ScalFmm - Parallel Algorithms (Draft)

Berenger Bramas

August 11, 2011

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.

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

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

The MPI I/O functions use a view model to manage data access. We first construct a view using the function MPI_File_set_view and then read the data with MPI_File_read as described in the fallowing $C++$ code.

```
// From FMpiFmaLoader
particles = new FReal[bufsize];
MPI_File_set_view(file, headDataOffSet + startPart * 4 * sizeof(FReal),
        MPI_FLOAT, MPI_FLOAT, const_cast<char*>("native"), MPI_INFO_NULL);
MPI_File_read(file, particles, 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.

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 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

```
1 myFirstIndex ← particles[0].index;
2 allFirstIndexes ← MortonIndex[nbprocs];
3 allGather(myFirstIndex, allFirstIndexes);
4 pivot ← Sum(allFirstIndexes) / nbprocs;
```

Algorithm 1: Choosing the QS pivot

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. Our idea is to use a two passes algorithm describes as a sand settling:

1. We can see each node as a heap of sand. This heap represents the leaves of the octree. Some nodes have lot of leaves and then are a big heap. On the contrary, some are small heaps because composed by a few leaves. Starting from the extremities, each node can know the sand there is on its left and on its right.

2. Because each node knows the total among of sand in the system which is the sum of the sand there is on each of its sides plus its own sand it can compute a balancing calculus. What should happen if we put a heavy plank above all our sand heaps? Well the system should balance until all heaps have the same size. The same happens here, each node can know what to do to balance the system. Each node communicates only with its two neighbors by sending or receiving entire leaves.

At the end of the algorithm our system is completely balanced with the same number of leaves on each process.

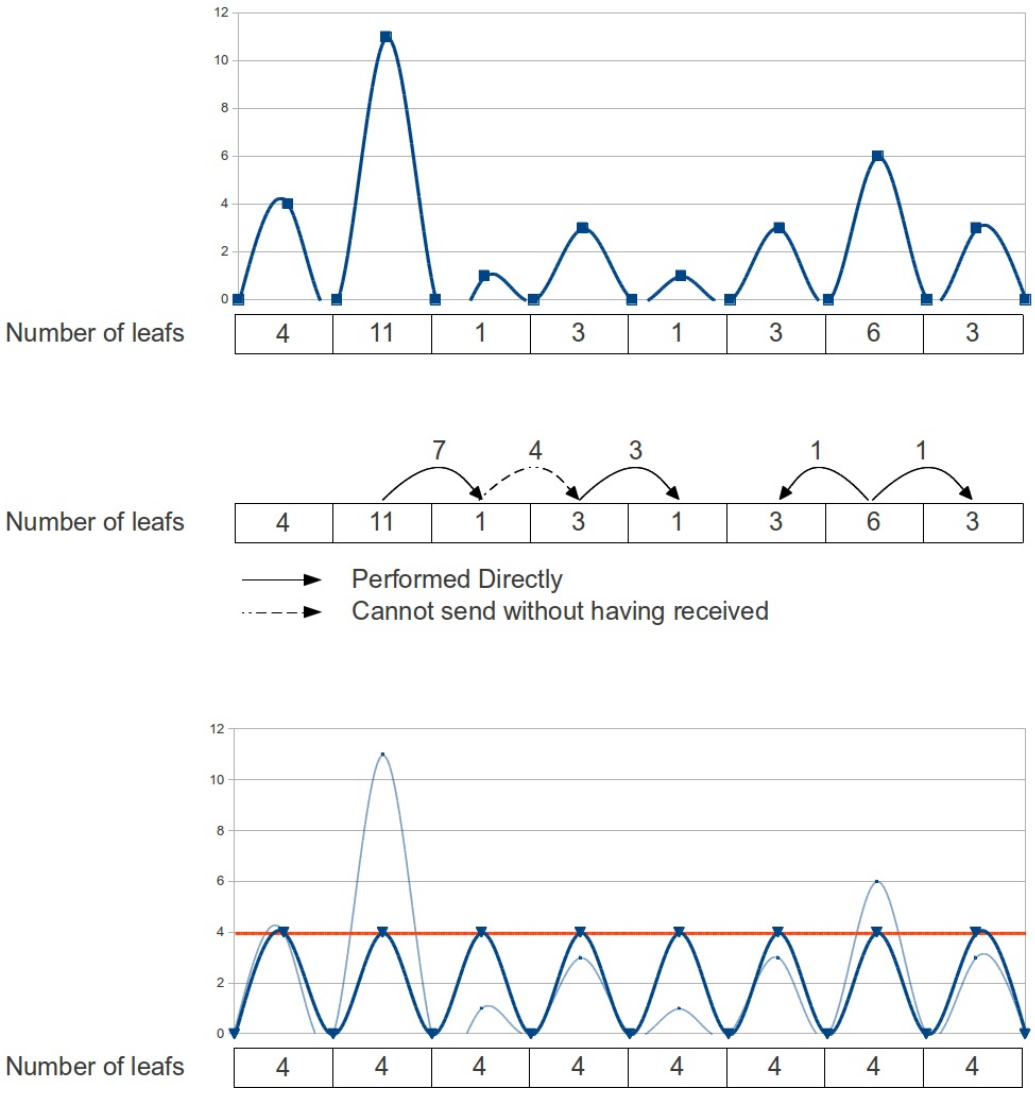

Figure 1: Sand Settling Example

Simple operators: P2M, M2M, L2L

We 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.

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.

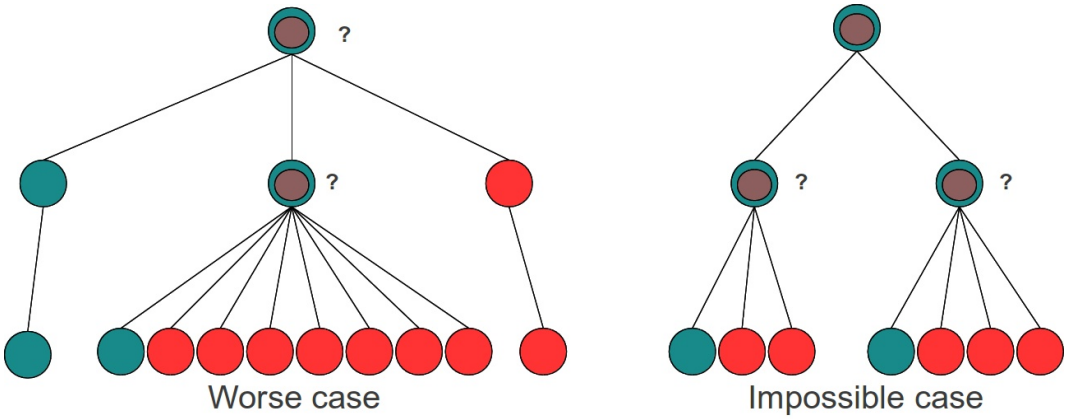

Figure 2: Potential Conflicts

Data: none
Result: none

