**FORSCHUNGSZENTRUM JÜLICH GmbH**
**Zentralinstitut für Angewandte Mathematik**
**D-52425 Jülich, Tel. (02461) 61-6402**

Technical Report

# PEPC:
# Pretty Efficient Parallel Coulomb-solver

*Paul Gibbon*

FZJ-ZAM-IB-2003-05

May 2003

(last change: 01.05.2003)

# PEPC: Pretty Efficient Parallel Coulomb-solver

Paul Gibbon

*John von Neumann Institute for Computing, ZAM,*
*Forschungszentrum Jülich GmbH, D–52425 Jülich, Germany*

ABSTRACT

An parallel tree code for rapid computation of long-range Coulomb forces based on the Warren-Salmon 'Hashed Oct Tree' algorithm is described. Communication overhead is minimised by bundling multipole data for large groups of particles prior to shipment. Implementations on the Cray T3E and the IBM-p690 cluster show the expected $O(N \log N)$ scaling with particle number, as well as good scaling properties with number of processors.

## 1. Introduction

The $N$-body problem for systems dominated by long-range potentials is still, even in the era of Teraflop computing, a considerable algorithmic and computational challenge. The brute-force approach, in which all $N(N-1)$ interactions are computed directly, is both inefficient and impractical for many $N$-body systems such as plasmas, gravitational objects, or large molecules in ionized solution. This is particularly true when the global dynamic behaviour of the system is of primary interest, rather than the microscopic details of individual particle trajectories. For this class of problem there is no need to compute potentials and forces to higher accuracy than the error incurred in integrating the equations of motion, which can be anywhere between $10^{-4}$ for a high-order Runge-Kutta scheme, to around $1\%$ for the simple 2nd-order Leap-Frog method.

In the mid-1980s two new algorithms were independently proposed in which multipole expansions are substituted for distant groups of particles, leading to vastly improved algorithmic scalings of $O(N \log N)$ or $O(N)$. These techniques—the hierarchical Tree Code developed by Barnes & Hut [1] and the Fast Multipole Method (FMM) by Greengard and Rohklin [2]—have revolutionized long-range $N$-body simulation for scientists across a broad range of fields [3]. The two methods differ in essentially one aspect: whereas the Barnes-Hut (BH) algorithm replaces the particle-particle sum by lists of multipole expansions, each of length $O(\log N)$, the FMM makes additional use of high-order cluster-cluster interactions (Taylor expansions) to transfer multipole information between well-separated regions of space, resulting in a theoretical $O(N)$ overall effort. Despite these advances, the advent of massively parallel architectures in the 1990s has prompted a further challenge: can hierarchical algorithms be effectively implemented on a parallel machine?

Not surprisingly, this problem has attracted considerable attention because of the potentially rewarding prospect of billion-particle simulation which modern Teraflop machines can then feasibly offer. A parallel version of the original, non-adaptive 2D FMM was proposed by Greengard himself as early as 1990 [4]. This scheme, based on task-sharing in each of the separate near- and far-field stages of the FMM, works well on shared memory machines, but less efficiently on distributed memory systems. Nonetheless, these ideas inspired the first parallel FMM for MD simulation of macromolecules, developed and demonstrated by the Duke University group in the early 1990s [5]. A recent, optimized implementation by Dachsel [6] has achieved machine-accuracy potential summation for up to $10^9$ charges on an IBM-p690 SMP-cluster.

At first sight, the hierarchical data structure of BH tree codes would seem to rule out parallelism altogether, but it was soon realised that the construction of both the tree and particle interaction lists could at least be *vectorised* on a level-by-level basis [7, 8, 9], leaving a straightforward $N \times N_{list}$ force summation to contend with. However, this only works with shared memory. On distributed memory machines, the tree structure either has to be global to all processors—restricting the maximum simulation size—or somehow divided up equally among them. Either way, access to the *whole* tree will be needed at some stage in order to build an interaction list. Various parallel tree codes have been proposed and sucessfully implemented, including virtual shared-memory versions [10], and distributed memory schemes using geometrical domain decomposition methods [11, 12].

One of the problems common to any distributed parallel scheme is that searching for nonlocal nodes on remote processors requires the exchange of complicated sets of messages to ensure that the requested data is returned. Local tree data is typically accessed via an address pointer, which if shared amoung all processors, would lead to heavy duplication. This problem was recognised early by Salmon and Warren [13, 14], who practically reinvented the BH algorithm by scrapping pointers in favour of a set of universal binary keys to represent particle and tree-node coordinates alike. Given its key and owner, locating any node in the tree is reduced to an $O(1)$ operation. A further advantage is that sorting the keys yields a space-filling curve, providing a natural and efficient means of domain decomposition. Various derivatives of this scheme are possible, depending on the choice and ordering of the keys. Warren & Salmon originally used a simple Morton or $Z$-order; other authors have advocated Peano-Hilbert ordering because of its smoother connectivity and hence improved communication properties [15, 16, 17].

