TACC Technical Report IMP-09

Tree codes in the Integrative Model

Victor Eijkhout*

May 2, 2018

* `eijkhout@tacc.utexas.edu`, Texas Advanced Computing Center, The University of Texas at Austin

Abstract

Many practical algorithms have tree-structured calculations. In this note we look in particular at N-body algorithms such as the Barnes-Hut octtree method. Such calculations have many aspects that are tricky to program in parallel. We show how the Integrative Model for Parallelism (IMP) makes their specification more transparent.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a 'communication avoiding' compiler in the Integrative Model (under construction)
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
IMP-19 Report on NSF EAGER 1451204.
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design, with John McCalpin (under construction)
IMP-25 Dense linear algebra in IMP (under construction)
IMP-26 Load balancing in IMP (under construction)
IMP-27 Data analytics in IMP (under construction)

1 Background

1.1 Barnes-Hut

Algorithms for the N-body problem need to compute in each time step the mutual interaction of each pair out of N particles, giving an $O(N^2)$ method.

for each particle i
  for each particle j
    let $\bar{r}_{ij}$ be the vector between i and j;
    then the force on i because of j is
    $f_{ij} = -\bar{r}_{ij}\frac{m_i m_j}{|r_{ij}|}$
    (where m_i, m_j are the masses or charges) and
    $f_{ji} = -f_{ij}$.

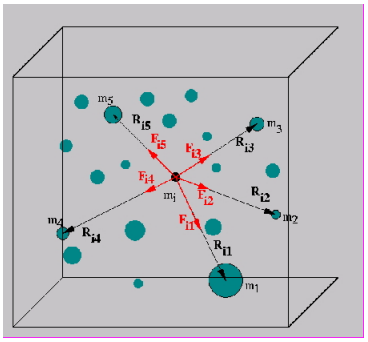


Figure 1: Summing of forces in an N-body problem

However, by suitable approximation of the 'far field' it becomes possible to have an $O(N \log N)$ or even an $O(N)$ algorithm, see the Barnes-Hut octree method [2] and the Greengard-Rokhlin fast multipole method [3].

The naive way of coding these algorithms uses a form where each particle needs to be able to read values of, in principle, every cell. This is easily implemented with shared memory or an emulation of it; however, it is difficult to express affinity this way, leading to potentially inefficient execution.

Pseudo-code would look like:

```
Procedure Quad_Tree_Build
    Quad_Tree = {empty}
    for j = 1 to N  // loop over all N particles
         Quad_Tree_Insert(j, root)  // insert particle j in QuadTree
    endfor
    Traverse the Quad_Tree eliminating empty leaves

Procedure Quad_Tree_Insert(j, n) // Try to insert particle j at node n in Quad_Tree
    if n an internal node              // n has 4 children
        determine which child c of node n contains particle j
        Quad_Tree_Insert(j, c)
   else if n contains 1 particle   //  n is a leaf
        add ns 4 children to the Quad_Tree
        move the particle already in n into the child containing it
        let c be the child of n containing j
        Quad_Tree_Insert(j, c)
    else                                          //  n empty
```

```
        store particle j in node n
    end
parallel over all particles p:
  cell-list = all top level cells
  sequential over c in cell-list:
    if c is far away, evaluate forces p<->c
    otherwise
      open c and add children cells to cell-list
// Compute the CM = Center of Mass and TM = Total Mass of all the particles
( TM, CM ) = Compute_Mass( root )

function ( TM, CM ) = Compute_Mass( n )
  if n contains 1 particle
     store (TM, CM) at n
     return (TM, CM)
  else       // post order traversal
             // process parent after all children
     for all children c(j) of n
           ( TM(j), CM(j) ) = Compute_Mass( c(j) )
     // total mass is the sum
     TM = sum over children j of n: TM(j)
     // center of mass is weighted sum
     CM = sum over children j of n: TM(j)*CM(j) / TM
     store ( TM, CM ) at n
     return ( TM, CM )
```

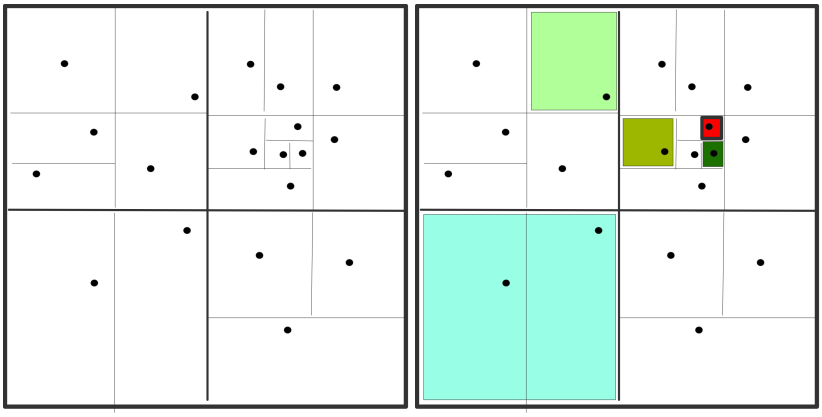


Figure 2: Quadrants at size/distance ratio

(This algorithm can also be implemented in distributed memory, with suitable *latency hiding* tech-

niques [7]. However, we prefer to reformulate it for distributed memory. Another computational discussion in [1].)

