ScalFmm - Parallel Algorithms (Draft)

Berenger Bramas, Olivier Coulaud, Cyrille Piacibello

April 10, 2014

Contents

Introduction

In this document we introduce the principles and the algorithms used in our library to run in a distributed environment using MPI. The algorithms in this document may not be up to date comparing to those used in the code. We advise to check the version of this document and the code to have the latest available.

Building the tree in Parallel

Description

The main motivation to create a distributed version of the FMM is to run large simulations. These ones contain more particles than a computer can host which involves using several computers. Moreover, it is not reasonable to ask a master process to load an entire file and to dispatch the data to others processes. Without being able to know the entire tree it may send randomly the data to the slaves. To override this situation, our solution can be viewed as a two steps process. First, each node loads a part of the file to possess several particles. After this task, each node can compute the Morton index for the particles he had loaded. The Morton index of a particle depends of the system properties but also of the tree height. If we want to choose the tree height and the number of nodes at run time then we cannot pre-process the file. The second step is a parallel sort based on the Morton index between all nodes with a balancing operation at the end.

Load a file in parallel

We use the MPI I/O functions to split a file between all the mpi processes. The prerequisite to make the splitting easier is to have a binary file. Thereby, using a very basic formula each node knows which part of the file it needs to load.

$$sizeperproc \leftarrow (filesize - headersize) / nbprocs \tag{1}$$

$$offset \leftarrow headersize + sizeperproc.(rank - 1) \tag{2}$$

We do not use the view system to read that data as it is used to write. The MPI_File_read is called as described in the following C++ code.

```
// From FMpiFmaLoader
MPI_File_read_at(file, headDataOffSet + startPart * 4 * sizeof(FReal),
particles, int(bufsize), MPI_FLOAT, &status);
```

Our files are composed by a header fallowing by all the particles. The header enables to check several properties as the precision of the file. Finally, a particle is represented by four decimal values: a position and a physical value.
Remark: The MPI IO function do not work if we use a MPI_Initthread(MPI_THREAD_MULTIPLE) and a version above 1.5.1.

Sorting the particles

Once each node has a set of particles we need to sort them. This problem boils down to a simple parallel sort where Morton index are used to compare particles. We use two different approaches to sort the data. In the next version of scalfmm the less efficient method should be deleted.

Using QuickSort

A first approach is to use a famous sorting algorithm. We choose to use the quick sort algorithm because the distributed and the shared memory approaches are mostly similar. Our implementation is based on the algorithm described in [1]. The efficiency of this algorithm depends roughly of the pivot choice. In fact, a wrong idea of the parallel quick sort is to think that each process first sort their particles using quick sort and then use a merge sort to share their results. Instead, the nodes choose a common pivot and progress for one quick sort iteration together. From that point all process has an array with a left part where all values are lower than the pivot and a right part where all values are upper or equal than the pivot. Then, the nodes exchange data and some of them will work on the lower part and the other on the upper parts until there is one process for a part. At this point, the process performs a shared memory quick sort. To choose the pivot we tried to use an average of all the data hosted by the nodes:

```
Result: A Morton index as next iteration pivot
myFirstIndex ← particles[0].index;
allFirstIndexes = MortonIndex[$nbprocs$];
allGather(myFirstIndex, allFirstIndexes);
pivot ← Sum(allFirstIndexes(:) / nbprocs);
```

Algorithm 1: Choosing the QS pivot

A bug was made when at the beginning, we did an average by summing all the values first and dividing after. But the Morton index may be extremly high, so we need to to divide all the value before performing the sum.

Using a Sorting Network

In [2], a proposition has been made to sort the data using a sorting network. We implemented a such sorting algorithm but the result were not extremly efficient. Contrary to Quick sort, a sorting network is extremly stable and all the nodes performs similar work. The quick sort is pivot dependant and some nodes may work much more than other. But, the average case the quick sort enable higher efficiency.

Using an intermediate Octree

The second approach uses an octree to sort the particles in each process instead of a sorting algorithm. The time complexity is equivalent but it needs more memory since it is not done in place. After inserting the particles in the tree, we can iterate at the leaves level and access to the particles in an ordered way. Then, the processes are doing a minimum and a maximum reduction to know the real Morton interval of the system. By building the system interval in term of Morton index, the nodes cannot know the data scattering. Finally, the processes split the interval in a uniform manner and exchange data with P^2 communication in the worst case.

