Towards a Dataflow FMM using the OmpSs Programming Model

Miquel Pericàs[1,a) Abdelhalim Amer[4,b)] Keisuke Fukuda[4,c)] Naoya Maruyama[2,d)]
Rio Yokota[3,e)] Satoshi Matsuoka[1,f)]

Abstract: This paper describes initial efforts towards the development of a dataflow implementation of the *ExaFMM* Fast Multipole Method code using the OmpSs programming model. We first develop several implementations based on task decomposition which overcome load balancing problems previously identified using traditional parallelization approaches. We then add dataflow extensions to improve task throughput by extracting distant parallelism and removing barriers. Execution profiles and scalability results for a single node of the Tsubame 2.0 supercomputer are then shown.

Keywords: Fast Multipole Method, Task-Dataflow Programming Models, Tsubame 2.0

1. Introduction

[*1] The Fast Multipole Method (FMM) is an algorithm to compute fast and efficient solutions to the N-Body problem. It has multiple applications such as molecular simulation [1], turbulence simulation [2] or computational electromagnetics [3]. The computational complexity increases as $O(N)$. This is a notable advantage over the direct method which, while offering inmense parallelism, is burdened by its $O(N^2)$ complexity. Large scale simulations need more efficient solutions, such as treecodes or the FMM.

As with treecodes [4], the Fast Multipole Method approximates distant clusters of source bodies while processing close bodies using the direct method. However, contrary to treecodes, the FMM also approximates target bodies, resulting in the hierarchical interaction of source and target clusters. The resulting algorithm has many more phases and kernels than the direct particle-to-particle (P2P) approach. In addition, many of the kernels operate bottom-up or top-down on an octree structure, which results in a dynamically varying level of parallelism over the execution time. The efficient implementation of the FMM algorithm and its parallelization on multiple cores becomes a much more complex problem.

Traditionally N-Body solvers have been parallelized by statically partitioning the input distribution among processors and allowing them to operate on independent data as much as possible [5], [6]. Changes in the input distribution can potentially lead to load balancing problems, thus a considerable fine tuning effort is necessary to make such implementations highly scalable [7]. An alternative strategy is to avoid addressing load balancing at the source code level and rely on the runtime to perform this task. Many modern programming models such as StarPU [8] or StarSs/OmpSs [9] try to do this by scheduling program sections as tasks and using dataflow techniques to avoid barrier synchronization.

In this paper we show initial efforts into applying one of these systems, namely the OmpSs programming model, to parallelize the ExaFMM code [10]. We explore several methods to parallelize the main computation phase of the FMM kernel over a single node of the Tsubame 2.0 supercomputer and analyze the complexity and the result of adding task dataflow execution into the code.

This paper is organized as follows. We begin by introducing the necessary background information on task-dataflow programming models and the fast multipole method together with the ExaFMM implementation. Next we explain several strategies to parallelize the main kernel of the ExaFMM code, the dual tree traversal (DTT). Finally, we report initial results obtained on our test platform, a single thin node of the Tsubame 2.0 supercomputer.

2. Task-based Dataflow Programming Models

Recently we are witnessing the growing popularity of task based programming models such as Intel's Threading Building Blocks [11], the OpenMP 3.0 tasking constructs [12] or CUDA [13]. This suggests that tasking is an intuitive abstraction

[1] Global Scientific Information and Computing Center, Tokyo Institute of Technology, Tokyo 152–8550, Japan
[2] Advanced Institute for Computational Science, Riken, Kobe, Japan
[3] King Abdullah University of Science and Technology, Saudi Arabia
[4] Department of Mathematical and Compute Sciences, Tokyo Institute of Technology, Tokyo 152–8550, Japan
[a)] pericas.m.aa@m.titech.ac.jp
[b)] amer@matsulab.is.titech.ac.jp
[c)] fukuda@matsulab.is.titech.ac.jp
[d)] nmaruyama@riken.jp
[e)] rio.yokoya@kaust.edu.sa
[f)] matsu@m.titech.ac.jp
[*1] IMPORTANT NOTICE: This work is a non-refereed publication

to address parallel programming in modern systems. Tasking also allows to improve system utilization by addressing load balancing via work stealing techniques. This enables higher levels of parallelization, but ultimately these schemes are also limited by system wide synchronization events such as barriers.

The main idea of task-based dataflow programming models is to relax the strict ordering of these schemes and provide a more dynamic task scheduling environment. These schemes add dependency information to individual tasks, allowing the runtime to schedule the tasks based only on its input and output dependencies. Modern programming languages following this design approach include, among others, OmpSs [14], StarPU [8] and Sequoia [15]. In this paper we focus on OmpSs, an implementation of the concepts of tasking and dataflow on top of the OpenMP 3.x programming model. By adding extensions on top of OpenMP constructs, OmpSs attempts to provide a familiar programming environment, and one in which developers can verify the correctness of their program sequentially, while having the runtime take care of dynamic parallelization.