This report describes a fresh implementation of a parallel tree code—PEPC—designed primarily with modelling of nonlinear, complex plasma systems in mind. Needless to say, with appropriate choice of units and boundary conditions, the code can easily be adapted for any molecular dynamics problem dominated by Coulomb forces. For PEPC, we have opted for the orginal WS scheme, largely because of its simplicity and ease of implementation: nearly all of the bit-operations needed for key manipulation are available via single internal function calls. As we shall see in Section 3, this choice does not seem to harm the code's performance significantly when combined with an asynchronous tree traversal (see Section 2.5.

From a very early stage, this code has been integrated with the visualisation package VISIT [18] both to assist in visualising the tree structure and domain decomposition for geometrically complex systems, and to permit computational steering of plasma simulations running on the T3E and the new IBM-p690 cluster (JUPP). This is particularly useful when modelling large-scale, 3-dimensional laser-plasma interactions, for example, in verifying start-up parameters such as the initial alignment of laser and target, to monitor the progress of a lengthy run without disrupting its progress, or for performing trial simulations with reduced particle number as a prelude to full-blown production. Live demonstrations of this capability have already been made at a number of recent conferences [19] and will be reported in detail elsewhere [20].

# 2. The Hashed Oct Tree algorithm

## 2.1. Construction of particle keys and domain decomposition

As mentioned earlier, the strategy adopted here is based on the scheme of Warren & Salmon (1995), who use binary coordinate keys to map the 3-dimensional spatial structure onto a one-dimensional space-filling curve. The basic idea is to convert the coordinate triple of each particle into a single, unique 64-bit integer key. The keys do not *replace* the coordinates, but as we shall see, provide a natural and rapid means of sorting the particles and building up the tree structure around them.

In the present code, the keys are constructed from the binary interleave operation:

$$\texttt{key} = \texttt{placebit} + \sum_{\texttt{j=0}}^{\texttt{nbits}-1} 8^j(4 \times \texttt{bit}(i_z, j) + 2 \times \texttt{bit}(i_y, j) + \texttt{bit}(i_x, j))$$

The function `bit()` selects the $j$th bit of the integer coordinate component $(i_x, i_y, i_z)$, which are computed from:

$$i_x = x/s, \text{etc.},$$

where

$$s = L/2^{\texttt{nlevels}}$$

and $L$ is the simulation box length; `nlevels` the maximum refinement level. The latter obviously depends on the machine precision, and for a 64-bit machine, we can have 21 bits per coordinate (or `nlevels=20`) plus a place-holder bit:

$$\texttt{placebit} = 2^{63}.$$

This procedure yields a space-filling curve following the so-called Morton or Z-ordering, a 2-dimensional example of which is shown in Fig. 1 below.

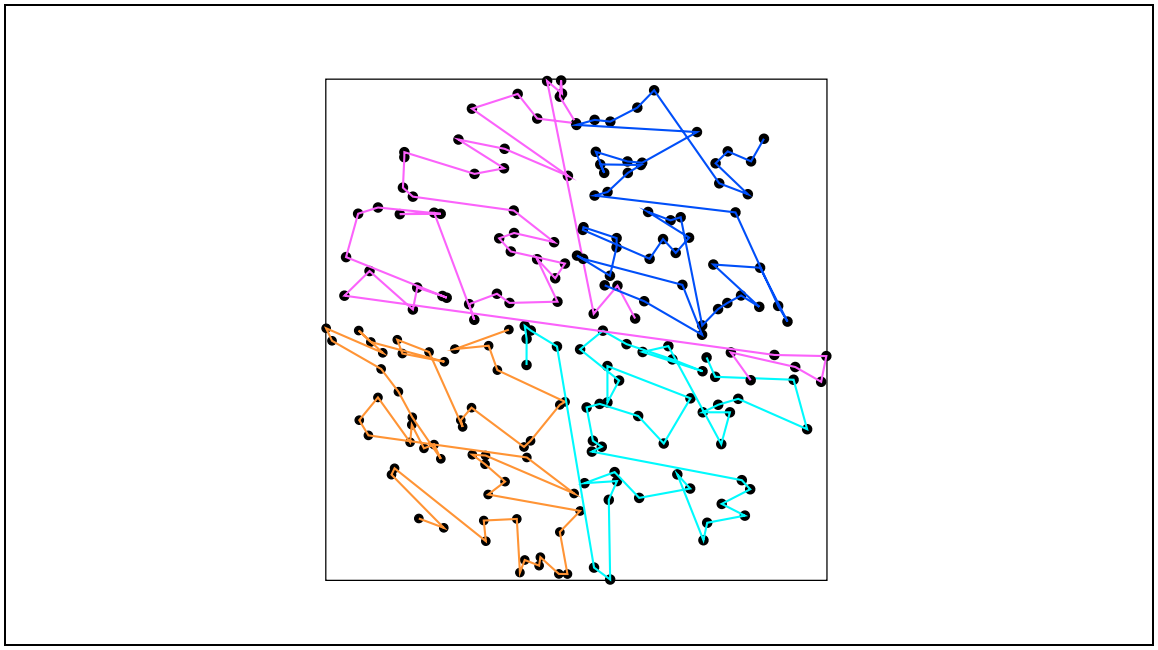

**Figure 1:** 2-dimensional Morton (Z-) ordering of 200 simulation particles, equally shared among 4 processor domains.

The simulation particles are then sorted according to the list of binary keys generated above. In an early version of the code, the key-sort was implemented sequentially, which requires all $N$ keys to be gathered onto the root processor. This would appear to ruin the overall parallelism, but in fact the $N \log N$ sorting effort which then results is a worst case—typically encountered at the beginning of the simulation when the particles have been randomly distributed. Once they are sorted, only a few particles cross processor domain-boundaries between timesteps, so that this routine actually takes a negligible amount of time. Nonetheless, the necessity of gathering all keys onto one processor limits the maximum simulation size to less than a million particles on the T3E.