1.2 Fast Multipole

See the analysis in [8] for communication complexity, which takes into account redundant storage of higher tree levels.

2 Algebraic framework

In order for the N-body problem to be efficiently implementable in the Integrative Model for Parallelism (IMP) model we use a different formulation. In a way this is a different arrangement of the statements of the naive formulation. Such reformulation was already used to arrive at implementations in distributed memory with message passing; see [6].

Let us then consider the following form of the N-body algorithms (see [4, 1]):

1. The field due to cell i on level ℓ is given by

$$g(\ell,i) = \oplus_{j\in C_i^{(\ell)}} g(\ell+1,j) \tag{1}$$

   where $C_i^{(\ell)}$ denotes the set of children of cell i and $\oplus$ stands for a general combining operator, for instance computing a joint mass and center of mass;
2. The field felt by cell i on level ℓ is given by

$$f(\ell,i) = f(\ell-1,p(i)) + \sum_{j\in S_i^{(\ell)}} g(\ell,j) \tag{2}$$

   where $p(i)$ is the parent cell of i, and $S_i^{(\ell)}$ is the interaction region of i: those cells on the same level ('cousins') for which we sum the field.

This is a structural description; by suitable choice of kernel it can correspond to both the *Barnes-Hut* or *Fast Multipole Method (FMM)*. In FMM terms [5], step 1 is the 'upward pass', by M2M or multipole-to-multipole translation. The direct calculation of $g(\cdot,\cdot)$ on the finest level is done by Multipole Expansion Evaluation. The first term in step 2 is the 'downward pass' by L2L or local-to-local translation. The second term is the M2L or multipole-to-local translation.

3 Implementation

Initially, we consider the tree-structured computation of g.

Let's look at a full binary tree.

3.1 Per-level treatment

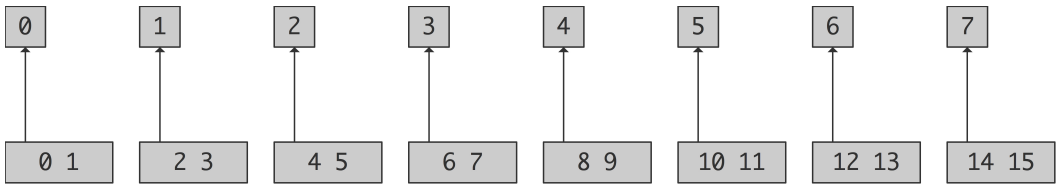
As long as we have more points (actually, blocks, but for now let's do the simple case) than processors, the gather is easy. Here we picture the gather to the level where the number of points equals the number of processors. This gives

$$\begin{cases} \gamma \equiv p \mapsto \{p\} \\ \alpha \equiv p \mapsto \{2p, 2p+1\} \end{cases}$$

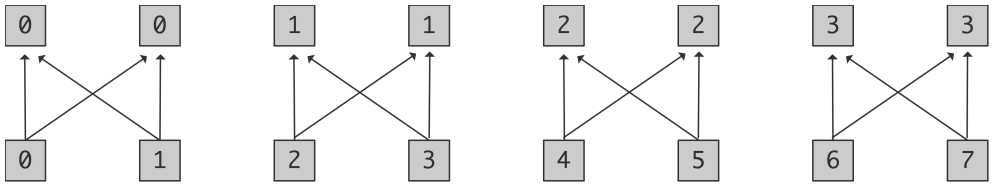
The communication here is completely processor-contained, and so

$$\beta = \alpha.$$

Picture:



As soon as the number of points is less than P, the situation gets more tricky. We could use a subset of processors, but let's store data redundantly. (For MPI that's not a bad idea, for OMP tasks we will have to explore reducing the number of processing elements on the higher levels.) Here's the picture:



We now have a slight problem that for the most populous levels we can simply use a disjoint block distribution, whereas for the top levels we need an entirely different mechanism for the redundant distribution, for instance specifying the map $p \mapsto u(p)$ explicitly. That's inelegant.