In OmpSs the programmer describes an application as a collection of tasks together with their dependencies. OmpSs tasks have similarities to Cilk tasks [16]. On startup, only the main thread (called *Master Thread*) executes. The master thread then starts generating tasks, which are executed asynchronously by a pool of workers which persists during the execution. Tasks are always single-threaded and non-preemptible. They execute on a single device and operate on local/global data as well as its input and output dependencies. The OmpSs workers hide the complexities and dynamic heterogeneities in the core and memory architectures, as well as adapt to dynamic changes in resource availability and workloads.

2.1 Dataflow Extensions

OmpSs adds a set of clauses (`ss_clauses`) to the OpenMP 3.0 task construct indicating the usage of data within the task. It includes three directionality clauses: `input, output` and `inout`.

```
#pragma omp task [omp_clauses] [ss_clauses]
  task_block
```

The task declarations can either be inlined or associated with a function declaration. In the latter case, every invocation of the function is executed as an asynchronous task.

```
#pragma omp task [omp_clauses] [ss_clauses]
  function definition | function prototype
```

The three `ss_clauses` accept a list of expressions that must evaluate to a set of lvalues and that are used by the runtime system to build the task dependency graph.

- *input*: Task not elegible to run until input dependency has been generated
- *output*: Subsequent tasks cannot run until output has been generated
- *inout*: Task reads and write dependency

Dependencies are described as memory ranges. The array section syntax from Fortran 90 is used to describe these dependencies. For example, in

```
#pragma omp task input([N] a [0:N-1], \
   [N] b [0:N-1]) output ([N] c [0:N-1])
void vec_add(double *a, double *b, double *c)
```

the pragmas instruct the compiler that `vec_add()` reads two vectors of N elements and writes into array `c` an output vector of the same length. This information is enough to build the dependency graph. Dependency based execution has many benefits: First, it enables asynchronous execution based on inputs/outputs, therefore hiding communication latencies and enabling the extraction of distant parallism. Second, global barriers are eliminated since synchronization is implicit.

Since OmpSs extends the OpenMP task construct with dependency clauses it is still possible to program a pure OpenMP 3.0 "dependency-less" code and use the OmpSs runtime to execute the program. In this programming style, synchronization must be explicitly encoded using the `#pragma omp taskwait` construct. This is necessary because, since OmpSs deprecates the `#pragma omp parallel` directive, there are no implicit synchronizations at the end of a parallel section. As will be seen later, only one of our multiple ExaFMM parallelizations uses the dependency extensions of OmpSs. Thus we will also compare with other OpenMP runtimes when executing the pure OpenMP 3.0 implementations of ExaFMM.

3. The Fast Multipole Method

In this section we provide an overview of the Fast Multipole Method and give details on one of its implementations, namely the ExaFMM code under development by Rio Yokota.

3.1 Overview

The FMM is a hierarchical N-Body Solver, which calculates the interactions of N bodies with a computational complexity of $O(N)$. The kernel offers high arithmetic intensity and has been shown to scale to large GPU based clusters [7].

The FMM consists of six kernels, with most of them having some kind of dependency on others. The domain is hierarchically partitioned into trees using an octree structure. The *root* cell of the octree represents the full domain, while the *leaf* cells are the smallest cells. Leaf cells may contain only up to a fixed number of bodies. This number, which is often called q in FMM literature, determines the depth of the tree structure. Another important parameter in FMM is the order of multipole expansions. The higher this order, the more accurate the approximations will be, but also the more computation needs to be conducted. In this work we keep the order of expansions p fixed at 8. For non-uniform distributions of bodies, the octree becomes highly adaptive and the performance very sensitive to load balancing.

In the following we define the functionality of the six main kernels that define the FMM functionality:

(1) **Particle-to-Multipole (P2M)**: This first stage translates the mass/charge of the particles into multipole expansions at the center of the leaf cells.

(2) **Multipole-to-Multipole (M2M)**: In this stage, the multipole expansions of smaller cells are translated into the multipole expansions of larger cells.

(3) **Multipole-to-Local (M2L)**: Once the multipole expansions

have been computed they can be translated to local expansions. In order to do so, cells need to be well separated. A Multipole Acceptance Criteria (MAC), based on size and distance is used to determine whether the cells fulfill the conditions to be approximated in this manner [17].

(4) **Particle-to-Particle (P2P)**: The neighboring cells at the leaf level are adjacent and thus never approximated using a M2L kernel. Instead they are handled by a particle-to-particle approach, which computes the interaction of all bodies within the cells using the direct method.

(5) **Local-to-Local (L2L)**: After the local expansions have been computed, they are translated to the center of smaller cells using the L2L kernel.