The fully parallel sort currently implemented is an adaptation of the PSRS (parallel sort by regular sampling) algorithm originally proposed in Ref. [21]. Since the distribution of keys depends sensitively on the geometry of the system simulated—that is, whether the particles are initially arranged in a cube, sphere or more complex object—regular sampling tends to produce highly imbalanced

particle numbers across the processors. To compensate this effect, we instead use weighted sampling, which allows for the actual distribution of keys along the whole space-filling curve (Fig. 1).

A big advantage of binary coordinate ordering over standard addressing techniques in tree codes is that the hierarchical structure is recovered automatically. Keys of parent and neighbour cells can be obtained by simple bit operations, so that the average access-time for any particle or node in the tree is $O(1)$ instead of the usual $O(\log N)$. The obvious drawback is that the number of possible keys, $2^{63} \simeq 10^{19}$ on a 64-bit machine, vastly exceeds the memory available, typically $\sim 10^5 - 10^6$ locations per processor. This mismatch is resolved by using a hashing function to map the key onto a physical address in memory, for example:

$$\texttt{address} = \texttt{key AND } (2^h - 1), \tag{1}$$

where $h$ is the number of bits available for the address. This address then acts as a pointer to the particle or cell properties. In case two or more keys give the same address (collision), a linked-list is constructed to resolve it. Clearly a high occurence of collisions will ultimately degrade performance; however, as Warren & Salmon pointed out [14], the distribution of particles and nodes between many processors with their own address-spaces helps to reduce their number considerably.

Domain decomposition is then reduced to the almost trivial task of cutting out equal portions of the sorted list and allocating these to the processors. An decomposition example for 200 particles divided amoung 4 processors is also displayed by Fig. 1. Note that with this scheme, load balancing can be easily introduced by biasing the key-list segments according to the number of interactions computed for each particle in the force summation (performed later).

## 2.2. Construction of local trees

Once a set of particles has been allocated to a particular processor, and their associated properties (mass, charge, velocity etc.) have been fetched from their original location, one can immediately begin to construct the local trees. This can be done very efficiently because the particle keys implicitly contain the necessary information on all their ancestor nodes up to the root. As the 2D example in Fig. 2 shows, the parent of a particle or twig-node is simply found by a 3-bit shift operation:

```
parent_key = RIGHTSHIFT(key,3)
```
(2)

Likewise, if a node's children are numbered from 0 to 7 (in a 3D oct-tree), their keys can be obtained by the inverse operation:

```
child_key = LEFTSHIFT(key,3) OR child(0-7),
```
(3)

The local sorted list of particle keys would thus provide a natural starting point for determining their parent nodes if we knew how they were distributed. In a dynamic application we cannot assume anything about their distribution, however, so instead we start from the highest (coarsest) level and work down to the leaves. As in a sequential algorithm [3], all particles are initially attached to the root, a cube encompassing the whole simulation region. Next, the region is subdivided into 8 sub-boxes, and the particles re-attached accordingly. A sub-box containing exactly one particle is defined as a leaf; a box with 2 or more constitutes a twig and empty boxes are discarded. This procedure is continued at the next highest level until each particle sits in its own box. Each new leaf or twig node created this way is added to the local hash-table via the same hash function (1) as the particles. Collisions are again dealt with via a simple linked list. As yet, no attempt has been made to optimise the distribution of hash-table addresses (thereby saving memory): the sharing of keys across a number of processors keeps the collision count down to tolerable levels.

