Task-based fast multipole method for clusters of multicore processors

Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Martin Khannouz, Luka Stanisic

Task-based fast multipole method for clusters of multicore processors

Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Martin Khannouz, Luka Stanisic

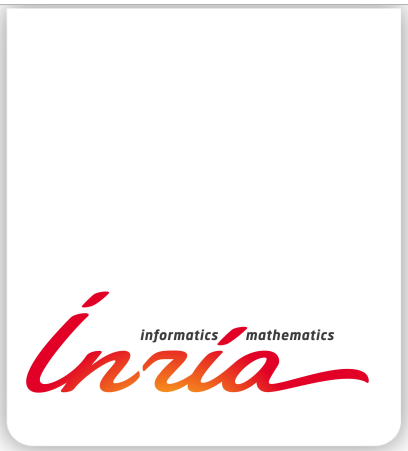
RESEARCH REPORT

N° 8970

October 2016

Project-Teams HiePACS and STORM

ISSN 0249-6399 ISRN INRIA/RR--8970--FR+ENG



Task-based fast multipole method for clusters of multicore processors

Emmanuel Agullo*, Bérenger Bramas†, Olivier Coulaud*,
Martin Khannouz*, Luka Stanisic*

Project-Teams HiePACS and STORM

Research Report n° 8970 — October 2016 — 16 pages

Abstract: Most high-performance, scientific libraries have adopted hybrid parallelization schemes - such as the popular MPI+OpenMP hybridization - to benefit from the capacities of modern distributed-memory machines. While these approaches have shown to achieve high performance, they require a lot of effort to design and maintain sophisticated synchronization/communication strategies. On the other hand, task-based programming paradigms aim at delegating this burden to a runtime system for maximizing productivity. In this article, we assess the potential of task-based fast multipole methods (FMM) on clusters of multicore processors. We propose both a hybrid MPI+task FMM parallelization and a pure task-based parallelization where the MPI communications are implicitly handled by the runtime system. The latter approach yields a very compact code following a sequential task-based programming model. We show that task-based approaches can compete with a hybrid MPI+OpenMP highly optimized code and that furthermore the compact task-based scheme fully matches the performance of the sophisticated, hybrid MPI+task version, ensuring performance while maximizing productivity. We illustrate our discussion with the ScalFMM FMM library and the StarPU runtime system.

Key-words: high performance computing (HPC), fast multipole method, FMM, cluster, multicore processor, runtime system, hybrid parallelization, task-based programming, MPI, OpenMP

* Inria, France, Email: surname.name@inria.fr
† Max Planck Computing and Data Facility (RZG), Garching, Germany, Email: Berenger.Bramas@mpcdf.mpg.de

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Méthode des multipôles rapide à base de tâches pour des clusters de processeurs multicoeurs

Résumé : La plupart des bibliothèques scientifiques très performantes ont adopté des parallélisations hybrides - comme l'approche MPI+OpenMP - pour profiter des capacités des machines modernes à mémoire distribuée. Ces approches permettent d'obtenir de très hautes performances, mais elles nécessitent beaucoup d'efforts pour concevoir et pour maintenir des stratégies de synchronisation/communication sophistiquées. D'un autre côté, les paradigmes de programmation à base de tâches visent à déléguer ce fardeau à un moteur d'exécution pour maximiser la productivité. Dans cet article, nous évaluons le potentiel de la méthode des multipôles rapide (FMM) à base de tâches sur les clusters de processeurs multicœurs. Nous proposons deux types de parallélisation, une première approche hybride (MPI+Tâche) à base de tâches et d'appels à MPI pour gérer explicitement les communications et la deuxième uniquement à base de tâches où les communications MPI sont implicitement postées par le moteur d'exécution. Cette dernière approche conduit à un code très compact qui suit le modèle de programmation séquentiel à base de tâches. Nous montrons que cette approche rivalise avec le code hybride MPI+OpenMP fortement optimisé et qu'en outre le code compact atteint les performances de la version hybride MPI+Tâche, assurant une très haute performance tout en maximisant la productivité. Nous illustrons notre propos avec la bibliothèque FMM ScalFMM et le moteur d'exécution StarPU.

Mots-clés : Calcul haute performance, méthode multipôles rapide, FMM, cluster, architecture multicœur, moteur d'exécution, parallélisation hybride, programmation à base de tâches, MPI, OpenMP

Contents

List of Algorithms

1 Introduction

The hardware complexity of computers used in scientific computing has continuously increased since the first parallel machines. In particular, distributed-memory machines now almost systematically rely on multicore processors. As a consequence, most parallel, scientific libraries have been revisited to ensure high-performance on such architectures. The most broadly used approach has consisted in developing two-level hybrid parallelization schemes that cope with the hardware hierarchy. The Message Passing Interface (MPI) paradigm is most often employed to allow for inter-node parallelism while a lower footprint threading system such as OpenMP ensures the parallelization within a node. Although such hybrid MPI+OpenMP approach has proved to be very efficient for a wide range of scientific applications, it tends to lack versatility and it is hard to maintain. For these reasons, the high performance computing (HPC) community strongly investigates whether higher-level programming paradigms could achieve a competitive performance.

The fast multipole method (FMM) [**?**] has been proposed by Leslie Greengard and Vladimir Rokhlin in 1987 to reduce the complexity of N-body simulations. While the accurate calculations of the motions of N particles interacting via gravitational or electrostatic forces was considered to require $\theta(N^2)$ operations (one for each pair of particles), the FMM does so in $\theta(N)$ operations by using multipole expansions (mass or net charge, dipole moment, quadrupole, and so forth) to approximate the effects of a distant group of particles on a local group. Considered among the

top ten algorithms of the twentieth century [?], the FMM has been strongly used in the twenty-first century by the HPC community, winning ACM Gordon Bell Prize awards for applications in turbulence [?] and blood flow [?] simulations beyond petascale for the latter. These astonishing breakthroughs have been achieved with hybrid parallelization schemes relying on MPI, OpenMP and possibly CUDA.

