A Massively Parallel Barnes-Hut Tree Code with Dual Tree Traversal

Benedikt STEINBUSCH [a,1], Marvin-Lucas HENKEL [b], Mathias WINKEL [a] and Paul GIBBON [a,c]

[a] *Institute for Advanced Simulation, Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany*
[b] *Faculty of Physics, University of Duisburg-Essen, 47048 Duisburg, Germany*
[c] *Centre for mathematical Plasma Astrophysics, Department of Mathematics, KU Leuven, Celestijnenlaan 200B, 3001 Heverlee, Belgium*

Abstract. Hierarchical methods like the Barnes-Hut (BH) tree code and the Fast Multipole Method (FMM) are important tools to decrease the algorithmic complexity of solving the N-body problem. We report on recent efforts to achieve massive scalability on IBM Blue Gene/Q and other highly concurrent supercomputers with our BH tree code PEPC by replacing a thread synchronization strategy based on traditional lock data structures with atomic operations. We also describe the integration of the Dual Tree Traversal a more recent algorithm that combines advantages of both BH tree code and FMM into PEPC. We explain how the scalability of this algorithm is influenced by the existing communication scheme and we propose a modification to achieve better distributed memory scalability.

Keywords. N-body problem, hierarchical methods, hybrid parallelism

Introduction

Solving the N-body problem is an important component of many applications in the field of computational science. A naive implementation can quickly become a bottleneck however, due to the $\mathcal{O}\left(N^2\right)$ complexity of calculating all pair-wise mutual interactions:

$$\Phi_i = \sum_{j \neq i} \phi(x_i, q_i, x_j, q_j) \quad i \in 1 \ldots N \tag{1}$$

While the computation of short-ranged interactions can usually be accelerated using e.g. spatial cutoff techniques, long-ranged interactions require a different approach. One possibility is to discretise the spatial domain in the form of a computational grid on which a partial differential equation for the interaction force field can be solved numerically (e.g. the particle-in-cell scheme). This solution strategy can readily benefit from

[1]Corresponding Author: Benedikt Steinbusch, Forschungszentrum Jülich GmbH, IAS - JSC, 52425 Jülich, Germany; E-mail: b.steinbusch@fz-juelich.de

the significant advances made in the field of numerical linear algebra to accelerate the computation. Grid-based techniques however favour certain forms of (typically closed) boundary conditions and require adaptive grid management in order to efficiently deal with strongly inhomogeneous densities.

Another possibility to solve the N-body problem efficiently is the family of mesh-free, hierarchical techniques which have first been proposed in the 1980s. Among the members of this family are the Barnes-Hut (BH) tree code [1] and the Fast Multipole Method (FMM) [2]. Common to these methods is a hierarchical decomposition of space into sub-regions containing fewer and fewer bodies.

The BH tree code computes interaction results for each individual body by selecting appropriate regions to interact with from the hierarchy via an acceptance criterion (AC) that can be based on both geometric and physical quantities (x and q in (1)). On average, the number of interaction partners for each body is typically $\mathcal{O}(\log N)$ and thus the total time complexity of the tree code algorithm is $\mathcal{O}(N \log N)$. BH tree codes to this day are the subject of active research [3,4].

The FMM makes more rigorous use of the hierarchical decomposition by also considering pair-wise interactions between regions and passing the results on to the multiple bodies contained therein. Interaction partners are selected based on an AC that is based primarily on the topology of the decomposition and that can be evaluated statically. The dual use of the decomposition allows the FMM to reach a time complexity of $\mathcal{O}(N)$.

Both methods usually apply a series expansion to the interaction law in order to retain the desired amount of accuracy when computing results of aggregate interactions in a single step. Mirroring the one sided use of the spatial decomposition, the BH tree code expands the interaction law in a single parameter (the source coordinate x_j), while the FMM applies expansions in two parameters (source coordinate x_j and destination coordinate x_i).

More recently a novel hybrid algorithm, the Dual Tree Traversal (DTT), has been proposed [5,6]. Like the FMM it maximises the benefits of the spatial decomposition and employs series expansion in both spatial parameters. The DTT combines this with the flexibility offered by the AC of the BH tree code. It has been shown empirically to also have a time complexity of $\mathcal{O}(N)$.

