Hybrid MPI-Thread Parallelization of the Fast Multipole Method

Olivier Coulaud, Pierre Fortin, Jean Roman

Hybrid MPI-Thread Parallelization of the Fast Multipole Method

Olivier Coulaud* Pierre Fortin*† Jean Roman*,
*INRIA Futurs - LaBRI, ScAlApplix project, Université Bordeaux 1, 33405 Talence Cedex - France
olivier.coulaud@inria.fr, {fortin|roman}@labri.fr
†Laboratoire d'Astrophysique de Marseille, 2 place Leverrier, F-13248 Marseille Cedex 4 - France

Abstract

We present in this paper multi-thread and multi-process parallelizations of the Fast Multipole Method (FMM) for Laplace equation, for uniform and non uniform distributions. These parallelizations apply to the original FMM formulation and to our new matrix formulation with BLAS (Basic Linear Algebra Subprograms) routines. Differences between the multi-thread and the multi-process versions are detailed, and a hybrid MPI-thread approach enables to gain parallel efficiency and memory scalability over the pure MPI one on clusters of SMP nodes. On 128 processors, we obtain 85% (respectively 75%) parallel efficiency for uniform (respectively non uniform) distributions with up to 100 million particles.

1. Introduction

The N-body problem in numerical simulations describes the computation of all pairwise interactions among N bodies (or particles). The Fast Multipole Method (FMM) [2] developed for gravitational potentials in astrophysics and for electrostatic potentials in molecular simulations, solves this N-body problem for any given precision with $\mathcal{O}(N)$ runtime complexity against $\mathcal{O}(N^2)$ for the direct computation. The potential field is decomposed in a near field part, directly computed, and a far field part approximated thanks to multipole and local expansions. The maximum degree in the expansions, denoted by P, determines the accuracy of the FMM: higher accuracies require higher values of P. This decomposition is performed thanks to a 3D octree, and the algorithm requires first an upward pass of this octree in order to build the multipole expansion of all cells in the octree. Then, during a downward pass, the local expansion of each cell c is computed from the multipole expansions of all cells in its *interaction list*, which are all *well-separated* from c: two cells are *well-separated* provided they do not share a boundary point. The final potential is deduced from the local expansions at the leaf level (far field) and from the direct computation with the nearest neighbors (near field). More details can be found in [2].

In the FMM for Laplace equation, the multipole-to-local operator (*M2L*), which converts a multipole expansion into a local expansion, represents most of the runtime of the far field computation. In [3, 4], we have proposed to rewrite this *M2L* operator as a matrix product. Thanks to the BLAS (Basic Linear Algebra Subprograms) routines [5], which are highly efficient routines performing matrix operations, we have thus gained substantial runtime speedup on modern superscalar architectures. When considering the memory needs, the numerical stability and the runtimes required for the targeted precisions in astrophysics and in molecular dynamics (between 10^{-2} and 10^{-7}), it appears that the BLAS formulation is better than the other improvements of the *M2L* computation: (block) Fast Fourier Transform, rotations and plane wave expansions. This comparison has been realized for the uniform FMM (used with uniform distributions of particles) [4], as well as for the adaptive FMM (used with non uniform distributions of particles) [3] where uniform areas can be detected in order to fasten the *M2L* computation in our BLAS version: bigger uniform areas imply shorter computation times. All this has been implemented in a code named FMB for Fast Multipole with BLAS [6].

In this paper, we focus on the parallelization of the FMM, where the computation space is decomposed equally in several subdomains, each domain being treated by one given processor. In practice, if efficient parallelizations have been realized in the uniform case [9, 11, 12, 16], the non uniform case is more difficult to handle. This is due to the irregularity of the particle distribution and to the hardly predictable communication scheme. Moreover, the different computation steps of the FMM algorithm have potentially different costs, depending on the number of particles per cell and on the number of expansions, which would imply a different decomposition for each phase [13].

Historically, the first decomposition proposed for N-body algorithms is the *Orthogonal Recursive Bisection* (ORB) for the Barnes-Hut algorithm [8, 14] (which can be extended to the FMM): the computation space is re-

cursively halved along one dimension. Other decompositions have then been introduced: they rely directly on the octree space decomposition thanks to space-filling curves and cost functions. The *costzones* decomposition has been shown to be more efficient that ORB on common address space architecture in [13] (multi-thread mode) and similar decompositions, based on Morton or Hilbert ordering, have also been prefered to ORB on distributed memory architecture with message passing [12, 15] (multi-process mode). Moreover, in order to obtain efficient parallelization in message passing environment, communications have to be overlaped with computation, small messages have to be aggregated into bigger ones (large-grain communications), communications have to ordered so as to avoid contention, and *sender-driven* communications have to be prefered to *receiver-initiated* communications [7, 8, 12, 15].