On the other hand, studies have been conducted to assess whether the FMM could be designed alternative programming paradigms. [?] and [?] for instance assessed FMM implemented on top of PGAS and Charm++, respectively. Among alternative approaches, the design of task-based FMM on top of runtime systems has deserved a special attention, yet at a more modest scale, either on a (single) multicore [?, ?, ?] or heterogeneous node [?]. While achieving equally good or higher performance than lower-level approaches, the main benefit of these task-based approaches is that they can be written as extremely compact codes following a sequential task-based programming model, namely the sequential task flow (STF) paradigm.

In this article, we assess the potential of task-based FMM on distributed memory machines, more specifically on clusters of multicore processors. We first propose a hybrid MPI+Task FMM parallelization. In this scheme, MPI communications are submitted as tasks so that they can be finely interleaved with computational tasks by the runtime system. However, communications must still be explicitly written by the programmer, leading to a relatively complex, error-prone code. Therefore, we propose a second algorithm where the MPI communications are implicitly handled by the runtime system. This latter approach yields a compact code, yet very competitive, following a sequential task-based paradigm.

Beyond the design of two new task-based codes (sections 3.1 and 3.2) for exploiting distributed-memory machines, the main contribution of this study is to show how a very competitive performance can be achieved with a compact sequential task-based algorithm on distributed memory machines for one of the most important algorithms in scientific computing. Although demonstrated at a relatively modest scale (960 cores), especially with respect to the aforementioned previous studies, we think that these results are important enough to draw more attention from the parallel and distributed processing community on the design of fully-featured, scalable numerical libraries on top of task-based runtime systems.

The rest of the paper is organized as follows. We present the FMM algorithm in Section 2 together with an MPI+OpenMP parallelization scheme (Section 2.2) and a task-based scheme for single node multicore architectures from [?] (Section 2.3). We then describe how we extended this task-based solution to distributed-memory parallelization using explicit or implicit communication approaches in sections 3.1 and 3.2, respectively. We finally assess the efficiency of the proposed algorithms in Section 4 before concluding (Section 5).

2 Background

The FMM is a widely used algorithm to reduce the quadratic complexity of dual-based interactions originally introduced in [?]. Among the top ten algorithms of the twentieth century [?], it has been the object of many studies and we refer the reader to [?] for a detailed overview. In the sequel, we present the key ingredients of the method together with an high-level description of the algorithm, yet accurate enough so that non specialists can have a clear view of the main data-structures FMM relies on.

Initially introduced in the context of N particles interacting via gravitational forces, FMM has since been used in a wide range of applications such as astrophysical simulations, molecular dynamics, the boundary element method, radiosity in computer-graphics and fluid dynamics. The crucial part of the FMM for reducing the complexity from $\theta(N^2)$ operations (one computa-

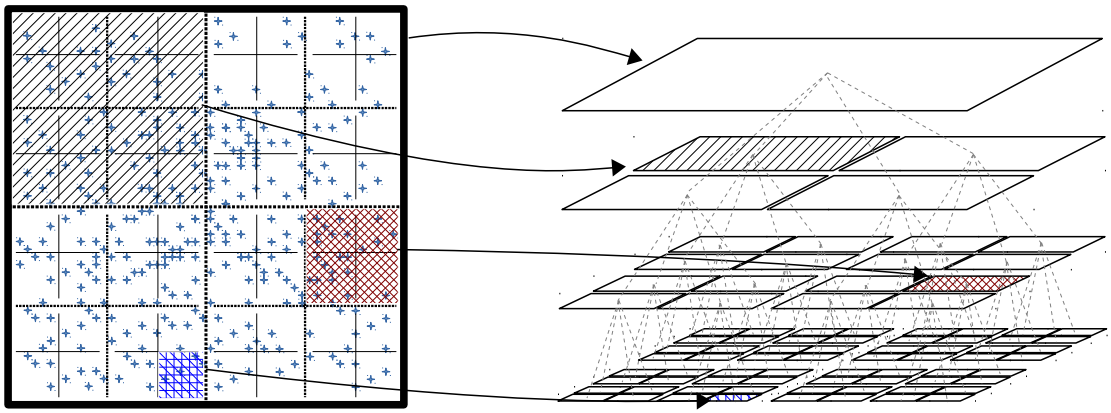


Figure 1: 2D hierarchical decomposition of the plane (left) and corresponding quadtree (right).

Algorithm 1: Sequential FMM

```
1  Function FMM(tree, kernel)
2      // Near-field
3      P2P(tree, kernel);
4      // Far-field
5      P2M(tree, kernel);
6      for l = tree.height-2 → 2 do
7          M2M(tree, kernel, l);
8      for l = 2 → tree.height-2 do
9          M2L(tree, kernel, l);
10         L2L(tree, kernel, l);
11     M2L(tree, kernel, tree.height-1);
12     L2P(tree, kernel);
13 Function M2M(tree, kernel, level)
14     foreach cell cl in tree.cells[level] do
15         kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
```

tion for each pair of particles) to $\theta(N)$ consists of approximating interactions between far-apart particles in a hierarchical manner. The approximation is done on the far-field - the interactions between far-apart particles - while maintaining a desired accuracy, exploiting the property that the underlying mathematical kernel decays with the distance between particles. As shown in Figure 1 for dimension $D = 2$, the original plane is recursively divided into four ($4 = 2^D$) sub-cells (left). This hierarchical decomposition is represented with a tree hence a quadtree in the particular 2D case (right), the root cell at the top of the tree being the original plane. We note h the height of this tree.