In both approaches the data may not be balanced at the end. In fact, the first method is pivot dependent and the second consider that the data are uniformly distributed. That is the reason why we need to balance the data among nodes.

Balancing the leaves

After sorting, each process has potentially several leaves. If we have two processes P_i and P_j with $i < j$ the sort guarantees that all leaves from node i are inferior than the leaves on the node j in a Morton indexing way. But the leaves are randomly distributed among the nodes and we need to balance them. It is a simple reordoring of the data, but the data has to stayed sorted.

1. Each process informs other to tell how many leaves it holds.

2. Each process compute how many leaves it has to send or to receive from left or right.

At the end of the algorithm our system is completely balanced with the same number of leaves on each process. If another kind of balancing algorithm is needed, one can only change the BalanceAlgorithm class that is given in parameter to the ArrayToTree static method in the step 2.

Balancing algorithms supported

Any balancing algorithm can be used, but it has to provide at least two method, as showed in the class FAbstractBalancingAlgorithm. Those methods are :

1. GetLeft : return the number of leaves that will belongs only to proc on the left of given proc.

2. GetRight : return the number of leaves that will belongs only to proc on the right of given proc.

In the parameters of those two methods, one can find the total number of leaves, the total number of particles, the number of proc, and the index of a proc to be treated.

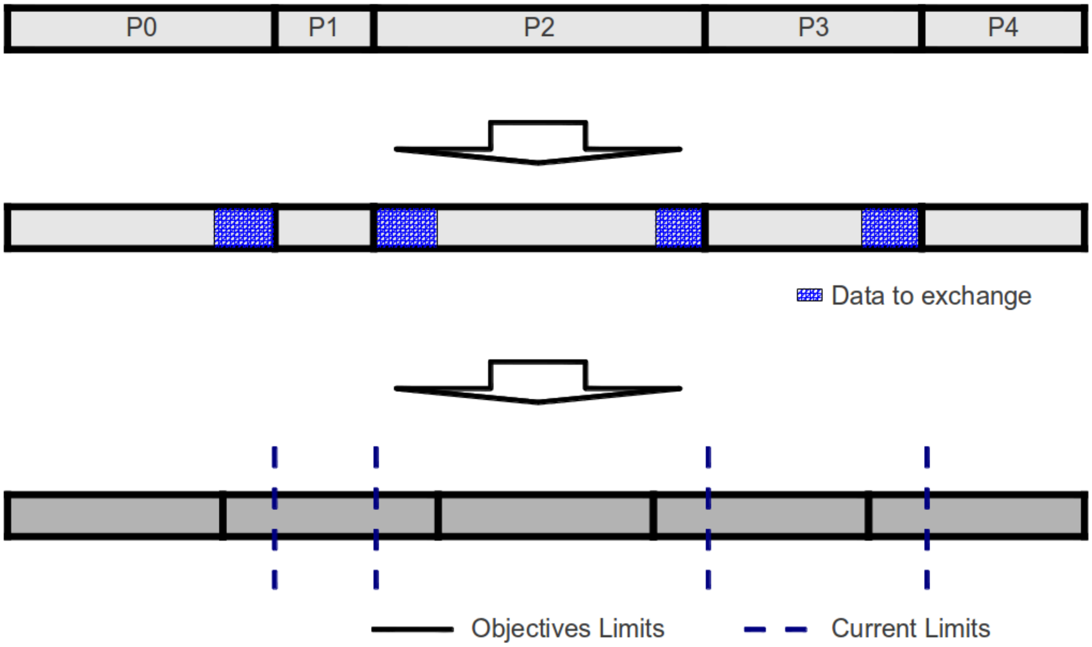

Figure 1: Balancing Example : A process has to send data to the left if its current left limit is upper than its objective limit. Same in the other side, and we can reverse the calculs to know if a process has to received data.

Mpi calls

Once every process know exactly what it needs to compute for itself and for any other proc the bound GetRight() and GetLeft(), there is only one Mpi communication AllToAll.