In this paper, we plan to improve the parallelizations of the FMM in both multi-thread (POSIX threads) and multi-process (MPI standard) modes, and for both uniform and non uniform distributions of particles. This will be done for the *M2L* computation scheme without BLAS (*classic M2L* computation scheme) as well as for the one with BLAS, and we will introduce the first (to our knowledge) hybrid MPI-thread parallelization of the FMM for Laplace equation. This hybrid version is indeed well-adapted for nowadays common architectures like clusters of SMP (symmetric multi-processors) nodes with multi-core processors. The principles and implementations of these parallelizations will be presented in section 2, and section 3 will validate our results with different particle distributions.

In the following, *P2M* (resp. *P2L*) will denote the computation of a multipole (resp. local) expansion, *M2M* (resp. *L2L*) the translation of a multipole (resp. local) expansion, *M2P* (resp. *L2P*) the evaluation of a multipole (resp. local) expansion, and *P2P* the direct computation.

2. Principles and implementation

In [3], we have justified the choice of the algorithm of Nabors *et al.* [10] for the adaptive FMM instead of the algorithm of Cheng *et al.* [2]. In the algorithm of Nabors *et al.* we do not limit the number of particles per leaf, but we only fix the height of the octree: the possible numerous empty cells are simply skipped. There is a special treatment for the cells along a chain in the octree which saves *M2M* and *L2L* operations, and a threshold value on the number of particles per cell determines whether or not the expansions must be used for each cell. Thanks to these improvements, and contrary to the algorithm of Cheng *et al.*, the linear operation count of the FMM is guaranteed, as proved in [10], without any assumption on the distribution of particles. Moreover, we keep in this algorithm the two size-bounded and easily computable lists of the uniform case; in the context of parallel computing in multi-process mode this results in more predictable communication patterns than with the algorithm of Cheng *et al.*. The algorithm of Nabors *et al.* is thus used here and the octree height H is optimally set by the user in order to minimize the computation time: usually, the H value must roughly balance the near field and far field computations, which depends on the particle distribution and on the P value used. Moreover, we use the distinction on the threshold value between multipole and local expansions, as presented in [3], but we renounce to the special treatment of cells along a chain as justified in [3]: this indeed extends in practice the computation times because of the precomputation of the *M2L* transfer functions.

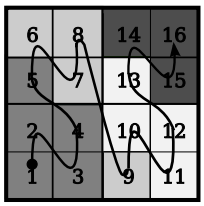

Figure 1. Morton decomposition for 4 processors at level 2 of a quadtree (2D octree).

For multi-thread and multi-process parallelizations, we need first to decompose the computation space among the threads or processes. In both cases, we will use the decomposition based on Morton ordering instead of the ORB decomposition in order to keep the octree space decomposition and to better adapt to highly non uniform distributions. An example of this Morton decomposition is pictured on figure 1: at each octree level, the total cost is first computed and each domain is then built with equal cost along Morton ordering. Hilbert ordering offers better data locality than Morton ordering (but with additional memory or computation cost), but as shown in [6], the Hilbert decomposition does not offer in practice better results than the Morton one. At each level in the octree, each thread or process has thus a continuous interval of Morton indexes (which represents its *domain*) and has to compute all FMM operations (*P2M*, *M2M*, *M2L*, *L2L*, *L2P*, *P2P*) for the target cells corresponding to these indexes.

2.1. Multi-thread parallelization

We first focus on parallel runs in shared-memory mode: there are one single process and as many threads as available processors (kernel threads), and we use here the POSIX thread standard. In multi-thread mode, there is no communication and therefore less constraint on the decompositions. Contrary to the other decompositions already proposed, like costzones [13], we will use different cost functions (and hence different intervals) for the different FMM operations in order to improve load balancing.

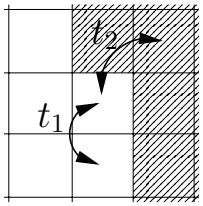

Figure 2. Conflict between two *P2P* operations performed by threads t_1 and t_2. t_2 domain is striped, and t_1 one is left blank.

2.1.1 Near field computation

For near field computation, the direct computation is performed by the *P2P* operator with the mutual interaction principle, which enables to update the particles of the target cell as well as the particles of the source cell at each *P2P* operation, saving thus half of the computation. This implies a time dependency and can result in write/write conflicts at the domain borders, as shown in figure 2. With our decomposition based on Morton ordering, no assumption can be stated on the geometrical shape of the domains. However, two Morton indexes whose values are close correspond to two cells that are likely to be close in space. Inversely, the first indexes of each interval should be different enough, so that the corresponding cells are not neighbor cells. So when each thread performs the near field computation by browsing its interval in increasing order, we should avoid most conflicts thanks to Morton ordering.