2.1 Sequential FMM

At the bottom of the tree, the interactions between close particles are computed with a classical, direct particle-to-particle (P2P) method (line 3 in Algorithm 1). The far-field is then processed as follows. An upward stage transfers the potentials from the leaves to the root by performing a computation from children to parents with a particle-to-multipole (P2M) operator at the bottom level (line 5) and a multipole-to-multipole (M2M) operator at the upper levels (lines 6-7). This M2M operator more precisely corresponds to a computation of a multipole expansion at a level

l based on the multipole expansions of the child cells at level $l+1$ as shown at lines 13-15 (for a matter of conciseness, we do not detail the other operators). A transfer stage is then applied at each level between each cell and its corresponding interaction list composed of cells at the same level but at an intermediate distance (closer cells being processed at a lower level and further cells at an upper level). This transfer pass corresponds to the computation of local expansions with the multiple-to-local (M2L) operator (lines 8-9, 11). Finally, the downward pass computes a potential from the root to the leaves, that is from parents to children through local-to-local (L2L) operators (lines 8,10) and local-to-particle (L2P) and the lowest level (line 12). Figure 2 illustrates how the contributions from different points of the grid are applied to a given leaf.

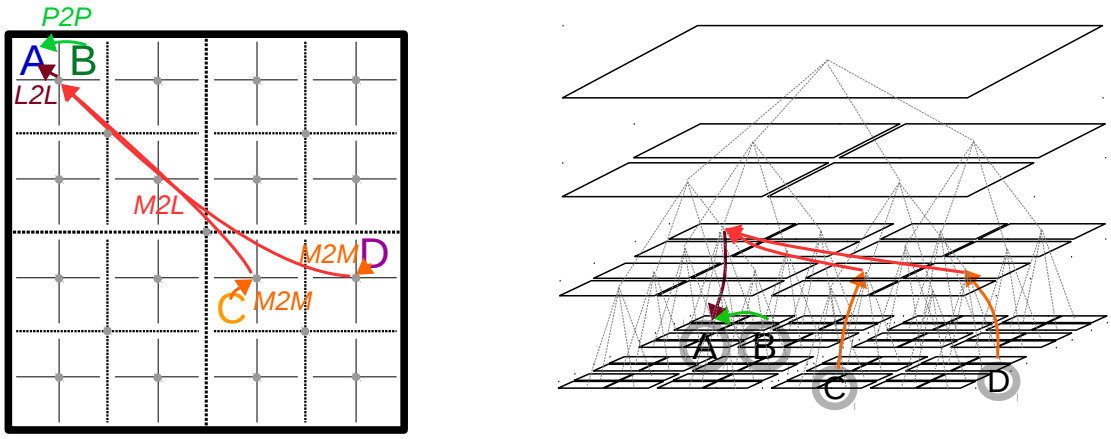


Figure 2: The contributions from leaves B, C and D are applied to A using the different operators depending on their respective distance to A.

2.2 MPI+OpenMP3 FMM (hierarchical clusters)

The first parallel implementations of the FMM for distributed memory machines were designed on top of MPI [**?**, **?**, **?**, **?**]. The advent of multicore processors and GPUs led the community to combine MPI for inter-node parallelism with either Posix or OpenMP threads within the node [**?**, **?**, **?**], possibly enhanced with accelerator directives [**?**, **?**, **?**]. We now present the hybrid MPI+OpenMP FMM algorithm implemented in ScalFMM. Although each FMM library has its own specificities and optimizations, this algorithm is representative of advanced FMM hybrid parallelization schemes and we refer to [**?**] for a more comprehensive overview. The entire FMM is divided into sub-trees mapped onto the different processes. In the particular case of ScalFMM (and in experiments conducted in this paper), the partitioning is done using the Morton space filling curve [**?**] such that each process owns a Morton index interval. We currently rely on a balancing heuristic ensuring that each process gets approximately attributed either the same number of particles or the same number of leaves.

Shaded lines in Algorithm 2 show how the sequential M2M operator FMM (lines 13-15 in Algorithm 1) is turned into an hybrid MPI+OpenMP algorithm, the main FMM function remaining unchanged as lines 1-12 in Algorithm 1. Once the threads are created (line 2 in Algorithm 2), one thread is dedicated to the communications stage (lines 4-9) while the other threads compute the local M2M (lines 11-13). Working at level l, the communicator thread sends the children of the first cell if needed (lines 5-6) and receives the children of the last cell if needed (line 7-8).

Algorithm 2: Hybrid MPI+OpenMP3 M2M (hierarchical clusters). The main FMM function remains as lines 1-12 in Algorithm 1.

```
1  Function M2M(tree, kernel, level)
2      #pragma omp parallel
3          // One thread dedicated to communication
4          #pragma omp single nowait
5              if must_send_first_children(tree, level) then
6                  MPI_ISend( tree.cells[level].first..children.multipole );
7              if must_recv_children_last_parent(tree, level) then
8                  MPI_IRecv(&recv_children);
9              MPI_Waitall();
10         // All threads compute local cells
11         #pragma omp for schedule(dynamic, 10) nowait
12         foreach cell cl in tree.local_cells[level] do
13             kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
14         // Wait completion of both communication and computation
15         #pragma omp barrier
16         // Compute received data if needed
17         if must_recv_children_last_parent(tree, level) then
18             kernel.M2M(tree.cells[level].last, recv_children);
```

Then, the communicator thread may join the computation if the communications are finished first. The barrier (line 15) ensures that the communication and computation are done when the threads potentially apply the received children (lines 17-18).

We refer to this algorithm as hybrid MPI+OpenMP3 in the sequel as it relies on OpenMP3 features, yet not employing dependent tasks from the revision 4 of the standard.