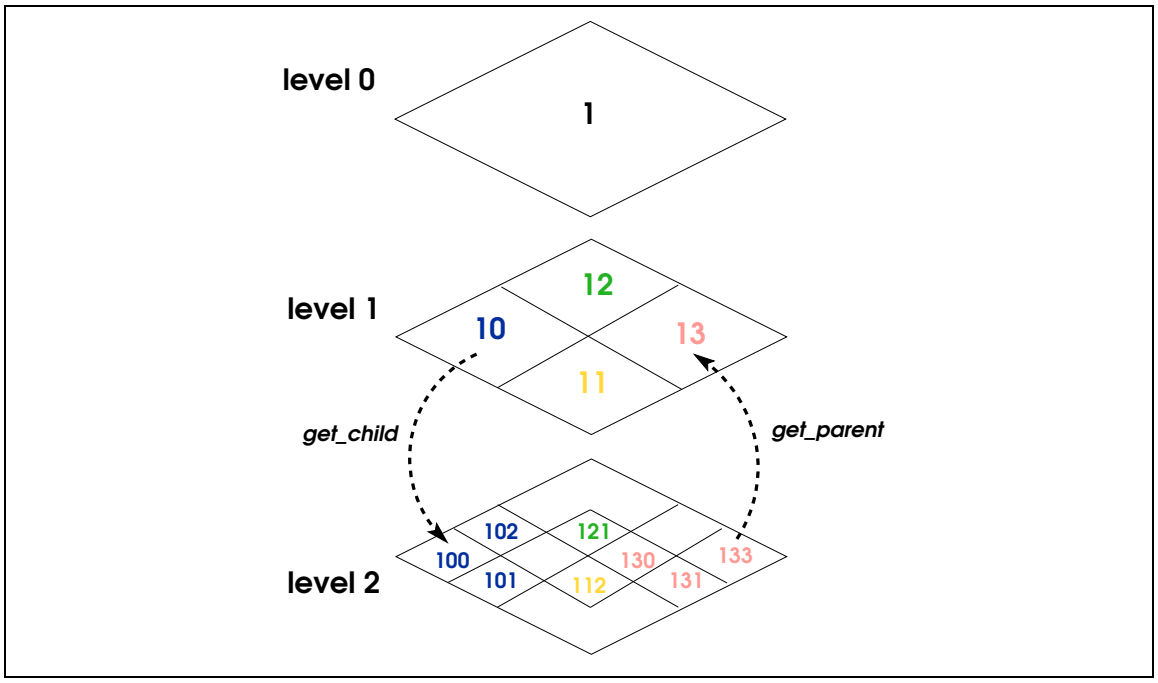


**Figure 2:** Level-by-level local tree construction.

## 2.3. Global branch nodes

At their coarsest level, the local trees will contain 'incomplete' twig nodes; that is, nodes which cross domain boundaries. Information from neighbouring domains is therefore needed to complete them. To facilitate the exchange of information (and later multipole moments) between processors, a set of local 'branch' nodes is defined first, comprising the minimum number of *complete* twig and leaf nodes covering the whole local domain—Fig.3.

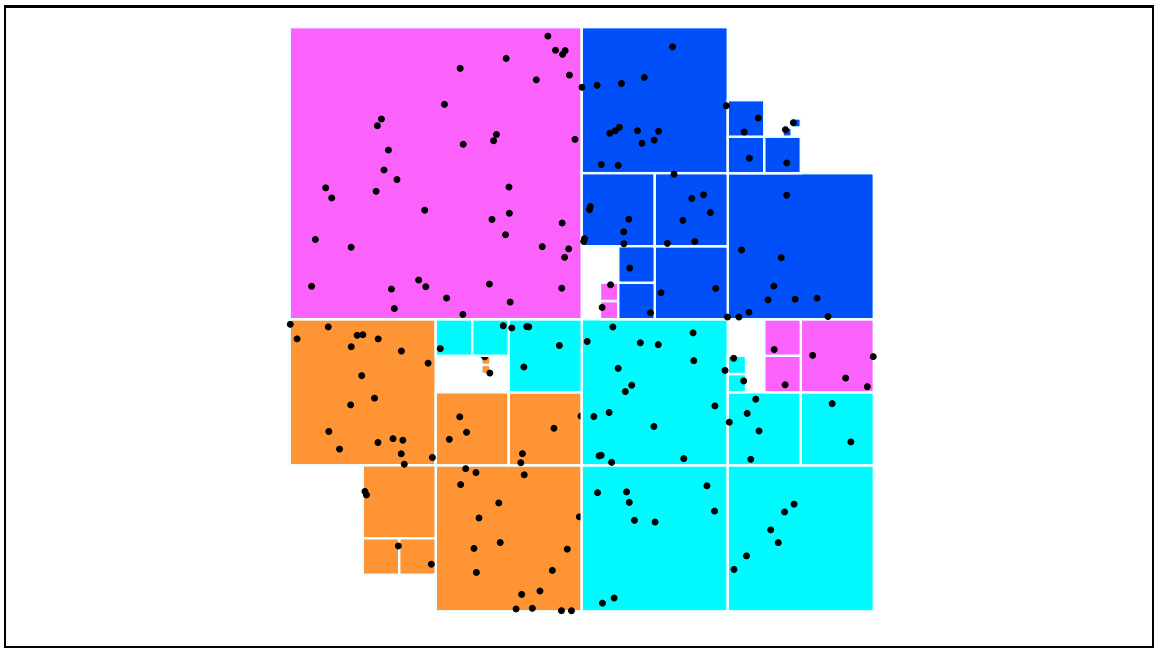

**Figure 3:** Branch nodes belonging to 4 processor domains.

This set of branch nodes is then broadcast to all other processors, so that each one subsequently knows where to find (or request) any missing non-local particle or tree node. For example, a branch's child nodes can immediately be found from a byte code stored with the hash-table entry, the first 8 bits of which declare which children exist at the next refinement level. Applying the operation (3) yields each (still non-local) child key. A branch's hash-entry will also contain the total

number of particles contained beneath it, so that the top level nodes above can now be filled in up to the root. At this point the local trees comprise 3 types of node: i) twig or leaf nodes covering the local domain, ii) branch nodes and iii) top level twig nodes, each covering the whole simulation region—Fig.4. Leaf node entries contain a pointer to the actual particle coordinates, charge and mass, as well as a globally unique label for tracking purposes. Twig nodes, including the special branch nodes, contain pointers to the multipole moments of their associated charge distributions, together with some flags indicating the status of non-local child nodes (in particular, whether a local copy already exists).