In order to treat the remaining conflicts, we choose to apply mutual exclusion mechanisms to all particles of a leaf at the same time: a single bit per leaf is set to '1' every time the particles of this leaf are updated by a *P2P* operation ('0' otherwise). This operation corresponds to a "test and set" operation on the bit, which is performed atomically thanks to *spinlocks* (busy waiting): there is one spinlock for each thread domain. When the bit of the leaf is already set to '1', we use two "FIFO" (*First In First Out*) structures to differ the *P2P* operation on this leaf. The first FIFO structure is used for the source cells and is flushed (possibly in several times) at the end of the interaction list browsing, whereas the second FIFO structure is used for the target cells and is flushed (possibly in several times) at the end of the thread domain browsing. Besides, in order to obtain good load balancing for highly non uniform distributions, the cost function used to build the intervals of the near field computation considers the number of particles in the target cell and in all neighbor source cells.

2.1.2 Far field computation

For far field computation, we have data dependencies for *M2M*, *M2L*, *M2P*, *L2L* and *L2P* operations, and we have to decompose the internal levels of the octree too. As there is no communication in multi-thread mode, we are here free to decompose the internal levels of the octree independently. In order to build the different thread domains for the *M2M* and *P2M* operations, we consider only for the cost function computation whether the target cell is empty or not: this cost is best adapted to the *M2M* operation, and we impose it to the *P2M* operation in order to avoid conflicts during the upward pass. For the *M2L* operation, we consider whether the source cells in the interaction list are empty or not. And for the *L2L* operation, we use the same intervals as for the *M2L* operation in order to avoid conflicts during the downward pass. Finally, the *L2P* operation cost function is linear with the number of particles inside each leaf.

As far as the algorithm of the FMM is concerned, we insert the direct computation phase between the upward pass and the downward pass, so that in practice almost all data dependencies are met for the *M2L* operation. In order to treat the unmet dependencies between the *P2M*, *M2M*, *L2L* and *L2P* operations during the upward and the downward passes, we use a mechanism similar to the one used for the near field computation: two bits per cell indicate whether or not the multipole and local expansions have been computed. A FIFO structure is also used to differ the treatment of the operation when the source expansion is not computed yet.

Finally, our BLAS computation scheme has also to be parallelized with multiple threads. However, it is not efficient to use already multithreaded BLAS libraries (like ESSL SMP [1]): for common P values, our matrix products are too small to offer a large enough computation grain to each processor. That's why we equally decompose the set of BLAS calls among the different threads while maximizing data locality to optimize cache effects. Each BLAS call is then performed by one thread only.

2.2. Hybrid MPI-thread parallelization

Now that we have a multi-threaded parallelization of our FMM, we aim to a multi-process MPI parallelization with multi-thread computation within each process, which is well-suited for clusters of SMP nodes. Contrary to the multi-thread mode, the required communications among the processes prevent us from using different decompositions for the different FMM operations, and from having independent decompositions at each level in the octree. We therefore use one single decomposition at the leaf level, induced by a cost function which is based on the number of particles per leaf. The internal cells in the octree are then assigned to the process which owns the first child. Like this, we minimize the number of communications required for the *P2M*, *M2M*, *L2L* and *L2P* operations between two consecutive levels. The difference between one of our decompositions in multi-thread mode and the decomposition

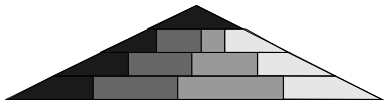

Figure 3. Multi-thread decomposition.

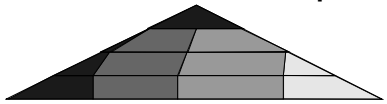

Figure 4. Multi-process decomposition.

in multi-process mode are pictured in figures 3 and 4: on these schematic octree views, the domain of each thread has a different gray value (4 threads here). Within each process, decompositions are then performed among the different computation threads just like in multi-thread mode. In practice, the two interval boundaries of each process are known by all processes.

In order to aggregate the messages corresponding to the interaction list and to the nearest neighbors, we use the *pack/unpack* mechanism offered by MPI. Despites its additional copy, this mechanism is indeed much simpler to use than a "zero-copy" mode, and it avoids potential conflicts between the different threads. Contention in the communication sub-system is simply reduced by ordering the messages to be sent according to a circular order.

As usual in FMM parallelization (see [12]), we renounce to the use of the mutual interaction principle between two processes. We also overlapp communications with computations in a classic way for the FMM (see [12] for example): nearest neighbor communications (for direct computation) are performed during the upward pass, whereas interaction list communications are performed during the direct computation step. More precisely, thanks to the choice of the algorithm of Nabors *et al.*, we have a total symmetry between the source cell and the target cell for *M2L* (interaction list) and *P2P* (nearest neighbors) operations. This is the principle of the inverse interaction list used in some uniform parallel FMM algorithms [7, 12]. Each computation thread can therefore determine among its cells which ones have to be sent to whom without any preliminary communications. All communications are thus *sender-driven* and we use non-blocking MPI calls for sending. For receiving, we use one additional thread, the *receiving* thread, with blocking MPI calls. This thread enables to receive incomming messages as soon as possible, and do not consume CPU time between two messages. The packings and unpackings, as well as the message sendings, are performed by the computation threads. Communications for *M2M* and *L2L* operations are treated differently. Due to the very short delay between the source data computation in the distant process and its requirement in the local process, we directly send (thanks to the MPI *eager limit*) one message per cell as soon as the source computation is over. Thanks to our decomposition based on the first child, the number of these messages is small and can be determined locally.