3.2 General treatment

The way out is to have further operators on distributions. For instance, the sequence of γ distributions is given as

$$\gamma^{(k-1)} = \gamma^{(k)}/2.$$

Somewhat remarkably, this formula gives the right distribution, both on disjointly and redundantly blocked levels. Mathematically, we describe this by saying that we have a fixed number P of processors, and levels

$$N = P2^k, P2^{k-1}, \ldots, P, P/2, \ldots, 1$$

where the level $N = P$ is the crossover between disjoint and redundant distributions. On each level, the distribution is

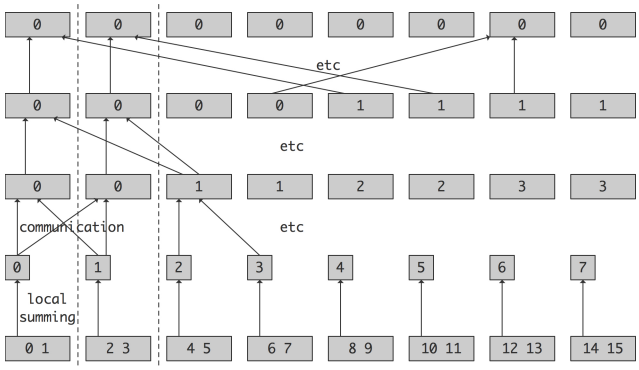
$$p \mapsto [\lceil p*(N/P) \rceil, \ldots, \lceil (p+1)*(N/P)-1 \rceil]$$

For the top level with $N = 1$ this gives $p \mapsto [0]$, meaning that each processor gets the root of the tree.

On each level, the β distribution is then given as an explicit function

$$\sigma(i) = \{2i, 2i+1\}.$$

Looking at a picture is instructive. We consider the case $N = 16, P = 8$, meaning that we start with two points per processor initially.



We note the following:

- At the finest levels, the summing is purely local; from the $N = P$ level up we have communication.
- The first time we have communication it is uniquely defined, on higher levels there are actually multiple ways to satisfy the data dependencies.
  The current implementation tries to satisfy data dependencies first locally, then cycling through the other processors. This has the effect we see in the $N = 4 \rightarrow N = 2$ transition, where two processors contain the same data, but all requests are satisfied by just one. This scheme can be advantageous, implemented as a multicast, or disadvantageous because of increased latency.

3.3 Software realization

The following shows how close the DSL implementation hews to the mathematical definition. First of all, the distributions are derived by subsequent divisions:

```
%% unittest_struct.cxx
auto div2 = std::shared_ptr<ioperator>( new ioperator("/2") );
auto mul2 = std::shared_ptr<ioperator>( new ioperator("x2") );
distribution *distributions[nlevels];
product_object *objects[nlevels];
for (int nlevel=0; nlevel<nlevels; nlevel++) {
  if (nlevel==0) {
    distributions[0]
        = new product_block_distribution(decomp,points_per_proc,-1);
  } else {
    distributions[nlevel] = distributions[nlevel-1]->operate(div2);
  }
  INFO( "level: " << nlevel << "; g=" << distributions[nlevel]->global_volume() );
  objects[nlevel] = new product_object(distributions[nlevel]);
```

The indirect functions of the kernel are then given by an explicit function pointer:

```
%% unittest_struct.cxx
product_kernel *kernels[nlevels-1];
for (int nlevel=0; nlevel<nlevels-1; nlevel++) {
  INFO( "level: " << nlevel );
  char name[20];
  sprintf(name,"gather-%d",nlevel);
  kernels[nlevel] = new product_kernel(objects[nlevel],objects[nlevel+1]);
  kernels[nlevel]->set_name( name );
  kernels[nlevel]->set_signature_function( &doubleinterval );
  kernels[nlevel]->set_localexecutefn( &scansum );
}
```

The most interesting part is the down tree.

```
%% mpi_ops.h
class mpi_sidewaysdown_kernel : virtual public mpi_kernel {
private:
  distribution *level_dist,*half_dist;
  std::shared_ptr<object> expanded,multiplied;
  mpi_kernel *expand,*multiply,*sum;
public:
  mpi_sidewaysdown_kernel( std::shared_ptr<object> top,std::shared_ptr<object> side,std::shared_ptr<ob
    : kernel(top,out),mpi_kernel(top,out),entity(entity_cookie::KERNEL) {
```

References

[1] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Pipelining the fast multipole method over a runtime system. Technical Report 7981, INRIA, 2012.

[2] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73:325, 1987.

[4] J. Katzenelson. Computational structure of the n-body problem. *SIAM Journal of Scientific and Statistical Computing*, 10:787–815, July 1989.

[5] J. Kurzak and B. M. Pettitt. Fast multipole methods for particle dynamics. *Molecular simulation*, 32:775–90, 2006. doi:10.1080/08927020600991161.

[6] John K. Salmon, Michael S. Warren, and Gregoire S. Winckelmans. Fast parallel tree codes for gravitational and fluid dynamical n-body problems. *Int. J. Supercomputer Appl*, 8:129–142, 1986.

[7] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 12–21, New York, NY, USA, 1993. ACM.

[8] Rio Yokota, George Turkiyyah, and David Keyes. Communication complexity of the fast multipole method and its algebraic variants. *CoRR*, abs/1406.1974, 2014.