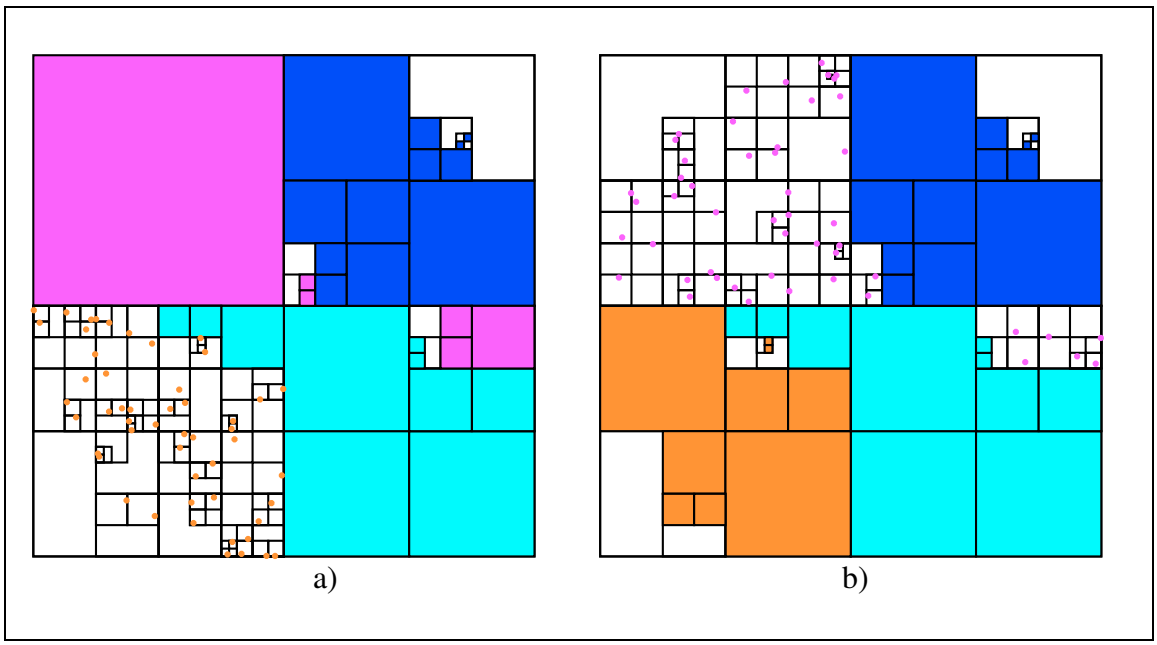


**Figure 4:** Local trees for a) processor 0 and b) processor 2 prior to tree-walk. The shaded boxes represent the branch nodes gathered from all remote processors.

## 2.4. Construction of multipole moments

Once the basic tree structure is in place, it is a straightforward matter to accumulate multipole moments for each node from the leaves up. Once again, this procedure is considerably simplified by sorting the keys for the twig-nodes contained within the list of local branch nodes. Twig nodes with the highest keys will, by definition, have the highest refinement levels:

$$\texttt{level} \;=\; \frac{\log(\texttt{key})}{\log 8} \tag{4}$$

This means that multipole moments at higher levels can be successively shifted up to their parent levels, for example:

$$\sum_i q_i x_i \quad \rightarrow \quad \sum_i q_i x_i - x_s \sum_i q_i$$

$$\sum_i q_i x_i^2 \quad \rightarrow \quad \sum_i q_i x_i^2 - 2x_s \sum_i q_i x_i + x_s^2 \sum_i q_i$$

$$\sum_i q_i x_i y_i \quad \rightarrow \quad \sum_i q_i x_i y_i - x_s \sum_i q_i y_i - y_s \sum_i q_i x_i + x_s y_s \sum_i q_i,$$

where $\boldsymbol{r}_s$ is the shift vector from the child nodes to their parent.

This procedure is continued by working through the sorted list of twigs in reverse order up to the local branch nodes, which then contain the *complete* multipole information for the local domain.

This information is then broadcast to all other processors, so that the remaining top-level nodes can be filled in using the above shifting rules. At the end of this procedure, each processor has the complete multipole expansion for the whole system contained in the root node.

## 2.5. Tree traversal: building interaction lists

By far the most important and algorithmically demanding part of this code is the tree-traversal, which combines a previous list-based vectorised algorithm [22] with the asynchronous scheme of Warren & Salmon [14] for requesting multipole information 'on the fly' from non-local processor domains. In the present scheme, rather than performing complete traversals for one particle at a time, as many 'simultaneous' traversals are made as possible, thus i) minimizing the duplication incurred when the same non-local multipole node is requested many times and ii) maximising the communication bandwidth by accumulating large numbers of nodes before shipment. In practice, this means creating interaction lists for batches of 200–1000 particles at a time, before actually computing their forces. The routine `TREE-WALK`, which finds the interaction list for each batch has the structure depicted in Fig. 2.5.