In this paper, we present our work on PEPC, the **P**retty **E**fficient **P**arallel **C**oulomb-solver. PEPC was conceived in 2003 as a distributed memory parallel BH tree code for electrostatic interactions. It has since evolved to include different laws of interaction and successfully makes use of a hybrid parallelisation scheme [7].

First we report on recent efforts to achieve massive scalability on IBM Blue Gene/Q and other highly concurrent supercomputers by replacing a thread synchronisation strategy based on traditional lock data structures with atomic operations. We show results for intra node scaling utilising all 64 hardware threads of the A2 compute chip and inter node scaling utilising all 458752 cores of the JUQUEEN supercomputer at JSC.

Then we describe the integration of the DTT algorithm into PEPC. We explain how the scalability of this algorithm is influenced by the existing communication scheme and we propose a modification to achieve better distributed memory scalability.

1. Massively Parallel Barnes-Hut Tree Code

1.1. The Barnes-Hut Algorithm

The Barnes-Hut algorithm proceeds in two phases. Phase 1 builds a tree decomposition of the source bodies $b_j, j \in 1, \ldots, N$. PEPC follows the Hashed Oct-Tree scheme proposed in [8], repeated here as algorithm 1. The algorithm rearranges the distribution of the bodies among parallel processes according to their spatial position making use of a space-filling curve. It builds a local part of the tree bottom-up on every process using only the information of the bodies assigned to that process. This local part spans from the leaves containing individual bodies to the "branch nodes". Branch nodes are those nodes that are farthest up the tree from the leaves yet still only contain bodies assigned to a single process. The set of all branch nodes of all processes contains all bodies. Next, all branch nodes are made available on every process and the global part of the tree is built bottom-up from all branch nodes on every process. Finally, all processes have a tree that contains the same global part (between root node and branch nodes) everywhere and a local part below the branch nodes with information about the local bodies. Figure 1 shows a simple distributed tree right after its construction as seen by one of the processes. In general, both global and local parts are deeper than one level, local parts end in more than one branch node and branch nodes are not all at the same level.

Algorithm 1 Phase 1 of the BH algorithm: distributed tree construction

```
1: function TREE(b)                                          ▷ Construct tree from bodies b
2:     calculate integer coordinates $k$ along space-filling curve
3:     sort bodies $b$ among processes according to $k$
4:     construct tree bottom-up until branch nodes
5:     distribute branch nodes
6:     finish construction bottom up from branch nodes to root
7:     return root
8: end function
```

Phase 2 of the algorithm (see algorithm 2) calculates the forces on all local bodies due to the ensemble of all bodies as described by the tree decomposition. Calculating the force on one particular body involves a top-down traversal of the tree (function FORCE)

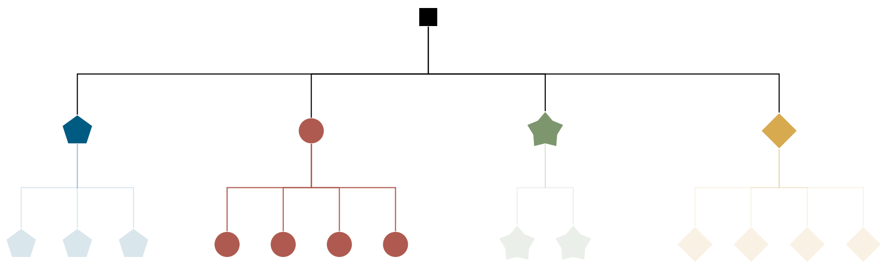

Figure 1. A distributed tree after tree construction as seen by process 1; information from different processes is distinguished by colour and shape (process 0: blue pentagons, 1: red circles, 2: green stars, 3: yellow diamonds). The global part of the tree is in black rectangles. Information that is not available locally on process 1 is half transparent.

Algorithm 2 Phase 2 of the BH algorithm: force computation

```
1:  function ForceAll(t)                          ▷ forces on all bodies b
2:      for $i \leftarrow 1, \ldots, N$ do
3:          $\Phi_i \leftarrow \textsc{Force}(root, b_i)$
4:      end for
5:      return $\Phi$
6:  end function
7:  function Force(t, b)                          ▷ force on body b due to tree t
8:      if IsLeaf(t) or AC(t, b) then
9:          return $\phi(t, b)$
10:     else
11:         return $\sum_{c \in t} \textsc{Force}(c, b)$
12:     end if
13: end function
```

starting at the root node. The force on a body due to a node in the tree (and all the bodies contained therein) can be calculated in a single application of ϕ (or its series expansion) if the node is found to be suitable – either by being a leaf or by fulfilling the acceptance criterion (AC). Otherwise the force is calculated as the sum of forces due to the children of the node. Applications of function FORCE for different bodies b_i are independent and can be performed in parallel by multiple processes and threads. The next section describes how this parallelism is exploited and how it interacts with the non-locality of certain parts of the tree.

