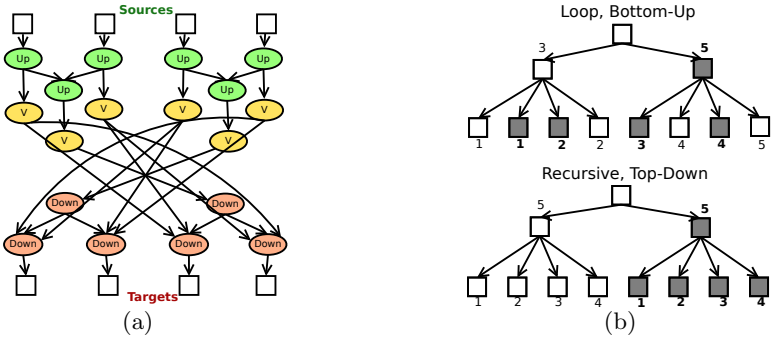


Fig. 1. (a) Simplified FMM far-field computation task dependencies. (b) Simple example of the Upward tasks executed by two *workers*: white for the first *worker* and gray for a second *worker*. The numbering shows a possible task execution order.

3.3 Effects of the data-driven FMM on data locality

The dependency between a `V` and a `Down` task, as described in Section 3.2, corresponds to a *read* after *write* hazard and results in temporal data reuse. Such data reuse can be observed along the paths going from sources to targets. However assessing the spatial data reuse is more subtle since it depends on how the tasks are scheduled. First, one may implement the task graph of Figure 1.(a) by traversing the leaf boxes and spawning the Upward and V-list tasks. This method requires *synchronization counters* where each Upward task atomically increments its parent's *counter* upon termination, and triggers the Upward task of its parent if it is the last child. In addition, *workers* will likely access non-contiguous cells in the tree. Indeed, the *serial* code to create all the leaf tasks is also considered as a task which will be preempted and put in the *worker*'s ready queue. A second *worker* will steal that task and create the second leaf task and so on. As a result, the *workers* will access randomly the data as shown by the upper part of Figure 1.(b).

To overcome this issue, we use a top-down recursive algorithm to spawn the tasks as shown in the lower part of Figure 1.(b). In this approach, the work stealing happens in the upper levels of the tree and results in a sub-tree working-set per *worker*, thus, ensuring a better spatial and temporal locality and avoiding additional synchronization variables.

4 Test-bed configuration

We choose to follow the same input problems as in [4]. That is, we simulate the evaluation of a single step with 4 million bodies following two distributions: a unit cube uniform and an elliptical non-uniform distribution. As for the interaction kernel we use the Laplace kernel. For each target machine, we manually tune the maximum number of bodies per cell parameter. We only consider double-precision computation because of its higher pressure on the memory subsystem and the document space limitations. As for the target architectures, we select representatives of NUMA multi-core architectures, with a 2 socket Intel Sandy-Bridge-EP, a 4 socket Intel Nehalem-EX, and a 4 socket 8 NUMA-nodes AMD Magny-Cours with their detailed specifications given in Table 1.

Table 1. Target machine specifications. We report the memory bandwidth as the maximum value achieved by the Stream benchmark [15]

	Sandy-Bridge-EP	Nehalem-EX	Magny-Cours
Processor	Xeon E5-2620	Xeon X7550	Opteron 6172
CPU Frequency (Ghz)	2.0	2.0	2.1
# Sockets	2	4	4
# NUMA-Nodes	2	4	8
#Cores/NUMA-Nodes	6	8	6
L3 Cache size (MB)	15	18	6-1
Memory BW (MB/s)	52590.4	68827.3	74720.4
Compiler	GCC 4.4.6	ICC 11.1	GCC 4.4.5