```
1  for idxLevel ← Height − 2 to 1 do
2  |  forall Cell c at level idxLevel do
3  |  |  M2M(c, c.child);
4  |  end
5  end
```

Algorithm 2: Traditional M2M

Data: none
Result: none

```
1  for idxLevel ← Height − 2 to 1 do
2  |  if cells[0] not in my working interval then
3  |  |  isend(cells[0].child);
4  |  |  hasSend ← true;
5  |  if cells[end] in another working interval then
6  |  |  irecv(recvBuffer);
7  |  |  hasRecv ← true;
8  |  forall Cell c at level idxLevel in working interval do
9  |  |  M2M(c, c.child);
10 |  end
11 |  Wait send and recv if needed;
12 |  if hasRecv is true then
13 |  |  M2M(cells[end], recvBuffer);
14 |
15 end
```

Algorithm 3: 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.

```
   Data: none
   Result: none
 1 for idxLevel ← 2 to Height − 2 do
 2 |   if cells[0] not in my working interval then
 3 |   |   irecv(cells[0]);
 4 |   |   hasRecv ← true;
 5 |   if cells[end] in another working interval then
 6 |   |   isend(cells[end]);
 7 |   |   hasSend ← true;
 8 |   forall Cell c at level idxLevel in working interval do
 9 |   |   M2M(c, c.child);
10 |   end
11 |   Wait send and recv if needed;
12 |   if hasRecv is true then
13 |   |   M2M(cells[0], cells[0].child);
14 |
15 end
```

Algorithm 4: 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 leaf that contains particles is 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
 1 forall Leaf lf do
 2 |   neighborsIndexes ← lf.potentialNeighbors();
 3 |   forall index in neighborsIndexes do
 4 |   |   if index belong to another proc then
 5 |   |   |   isend(lf);
 6 |   |   |   Mark lf as a leaf that is linked to another proc;
 7 |   |
 8 |   end
 9 end
10 all gather how many particles to send to who;
11 prepare the buffer to receive data;
12 forall Leaf lf do
13 |   if lf is not linked to another proc then
14 |   |   neighbors ← tree.getNeighbors(lf);
15 |   |   P2P(lf, neighbors);
16 |
17 end
18 Wait send and recv if needed;
19 Put received particles in a fake tree;
20 forall Leaf lf do
21 |   if lf is linked to another proc then
22 |   |   neighbors ← tree.getNeighbors(lf);
23 |   |   otherNeighbors ← fakeTree.getNeighbors(lf);
24 |   |   P2P(lf, neighbors + otherNeighbors);
25 |
26 end
```

Algorithm 5: Distributed P2P

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. 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
 1 forall Level idxLeve from 2 to Height - 2 do
 2 |   forall Cell c at level idxLevel do
 3 |   |   neighborsIndexes ← c.potentialDistantNeighbors();
 4 |   |   forall index in neighborsIndexes do
 5 |   |   |   if index belong to another proc then
 6 |   |   |   |   isend(c);
 7 |   |   |   |   Mark c as a cell that is linked to another proc;
 8 |   |   |
 9 |   |   end
10 |   end
11 end
12 Normal M2L;
13 Wait send and recv if needed;
14 forall Cell c received do
15 |   lightOctree.insert(c);
16 end
17 forall Level idxLeve from 2 to Height - 2 do
18 |   forall Cell c at level idxLevel that are marked do
19 |   |   neighborsIndexes ← c.potentialDistantNeighbors();
20 |   |   neighbors ← lightOctree.get(neighborsIndexes);
21 |   |   M2L( c, neighbors);
22 |   end
23 end
```

Algorithm 6: Distributed M2L

Bibliography

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