2.3 Task-based FMM (single node)

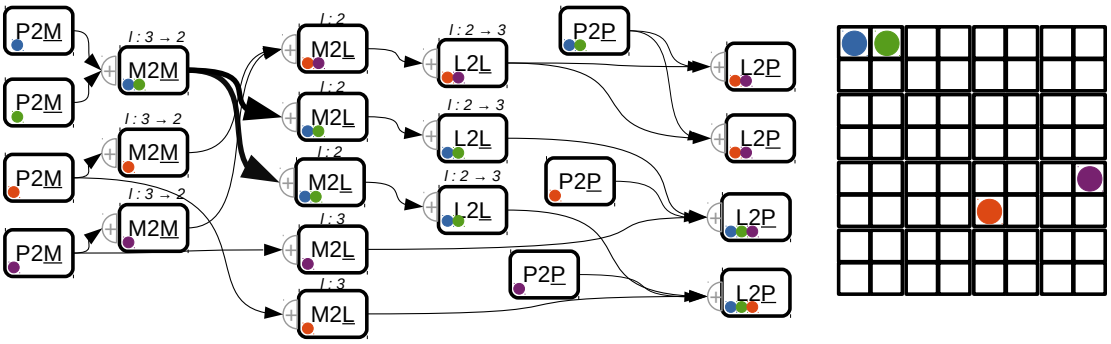


Figure 3: FMM DAG associated with the configuration depicted in Figure 2.

Task-based programming paradigms consist in designing an algorithm as a set of tasks and dependences between them. A directed acyclic graph (DAG) can be conveniently used to represent such an algorithm, tasks being associated with vertices and dependences with edges. DAGs have long been used in computer science to represent parallel programs. They are for instance commonly used at the instruction level in compilation [?] or, on the other side of the spectrum, at the job level in grid computing where they are commonly referred to as workflows [?]. With

the advent of multicore processors and accelerators as main processing units, the increase of hardware complexity led the parallel and distributed communities to rely once again on this principle as it allows a fine level of abstraction. Figure 3 shows for instance the FMM DAG associated with the particle distribution depicted in Figure 2.

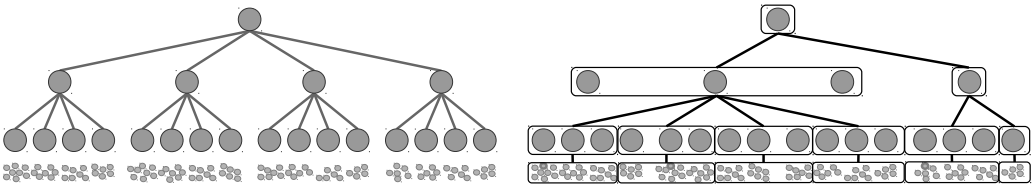


Figure 4: From a quadtree (left) to a group tree (right), $n_g = 3$.

One of the challenges of task-based programming is to ensure an appropriate granularity of the tasks. They must be fine enough so that many tasks can get executed concurrently but coarse enough so that the dependences between tasks can be handled at runtime without a significant overhead. In the particular case of the FMM, [**?**] showed that designing tasks as baseline FMM operators led to a significant overhead of the runtime system management. They introduced a new data structure, the group tree, to reduce this overhead. The subsequent algorithm indeed operates on groups of n_g (the *group size*) consecutive leaves or cells (following the Morton indexing) as illustrated in Figure 4.

Another challenge consists in (correctly) encoding the DAG. [**?**] proposed a task-based model, namely the *Parameterized Task Graph* (PTG), where both tasks and dependences (list of predecessors and successors) between them consist of parametrized expressions. This model was initially thought to describe regular algorithms (dense linear algebra factorizations) and later on extended to support irregular algorithms [**?**, **?**]. Let us once again consider the configuration and associated DAG represented in figures 2 and 3, respectively. Because they have no predecessors, P2M and P2P tasks can be immediately processed. When, say, the P2M task associated with the orange particle is completed, its M2M and M2L successors can be triggered since they have no other predecessor. In this model, the local expansion of the DAG traversal can thus be performed with the unique knowledge of the related P2M successors and M2M predecessors, without requiring any global view of the DAG. This property is exploited by the PaRSEC [**?**] runtime system to perform decentralized DAG unrolling and ensure an excellent scalability to the numerical task-based libraries that have been developed on top of it [**?**]. However, the expression of the dependences is very challenging.

On the contrary, the *Sequential Task Flow* (STF) model allows for automatically computing dependences based on sequential consistency. Tasks are inserted in a sequential (valid) order as one would write a sequential algorithm. Shaded lines in Algorithm 3 show how the sequential FMM (Algorithm 1) is turned into an STF algorithm. Instead of executing a kernel (such as the M2M kernel at line 15 in Algorithm 1), the execution of the kernel is submitted to the runtime system. The exact code for performing this submission depends on the runtime system that is employed but the principle is common to all runtime systems supporting the STF model such as StarPU [**?**], OmpSs [**?**], Quark [**?**] and many others. Without loss of generality, we consider the particular syntax of StarPU as it will be employed to illustrate the discussion in our experiments (Section 4). The submission is performed with the *starpu_insert_task*() instruction (line 18 in Algorithm 3). Based on the access modes (READ, WRITE or READ_WRITE), the runtime system automatically computes the dependences with previously submitted tasks. Contrary to compiler theory where the dependence analysis is performed statically and on scalars [**?**], the

Algorithm 3: STF FMM (single node)

```
1  Function FMM(tree, kernel)
2      // Near-field
3      P2P_insert_tasks(tree, kernel);
4      // Far-field
5      P2M_insert_tasks(tree, kernel);
6      for l = tree.height-2 → 2 do
7          M2M_insert_tasks(tree, kernel, l);
8      for l = 2 → tree.height-2 do
9          M2L_insert_tasks(tree, kernel, l);
10         L2L_insert_tasks(tree, kernel, l);
11     M2L_insert_tasks(tree, kernel, tree.height-1);
12     L2P_insert_tasks(tree, kernel);
13     // Wait for completion
14     starpu_wait_all_tasks();
15 Function M2M_insert_tasks(tree, kernel, level)
16     foreach cell_group cl in tree.cell_groups[level] do
17         foreach cell_group cl_child in cl.sub_groups do
18             starpu_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                   cl_child.multipole);
```

dependences are evaluated at runtime and at the granularity of tasks operating on groups of cells (the tree now being a group tree). The submission of the tasks being asynchronous, their completion is waited at the end of the algorithm (line 15).