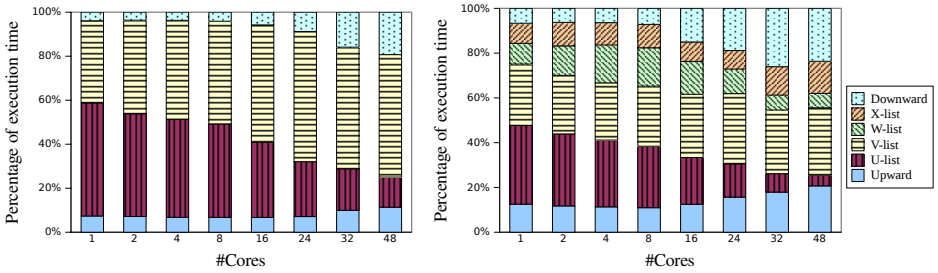
5 Performance evaluation

In this section we will conduct a performance analysis of the original KIFMM design approach as described in [4] and [5]. However we do not consider the intermediate and advanced tuning techniques introduced in [5], as these techniques can also be adapted for a data-driven execution. Our methodology is guided by the high-level knowledge of the application and also relying on hardware performance monitoring tools. For the latter purpose, we use the Vampir tool-set [16][17] combined with native hardware counters accessible through the PAPI library [18]. In addition, we use the VTune tool to report memory-bandwidth measurements on the Sandy-Bridge-EP machine [19].

5.1 FMM stages at large scale

It is well known in the FMM literature that the U-list and V-list computations dominate the serial execution time. However, after parallelization, not all of the stages scale in the same way, and the dominant stages at larger scale may differ.

To verify our assumptions, we run strong scaling simulations using the original implementation on the Magny-Cours machine and we report the percentage of execution time taken by each stage as shown in Figure 2. These results were reported using high resolution timers without tracing the execution in order to avoid unnecessary overheads. We observe that although the U-list stage takes the longest time when running sequentially, at full concurrency it takes the smallest amount of time while the opposite is observed for the other stages. In the following section, we shed light on the reasons behind this disparity in parallel efficiency of the stages while performing a deep comparative analysis of both FMM implementations on the target machines.



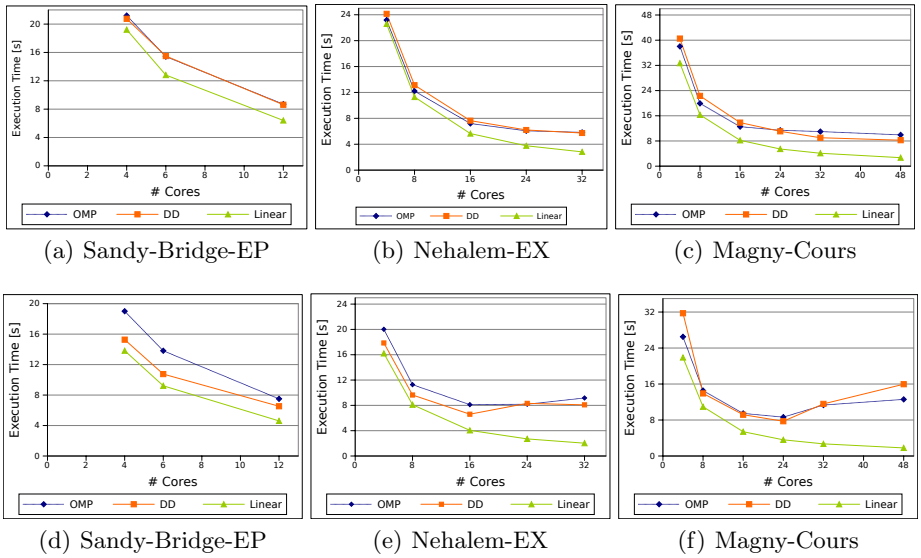
(a) Uniform distribution (b) Elliptical distribution

Fig. 2. Percentage of execution time for each stage on a the Magny-Cours machine for uniform and elliptical distributions.

5.2 Comparative analysis

To decrease the negative effects of NUMA in the data-driven execution, we use the `numactl` command to interleave the memory allocation on the NUMA-nodes where there exists a MassiveThreads *worker*. For both implementations, the OS-threads are scattered across the sockets and bound to the cores to optimize the memory bandwidth. Figure 3 shows the strong scaling of each implementation on each machine using both distributions and indicates an overall better scaling of the data-driven execution. However we observe that both implementations exhibit very limited speed-ups after using more than half the cores. In particular, the Magny-Cours machine has the worst scaling likely due to a smaller last level cache. Also, for the elliptical distribution, there is a 22%, 18%, and 10% speed-up of the data-driven execution over fork-join when using half of the cores on the Sandy-Bridge-EP, Nehalem-EX, and Magny-Cours machine, respectively. To better understand this scaling disparity, a deeper analysis of the latter case is performed as it showed the greatest gap between the two methods.