In the first half of this routine, traversals are made through the local tree using the familiar divide-and-conquer strategy common to sequential tree codes [22]. The multipole acceptance criterion (MAC) determines whether to accept or subdivide local nodes as usual, but also provides for a third possibility: the subdivision of a *non*-local node for which child data is not yet available. This is then placed on a special 'request list' to be processed in the 2nd half of the routine when all particles have completed their traversals as far as they can with the available node data. Each processor then compiles a lists of nodes it needs child data from, and sends them to the owners of the parent nodes. In the first pass, these will just be the branch nodes. On receipt of a request list, a processor packages and ships back the multipole data for the children. The use of non-blocking SENDS and RECEIVES for the multipole information allows some overlap of communication with the creation of new hash-table entries locally. At the end of all the traversals, each processor's local tree contains all the nodes required to compute the forces on its own particles. The nodes fetched during the traversals actually take up most of the space in the local hash-table, as Fig. 6 illustrates.

## 2.6. Force summation

Once an interaction list has been found for a particle, it is a straightforward task to compute its force and/or potential. Separation of the actual force sum from the tree traversal has the advantage that this floating-point-intensive routine can be hardware-optimised. Also, the physics and algorithm are kept naturally apart, so that additional forces (for example, short-range components or magnetic fields) and/or boundary conditions (for example, corrections from a periodic Ewald summation) can be added with relative ease. In the present implementation, forces are computed for each batch of interaction lists returned from the tree-walk routine. One subtlety which arises here is that even if overall load-balancing has been arranged during the domain decomposition, it is not necessarily guaranteed for each batch of particles (which may comprise only 1/100 of the total number on each processor). To redress this problem, the batch size $N_b$ for each processor is determined individually, so that the integral

$$\sum_{p=1}^{N_b} N_{\text{int}}(p) \tag{5}$$

is the same, and each processor computes the same number of interaction pairs during each pass.

```
do while (any defer list still > 0)

    do while (any particle not finished walk)
        find next node on particles' tree-walks

        if (MAC OK)
            put node on interaction list
            walk-key = next-node
        else if (MAC not OK for local node)
            subdivide: walk-key = first-child
        else if (MAC not OK for non-local node)
            walk-key = next-node
            put particle on 'defer' list
            put node on request list
        endif

        remove finished particles from walk list
    end do

    gather request lists for non-local nodes from all processors and discard duplicates

    do for all remote processors
        initiate receive buffer for incoming child data
        send off requests for remote child data
    end do

    do for all remote processors
        test for incoming request
        package and ship back child multipole data to processor that requested it
    end do

    do for all requests
        test if data has arrived for requested node
        if so, create new hash-table entries for each child
    end do

    copy particle defer lists to new walk lists for next pass through tree

end do
```

**Figure 5:** Parallel tree-walk algorithm for determining interaction lists for a batch of particles.

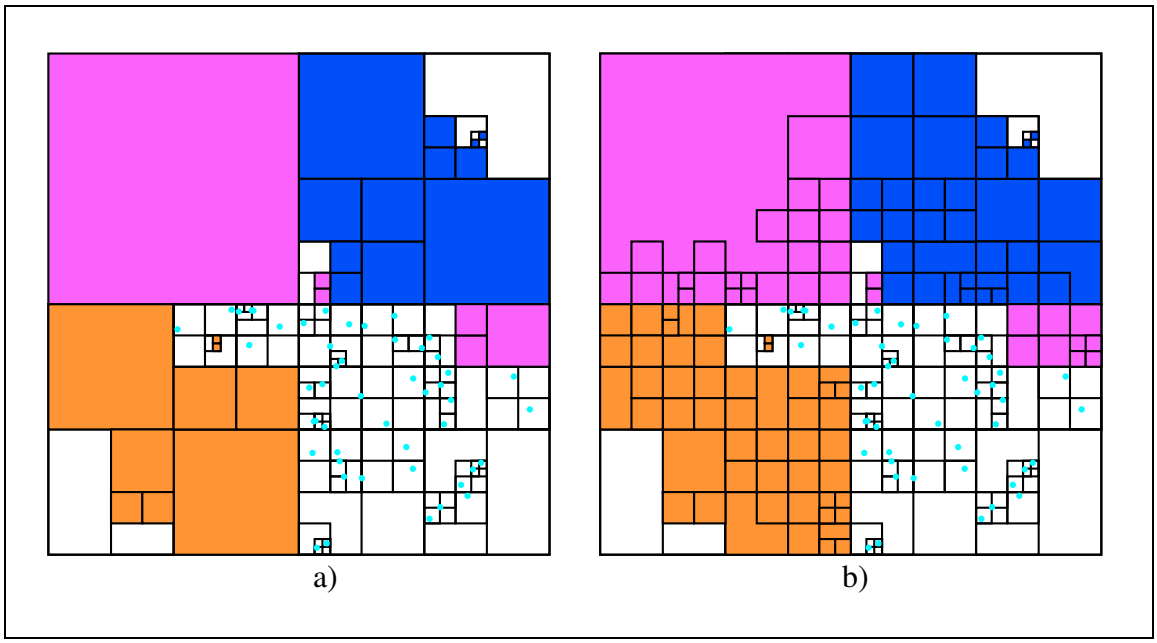


**Figure 6:** Tree for processor 1 (domain in bottom right quadrant): a) before and b) after traversals for all locally held particles.