3 Task-based FMM for hierarchical clusters

We now propose the design of two new task-based FMM algorithms for distributed memory machines. The first algorithm (Section 3.1) aims at exploiting the hierarchy of the cluster by combining explicit MPI communications for inter-node management and a task-based scheme within the node. While this scheme is designed to deliver very high performance by combining the benefits of explicit MPI communications together with the DAG parallelism inherent to task-based schemes, it leads to another hybrid scheme, whose synchronizations are as complex to handle as in the case of the MPI+OpenMP3 scheme discussed above. This is why we then investigate whether a competitive performance could be delivered with a compact STF algorithm (Section 3.2). The key idea of this second approach is to automatically infer MPI communications from the DAG [**?**, **?**, **?**].

3.1 Hybrid MPI+Task FMM for hierarchical clusters

The first task-based FMM algorithm for distributed memory machines we propose aims at exploiting the FMM DAG information within a node. The tree is built in parallel with the same procedure and mapping as in the case of the above hybrid MPI+OpenMP3 algorithm, except that it operates on a group data-structure. In the MPI+OpenMP3 FMM code, one thread was explicitly dedicated to the communications (lines 4-9 in Algorithm 2 for the M2M operator). In the MPI+Task algorithm, the communications are submitted to the runtime system with the *starpu_mpi_irecv/isend_detached* instructions (lines 6/8 in Algorithm 4). The runtime system takes in charge the posting of the actual communication requests to the MPI layer when the data gets available, transparently ensuring an advanced interleaving of the communications with computational tasks. For instance, StarPU-MPI [**?**] dedicates one thread to ensure the progress

Algorithm 4: Hybrid MPI+Task M2M (hierarchical clusters). The main FMM function remains as lines 1-16 in Algorithm 3.

```
1 Function M2M_insert_tasks_and_comms(tree, kernel, level)
2     foreach cell_group cl in tree.cell_groups[level] do
3         foreach cell_group cl_child in cl.sub_groups do
4             starpu_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                cl_child.multipole);
5         foreach cell_group cl_child in cl.remote.sub_groups do
6             starpu_mpi_irecv_detached(cl_child.multipole);
7             starpu_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                cl_child.multipole);
8         if cl has remote parent then
9             starpu_mpi_isend_detached(cl.multipole)
```

of communications while other threads perform computational tasks, submitted at lines 4 and 7 in Algorithm 4 in the case of the M2M operator.

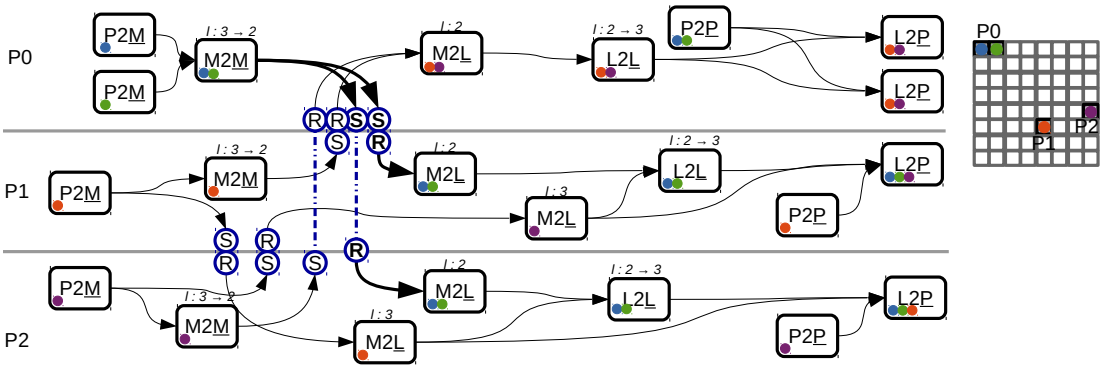


Figure 5: FMM DAG associated with the configuration depicted in Figure 2 scattered onto three processes. Communication tasks are represented with R (reception) and S (send) symbols.

We illustrate the behaviour of our algorithm considering once again the configuration depicted in Figure 2. We assume that leaves A (blue) and B (green) are mapped on process P0, whereas leaves C (orange) and D (purple) are mapped on processes P1 and P2, respectively, as shown in Figure 5 (right). Once P0 has processed the M2M task related to the blue and green cells, it sends (rightmost bold S tasks in Figure 5 as well as line 9 in Algorithm 4) the resulting multipole values to P1 and P2. Conversely, P1 and P2 post the matching receives (rightmost bold R tasks in Figure 5) within the hybrid MPI+Task M2L algorithm (not reported here).

3.2 STF FMM for hierarchical clusters

The hybrid MPI+Task algorithm proposed above strongly reduces the burden of pipelining the computational tasks with the communications compared to the hybrid MPI+OpenMP3 scheme, which had to rely on an advanced usage of OpenMP3 features to achieve a similar behaviour. We now propose to let the runtime furthermore infer the communications. The idea is relatively simple. The edges of the DAG represent the dependences between tasks. If two consecutive computational tasks are not mapped on the same process, the process associated with the predecessor submits a send request while the process associated with the successor submits a receive request.

For instance, assuming the distribution and mapping proposed in Figure 5, the communication tasks related to the blue and green cells occurring on P0 (send) and P1 or P2 (receive), respectively, can be readily inferred from the dependences between related M2M and M2L tasks in the global DAG (bold edges in Figure 3). As a consequence, one may write a compact STF code while achieving high performance on distributed memory machines. Successive studies [**?**, **?**, **?**] assessed (and refined) this protocol showing its potential on relatively regular numerical algorithms. However, none of them assessed it on an irregular, numerical algorithm such as we do in the present study with the FMM.