1.2. A Hybrid Parallelisation for the Barnes-Hut Algorithm

PEPC uses a heterogeneous hybrid parallelisation for the BH force computation phase, i.e. it assigns structurally different tasks to different POSIX threads [7]. While most threads contribute to the computation of body-region and body-body interactions (worker threads), at least one thread on each distributed memory process is responsible for handling communication with other processes (communicator thread).

The loop that starts the tree traversal for different bodies on line 2 of algorithm 2 is parallelised among the worker threads. If the children of a node have to be visited by a traversal (line 11 in algorithm 2), but are not available locally, the worker thread enqueues a request for the children of that node to be processed by the local communicator thread and defer the remainder of the traversal until later, starting a new traversal for a different body in the meantime. The communicator thread asynchronously sends requests from the queue to other processes on behalf of the worker threads. It also receives and acts upon messages from other processes.

This latency-hiding scheme helps to achieve an overlap of computation and communication [8]. The dependency between the two kinds of threads necessitates an inter-thread synchronisation mechanism.

1.3. Concurrent Access to Shared Resources

The acquisition of bodies from the list by the worker threads (loop in line 2 of algorithm 2) is coordinated via a shared counter. Access to this shared resource from multi-

Algorithm 3 Concurrent access to a shared counter, synchronised via a lock

```
function AcquireBody(c)                ▷ acquire a body by incrementing a counter
    Lock(c)
    $i \leftarrow c$
    $c \leftarrow c + 1$
    Unlock(c)
    return i
end function
```

ple worker threads was originally synchronised using the read/write lock facilities of the POSIX threads API as described in algorithm 3.

While this solution scaled well to a moderate number of threads as found e.g. on IBM Blue Gene/P systems, it could not efficiently handle the 64 hardware threads that are available on the A2 compute chip of the Blue Gene/Q system. It is possible to keep the number of concurrent threads low by splitting the 16 cores available on each compute node among multiple distributed memory processes. In practice however, it is desirable to utilise all cores on a single compute node in a shared-memory parallel fashion since a larger number of processes leads to unnecessary fragmentation of the tree data structure which in turn leads to increased need for communication.

As an alternative to the read/write locks, we investigated atomic operations, which enable consistent (atomic) manipulation of primitive types by multiple concurrent threads. One such operation performs an addition on a variable and returns its value before or after the addition. We change the algorithm to acquire a body for force computation to manipulate the shared counter via atomic operations only and remove the lock as illustrated in algorithm 4.

Algorithm 4 Concurrent access to a shared counter, synchronised via atomic operations

```
function AcquireBody(c)                ▷ acquire a body by incrementing a counter
    atomic
        $i \leftarrow c$
        $c \leftarrow c + 1$
    end atomic
    return i
end function
```

Communication between worker threads and communicator threads is performed by enqueueing requests in a single-consumer multi-producer ring buffer. The buffer is filled with valid entries between a head and a tail pointer that are incremented using modulo arithmetic. These operations were previously also protected from data races using locks.

Algorithm 5 Atomic modulo arithmetic using *compare-and-swap*

```
function AtomicModIncrement(x, mod)
    repeat
        $y \leftarrow x$
    until CAS(x, y, (y + 1) % mod)
end function
```

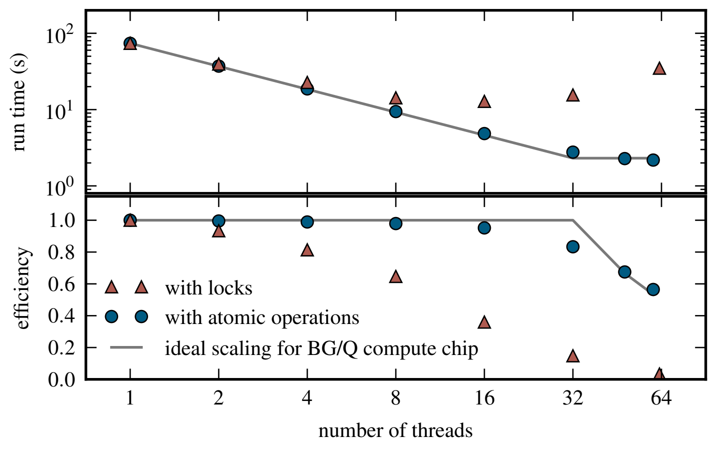

Figure 2. A strong scaling of PEPC on a single node of the IBM Blue Gene/Q installation JUQUEEN from 1 to 64 threads.