To prepare the buffers to be sent and received, each proc count the number of leafs (and the size) it holds, and divide them into potentially three parts :

1. The datas to send to proc on left (Can be null).

2. The datas to keep (can be null).

3. The datas to send to proc on right(can be null).

Distributed algorithm

Here present the different FMM operators in two separated parts depending on their parallel complexity. In this first part, we present the three simplest operators P2M, M2M and L2L. Their simplicity is explained by the possible prediction to know which node hosts a cell and how to organize the communication.

We will first present how the different processus can know which cell or leaf belongs to which processus.

Morton Index Intervals

A Morton Index Interval is a simple structure with two Morton indexes inside, referencing the first a last leaf of each processus. Each processus compute its Morton Index Interval at first by scanning all its leafs.

Once each processus compute its interval, there is a global communication for the processus to know the interval of the others, and the result is stored in an array of interval structures.

P2M

The P2M still unchanged from the sequential approach to the distributed memory algorithm. In fact, in the sequential model we compute a P2M between all particles of a leaf and this leaf which is also a cell. Although, a leaf and the particles it hosts belong to only one node so doing the P2M operator do not require any information from another node. From that point, using the shared memory operator makes sense.

M2M

During the upward pass information moves from a level to the upper one. The problem in a distributed memory model is that one cell can exist in several trees i.e. in several nodes. Because the M2M operator computes the relation between a cell and its child, the nodes which have a cell in common need to share information.

Moreover, we have to decide which process will be responsible of the computation if the cell is present on more than one node. We have decided that the node with the smallest rank has the responsibility to compute the M2M and propagate the value for the future operations.

Despite the fact that others processes are not computing this cell, they have to send the child of this shared cell to the responsible node.

We can establish some rules and some properties of the communication during this operation. In fact, at each iteration a process never needs to send more than 7 cells, also a process never needs to receive more than 7 cells. The shared cells are always at extremities and one process cannot be designed to be the responsible of more than one shared cell at a level.

There are to cases :

- My first cell is shared means that I need to send the children I have of this cell to the processus on my left.

- My last cell is shared means that I need to receive some children from the processus on my right.

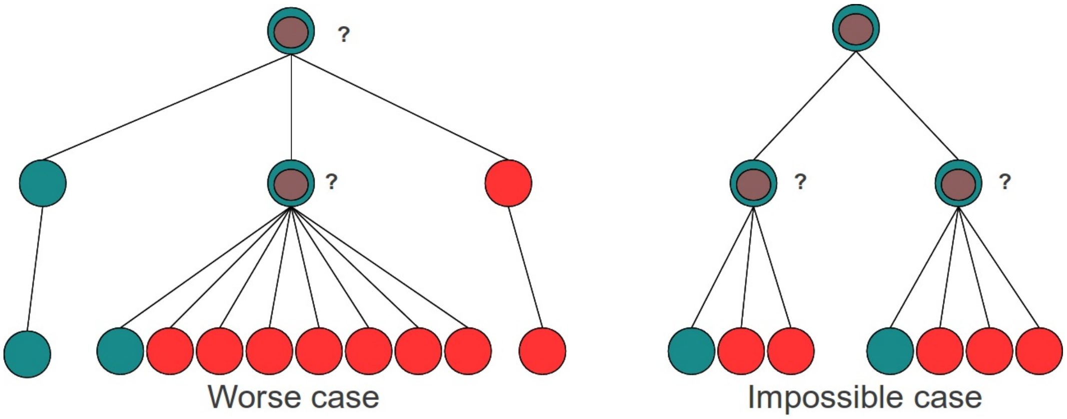

Figure 2: Potential Conflicts

```
Data: none
Result: none
for idxLevel ← $Height - 2$ to 1 do
    forall the Cell c at level idxLevel do
        M2M(c, c.child);
    end
end
```

Algorithm 2: Traditional M2M

```
Data: none
Result: none
for idxLevel ← $Height - 2$ to 1 do
    if $cells[0]$ not in my working interval then
        isend($cells[0].child$);
        hasSend ← true;
    if $cells[end]$ in another working interval then
        irecv(recvBuffer);
        hasRecv ← true;
    forall the Cell c at level idxLevel in working interval do
        M2M(c, c.child);
    end
    Wait send and recv if needed;
    if hasRecv is true then
        M2M($cells[end]$, recvBuffer);
end
```

Algorithm 3: Distributed M2M

In the oct-tree, a cell or a leaf only exists if it has some children or particles in. When the processus receive some cells, it need to know their positions in the tree, because maybe one of the cells has not be sent since it didn't exist.