Finally, as far as our BLAS computation scheme is concerned, each process proceeds independently with its local cells only: the available uniform areas are thus possibly decomposed among the different processes.

3. Experimental validation

We present here results obtained on the IBM p575 cluster of the M3PEC center (University of Bordeaux 1). There are 8 SMP nodes linked with a 2×12 Gb/s Federation network. Each node has 28 GB memory and 8 dual-core 1.5 Ghz Power5 processors, viewed here as 16 processors since the memory caches (1.9 MB L2 cache and 36 MB L3 cache shared by the two cores) are big enough for our computations. Our hybrid MPI-thread parallelization requires a thread compliant MPI implementation (here, the IBM vendor MPI implementation (version 1.2) on AIX 5.3).

In the following, we will present execution times and parallel efficiencies of one single FMM computation. The times used to build the octree and to obtain our static decompositions are not considered here; these steps are considered as precomputation ones, which can be themselves partly parallelized, and their costs can be amortized over several simulation timesteps. We will consider here different distributions of particles (from 1 million - 1M - to 100 million particles): complete uniform distributions or distributions with large available uniform areas (like a ball or a bar filled with particles), non uniform distributions (like a sphere or a cylinder covered with particles), and highly non uniform distributions used in astrophysics (the Plummer model, and the *hcg027* test case composed of 1.5M particles grouped in 50 galaxies). We use here $P = 3$ (small computation grain) and $P = 7$ (large enough computation grain): greater P values do not offer indeed better results than $P = 7$. For the low precision astrophysic simulations however, we use only $P = 2$ and no BLAS computation (due to the lack of uniform areas, see [3]). The octree height H ranges from 6 to 7 for the distributions with large uniform areas, from 7 to 10 for the cylinder and the sphere, and from 10 to 13 for the astrophysic simulations.

3.1. Multi-thread parallelization

3.1.1 Classic *M2L* computation

Results for our multi-thread parallelization on one SMP node are first presented with classic *M2L* computation on table 1 (the efficiencies in parentheses are for the near field computation only). Excellent efficiencies are obtained here for large enough computation grain ($P = 7$) as well as for

Threads		1	4	8	16
Uniform (3M)	P=3	109.0s 100% (100%)	28.0s 97.2% (95.6%)	14.0s 97.0% (93.6%)	7.0s **97.2%** (93.1%)
	P=7	1544.7s 100% (100%)	387.0s 99.8% (95.3%)	193.7s 99.7% (93.6%)	97.3s **99.2%** (92.9%)
Ball (2M)	P=3	64.5s 100% (100%)	16.5s 97.8% (98.6%)	8.2s 98.2% (98.5%)	4.1s **98.1%** (98.4%)
Bar (3M)	P=7	954.8s 100% (100%)	238.9s 99.9% (98.9%)	119.5s 99.9% (98.7%)	60.0s **99.4%** (98.6%)
Cylinder (1M)	P=3	15.5s 100% (100%)	4.1s 95.6% (93.2%)	2.0s 94.8% (93.1%)	1.0s **95.5%** (92.3%)
	P=7	60.1s 100% (100%)	15.2s 99.1% (99.8%)	7.6s 99.1% (99.8%)	3.8s **98.6%** (99.7%)
Sphere (2M)	P=3	45.5s 100% (100%)	11.9s 96.0% (94.0%)	6.0s 94.9% (89.5%)	3.3s **85.0%** (72.3%)
	P=7	176.0s 100% (100%)	45.2s 97.4% (92.3%)	23.3s 94.4% (84.1%)	13.0s **84.3%** (64.7%)
Plummer (1M)	P=2	56.4s 100% (100%)	15.2s 92.6% (93.3%)	8.2s 86.2% (79.7%)	4.7s **75.0%** (71.3%)
hcg027 (1.5M)	P=2	89.6s 100% (100%)	22.6s 99.1% (99.6%)	11.3s 99.3% (99.8%)	5.7s **98.4%** (99.6%)

Table 1. Execution times and parallel efficiencies with multithreading (classic *M2L*).

small computation grain ($P = 3$), and for uniform distributions (the completely uniform one, or the ball and the bar) as well as for non uniform distributions (like the cylinder). The sphere is harder to treat because of the higher particle concentration at the poles. The astrophysic Plummer distribution is highly non uniform and the computation grain is too small to accurately estimate the cost of the numerous empty cells in this case. Nevertheless, our efficiency is greater than 75% with 16 threads thanks to our different decompositions. Finally, the near field computation is very efficiently parallelized with more than 92% efficiency for 16 threads in several cases, which validates our assumption that most conflicts are avoided with Morton ordering.

