John von Neumann Institute for Computing

Performance Analysis and Visualization of the N-Body Tree Code PEPC on Massively Parallel Computers

P. Gibbon, W. Frings, S. Dominiczak. B. Mohr

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 367-374, 2006.

© 2006 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume33

Performance analysis and visualization of the N-body tree code PEPC on massively parallel computers

P. Gibbon[a], W. Frings[a], S. Dominiczak[a], B. Mohr[a],

[a]John-von-Neumann Institute for Computing, Forschungszentrum Jülich GmbH, ZAM, D–52425 Jülich, Germany

The performance and scalability of a parallel tree code for rapid computation of long-range Coulomb forces is investigated using both visual and analytical techniques. The present code uses a variation of the Hashed-Oct-Tree algorithm, in which communication overhead is minimised by bundling multipole data for large groups of particles prior to shipment between processors. The two critical components of this algorithm, the tree traversal and load-balancing, are examined in highly dynamic physical context with the help of the KOJAK performance analysis toolkit and the online visualisation packages VISIT and XNBODY. The parallel scalability of PEPC is investigated on the Jülich IBM p690 and BlueGene/L machines.

1. Introduction

Even in the era of Teraflop computing, the N-body problem for systems dominated by long-range potentials remains a formidable algorithmic and computational challenge. The brute-force approach, in which all $N(N-1)$ mutual interactions between simulation particles are computed directly, is simply impractical for many N-body systems such as plasmas, gravitational systems, 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 often no need to compute potentials and forces to higher accuracy than the error incurred in integrating the equations of motion, typically in the $10^{\ 4}$–$10^{\ 2}$ range.

Two techniques developed in the mid-1980s—the hierarchical Tree Code [1] and the Fast Multipole Method (FMM) [2], with respective algorithmic scalings of $O(N \log N)$ and $O(N)$—have revolutionized long-range N-body simulation across a broad range of fields [3]. These methods reduce the number of direct particle-particle interactions through the systematic use of multipole expansions, making it possible perform simulations with many millions of particles. Despite this progress on the algorithmic side, recent advances in the massively parallel computing paradigm have prompted a further challenge: can hierarchical algorithms be effectively implemented on a parallel machine with thousands of processors?

At first sight, the recursive data structure of tree codes would seem to rule out parallelism altogether, but in fact the construction of both the tree and particle interaction lists can be cast in data-parallel form on a shared-memory machine [4], leaving a straightforward $N \times N_{list}$ force summation to contend with. On a distributed-memory machine, the tree structure either has to be known to all processors—restricting the maximum simulation size—or somehow divided up equally among them. In the latter case, a *locally essential* tree can be built comprising only the information required to compute forces for locally held particles. Over the past decade various parallel tree algorithms have been proposed and implemented, including virtual shared-memory approaches [5], and distributed memory schemes [6,7].

This paper describes an efficient, portable implementation of a parallel tree code—PEPC (Pretty Efficient Parallel Coulomb-solver)—initially designed for mesh-free modelling of nonlinear, com-

plex plasma systems [8], but recently extended to other application areas ranging from molecular dynamics to protoplanetary accretion discs [9]. For PEPC, we have adopted the Warren-Salmon 'hashed oct-tree' scheme based on a space-filling 'Morton' curve, derived from 64-bit particle-coordinate keys. The discontinuities inherent in this curve, potentially leading to disjointed domains and additional communication overhead [10] is found to be a relatively minor issue compared to load-balancing and geometrical factors.

While the performance of PEPC on both commodity and high-end clusters is such that multi-million-body simulations can already be routinely performed, porting the code to new architectures with many *thousands* of processors such as BlueGene/L presents a much tougher challenge. To isolate and unravel the communication-critical parts of the code more systematically, we have made use of the automatic performance toolkit KOJAK [11].

PEPC has also been equipped with a combination of visualization toolkits VISIT [12] and XNBODY [13] to assist in tracking progress and enable real-time computational steering (user-feedback) of simulations. We have extended this online visualisation and steering (OVS) capability for PEPC by incorporating details of the tree structure on each processor, thus allowing visual monitoring of the inner, dynamic workings of the algorithm, such as the domain decomposition or load balancing.