The first thing to read from the buffer received is the heading, which is a bit vector of length 8 (practically a char), indexing each cells send.

Example :

Header	Data	...	Data
00001011	Data of cell 5	Data of cell 7	Data of cell 8

Modified M2M

The algorithm may not be efficient for special cases. Since the communications do not progress (even in asynchrone way) while computing the M2M, the algorithm has been modified, in order to set one of the OMP thread to the communications.

```
Data: none
Result: none
for $idxLevel \leftarrow Height - 2$ to $1$ do
    // pragma omp single
    begin To be done by one thread only
        if $cells[0]$ not in my working interval then
            isend($cells[0].child$);
            hasSend $\leftarrow$ true;
        if $cells[end]$ in another working interval then
            irecv(recvBuffer);
            hasRecv $\leftarrow$ true;
        Wait send and recv if needed;
        if hasRecv is true then
            M2M($cells[end]$, recvBuffer);
    end
    // pragma omp for
    begin To be done by all the other threads
        forall the Cell c at level idxLevel in working interval do
            M2M(c, c.child);
        end
    end
end
```

Algorithm 4: Distributed M2M

L2L

The L2L operator is very similar to the M2M. It is just the contrary, a result hosted by only one node needs to be shared with every others nodes that are responsible of at least one child of this node.

The L2L operator fill child local array from parent local array, so there is no need to precise wich cell is send, since it's the parent cell that is send. Consequently, there is no need for a heading.

```
Data: none
Result: none
for idxLevel ← 2 to $Height - 2$ do
    if $cells$[0] not in my working interval then
        irecv($cells$[0]);
        hasRecv ← true;
    if $cells$[end] in another working interval then
        isend($cells$[end]);
        hasSend ← true;
    forall the Cell c at level idxLevel in working interval do
        M2M(c, c.child);
    end
    Wait send and recv if needed;
    if hasRecv is true then
        M2M($cells$[0], $cells$[0].child);
end
```

Algorithm 5: Distributed L2L

Complex operators: P2P, M2L

These two operators are more complex than the ones presented in the previous chapter. In fact, it is very difficult to predict the communication between nodes. Each step requires pre-processing to know what are the potential communications and a gather to inform other about the needs.

P2P

To compute the P2P a leaf need to know all its direct neighbors. Even if the Morton indexing maximizes the locality, the neighbors of a leaf can be on any node. Also, the tree used in our library is an indirection tree. It means that only the leaves that contain particles are created.

That is the reason why when we know that a leaf needs another one on a different node, this other node may not realize this relation if this neighbor leaf do not exist on its own tree.

At the contrary, if this neighbor leaf exists then the node wills require the first leaf to compute the P2P too. In our current version we are first processing each potential needs to know the communication we should need. Then the nodes do an all gather to inform each other how many communication they are going to send. Finally they send and receive data in an asynchronous way and cover it by the P2P they can do.

```
Data: none
Result: none
forall the Leaf lf do
    neighborsIndexes ← $lf.potentialNeighbors()$;
    forall the index in neighborsIndexes do
        if index belong to another proc then
            isend(lf);
            Mark lf as a leaf that is linked to another proc;
    end
end
all gather how many particles to send to who;
prepare the buffer to receive data;
forall the Leaf lf do
    if lf is not linked to another proc then
        neighbors ← $tree.getNeighbors(lf)$;
        P2P(lf, neighbors);
end
while We do not have receive/send everything do
    Wait some send and recv;
    Put received particles in a fake tree;
end
forall the Leaf lf do
    if lf is linked to another proc then
        neighbors ← $tree.getNeighbors(lf)$;
        otherNeighbors ← $fakeTree.getNeighbors(lf)$;
        P2P(lf, neighbors + otherNeighbors);
end
```

Algorithm 6: Distributed P2P

Shared Memory Version

The P2P algorithm is computed once for each pair of leafs belonging to the same proc. This means that when a proc will compute the force on the particles of leaf 1 due to the particles of leaf 2, both leafs 1 and 2 will be updated.