3.1.2 *M2L* computation with BLAS routines

Results with our BLAS computation scheme are also given on table 2. The efficiencies are good or very good, depending on the distribution and on the computation grain. The efficiency indeed improves when the computation grain increases (the case of the sphere with $P = 7$ is misleading since most of the computation time is here due to the near field computation, which is harder to balance for the sphere, see table 1). Sometimes $P = 3$ implies a too small computation grain to have optimal efficiencies for BLAS computation among all threads: see for example the complete uniform case. Besides, it has to be noted that we have some parallel efficiencies greater than 100% here! This is simply due to the BLAS efficiency: the number of columns in our matrix products differs according to the number of threads used, which can improves the BLAS efficiency and thus the parallel efficiency. When considering execution times, it can also be noted that, compared to the classic *M2L* computation, our BLAS version is not always the fastest for $P = 3$ and mainly for non uniform distributions: this is due to the too small computation grain and to the lack of uniform areas (see [3]). But otherwise, the BLAS computation scheme greatly improves the execution times.

Threads		1	4	8	16
Uniform (3M)	P=3	72.3s 100% (100%)	19.6s 92.4% (95.6%)	11.2s 81.0% (93.6%)	7.3s **62.0%** (93.1%)
	P=7	311.0s 100% (100%)	79.9s 97.3% (95.3%)	41.3s 94.1% (93.6%)	22.5s **86.5%** (92.9%)
Ball (2M)	P=3	61.1s 100% (100%)	16.0s 95.1% (98.6%)	8.4s 91.0% (98.5%)	4.6s **82.4%** (98.4%)
Bar (3M)	P=7	197.5s 100% (100%)	49.9s 98.8% (98.9%)	25.6s 96.5% (98.7%)	13.6s **90.4%** (98.6%)
Cylinder (1M)	P=3	24.7s 100% (100%)	5.9s 104.0% (93.2%)	3.0s 103.9% (93.1%)	1.6s **99.3%** (92.3%)
	P=7	30.6s 100% (100%)	7.6s 101.0% (99.8%)	3.8s 101.1% (99.8%)	1.9s **100.0%** (99.7%)
Sphere (2M)	P=3	76.6s 100% (100%)	17.7s 107.9% (94.0%)	8.6s 111.0% (89.5%)	5.0s **96.6%** (72.3%)
	P=7	105.0s 100% (100%)	26.7s 98.1% (92.3%)	14.0s 93.6% (84.1%)	8.6s **76.5%** (64.7%)

Table 2. Execution times and parallel efficiencies with multithreading (BLAS *M2L*).

3.2. Hybrid MPI-thread parallelization

3.2.1 Classic *M2L* computation

We now present results for our MPI multi-process parallelization as well as for our MPI-thread parallelization. We start with *M2L* classic computation and we first determine the best layout between processes and threads on one node. As shown in table 3 (best values are written with **bold** type), this best layout is always achieved with the 1×16 layout (one process and 16 computation threads within). This is due to our different decompositions for the different FMM operations which leads to better load balancing. With fewer processes we also use more often the mutual interaction principle, which saves computation. Moreover, the 1×16 layout has always the lowest memory requirements: the octree data structure is indeed shared among all threads within the same process. From now on, the layout with 1 process and 16 computation threads will thus be used on each node in the MPI-thread version with classic *M2L* computation.

We can now presents results for the "pure MPI" and MPI-thread parallelizations on several nodes in figures 5, 6 and 7. Table 4 presents the corresponding execution times for the MPI-thread version. The pure MPI version corresponds to one single computation thread per process, and in this case we group MPI processes by 16 on a same node in order to exploit at best MPI communications with shared

Layout (Processes ×Threads)		16×1	8×2	4×4	2×8	1×16
Uniform (10M)	P=3	26.8s 4.9GB	25.5s 4.5GB	25.2s 4.1GB	24.8s 3.2GB	**23.7s** **1.5GB**
	P=7	142.1s 5.7GB	141.8s 5.0GB	140.6s 4.5GB	139.9s 3.5GB	**135.9s** **1.7GB**
Cylinder (10M)	P=3	13.0s 4.4GB	11.2s 4.4GB	11.0s 4.4GB	10.9s 4.33GB	**10.3s** **4.28GB**
	P=7	38.2s 5.5GB	37.6s 5.4GB	37.1s 5.3GB	36.8s 5.2GB	**35.7s** **5.1GB**
Sphere (10M)	P=3	26.3s 4.2GB	20.3s 4.1 GB	19.6s 4.1GB	19.8s 4.05GB	**18.2s** **4.00GB**
	P=7	94.5s 5.4GB	71.9s 5.2GB	71.9s 5.1GB	71.0s 5.0GB	**66.5s** **4.9GB**
Plummer (10M)	P=2	73.9s 5.6GB	51.4s 5.1GB	48.4s 4.8GB	46.5s 4.5GB	**44.3s** **4.3GB**
hcg027 (1.5M)	P=2	7.9s 0.84GB	7.1s 0.77GB	6.5s 0.73GB	6.3s 0.70GB	**5.8s** **0.68GB**

Table 3. Execution times and memory requirements for different layouts on 16 processors (classic *M2L*).