(6) **Local-to-Particle (L2P)**: Finally, the local expansions at the center of the leaf cells are used to compute the final force on the particles by means of a Local-to-Particle (L2P) kernel.

Except for the P2P kernel, all the stages depend on the data generated by the previous stages and thus need to wait for its completion before progressing. The P2P kernel, on the other hand, is independent on all other phases and can be executed concurrently to all of them. The only concern is the update to the particles. Since the L2P stage also updates the particles in the target cell, such an update needs to be guarded with some kind of lock. Out of all the kernels, M2L and P2P dominate the execution time. While P2M, M2M, L2L and L2P are called at most once per cell, the M2L kernel is called many times per cell. Because P2P is often executed in parallel to all other stages, to achieve good load balancing it is sized so that it takes similar execution time to M2L. This balance between M2L and P2P is achieved by finding the optimal value for the number of particles per cell (q). The higher this number, the more dominant P2P becomes. On the other hand, when q is small, the M2L kernel dominates the execution time.

3.2 Motivation Example

To motivate our work we start by analyzing a classic FMM parallelization case. A typical parallelization scheme for the n-body problems consists in partitioning the domain equally between workers at each timestep and have the workers operate on their own data subset. We call this kind of partitioning *octsection*. Figure 1 shows the execution profile generated with extrae and paraver for 8 processors performing an FMM simulation of 80000 bodies on a input data set following a plummer model. The value of q is 4000. A plummer model is a common data distribution which tries to model a globular star cluster. In this distribution the bodies are highly clusterd towards the center and only a few bodies are in the far regions. In the simulation of Figure 1, each processor is assigned one eighth of the domain before starting the computation. The profile clearly shows the upward phase (L2M, M2M), the generation of the Local Essential Trees (Group Communication in orange color) and the downward phase (M2L, P2P, L2L, L2P). Since each worker solves almost the same problem, performance is good and features good load balancing. Figure 2 shows the same expriment but using a plummer distribution that is not centered exactly in the middle of the domain, but instead is moved slightly (about 1.22%) towards one of the edges of the simulation cube. As can be seen, this small change has a massive effect on performance. The code now runs more than 5 times slower. This occurs because the static partitioning scheme generates a large load imbalance for this input distribution, with some cores having almost no data to compute on and a single core handling the bulk of the computation.

3.3 Load Balancing and Task Scheduling

When the particle distribution is irregular, or is not well positioned as in the previous example, balancing the load among cores is not a simple task. Just assigning each engine to work on a equally sized partition of the domain will result in large load imbalance. This will be particularly bad when many processors collaborate to solve the task, as some processes might not have enough data to operate one while others will be overloaded. This needs to be avoided. More advanced domain partitioning schemes such as orthogonal recursive bisection (ORB) [5] will generate a much more balanced work distribution by ensuring that every processing engine handles the same number of particles. However, such a scheme has still drawbacks. First, it adds complexity to the partitioning phase, and second, the work distribution is still static. Any remaining load imbalance will still reduce performance. Given that some FMMs have timesteps executing over 100 seconds each, a runtime approach can be a better solution to the problem.

In task-based execution schemes, one of the main requirements for performance is to keep the task pipeline as busy as possible. Barriers and synchronization points require the pipeline to be halted. An additional goal is therefore to remove as much synchronization as possible. One way to relax synchronization and increase task throughput is to allow tasks to execute across synchronization points. This can be done as long as all the inputs of the future task have been computed. In such a scenario one can remove the global barrier between two different parallel sections and allows the runtime to interleave the execution of both, thus increasing the task throughput and performance.

In this paper we explore the possibility of using such strategies in the ExaFMM code to achieve both a good load balancing and high utililtzation of the task pipeline by using a unified programming methodology.

4. Testing environment

The development of the OmpSs ExaFMM code and its execution and analysis involve the utilization of many different tools. The software environment consists, in addition to our modified ExaFMM sources, of the gcc compiler (version 4.5.4), the OmpSs runtime and source-to-source compiler (nanox-0.7a and mcxx-1.3.5.8, respectively) and the extrae-2.2.1 tracing library, whose output is then visualized by the Paraver tool (version 4.3.4). On the hardware side, we ran all our experiments on a thin node of the Tsubame 2.0 supercomputer. Each node has up to 12 physical cores (two Intel Xeon X5670 CPUs operating at 2.93GHz) and 54 GB of main memory.