Since modulo arithmetic is not available as an atomic primitive itself, we implement it using another primitive, the *compare-and-swap* CAS(x, old, new). CAS atomically replaces the value of a variable x with a new value and returns true, if the variable is currently holding a certain old value, otherwise it leaves the value untouched, returning false. The atomic modular arithmetic operation is given in algorithm 5.

Figure 2 shows a strong scaling of PEPC utilising 1 to 64 threads. It compares two versions, one using read/write locks and one using atomic operations. Note that for more than 32 threads, the theoretically available SMP concurrency is saturated due to hardware restrictions.

With these changes in place, the worker threads and the communicator threads cooperate efficiently and allow PEPC to achieve high parallel efficiency on the whole of JUQUEEN, see figure 3. PEPC is thus also a member of the High-Q club established at the Jülich Supercomputing Centre in 2013 [9].

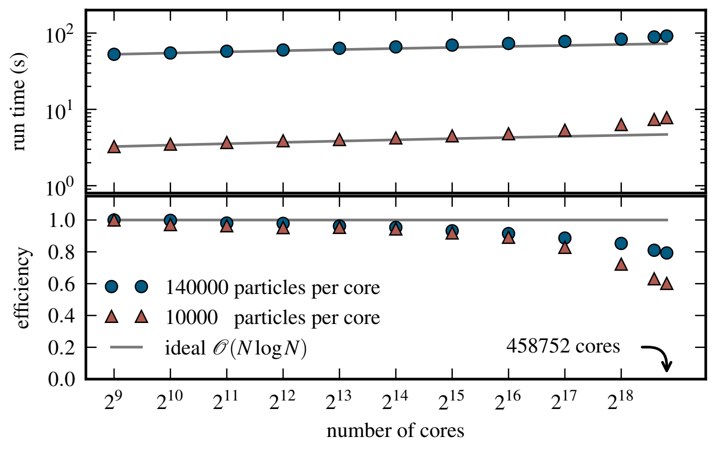

Figure 3. A weak scaling of PEPC on the IBM Blue Gene/Q installation JUQUEEN from a single node board to the whole machine.

2. Dual Tree Traversal

2.1. The Dual Tree Traversal Algorithm

Despite the remarkable scalability achieved with PEPC demonstrated in figure 3, the fact remains that the algorithm is fundamentally $\mathcal{O}(N \log N)$: many more operations are being performed than would be computed by an equivalent fast multipole method for the same precision in the force calculation. To improve the original BH tree code algorithm to $\mathcal{O}(N)$ therefore, we extend it to use the DTT algorithm as described in [6]. Due to the similarities between the two algorithms, this enhancement does not require a full rewrite of the application, but can be accomplished with a few self-contained changes that take advantage of the existing parallel framework. These changes are:

1. Introducing additional objects and operators in the mathematical back-ends.
2. Constructing the DTT algorithm on top of the existing implementation of the tree data structure and the newly added mathematical tools.

In particular, the tree construction phase as shown for the BH algorithm (algorithm 1) can be re-used almost completely unchanged, save for the initialisation of the aforementioned additional mathematical objects. Phase 2 of the BH algorithm is replaced

Algorithm 6 Phase 2 of Dual Tree Traversal: Sow procedure

```
 1: procedure SOW(s, t)
 2:     if (ISLEAF(s) and ISLEAF(t)) or AC(s, t) then
 3:         $\Phi_t \leftarrow \Phi_t + \phi(s,t)$
 4:     else if ISLEAF(t) then
 5:         SPLITSOURCE(s, t)
 6:     else if ISLEAF(s) then
 7:         SPLITTARGET(s, t)
 8:     else if DIAMETER(s) > DIAMETER(t) then
 9:         SPLITSOURCE(s, t)
10:     else
11:         SPLITTARGET(s, t)
12:     end if
13: end procedure
14: procedure SPLITSOURCE(s, t)
15:     for all children c of source do
16:         SOW(c, t)
17:     end for
18: end procedure
19: procedure SPLITTARGET(s, t)
20:     for all children c of target do
21:         task
22:             SOW(s, c)
23:         end task
24:     end for
25:     taskwait
26: end procedure
```

by two distinct phases in the DTT algorithm. The first, called Sow, is given in algorithm 6.