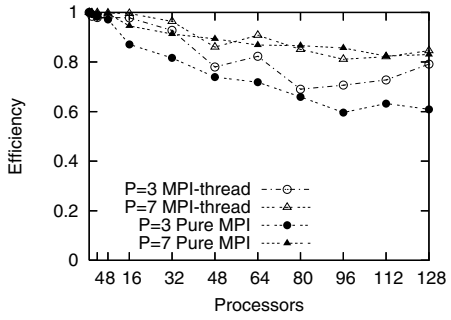

Figure 5. Parallel efficiencies for a 10M uniform distribution (classic *M2L*).

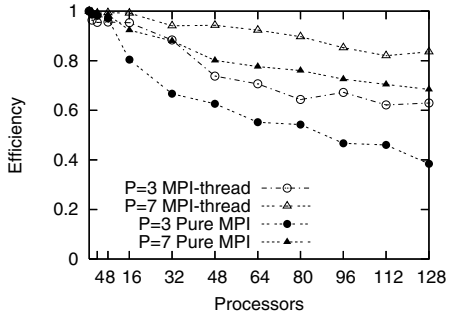

Figure 6. Parallel efficiencies for a 10M cylinder (classic *M2L*).

Processors		1	128
Uniform (10M)	P=3	371.5s	3.7s
	P=7	2163.6s	20.0s
Cylinder (10M)	P=3	156.3s	1.9s
	P=7	568.1s	5.3s
Sphere (10M)	P=3	240.5s	6.8s
	P=7	899.8s	26.3s
	P=7 (H+1)	1232.9s	18.9s
Plummer (10M)	P=7	618.7s	17.0s
hcg027 (1.5M)	P=7	90.9s	1.3s

Table 4. Execution times with MPI-thread for figures 5, 6, 7 and 8 (classic *M2L*).

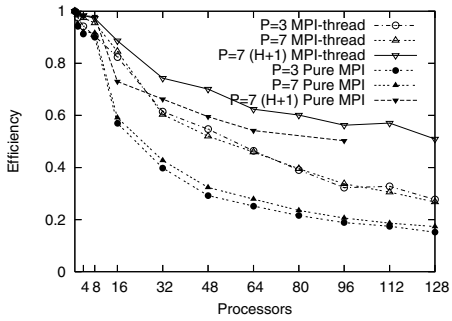

Figure 7. Parallel efficiencies for a 10M sphere (classic *M2L*).

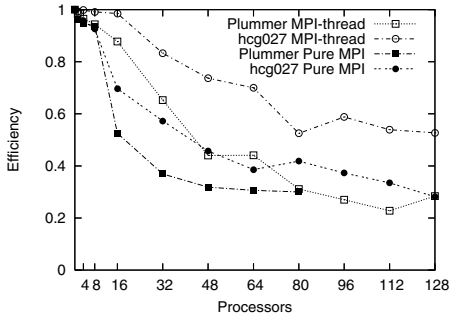

Figure 8. Parallel efficiencies for a 10M Plummer model and *hcg027* (classic *M2L*, $P = 2$).

memory on a same node. As far as pure MPI is concerned, the efficiencies are good for the uniform case and for the cylinder, and increase with the P value (that is to say with the computation grain size). This validates the choice of our adaptive algorithm as well as the choices done in section 2.2 concerning the multi-process parallelization. Because of the particle concentration at its poles, the sphere test case presents a load unbalancing which leads to much waiting for *M2M* and *L2L* communications. This is confirmed with a greater octree height ($H + 1$): in this case the number of particles decreases in every leaf, so that the load balancing and the efficiency improve significantly. This problem is mainly due to our unique decomposition in multi-process mode which can not balance equally all FMM steps.

That's why the MPI-thread version offers better results thanks to its different decompositions within each process, and thanks to the more important use of the mutual interaction principle. Since the serial times are similar, these gains in parallel efficiency really correspond to gains in execution times. Moreover, since the octree data structure is shared among all processors on a same node, the MPI-thread version also enables to improve the memory scalability. Hence, while the 10M particle sphere with $H+1$ can not be run on 80, 112 and 128 processors in the pure MPI version, the MPI-thread implementation succeeds. This is also valid for astrophysic distributions as shown on figure 8.