5. Parallelization Strategies and Performance Results

This section describes several schemes that we developed to

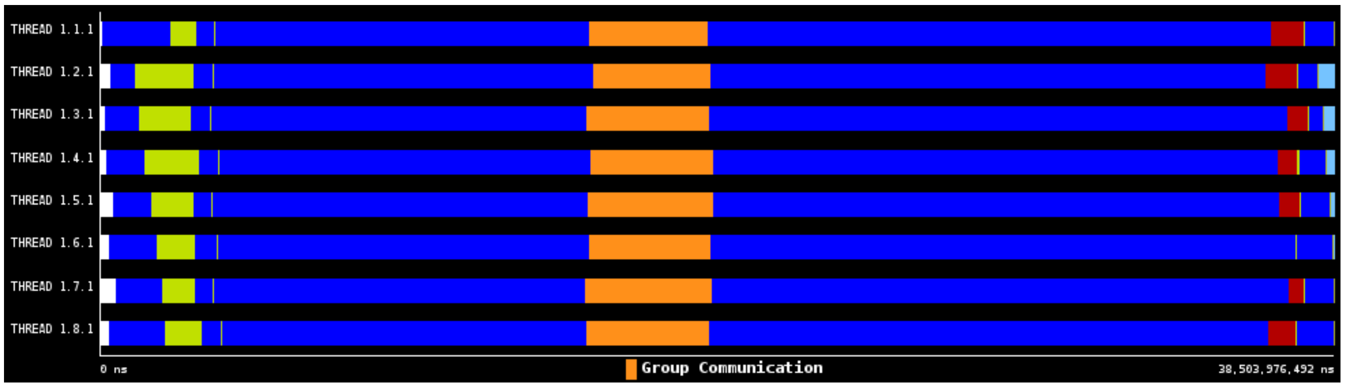

Fig. 1 Execution profile for a Plummer Distribution with 8 processors. Static Partitioning is conducted using binary octsection (Color Code: Blue="Running", Orange="MPI Group Communication", Red="Synchronization")

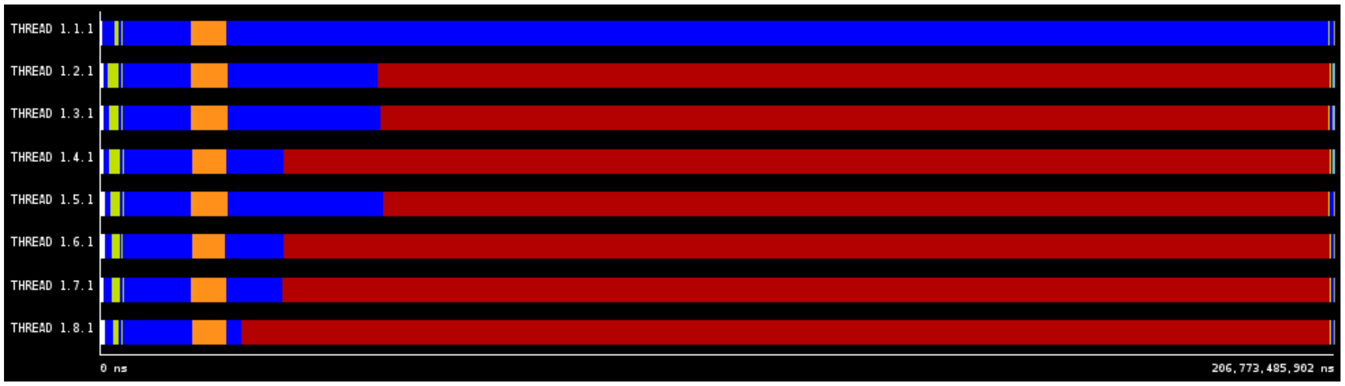

Fig. 2 Execution profile for 8 cores simulating the slightly deviated Plummer Distribution. Static Partitioning is performed using binary octsection

parallelize the ExaFMM code.

5.1 Dual Tree Traversal

To parallelize the ExaFMM code we consider the Dual Tree Traversal, ExaFMM's main kernel. The dual tree traversal (DTT) simultaneously traverses the source and target octrees top-down and constructs the interaction lists for the M2L and P2P kernels. The interaction lists contain the lists of source cells to be used by the M2L and P2P kernels. Given that the octree has many paths to analyze, the DTT keeps an explicit stack to stor pairs of source and target cells to be analyzed later. The original ExaFMM code can handle the Dual Tree Traversal in two different ways:

- *Default*: In the default configuration, interactions are computed as soon as they are recognized. This means that the local expansions and particle interactions are computed incrementally.
- *Queue*: In this alternative mode, all interactions are first stored in the interaction lists. After the DTT phase completes, the code traverses the target cells list and computes all the interactions for the cell at once.

Both strategies are interesting from the point of view of parallelization. We will analyze their strengths and weaknesses in the following sections. More details on the operation of the Dual Tree Traversal can be found in a recent publication by the ExaFMM authors [2].

5.2 Parallel Task Execution Strategies

We focus on the dual tree traversal and taskify the execution of the two kernels: M2L and P2P. Given that M2L and P2P are executed hundreds of thousands of times during a timestep for a input set of about 50000 bodies, focusing on these kernels should provide enough granularity to balance the overall execution workload.