Algorithm 5: STF FMM (hierarchical clusters)

```
1  Function FMM(tree, kernel)
2      // Register and map the multipole/local/particles groups
3      map_data(tree);
4      // Near-field
5      P2P_mpi_insert_tasks(tree, kernel);
6      // Far-field
7      P2M_mpi_insert_tasks(tree, kernel);
8      for l = tree.height-2 → 2 do
9          M2M_mpi_insert_tasks(tree, kernel, l);
10     for l = 2 → tree.height-2 do
11         M2L_mpi_insert_tasks(tree, kernel, l);
12         L2L_mpi_insert_tasks(tree, kernel, l);
13     M2L_mpi_insert_tasks(tree, kernel, tree.height-1);
14     L2P_mpi_insert_tasks(tree, kernel);
15     // Wait for completion
16     starpu_wait_all_tasks();
17 Function M2M_mpi_insert_tasks(tree, kernel, level)
18     // Same tasks inserted on all procs
19     foreach cell_group cl in tree.cell_groups[level] do
20         foreach cell_group cl_child in cl.sub_groups do
21             starpu_mpi_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                cl_child.multipole);
```

We report the entire FMM algorithm following this STF model in Algorithm 5. Shaded lines show how the baseline STF FMM operating on a single node (Algorithm 3) is turned into an STF algorithm for a distributed memory machine, emphasizing the compactness of the resulting code. The data must first be registered to the runtime system and mapped on processes. The runtime system on each process registers the owner of all the data. After this registration and mapping step, the algorithm is exactly the same as the STF code operating on a single node, except that the *starpu_mpi_insert_task*() primitive is used instead of *starpu_insert_task*(). All processes fully unroll the DAG. This primitive uses the mapping information to automatically decide whether the task must be actually processed (in the case the process is the owner), whether a receive must be posted (the process is the owner of the task but not of its predecessors), and whether a send must be posted (the process is the owner of the task but not of a successor). Using this simple protocol yields the exact same DAG as the one in Figure 5 but communication tasks (S and R) are now implicitly inserted by the runtime system according to this protocol.

Data extraction While the MPI+OpenMP3 implementation from Section 2.1 explicitly builds the MPI messages by aggregating the cells/leaves, the task-based algorithms we propose rely on groups of cells/particles to perform the communication. Therefore, a complete group of n_g cells might be sent to compute a task by a different process even if this task requires only one cell from

Algorithm 6: STF M2M with extract (hierarchical clusters)

```
1  Function M2M_mpi_insert_tasks_extract(tree, kernel, level)
2      // Same tasks inserted on all procs
3      foreach cell_group cl in tree.cell_groups[level] do
4          foreach cell_group cl_child in cl.sub_groups do
5              proc_cl ← proc_owner(cl.multipole) ;
6              proc_child ← proc_owner(cl_child.multipole) ;
7              if proc_cl == proc_child then
8                  cl_child_multipole ← cl_child.multipole;
9              else
10                 cl_child_extracted ← init_data_on_node(proc_child);
11                 starpu_mpi_insert_task(extract, WRITE, cl_child_extracted, READ,
                     cl_child.multipole);
12                 cl_child_multipole ← init_data_on_node(proc_cl);
13                 starpu_mpi_insert_task(restore, WRITE, cl_child_multipole, READ,
                     cl_child_extracted);
14             starpu_mpi_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                 cl_child_multipole);
```

the group. If the granularity is not fine enough, this scheme may lead to exhange a non negligible amount of data. We thus propose a simple method that avoids such extra data movement. The key idea is to insert data extraction and restoration tasks when we know that a communication will happen. Algorithm 6 shows the application of this extraction protocol for the M2M operator. Before inserting a computational task, we test if its dependences have been mapped to the same process (line 7). If it is not the case, we create a new data on the same node where is located the data in read mode (line 10) and we insert a task to extract from the complete data only the useful part (line 11). We then allocate a data on the same node as the data in write mode (line 12) and insert a task to re-build a group in the correct form (line 13). Our protocol is valid thanks to the assumption that StarPU computes a task where the data in write mode is mapped. We refer to this variant as "STF with extract".

4 Experiments

4.1 Experimental Setup

Particle distributions We consider two types of particle distributions. *Cube (volume)* distributions are composed of particles uniformly distributed in a unit box, leading to a regular octree and a high and well balanced amount of work for each cell/leave. *Ellipsoid (surface)* distributions are composed of particles distributed on the surface of an ellipsoid with a high density at the poles, leading to an irregular octree and a highly variable amount of work associated with the nodes of the octree.

Hardware and software The experiments have been performed on the *Draco* computer from the Max Planck Institute. It is composed of Intel Xeon E5-2698 processors with 32 cores (2.3 GHz each). Each node has 128 GB of main memory and the nodes are inter-connected with InfiniBand FDR14. We use gcc 5.4, Intel MKL 11.3, Intel MPI 5.1.3 and StarPU 1.2. We exploit the commutativity of FMM operations (see [**?**] for details). Task-based algorithms run on top of StarPU-MPI [**?**]. The STF algorithms furthermore rely on a submission window to limit the pressure on the MPI layer [**?**]. All these features we rely on are embedded in StarPU 1.2.

Except stated otherwise, we assign 32 workers per node (OMP_NUM_THREADS=32 for the MPI+OpenMP3 algorithm and STARPU_NCPU=32 for task-based algorithms).

FMM The mathematical kernel is a non-adaptive Chebyshev FMM derived from [**?**] with an accuracy of 10^{-6}. The particles are divided among the processes to balance the number of leaves and the height of the tree h is chosen such that the execution using one node is minimal (assessed with the MPI+OpenMP3 code).

Mapping In all methods, the particles are first distributed among the processes to make each of them owning the particles inside a Morton index interval. In the MPI+OpenMP3 and MPI+Task algorithms, each process builds its own local tree and all the processes exchange some symbolic information later on used for performing communications. In the STF approaches, each process builds the complete symbolic octree. The symbolic information is thus replicated, however, the computational parts of the cells and the particles are allocated only on the node they are mapped to. Therefore, the processes that do not own a particular data can be seen as having an empty memory pointer on that data. All the data are registered by all the processes to StarPU using a function which is able to record the owner process. To be consistent with hybrid schemes, we do not build the groups over the complete tree but on the Morton interval of each process.