Layout (Processes ×Threads)		16×1	8×2	4×4	2×8	1×16
Uniform (10M)	P=3	25.1s 5.1GB	23.8s 4.7GB	23.3s 4.3GB	**23.2s** 3.3GB	25.0s **1.6GB**
	P=7	44.2s 7.4GB	41.9s 6.1GB	**40.4s** 5.4GB	40.8s 4.2GB	42.4s **2.4GB**
Cylinder (10M)	P=3	14.1s 4.6GB	13.3s 4.5GB	13.4s 4.5GB	**12.7s** 4.44GB	12.8s **4.39GB**
	P=7	30.5s 7.3GB	29.5s 6.4GB	28.5s 5.9GB	27.8s 5.6GB	**27.5s** **5.4GB**
Sphere (10M)	P=3	34.2s 4.4GB	26.7s 4.3GB	26.1s 4.3GB	**26.0s** 4.2GB	26.2s **4.15GB**
	P=7	68.7s 7.2GB	52.8s 6.2GB	51.9s 5.7GB	50.5s 5.4GB	**48.0s** **5.2GB**

Table 5. Execution times and memory requirements for different layouts on 16 processors (BLAS *M2L*).

Compared to the pure MPI version, our MPI-thread implementation enables to improve the parallel efficiencies for these highly non uniform distributions with fine computation grain. Moreover, the improvement in memory scalability is also clear here: while the 10M Plummer test case requires too much memory for 96 (or more) processors with the pure MPI version, this test can be run with the MPI-thread one thanks to its lower memory requirements.

3.2.2 *M2L* computation with BLAS routines

In order to present results for our MPI-thread parallelization with our BLAS *M2L* computation scheme, we need first to find the best layout on one node like for the classic *M2L* computation. As shown on table 5 (best values are written with **bold** type), when considering execution times, there is however no best layout for all P values and all distributions. This is due to the BLAS efficiency: when we use different numbers of processes (and also different numbers of threads), we modify the number of columns in our matrix products which affects the BLAS efficiency. This behaviour is difficult to plan, but all the multithreaded versions are faster than the pure MPI one, and the differences remain small among them. Moreover, the lowest memory requirements are always achieved with the 1×16 layout: all threads within the same process share the octree data structure, and also here the *M2L* transfer matrices of the BLAS version (which can be large for high P values, see [4]). That is why we keep the layout of the classic *M2L* computation: 1 process with 16 threads within.

Results for our pure MPI and MPI-thread parallelizations, both with the BLAS *M2L* computation scheme, are given on figure 9 for uniform distributions, on figure 10 for large uniform distributions, and on figures 11 and 12 for the cylinder and the sphere. Table 6 presents the corresponding execution times for the MPI-thread version. With the pure MPI version, we obtain good efficiencies for uniform distributions, as well as for the cylinder, for both small and

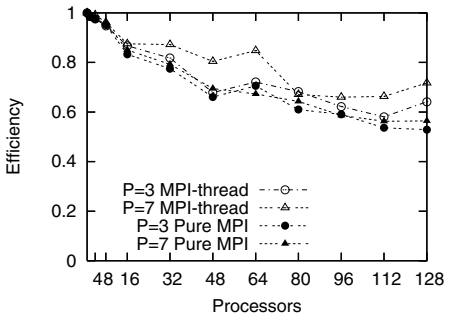

Figure 9. Parallel efficiencies for a 10M uniform distribution (BLAS *M2L*).

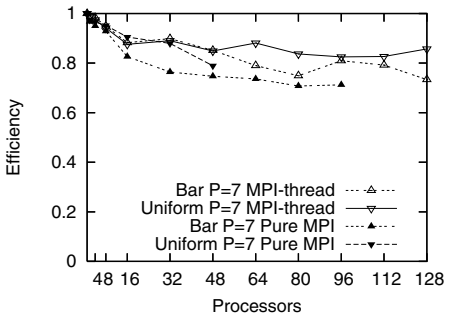

Figure 10. Parallel efficiencies for a 100M uniform distribution and a 30M bar (BLAS *M2L*).

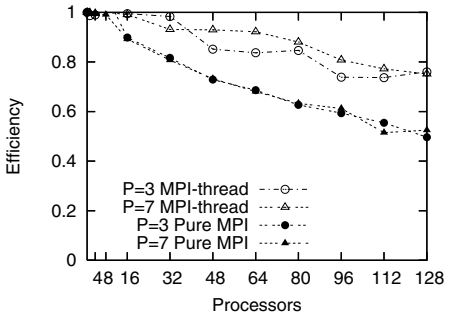

Figure 11. Parallel efficiencies for a 10M cylinder (BLAS *M2L*).

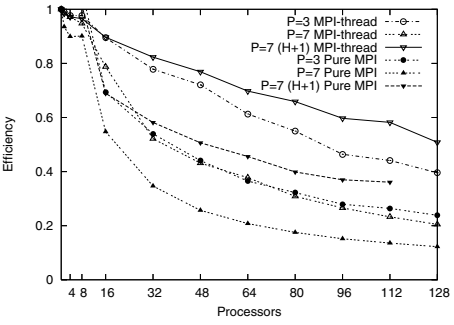

Figure 12. Parallel efficiencies for a 10M sphere (BLAS *M2L*).