The Sow phase traverses two trees which may be identical in a mutually recursive fashion, using them for information about the source and target body distribution. It calculates interactions between pairs of nodes that either fulfil the acceptance criterion AC or cannot be split further. Otherwise it tries to split the bigger of the two nodes under consideration (keeping the potential interaction partners at a similar level) or split the node that is not a leaf. Figure 4 illustrates the evolution of this algorithm. Parallelism is exploited during this phase by performing traversals for distinct target nodes in separate tasks using OpenMP (lines 21 and 25 in algorithm 6).

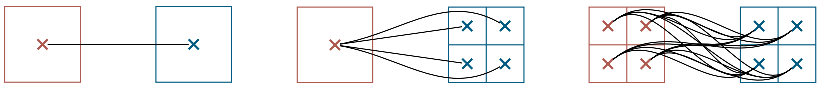

Figure 4. Evolution of the Sow phase of the DTT algorithm: starting at the root nodes of both source and target trees (left) the algorithm first splits the target tree root (middle) and then the source tree root (right)

The last phase of the DTT algorithm, called Reap, translates the results of interactions between pairs of nodes to results for the bodies contained therein. It operates in a top-down manner on the local part of the tree only.

Algorithm 7 Phase 2 of Dual Tree Traversal: Reap procedure

```
 1: procedure Reap(n)
 2:     if IsLeaf(n) then
 3:         $\Phi_{\text{body}} \leftarrow \Phi_n$
 4:     else
 5:         for all children c of n do
 6:             $\Phi_c \leftarrow \Phi_n$
 7:             Reap(c)
 8:         end for
 9:     end if
10: end procedure
```

2.2. *Request/Response Communication in the DTT Algorithm*

The DTT algorithm re-uses the request/response communication scheme presented in the BH section of this paper. However, while the DTT works as intended in combination with this scheme, it does not scale as well to large distributed memory partitions. This deficiency stems from the fact that the latency-hiding scheme employed in the BH algorithm does not easily translate to the DTT. Specifically, in the BH algorithm, a series of independent traversals of the input tree data structure is performed for every element of the output data structure, a flat list of bodies. Each one of these traversals can be suspended and another one can be performed in its place while waiting for remote information. In the DTT algorithm, both input and output are hierarchical data structures (the spatial tree decomposition) which are traversed in a mutually recursive manner. If the traversal tries to split a source node in line 15 of algorithm 6 and the children are not available locally it has to stop and wait for the remote information to be fetched by the communicator thread. In the meantime, other tasks might progress, but, typically, the need for remote information first arises at the level of the branch nodes (all information above is available locally after tree construction). Consequently, the number of tasks spawned at that point is equal to the number of branch nodes owned by the encountering process, which is comparable to the amount of hardware threads offered by current processors, so there is no remaining work to replace the waiting traversal.

To remedy this problem we evaluate an eager request/response mechanism. Along with the request for the missing children of a source node, the requesting process sends the target node that is currently being considered as an interaction partner. This allows the communicator thread on the process receiving the request to perform a traversal of its local tree. Starting at the requested nodes, it evaluates the AC for those nodes and their descendants and the target node sent along with the request. All nodes up to and including those that finally meet the AC are collected and sent as part of the response. This procedure anticipates all future communications due to the target node and its descendants based on a worst-case estimate, i.e. the nodes sent along in the response fulfil the AC for the original target node (by definition) and all its descendants (under the reasonable assumption that $\mathrm{AC}(t,s) \Rightarrow \mathrm{AC}(c,s)\ \forall\, c \in t$).

In practice, however, the eager mechanism lead to a significant decrease in performance. The decrease in request/response round trips won by anticipating multiple levels of communication is not enough to offset the increased work load put on the communicator threads by the collecting tree traversal. Additionally, for larger problem sizes, the increased size of the response messages is often enough to overflow the buffers allocated for MPI buffered communication. We therefore revert to the unmodified request/response scheme.