We implement the following strategies to taskify the execution.

- **Fine-Grained**: This version works like the default ExaFMM execution except that all interactions are spawned asynchronously as tasks. To avoid concurrency issues when multiple tasks access the same target cell OpenMP locks are used, with one lock per cell. This version generates many fine grained tasks, which might put a lot of pressure on the runtime system. On the other hand, it has the potential to finish earlier as it does not delay the execution of the kernels.
- **Queue-PAR**. This version is like *Queue* but it generates one asynchronous task for each cell during the kernel evaluation phase. Because all the interactions for a single cell are computed by the same task there is no need to set OpenMP locks. In addition, since it computes multiple interactions per cell and spawns just one task per each target cell, the granularity is much larger and it is much less likely to be impacted by runtime overheads.
- **Queue-EE-PAR**: This version is like *Queue-PAR* in that it spawns one asynchronous task to compute all the P2P and M2L interactions of a cell. However, it does not wait until

completion of the Dual Tree Traversal and instead launches the task as soon as it decides that no more interactions can be added to this cell. After each interaction evaluation it analyzes the explicit stack to check if a cell can no longer receive interactions in the future. It then conditionally spawns a task to compute the interactions. The 'EE' portion in the name is a shortcut for *Early Execution*.

The trace in Figure 3 shows the effect of applying the *Queue-EE-PAR* implementation to the example from Figure 2, keeping all other parameters identical. As can be seen, the usage of work-stealing techniques has a very positive impact on the load balance. The same results can also be achieved using the *Queue-PAR* and *Fine-Grained* schemes. Later we will analyze the performance of all three schemes in detail.

5.3 Dataflow Scheduling

The performance achieved by tasking can be further improved by applying dataflow scheduling. When applying task-dataflow techniques the programmer explicitly declares the dependency relationship between tasks, which are then executed in a dataflow manner by the runtime. This has two main benefits. First, the number of synchronization points is reduced. Global barriers are very detrimental to performance, particularly in large parallel runs because they force all processes to block at the same time, temporarily reducing the concurrency to just one. In addition, dataflow scheduling enables the explotation of distant parallelism. It allows tasks that belong to future phases of the program to start execution ahead of time, as long as all their input dependencies are ready.

Applying a dataflow execution scheme to the complete Fast Multipole Method is challenging. In the case of the ExaFMM code, specifying the dependencies between the M2M phase and the Dual Tree Traversal is non-trivial. Thus in our first approach described here, only the M2L→L2L→L2P interactions have been implemented in a dataflow scheme. The interactions are described at the level of cells. This means they are very fine-grained, but at the same time this programming methodology maximizes the extraction of parallelism. The dataflow scheme has been implemented as an extention to the *Queue-EE-PAR* scheme to allow the L2L tasks start as soon as possible. Of course, in order to trigger the L2L tasks it is also a condition that the local expansion (L2L kernel) of the parent cell has completed.

Figure 4 shows the last part of the downward phase execution profile of the dataflow execution. In this trace the M2L/P2P kernels are shown in blue color, while the L2Ls are shown in light green and L2P is shown in pink. Idle sections are shown in whilte. The L2L and L2P portions of the trace are the most interesting. The Figure allows to observe how these kernels execute in dataflow order and how distant tasks are overlapped. Although in this part of the profile many L2L and L2P kernels can be observed, the main part of the downware phase (not shown) is dominated by the blue color of the M2L and P2P kernels. This explains why the speed-up achieved by dataflow scheduling versus non-dataflow scheduling is rather small (around 4%). Another interesting observation is the middle section of the profile. During this time there are no M2L or P2P executions taking care,

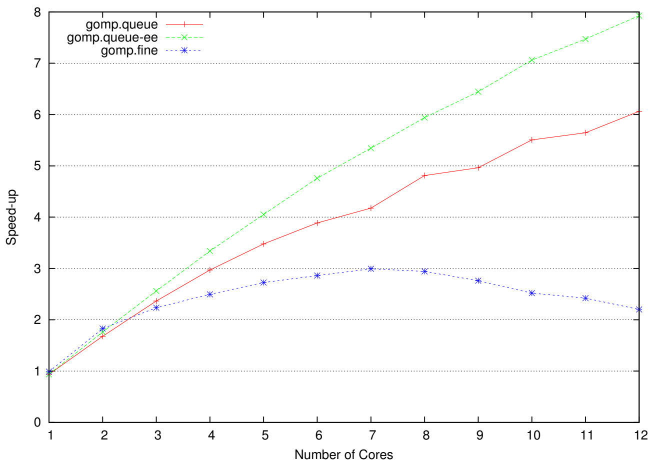

Fig. 5 Scalability results for the GOMP versions

there are some L2L and L2P kernels executing, but otherwise the workers are either idle or are busy performing other tasks. What exactly is ocurring needs yet to be studied. It remains to be seen if these wait times can be overlapped with later execution of P2P/M2L kernels in order to further improve the performance.