We record statistics for each stage and also for the total force evaluation as shown in Table 2. The computation times do not include scheduling and syn-



(a) Sandy-Bridge-EP (b) Nehalem-EX (c) Magny-Cours

(d) Sandy-Bridge-EP (e) Nehalem-EX (f) Magny-Cours

Fig. 3. Strong scaling of the OpenMP fork-join (OMP) and the data-driven (DD) implementations for uniform (a,b,c) and elliptical (d,e,f) distributions. To better appreciate the scaling results we added a linear scaling plot.

chronization overheads thus, the differences between the methods are only due to data movements. We used native counters rather than PAPI preset counters which were not enough to gather the information of interest. Native counters are machine dependent, thus we follow the guidelines of the hardware manufacturers to derive our metrics for the Magny-Cours [20] and the Sandy-Bridge-EP [21] machines. However, to the best of our knowledge, similar guidelines are not available for the Nehalem-EX machine, thus we do not report its memory-bandwidth measurements.

We can observe that the data-driven computation time is close to that of the original implementation, indicating that the synchronization overheads eliminated by our method did not detrimentally affect the data locality. In this experiment we observed improvements of 14%, 17.3%, and 14% resp. for the data-driven which deviate slightly from the above mentioned speed-ups due likely to tracing overheads and operating system noise.

We notice that our method ensures a better locality for the Upward computations, which hides the slower V-list execution time. In order to verify the locality benefit of using sub-tree based working-sets, a similar approach was implemented for the Upward stage using the fork-join model by manually partitioning the tree among the threads. The results, as reported in the last row of Table 2, show that the computation runs faster at the cost of a very large synchronization overhead. We also observe that most of the synchronization overheads stem from the X-list and W-list computations. This is not surprising since these computations exhibit

Table 2. Computation time (without scheduling and synchronization overheads), OpenMP synchronization overhead, average bandwidth consumption, and relative computation time of the data-driven execution per machine for the elliptical distribution running on half the cores. Note that important information is highlighted. Abbreviations: DD (Data-Driven), SB (Sandy-Bridge-EP), NH (Nehalem-EX), MC (Magny-Cours), and N/A (Not Available).

	Comput. Time(s)			Sync. Overhead(%)			Bandwidth(GB/s)			DD Relative Time(%)		
	SB	NH	MC	SB	NH	MC	SB	NH	MC	SB	NH	MC
U-list	27	30.5	38.5	7	14.8	14	0.1	N/A	0.4	-2.96	-2.30	-11.69
Upward	9.56	15.7	43.2	1.2	0.2	7.36	1.2	N/A	0.68	**-4.60**	**17.20**	**44.44**
V-list	13.42	24.7	36.7	1.8	8	1.34	**6**	N/A	**6.8**	**-8.05**	**-10.12**	**-32.15**
W-list	7.3	8.05	10.67	**56.7**	**62**	**61**	0.1	N/A	0.2	0.00	-4.60	-7.78
X-list	7	7.96	15	**29.4**	**25.5**	**24**	0.1	N/A	0.38	-2.86	-3.27	23.00
Downward	5.9	14.9	51.9	2.1	0.2	1.9	1.8	N/A	0.4	0.00	-0.54	1.54
Total OpenMP	70.18	101.8	195.9	**15.6**	**18.8**	**15**	1.3	N/A	1.8	**-3.59**	**-1.19**	**3.22**
Data-Driven	72.7	103	190	0	0	0	**2.7**	N/A	**4.4**			
Upward static	**9.32**	**13.4**	**18.58**	**55**	**24.5**	**45.3**						

the highest variation in the work per cell and results in high load-imbalance. An attempt to fix this by means of a `dynamic` or a `guided` OpenMP scheduler resulted in worsening the data locality and increasing the OpenMP scheduling overhead leading to a longer execution time. This validates our strategy which achieves a better trade-off between locality and synchronization overhead.

For a uniform distribution, we observed less synchronization overheads, a worse data locality, and more bandwidth consumption, due to a larger V-list computation, which reduces the effectiveness of the data-driven execution.