Figure 5 shows a weak scaling of the DTT algorithm as implemented in PEPC on top of the unmodified request/response communication scheme. Compared to the scaling behaviour of the BH algorithm in figure 3, the DTT shows a more pronounced fall-off of efficiency for the medium sized problem. However, in terms of efficiency, it performs similarly for large problem sizes, where both algorithms achieve a parallel efficiency of about 0.8 on the whole machine. In terms of total wall time, the DTT outperforms the BH algorithm for large problem sizes, due to its $\mathcal{O}(N)$ complexity.

3. Conclusion

In this paper, we have discussed aspects of parallel implementations of both the Barnes-Hut algorithm and the Dual Tree Traversal algorithm.

We have demonstrated how atomic operations can be used to replace more general locking mechanisms in order to prevent multi-threading bottlenecks on processors offering a high degree of parallelism. Our implementation of the BH algorithm in PEPC based on a general request/response communication scheme scales well on distributed memory machines at medium and large problem sizes.

Furthermore, we have shown results for the scaling behaviour of our implementation of the DTT algorithm in PEPC utilizing up to nearly 0.5 million cores on JUQUEEN. It achieves a parallel efficiency similar to that of the BH algorithm at large problem sizes and offers a better time to solution due to the $\mathcal{O}(N)$ complexity. At medium and small

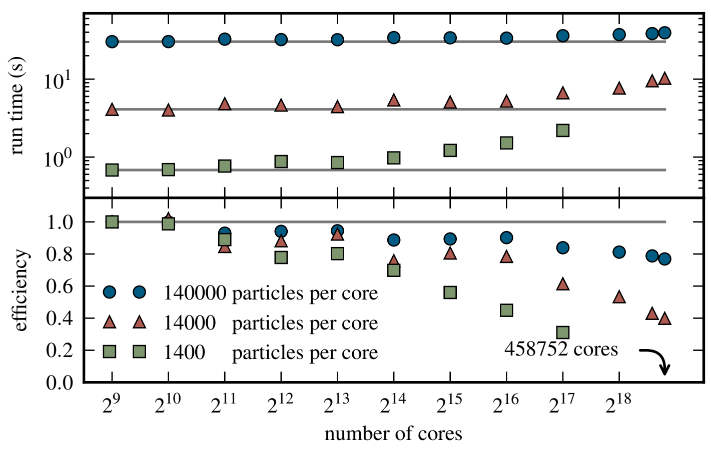

Figure 5. A weak scaling of the dual tree traversal algorithm in PEPC on the IBM Blue Gene/Q installation JUQUEEN from a single node board to the whole machine.

problem sizes the DTT does not yet reach the efficiency of the BH algorithm. We plan to investigate this issue further in the future.

Acknowledgements

Computing time for this work has been granted through the JARA-HPC partition. M. Henkel's contribution to this work has been funded by the Jülich Supercomputing Centre guest student programme.

References

[1] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[2] L Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.

[3] Michael S. Warren. 2HOT: An improved parallel hashed oct-tree N-body algorithm for cosmological simulation. *Scientific Programming*, 22:109–124, 2014.

[4] Jeroen Bedorf, Evghenii Gaburov, Michiko S. Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 Pflops on a Gravitational Tree-Code to Simulate the Milky Way Galaxy with 18600 GPUs. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 54–65, 2014.

[5] Michael S. Warren and John K. Salmon. A portable parallel particle program. *Computer Physics Communications*, 87(1-2):266–290, May 1995.

[6] Walter Dehnen. A Very Fast and Momentum-conserving Tree Code. *The Astrophysical Journal*, 536(1):L39–L42, June 2000.

[7] Mathias Winkel, Robert Speck, Helge Hübner, Lukas Arnold, Rolf Krause, and Paul Gibbon. A massively parallel, multi-disciplinary BarnesHut tree code for extreme-scale N-body simulations. *Computer Physics Communications*, 183(4):880–889, April 2012.

[8] M. S. Warren and J. K. Salmon. A parallel hashed Oct-Tree N-body algorithm. *Proceedings of the 1993 ACM/IEEE conference on Supercomputing - Supercomputing '93*, pages 12–21, 1993.

[9] Forschungszentrum Jülich – JSC – High-Q Club. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/_node.html`. [Online; accessed 31-July-2015].