| | |
|---|---|
| Initialise particle properties $\boldsymbol{r}_i, \boldsymbol{v}_i, q_i, m_i$ | $N/P$ |
| Key construction: $(x_i, y_i, z_i) \rightarrow k_i$ | $N/P$ |
| Sort keys: $k_1, k_2, ...k_N$ | $N/P \log N$ |
| Domain decomposition: $k_1, ..k_n; k_{n+1}, ..k_{2n}; ...; k_{N-n}...k_N$ | $N/P$ |
| Construct branch nodes | P $\log N/P$ |
| Fill in top level local tree nodes | $\log P$ |
| Build multipole moments | $\log N/P$ |
| Construct interaction lists (tree traversal) | $N/P \log N$ |
| Compute forces and potential | $N/P \log N$ |
| Update particle velocities and positions | $N/P$ |

**Table 1:** Algorithmic scaling of major routines in PEPC. The symbols $N$ and $P$ represent the total number of particles and processors respectively, and $n = N/P$.

## 3. Algorithm scaling and benchmarking

The overall algorithm is depicted together with the theoretical scaling of each major routine in Table 1. We see that in principle, all of the above routines can be performed in parallel, and thus require a computational effort $O(N/P)$, give or take a slowly varying logarithmic factor. Single-timestep benchmarks with this new code broadly confirm the theoretical scalings indicated in Table 1. As expected, most of the time is spent in the tree-traversal and force-summation routines: the total overhead incurred by the tree construction (which includes the steps 2.3, 2.3 and 2.4 described previously) is around 3%, although this figure excludes tree-nodes copied locally during the traversal—Table 2.

The increasing fraction of time spent in the tree traversal reflects the rising communication overhead with number of CPUs. This fraction can also vary depending on the geometry: high clustered systems will require deeper searches for interaction partners, and so longer traversals. Benchmarks for both T3E and JUPP (IBM-p690 SMP-cluster) machines are displayed in Fig. 7. This test used a sphere of randomly distributed charges: 3 force components and the potential of each charge were computed with a multipole acceptance ('$s/d$') parameter $\theta = 0.5$, giving a 1% rms force error

| Routine/ No. CPUs | 8 | 16 | 64 |
|---|---|---|---|
| Domain decomposition | 0.2 | 0.24 | 0.33 |
| Tree building | 2.3 | 2.3 | 2.7 |
| Tree traversal | 32.9 | 36.1 | 40.8 |
| Force summation | 64.4 | 61.2 | 55.7 |

**Table 2:** Breakdown of relative computational effort (percentage of wall-clock time spent in each routine) in the parallel tree code for a test case with 100k particles and 8, 16 and 64 processors respectively on the T3E-1200.

compared to direct (particle-particle summation). A glance at Fig. 7 shows that the IBM machine is over 10 times faster than the Cray for the same problem size and number of processors. Not surprisingly though, speedup saturation sets in considerably earlier on the IBM than for the Cray. Nevertheless, the parallel performance on both machines is respectable: for the $10^5$-particle system, we have a $52\times$ speedup on 64 processors of the T3E; $14\times$ on 16 IBM CPUs. The efficiency also improves with system size, since the ratio of work to communication goes up accordingly: for $10^6$ particles, we achieve a $15.2\times$ speedup on 16 CPUs of the IBM cluster.