5.3 Analysis of the memory bandwidth consumption

According to the memory bandwidth measurements of Table 2, most of the memory traffic comes from the V-list stage which is known to be memory bound. The memory behavior of this computation can be explained as follows: V-list target-source interactions can be seen as a sparse matrix pattern with high spatial locality and temporal reuse regions at the diagonal [5]. These regions are limited (roughly half of the total sources for a uniform distribution) while the rest of the sources are streamed in a non-unit-stride fashion. In addition to the source cells, V-list uses translation vectors, which are also accessed in a non-unit-stride pattern, and further increases the working-set size. We conclude that the V-list bandwidth is consumed by streaming a large working-set following mostly a non unit-stride memory access pattern. Furthermore, reading the sources and translation vectors in a NUMA-aware fashion is not guaranteed.

The data-driven execution of FMM resulted in a homogeneous memory bandwidth consumption rather than concentrated only in the V-list computation. Thus, on a machine with a low memory bandwidth, this execution model will help reduce localized high memory traffic and the performance may improve as long as the overall data locality is not severely hindered. However, for our target

machines this was not observed when comparing the V-list bandwidth in Table 2 with the Stream bandwidth in Table 1 for each machine. The limited scaling of V-list can be explained by the Roofline model [22]. We draw the Roofline plot for the the Sandy-Bridge-EP machine along with the performance achieved by U and V-list computations at full concurrency in Figure 4. V-list has a low arithmetic intensity, as opposed to the compute bound U-list, a mixture of unit-stride and non-unit-stride memory accesses, and also local and remote DRAM accesses. Hence, V-list is partially affected by each memory ceiling in the Roofline plot which explains the limited achievable bandwidth and the performance.

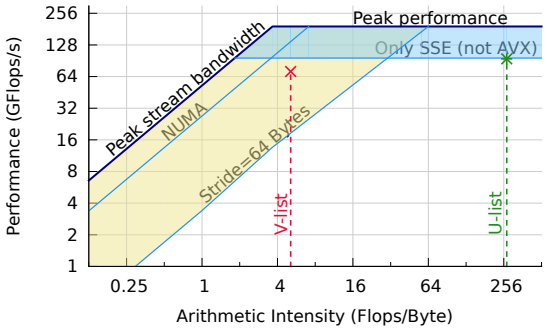


Fig. 4. Roofline of the Sandy-Bridge-EP machine. The NUMA memory ceiling was obtained using the Stream benchmark with only remote accesses, which was then augmented with 64 bytes strided accesses to plot the stride ceiling. We use SSE vector instructions and do not exploit AVX instructions which halves the computational power. The arithmetic intensity of the computations were derived from machine counters

6 Related work

Data-driven execution of FMM is not novel. Yokota et al. [23] proposed a data-driven execution of FMM in order to overcome load-balancing issues. Agullo et al. proposed to pipeline the FMM computations on heterogeneous architectures over a runtime [24]. Pericàs et al. implemented a data-driven execution of ExaFMM, a fast open-source FMM [25]. Although these works used a data-driven approach to implement the FMM, their objectives were to load-balance the work among the computational units. Our work also achieves this goal, proposes a novel distributed scheduling scheme, and further presents an in-depth comparison with a fork-join execution model. Using also the MassiveThreads library, Taura et al. described parallel recursions as an alternative to parallel loops for implementing ExaFMM [26]. Our methodology can be applied to this work in order to evaluate the implications of using recursions to implement task parallelism. Towards understanding the performance of multi-core machines, some

authors used stencil-computation and sparse matrix-vector multiplication kernels and optimized them for state-of-the-art multi-core architectures[27] [28]. These works use small kernels as benchmarks while going deeper in architectural details in order to get insight into performance trade-offs. In our work, we use FMM, a sizable algorithm which uses multiple kernels, as a benchmark to get insight into the performance of different execution models.