This way of compute the interaction is faster, but leads to concurrency problems.

M2L

The M2L operator is relatively similar to the P2P. Hence P2P is done at the leaves level, M2L is done on several levels from $Height - 2$ to 2. At each level, a node needs to have access to all the distant neighbors of the cells it is the proprietary and those ones can be hosted by any other node. Anyway, each node can compute a part of the M2L with the data it has.

Original Algorithm

The algorithm can be viewed as several tasks:

1. Compute to know what data has to be sent
2. All gather to know what data has to be received
3. Do all the computation we can without the data from other nodes
4. Wait *send/receive*
5. Compute M2L with the data we received

```
Data: none
Result: none
forall the Level idxLevel from 2 to Height - 2 do
    forall the Cell c at level idxLevel do
        neighborsIndexes ← c.potentialDistantNeighbors();
        forall the index in neighborsIndexes do
            if index belong to another proc then
                isend(c);
                Mark c as a cell that is linked to another proc;
        end
    end
end
Normal M2L;
Wait send and recv if needed;
forall the Cell c received do
    lightOctree.insert(c);
end
forall the Level idxLeve from 2 to Height - 1 do
    forall the Cell c at level idxLevel that are marked do
        neighborsIndexes ← c.potentialDistantNeighbors();
        neighbors ← lightOctree.get(neighborsIndexes);
        M2L( c, neighbors);
    end
end
```

Algorithm 7: Distributed M2L

Algorithm Modified

The idea in the following version is to cover the communications between process with the work (M2L Self) that can be done without anything from outside.

```
Data: none
Result: none
begin To be done by one thread only
    forall the Level idxLevel from 2 to Height - 2 do
        forall the Cell c at level idxLevel do
            neighborsIndexes ← $c.potentialDistantNeighbors()$;
            forall the index in neighborsIndexes do
                if index belong to another proc then
                    isend(c);
                    Mark c as a cell that is linked to another proc;
            end
        end
    end
    Wait send and recv if needed;
end
begin To be done by everybody else
    Normal M2L;
end
forall the Cell c received do
    $lightOctree.insert(c)$;
end
forall the Level idxLeve from 2 to Height - 1 do
    forall the Cell c at level idxLevel that are marked do
        neighborsIndexes ← $c.potentialDistantNeighbors()$;
        neighbors ← lightOctree.get(neighborsIndexes);
        M2L( c, neighbors);
    end
end
```

Algorithm 8: Distributed M2L 2

Cheat sheet about using EZtrace with ViTE on ScalFMM

In this appendix, one can find usefull information about using EZtrace on ScalFMM, and visualisation with ViTE.

EZtrace

EZTrace is a tool that aims at generating automatically execution trace from HPC (High Performance Computing) programs.

It does not need any source instrumentation. Usefull variables :

- EZTRACE_FLUSH : set the value to 1 in order to flush the event buffer to the disk in case of uge amouts of datas.
- EZTRACE_TRACE : choice of the type of event one wants to have. Example : EZTRACE_TRACE="mpi stdio omp memory". Remark : Mpi do a lot of call to pthread, so I suggest to not trace pthread events in order to visualize the results.
- EZTRACE_TRACE_DIR : path to a directory in wich eztrace will store trace for each MPI Proc. (Set to /lustre/username/smt to avoid overhead)

Once the traces are generated, one need to convert them, in order to visualize its.

ViTE

ViTE is a high memory consumption software, so in order to use it, avoid tracing pthread for example.

One can zoom in and out in the gant chart.

Plugin : One can get a histogram of each proc display the percentage of time spend in different section.

- Got to Preferences → Plugin.

Sorting the gant charts : Sometimes the process are badly sorted (like 0,1,10,11,2,3,4,5,6,7,8,9). It's possible to sort them with the mouse, or with editing an xml file :

- Got to Preferences → Node Selection and then Sort, or export/load xml file .

Bibliography

[1] Ananth Grama, George Karypis, Vipin Kumar, Anshul Gupta, *Introduction to Parallel Computing*. Addison Wesley, Massachusetts, 2nd Edition, 2003.

[2] I. Kabadshow, H. Dachsel, *Passing The Three Trillion Particle Limit With An Error-Controlled Fast Multipole Method*. 2011.