4.2 Results

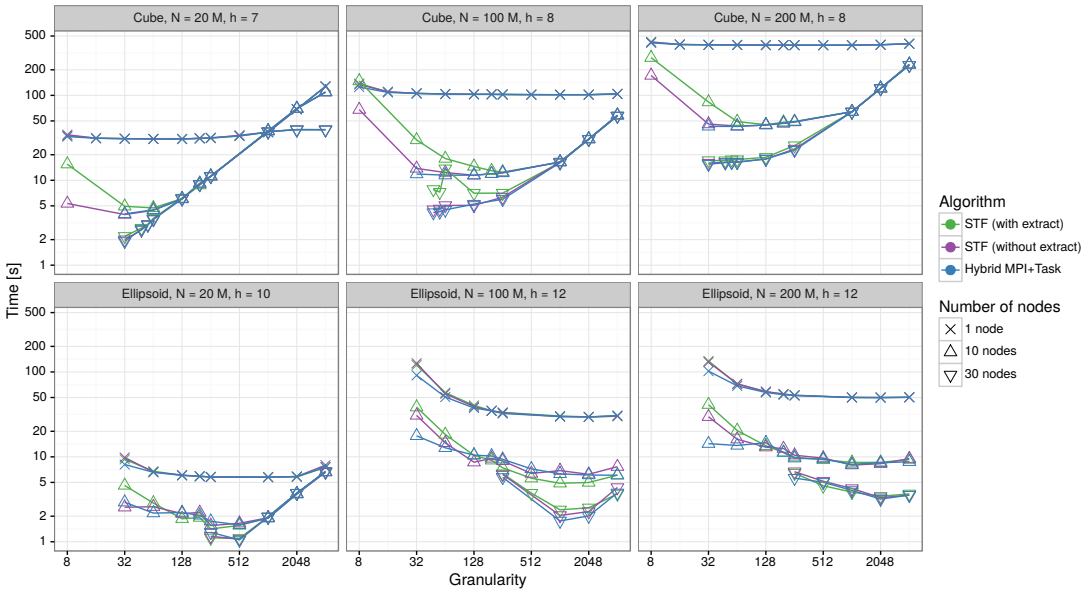


Figure 6: Impact of the granularity (log2 scale) on the execution time (seconds, log10 scale).

Effect of the granularity As discussed above, one of the key factor for achieving high performance with a task-based algorithm is to rely on an appropriate granularity of the tasks. We present its impact on the overall execution time and total communication volume in figures 6 and 7, respectively. We performed intensive tuning and the main observation is that maximum

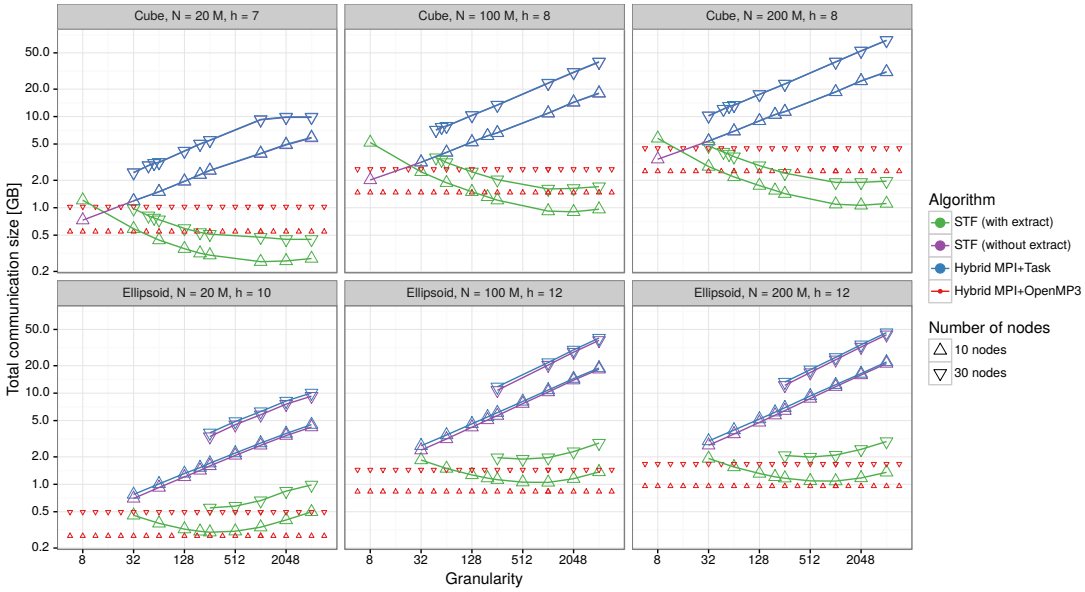


Figure 7: Impact of the granularity (log2 scale) on the total volume of communication (GB, log10 scale). Note that the STF (without extract) and the Hybrid MPI+task perfectly overlap with each other in the cube case (and almost overlap in the ellipsoid case).