7 Conclusion and future work

In this work, we study multi-core architectures' performance given a fork-join and a data-driven execution models. We implemented a data-driven execution of the FMM using a distributed scheduling approach and observed improvements of up to 22% in execution time as compared to the fork-join approach. We concluded that for an algorithm such as a FMM, a data-driven execution is more suitable on our target machines as trading-off the inferior data locality by removing the synchronization overheads was beneficial. The benefit of the data-driven execution grows at scale reaching the best speed-ups with half of the cores, after which both methods are limited by the scalability of the memory intensive kernel. This kernel is not limited by the memory bandwidth in our experiments, but it is rather a combination of low arithmetic intensity and a sparse NUMA pattern that reduces the achievable bandwidth. This work can be extended to analyze the effects of task-coarsening and adapting the advanced optimizations applied to the original OpenMP implementation [5]. Our preliminary attempts in task-coarsening, by aggregating the work of leaf siblings in the tree, resulted in an average 5% speed-up, which encourages pursuing this direction.

References

1. Dehnen., W.: A hierarchical o(n) force calculation algorithm. Journal of Computational Physics **179(1)** (2002) 2742
2. S. Chaillat, M.B., Semblat, J.F.: A multi-level fast multipole bem for 3-d elastodynamics in the frequency domain. Computer Methods in Applied Mechanics and Engineering **197** (2008) 42334249
3. Yokota, R., Narumi, T., Barba, L.A., Yasuoka, K.: Petascale turbulence simulation using a highly parallel fast multipole method. (2011)
4. Chandramowlishwaran, A., Williams, S., Oliker, L., Lashuk, I., Biros, G., Vuduc, R.: Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on. (april 2010) 1 –12
5. Chandramowlishwaran, A., Madduri, K., Vuduc, R.: Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC '10, Washington, DC, USA, IEEE Computer Society (2010) 1–12
6. Augonnet, C., Thibault, S., Namyst, R.: StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Rapport de recherche RR-7240, INRIA (March 2010)

7. Duran, A., Ayguade, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters (2011) 173–193
8. YarKhan, A., K.J.D.J.: Quark users' guide: Queueing and runtime for kernels. Technical report, University of Tennessee Innovative Computing Laboratory (April 2011)
9. http://code.google.com/p/massivethreads/
10. Barnes, J., Hut, P.: A hierarchical O(N log N) force-calculation algorithm. Nature **324**(6096) (December 1986) 446–449
11. Greengard, L.F.: The rapid evaluation of potential fields in particle systems. PhD thesis, New Haven, CT, USA (1987) AAI8727216.
12. Ying, L., Biros, G., Zorin, D., Langston, H.: A new parallel kernel-independent fast multipole method. In: Supercomputing, 2003 ACM/IEEE Conference. (nov. 2003) 14
13. Ying, L., Biros, G., Zorin, D.: A kernel-independent adaptive fast multipole algorithm in two and three dimensions (2003)
14. Greengard, L.: The Rapid Evaluation of Potential Fields in Particle Systems. Volume 52. MIT Press (1988)
15. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture TCCA Newsletter (1995) 19–25
16. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set. In Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A., eds.: Tools for High Performance Computing. Springer Berlin Heidelberg (2008) 139–155
17. Brunst, H., Knüpfer, A.: Vampir. In: Encyclopedia of Parallel Computing. Springer (October 2011)
18. http://icl.cs.utk.edu/PAPI/
19. http://software.intel.com/en-us/intel-vtune-amplifier-xe
20. Drongowski, P.J.: Basic performance measurements for amd athlontm 64, amd opterontm and amd phenomtm processors. (25 September 2008)
21. : Intel xeon processor e5-2600 product family uncore performance monitoring guide. (March 2012)
22. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52** (April 2009) 65–76
23. Ltaief, H., Yokota, R.: Data-driven execution of fast multipole methods. (2012)
24. Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., Takahashi, T.: Pipelining the fast multipole method over a runtime system. (2012)
25. Pericas, M., Amer, A., Fukuda, K., Maruyama, N., Yokota, R., Matsuoka, S.: Towards a dataflow fmm using the ompss programming model. 136th IPSJ Conference on High Performance Computing
26. Taura, K., Yokota, R., Maruyama, N.: A task parallelism meets fast multipole methods. In: Proceedings of SCALA'12. (November 2012)
27. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors (2009)
28. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proceedings of the 2007 ACM/IEEE conference on Supercomputing. SC '07, New York, NY, USA, ACM (2007) 38:1–38:12