5.4 Scalability Results

The final set of experiments that we conducted focuses on the scalability to larger numbers of cores. Given that of our four parallel schemes only one uses the OmpSs extensions, we take the opportunity to check not only the scalability of the OmpSs runtime, but also the scalability of another OpenMP library such as GOMP. We choose to use the GOMP OpenMP implementation here as our OmpSs-based FMM already uses the GNU g++ compiler for the native compilation of the code.

Figure 5 shows the speed-ups achieved by the three implementations described in Section 5.2 executing the downward phase of a single FMM timestep on a Plummer distribution centered in the middle of the domain. The runtime used for this experiment is GOMP. All speed-ups are reported against the serial execution of the benchmark, which takes 4.57 seconds to execute the Downward phase. The behavior varies considerably from implementation to implementation. *Queue-EE-PAR* offers the best performance, achieving an 8× speed-up when using all of the 12 cores on the node. *Queue-PAR* has a similar behavior for a low number of cores, but performance quickly degrades, reaching only about a 6× speed-up for 12 cores. The *Fine-Grained* implementation has the worst performance, scaling only to 3-4 cores. After this point performance almost does not improve any more, and after 7 cores it actually gets worse. This means that for larger number of cores the execution is dominated by the overhead of creating so many fine-grained tasks. An important property of task-based runtime systems is that the maximum acceptable time for task creation decreases proportionally to the inverse of the number of cores. This is because a larger number of cores can absorb more tasks. As a result the master thread needs to be able to deliver tasks at a higher rate. For the case of `libgomp`, after 3 cores, task creation starts being a major part of the execution time, and after 7 cores the runtime becomes the execution bottleneck.

Figure 6 shows the results for the OmpSs runtime. These re-

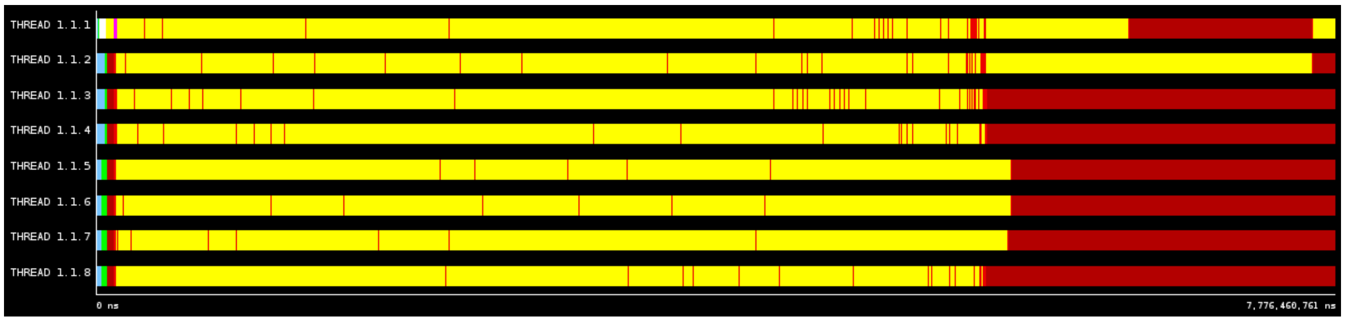

Fig. 3 Profile for the non-centered Plummer distribution using *Queue-EE-PAR*

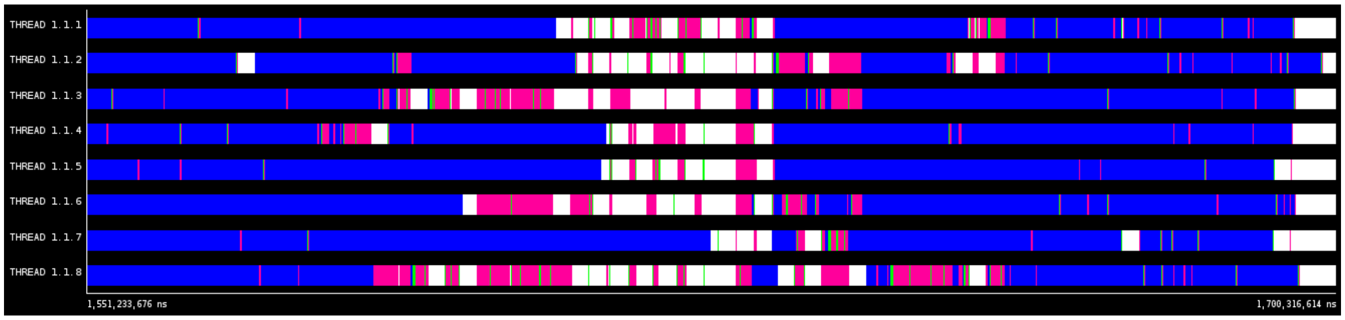

Fig. 4 Profile showing a section of the dataflow execution in the FMM downward phase (Color code: Blue = M2L/P2P, Green = L2L, Pink = L2P, White = other)