Processors		1	128
Uniform (10M)	P=3	339.3s	4.1s
	P=7	606.3s	6.6s
Bar (30M)	P=7	1942.5s	20.7s
Uniform (100M)	P=7	6164.6s	56.2s
Cylinder (10M)	P=3	204.5s	2.1s
	P=7	435.4s	4.5s
Sphere (10M)	P=3	373.6s	7.4s
	P=7	605.9s	23.1s
	P=7 (H+1)	663.8s	10.2s

Table 6. Execution times with MPI-thread for figures 9, 10, 11 and 12 (BLAS *M2L*).

large computation grains. Moreover, these efficiencies increase with the distribution size since the computation load increases faster than the communication volume. Like for classic *M2L* computation, the case of the sphere is more difficult and leads to a load unbalancing because of the particle concentration at its pole, and a greater octree height ($H+1$) enables to greatly improve the load balancing.

Here again, our MPI-thread version significantly improves the parallel efficiencies, for all distributions and for all computation grain sizes, and thus the execution times (since the serial times are similar). With 128 processors, we obtain 85% efficiency for the 100M uniform distribution and 75% for the 10M cylinder. The memory scalability is also improved: in the pure MPI version, the 100M uniform distribution, the 30M bar and the 10M sphere (with $H+1$) can not be run up to 128 processors. Besides, with the MPI-thread version we have now bigger uniform areas within each process for our BLAS version. This explains partly the gains of the MPI-thread version for the 10M and 100M uniform distributions ($P = 7$), as well as the peaks for 64 and 128 processors: in these cases each of the 4 or 8 processes has indeed a cubical or 3D rectangular domain, corresponding exactly to an uniform area.

4. Conclusion

We have presented multi-thread and multi-process parallelizations of the FMM for the classic *M2L* computation scheme and for our new *M2L* computation scheme with BLAS routines. To our knowledge, we have also introduced the first MPI-thread parallelization of the FMM for Laplace equation. Our parallelizations are efficient for both uniform and non uniform distributions of particles, and for both large and small computation grains, which validates our choice of the adaptive algorithm of Nabors *et al.*. In addition, our MPI-thread implementation enables to reach better parallel efficiency and better memory scalability.

In the future, the octree height could be corrected from one timestep to another when the costs of the far field and the near field computations become unbalanced. We also plan to further improve the load balancing for highly non uniform particle distributions with small computation grain thanks to dynamic load balancing at the thread level, and thanks to load balancing correction over several simulation timesteps at the process level.

Acknowledgments

The authors wish to thank P. Ramet (ScAlApplix project, INRIA Futurs - LaBRI) and N. Ky at the M3PEC center.

References

[1] *Engineering and Scientific Subroutine Library (ESSL) for AIX, Version 3 Release 3, Guide and Reference*, 2001.

[2] H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155:468–498, 1999.

[3] O. Coulaud, P. Fortin, and J. Roman. High-performance BLAS formulation of the Adaptive Fast Multipole Method. *Journal of Supercomputing*. (to appear).

[4] O. Coulaud, P. Fortin, and J. Roman. High performance BLAS formulation of the multipole-to-local operator in the Fast Multipole Method. *Journal of Computational Physics*. (submitted, available at: http://hal.inria.fr/inria-00000957).

[5] J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1), 1990.

[6] P. Fortin. *Algorithmique hiérarchique parallèle haute performance pour les problèmes à N-corps*. PhD thesis, Université Bordeaux 1, 2006.

[7] J. Kurzaka and B. M. Pettitt. Communications overlapping in fast multipole particle dynamics methods. *Journal of Computational Physics*, 203(2), 2005.

[8] P. Liu and S. Bhatt. Experiences with parallel n-body simulation. *IEEE Trans. Parallel Distrib. Syst.*, 11(12):1306–1323, 2000.

[9] E. J.-L. Lu and D. I. Okunbor. A massively parallel fast multipole algorithm in three dimensions. In *Proc. of the High Performance Distributed Computing*, 1996.

[10] K. Nabors, F. Korsmeyer, F. Leighton, and J. White. Preconditioned, adaptive, multipole-accelerated iterative methods for three-dimensional first-kind integral equations of potential theory. *SIAM J. on Scientific Computing*, 15(3), 1994.

[11] S. Ogata, T. Campbell, R. Kalia, A. Nakano, P. Vashishta, and S. Vemparala. Scalable and portable implementation of the fast multipole method on parallel computers. *Computer Physics Communications*, 153(3):445–461, July 2003.

[12] W. Rankin. Efficient parallel implementations of multipole based N-body algorithms. PhD. Diss., Duke Univ., 1999.

[13] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Supercomputing '93*.

[14] M. Warren and J. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Supercomputing '92*, pages 570–576.

[15] M. Warren and J. Salmon. A parallel hashed oct-tree N-body algorithm. In *Supercomputing '93*, pages 12–21.

[16] L. Ying, G. Biros, D. Zorin, and H. Langston. A new parallel kernel-independent fast multipole method. In *Supercomputing '03*.