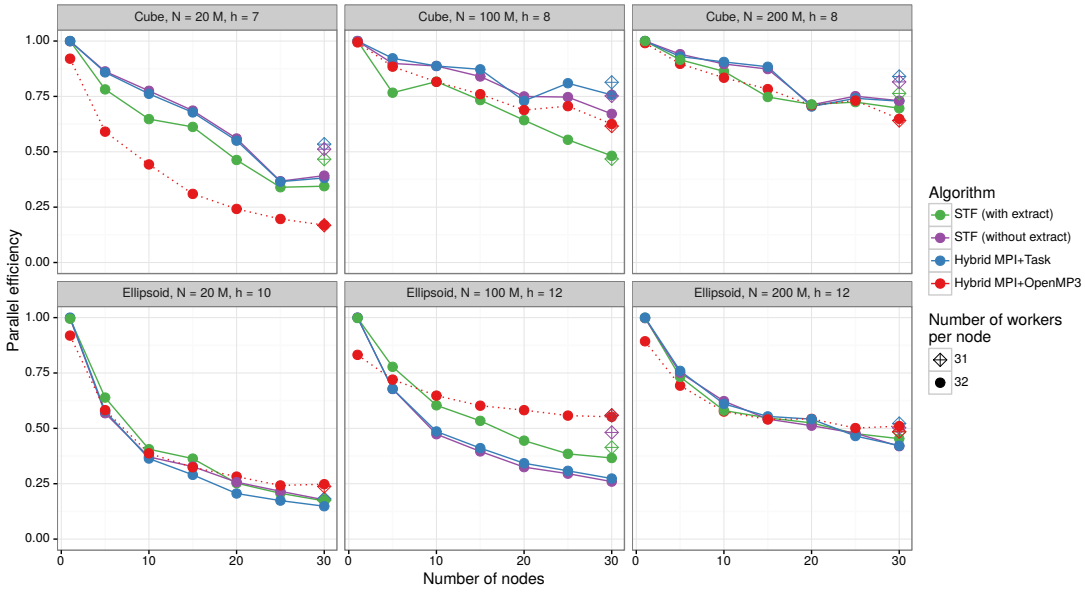


Figure 8: Parallel efficiency (normalized with the fastest algorithm on a single node).

performance is achieved with relatively low granularity for cube test cases. Furthermore, cube test cases have a high computation to communication volume ratio. As a consequence, in spite of the much higher volume of communication, the schemes without extract (purple plot for STF, and blue plot for hybrid MPI+Task) achieve a maximum performance. Indeed, they are not penalized by the high volume of communication, which is fully overlapped with computation, while not requiring to perform extraction operations. They achieve the optimum performance with relatively low granularity.

On the contrary, the computation to communication volume ratio being low for ellipsoid distributions, they significantly benefit from the extraction scheme (green plot). In particular, the STF with extract scheme remains efficient for large granularities where the ellipsoid achieves higher performance due to the performance kernel of the individual tasks, as shown by the behaviour on a single node execution (crosses) where there are no communications.

Efficiency Figure 8 shows the parallel efficiency of the different strategies normalized with the fastest execution on one node. We first discuss the results where the number of workers corresponds to the number of cores (circle points), consistently to the above results with variable granularity, and we report the performance obtained with optimum granularity. The main result is that the task-based schemes are very competitive with respect to the optimized MPI+OpenMP3 scheme. Furthermore, the STF variant (purple plot) almost consistently matches the performance of the hybrid MPI+Task code (blue plot). The STF with extract scheme (green plot) brings significant benefits on the ellipsoid test case over both the STF without extract (purple plot) and MPI+Task (blue plot) schemes and is in general competitive with the MPI+OpenMP3 scheme (red plot), except for the 100 M test case, which is a hard numerical configuration (few particles with respect to the tree height) for which further tuning in the priorities shall be investigated.

Finally, when the number of nodes become large, it may be interesting to dedicate one CPU core to perform task insertions and manage the communications. We show the benefit of this strategy with 30 nodes (diamond points). In this configuration, all in all, the compact STF achieves a significant speed-up over the MPI+OpenMP3 code on all three assessed cube test cases (20 M, 100 M, 200 M) and also significalntly improves the performance on the 100 M ellipsoid test cases., especially for the hard 100 M configuration where the task-based algorithm MPI+Task version now outperforms the MPI+OpenMP code. This illustrates one of the strength of task-based approaches: relying on a runtime system allows to benefit from advanced features (here, communication progress) only by setting up a configuration variable (here, STARPU_NCPU=31) without requiring to fully re-design an advanced scheme (here, a communication scheme).

5 Conclusion

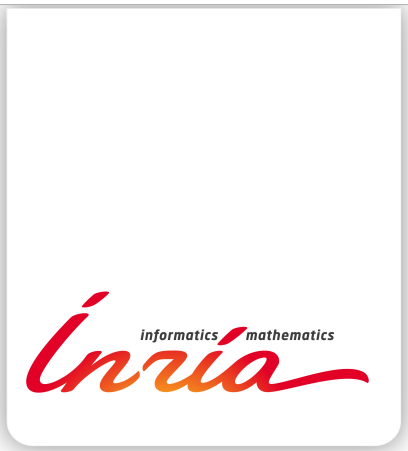
We have proposed two task-based FMM algorithms for clusters of multicore nodes. Similarly to most high-performance codes for such types of architectures, the first algorithm is a hybrid approach that aims at combining the benefits of explicit MPI communications together with the DAG parallelism inherent to task-based schemes. The second algorithm is a very compact code that delegates the burden of both synchronizations and communications to the StarPU runtime system. Both approaches have been implemented within the ScalFMM library to fairly compare them with the optimized MPI+OpenMP3 code while relying on the same computational kernels and data distribution schemes. We showed that a very competitive performance can be achieved with a compact sequential task-based algorithm on a cluster of multicore processors. Although demonstrated at a relatively modest scale (960 cores), we hope that these results will encourage

the parallel and distributed processing community to further consider the design of fully-featured, scalable numerical libraries on top of task-based runtime systems.

We considered a variant for the second algorithm that performed the extraction of the exact data to be communicated. On (very) irregular data distribution such as the ellipsoid test case, this extraction mechanism is vital in order to allow the algorithm to operate at a relatively coarse granularity (for ensuring the efficiency of the kernels when operating on irregular data) while maintaining a reasonable volume of communications. We plan to study the opportunity to also delegate this extraction scheme to the runtime system. Another important feature of the STF model is that it provides the opportunity to perform any data and task mapping, as the mapping is then orthogonal to the design of the code. We thus plan to develop new mapping strategies for ensuring load balancing both in the near-field and in the far-field.

Acknowledgment

Computations were performed on the HPC system "Draco" of the Max Planck Computing and Data Facility (RZG). We thank the StarPU development team for their support as well as Grégoire Pichon for proofreading an early draft of this paper.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399