sults also include the dataflow implementation. The speed-ups for *Queue-EE-PAR* and *Queue-EE* are somewhat smaller than the *libgomp* results. This might indicate that the GOMP task creation overhead is smaller than the OmpSs task creation overhead, but overall the behavior is very similar. The performance of *Fine-Grained* is very bad in this case, not scaling to even just 2 processors. In fact, at its best performance, with about 9 cores, it reaches only a speed-up of 1.8×. The reason is probably the larger task creation overhead. *Fine-Grained* spawns more than 700000 tasks in a little less than 2.6 seconds, thus the runtime needs to be able to spawn more than 300000 tasks per second if speed-ups are desired. *Fine-Grained* also makes extensive usage of locks to avoid race conditions which can also be a source of performane degradation.

We now analyze the dataflow implementation. As it is implemented as an optimization to *Queue-EE-PAR*, which was already the best performer, it is no surprise that it beats all other configurations. *Dataflow* reaches about an 8.5× speed-up with 12 cores, the highest of all experiments (including GOMP). The speed-up over *Queue-EE-PAR* is not very large, but this is expected as the pipelining of M2Ls with L2Ls and L2Ps just enables the overlapping of a small amount of computations after the Dual Tree Traversal. However, we also observe that the gap seems to increase with the number of processors, reaching 8% with 12 cores. This makes sense, as the impact of barriers on performance grows the more processors are being used. In this case, the barrier between the DTT and the L2L kernels present in the *Queue*-derived implementations is causing a scalability problem. On the other hand, the dataflow implementation is not affected by this problem as it removes the barrier.

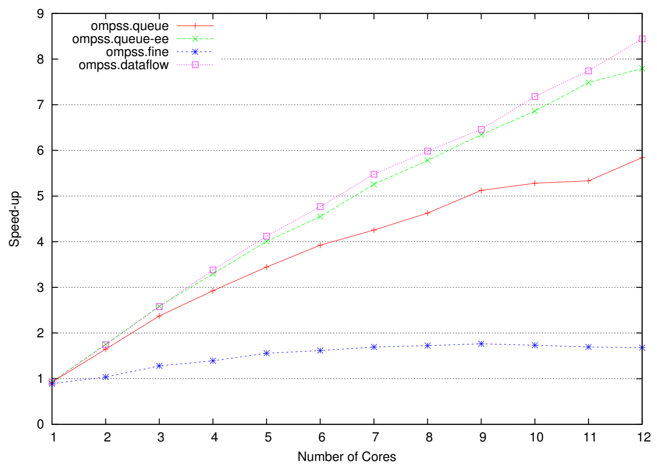

Fig. 6 Scalability results for the OmpSs versions

6. Related Work

Task-based runtimes are currently a hot topic of research, but their history goes far into the past. Perhaps the most important work in this topic was the development of the Cilk programming language at MIT during the 90s [16]. One particular concept that can be found in many implementation is that of FIFO work-stealing and LIFO execution by each worker [18]. Many later languages such as Nanos [19], Qthreads [20] or MassiveThreads [21] have adapted these principles.

Recently, several groups have proposed to combine tasking with dataflow programming principles. Dataflow promises to remove barriers and exploit distant parallelism in programs, increasing task throughput and, therefore, performance. Systems such as StarSs [9], StarPU [8] or QUARK [22] are all based on this principle. OmpSs [14], the runtime used for the currrent work, is an extension to OpenMP based on the ideas introduced in StarSs.

Fast Multipole Methods are a hot topic due to their multiple applications and good characteristics for petascale computing and beyond. ExaFMM has been recently parallelized using MassiveThreads [23]. In the field of task-dataflow programming models, we are aware of at least two other works that have tried to run the FMM on a task-dataflow runtime. The first [24] is an attempt to run the Black Box FMM on the StarPU system, while the second is an attempt to run ExaFMM on top of QUARK [25]. The main difference with the present work is that the latter ([25]) focuses only on the M2L and P2P kernels and does not pipeline the L2L and L2P kernels.

7. Conclusions

We present our initial work towards implementing a Fast Multipole Methods using the OmpSs task-dataflow programming model. For simplicity we first taskify the application and then apply dataflow constructs.This initial implementation does not yet cover the full application, but the main execution phase (called *downward*) is covered. Experiments show that the dataflow version outperforms the task-only versions thanks to the extraction of distant parallelism between the M2L, L2L and L2P kernels. The speed-up over the task-only versions is not large, but we also observe that it increases with the number of cores, an observation which we plan to study in more detail. We also compared the OmpSs runtime with the GOMP runtime. GOMP seems to have less overhead in task creation, but this turns out not to be an important factor as the best performing schemes do not put much pressure on the runtime system.