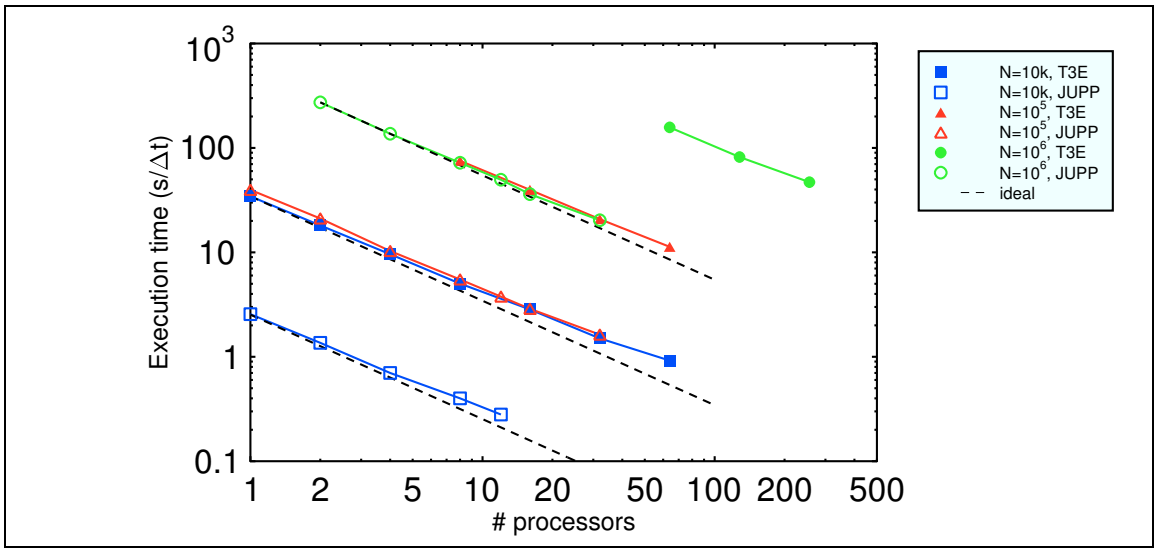


**Figure 7:** Timings on T3E-1200 and IBM-p690 cluster for an 10k- and 100k- and 1M-particle randomly distributed spheres.

Whether this performance can be maintained across a large number of cluster nodes remains to be seen: a hybrid approach combining shared-memory tree-access on each node with on-the-fly exchange of multipole data between nodes may ultimately prove to be more effective (albeit less portable) than the pure distributed-memory MPI implementation described here.

## Acknowledgements

During the course of this work, the author has benefitted from numerous discussions on parallel programming and tuning issues with W. Frings, G. Sutmann, K. Scholtyssik and B. Mohr.

REFERENCES


1. J. Barnes and P. Hut, *A hierarchical $O(N \log N)$ force-calculation algorithm*, Nature **324**, 446–449 (1986).
2. L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, J. Comp. Phys. **73**, 325–348 (1987).
3. S. Pfalzner and P. Gibbon, *Many Body Tree Methods in Physics*, Cambridge University Press New York (1996).
4. L. Greengard and W. D. Groop, *A parallel version of the fast multipole method*, Comput. Math. Applic. **20**, 63–71 (1990).
5. J. A. Board Jr., J. W. Causey, J. F. Leathrum Jr., A. Windemuth, and K. Schulten, *Accelerated molecular dynamics simulation with the parallel fast multipole algorithm*, Chem. Phys. Lett. **198**, 89–94 (1992).
6. H. Dachsel, "An improved implementation of the fast multipole method", Tech. Rep. FZJ-ZAM-IB-2002-14 Forschungszentrum Jülich GmbH, ZAM (2002).
7. J. E. Barnes, *A modified tree code: don't laugh; it runs*, J. Comp. Phys. **87**, 161–170 (1990).
8. J. Makino, *Vectorization of a treecode*, J. Comp. Phys. **87**, 148–160 (1990).
9. L. Hernquist, *Vectorization of tree traversals*, J. Comp. Phys. **87**, 137–147 (1990).
10. U. Becciani, V. Antonuccio-Delogu, and M. Gambera, *A modified parallel tree code for $N$-body simulation of the large-scale structure of the universe*, J. Comp. Phys. **163**, 118–132 (2000).
11. J. Dubinski, *A parallel tree code*, New Astronomy **1**, 133–147 (1996).
12. H. Yahagi, M. Mori, and Y. Yoshii, *The forest method as a new parallel tree method with the sectional voronoi tessellation*, Ap. J. Supp. **124**, 1–9 (1999).
13. M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree $n$-body algorithm", in *Supercomputing '93* Los Alamitos (1993) IEEE Comp. Soc. pp. 12–21.
14. M. S. Warren and J. K. Salmon, *A portable parallel particle program*, Comp. Phys. Commun. **87**(266–290) (1995).
15. J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, *Load balancing and data locality in hierarchical $N$-body methods*, J. Par. Dist. Comp. **27**, 118–141 (1995).
16. A. Grama, V. Kumar, and A. Sameh, *Scalable parallel formulations of the Barnes-Hut method for n-body simulations*, Parallel Comp. **24**, 797–822 (1998).
17. A. Caglar, M. Griebel, M. A. Schweitzer, and G. Zumbusch, "Dynamic load-balancing of hierarchical tree algorithms on a cluster of multiprocessor PCs and on the Cray T3E", in *Proceedings 14th Supercomputer Conference, Mannheim*, H. W. Meuer, Ed. Mannheim, Germany (1999) Mateo.
18. T. Eickermann and W. Frings, "VISIT – a visualization interface toolkit", Tech. Rep. Central Insitute for Applied Mathematics (2000).
19. P. Gibbon, "Introducing PEPC – a parallel electrostatic plasma coulomb-solver", in *Proc. DPG Spring Meeting* Aachen March 24–28 (2003).
20. T. Eickermann, W. Frings, P. Gibbon, A. Haeming, L. Kirtchakova, and D. Mallmann, "Steering UNICORE applications with VISIT", in *Proc. Supercomputing 2003* (2003), submitted.
21. H. Shi and J. Schaeffer, *Parallel sorting by regular sampling*, J. Par. Dist. Comp. **14**, 361–372 (1992).
22. S. Pfalzner and P. Gibbon, *A hierarchical tree code for dense plasma simulation*, Comp. Phys. Commun. **79**, 24–38 (1994).