References

[1] Board, J. A., Causey, J. W., Leathrum, J. F., Windemuth, A., and Schulten, K.: *Accelerated molecular dynamics simulation with the parallel fast multipole algorithm*, Chemical Physics Letters, Volume 198, Issues 12, 2 October 1992, Pages 89-94.

[2] Yokota, R., Narumi, T., Barba, L. A., and Yasuoka, K.: *Petascale turbulence simulation using a highly parallel fast multipole method on GPUs*, available from ⟨http://arxiv.org/abs/1106.5273⟩ (accessed 2012-9-3).

[3] Engheta, N., Murphy, W. D., Rokhlin, V., and Vassiliou, M. S.: *The Fast Multipole Method (FMM) for Electromagnetic Scattering Problems*, IEEE Transactions on Antennas and Propagation, Volume 40, Issue 6, June 1992, pages 634-641.

[4] Barnes, J., and Hut, P.: *A hierarchical O(N log N) force-calculation algorithm*, Nature 324 (4): 446-449, December 1986

[5] Warren, M. S., and Salmon, J. K.: *Astrophysical N-body simulation using hierarchical tree data structures*, ACM/IEEE Conference on Supercomputing, pages 570-576, 1992.

[6] Warren, M. S., and Salmon, J. K.: *A parallel hashed oct-tree N-body algorithm*, ACM/IEEE conference on Supercomputing, pages 1221, 1993.

[7] Rahimian, A., Lashuk, I., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Veerapaneni, S., Vetter, J., Vuduc, R., Zorin, D., and Biros. G.: *Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures*, ACM/IEEE Conference on Supercomputing (SC10), 2010.

[8] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.: *StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures*, Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, February 2011

[9] Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J.: *CellSs: a programming model for the Cell BE architecture*, ACM/IEEE conference on Supercomputing (SC06), 2006.

[10] *ExaFMM | Boston University*, available from ⟨http://www.bu.edu/exafmm/⟩ (accessed 2012-9-3)

[11] *Intel Threading Building Blocks*, available from ⟨http://software.intel.com/en-us/intel-tbb⟩ (accessed 2012-9-3)

[12] *OpenMP.org*, available from ⟨http://openmp.org/wp/⟩ (accessed 2012-9-3)

[13] *Parallel Programming and Computing Platform | CUDA | NVIDIA*, available from ⟨http://www.nvidia.com/object/cuda_home_new.html⟩ (accessed 2012-9-3)

[14] Bueno-Hedo, J, Planas, J, Duran, A, Badia, R. M., Martorell, X, Ayguad, E, and Labarta, J.: *Productive Programming of GPU Clusters with OmpSs*, IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2012.

[15] Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P.: *Sequoia: programming the memory hierarchy*, ACM/IEEE conference on Supercomputing (SC06), 2006

[16] Frigo, M., and Leiserson, C. E., and Randall, K. H.: *The implementation of the Cilk-5 multithreaded language*, ACM SIGPLAN Conference on Programming language design and implementation (PLDI'98), 1998

[17] Dehnen, W.: *A hierarchical O(N) force calculation algorithm*, Journal of Computational Physics, 179(1):27 42, 2002.

[18] Mohr, E., Kranz, D. A., and Halstead, R. H.: *Lazy task creation: A technique for increasing the granularity of parallel programs*, IEEE Transactions on Parallel and Distributed Systems, 2(3):264, July 1991.

[19] *NANOS++*, available from ⟨https://pm.bsc.es/projects/nanox⟩ (accessed 2012-9-3).

[20] *Qthreads*, available from ⟨http://www.cs.sandia.gov/qthreads/⟩ (accessed 2012-9-3).

[21] *massivethreads - A Lightweight Thread Library for High Productivity Languages*, available from ⟨http://code.google.com/p/massivethreads/⟩ (accessed 2012-9-3).

[22] *QUARK*, available from ⟨http://icl.cs.utk.edu/quark/⟩ (accessed 2012-9-3).

[23] Taura, K., Nakashima, J., Yokota, R., Maruyama, N.: *Parallelizing ExaFMM with MassiveThreads Task Parallel Library and Its Evaluation*, HPC-135, 2012.

[24] Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., and Toru, T., *Pipelining the Fast Multipole Method over a Runtime System*, available from ⟨http://arxiv.org/abs/1206.0115⟩ (accessed 2012-9-3).

[25] Ltaief, H., and Yokota, R.: *Data-Driven Execution of Fast Multipole Methods*, available from ⟨http://arxiv.org/abs/1203.0889⟩ (accessed 2012-9-3).