The structure of this paper is as follows: In Section 2 we briefly review the hashed oct-tree algorithm and the various implementations of the tree-traversal routine available in PEPC. In Section 3 some benchmarks for 'static' systems are presented demonstrating the code's scalability on the Jülich IBM p690 cluster JUMP. The tree-traversal variations are then examined with the help of the KOJAK toolkit with a view to porting the code onto the BlueGene system. Finally, in Section 4 the dynamic evolution of the code performance for a plasma physics application is then discussed with the help of online visualization techniques.

2. Variations on the Hashed Oct Tree algorithm: asynchronous *vs.* collective

The Hashed Oct Tree algorithm is well documented in the literature [6] so we need not dwell on the details of tree construction here: features particular to the code PEPC are also described elsewhere [14]. A summary of the algorithm implemented in PEPC is depicted in Table 1, along with the theoretical scaling and relative effort for each major routine. All of the above routines can be performed in parallel, requiring an effort $O(N/P)$, give or take a slowly varying logarithmic factor.

Code region	Scaling	% CPU time
Domain decomposition: weighted key-sort	N/P	3
Construct local trees and multipole moments	$P \log N/P$	4
Construct interaction lists (tree walk)	$N/P \log N$	43
Compute forces and potential	$N/P \log N$	49
Update particle velocities and positions	N/P	1

Table 1
Algorithmic scaling and relative computational effort of major routines in PEPC. The symbols N and P represent the total number of particles and processors respectively.

As mentioned above, the HOT algorithm employs 64-bit keys derived from the (3-dimensional) particle-coordinates. Domain decomposition is acheived by cutting out equal portions of the sorted

particle key-list and allocating these to the processors. The fully parallel sort currently implemented is an adaptation of the PSRS (parallel sort by regular sampling) scheme originally proposed in Ref. [15]. 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 geometry—regular sampling tends to produce highly imbalanced particle numbers across the processors. To compensate this effect, we instead use load-weighted sampling, which allows for the actual distribution of keys along the whole space-filling curve. Problems may arise here if the key distribution is not finely enough resolved, a feature which we return to in Sec. 4.

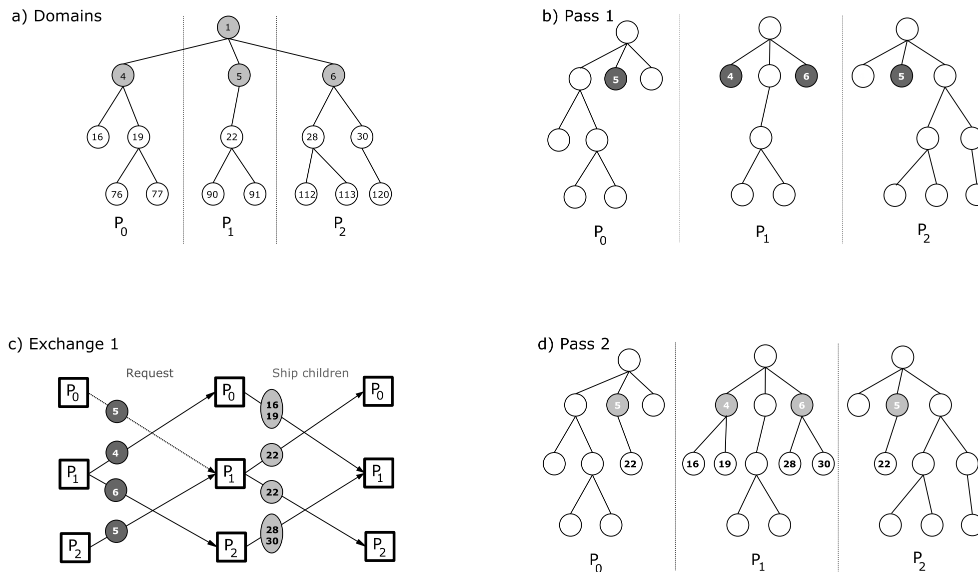

Figure 1. a) Domain decomposition and b)-d) parallel tree-walk for 3 CPUs.

To facilitate the exchange of information (in particular, multipole moments) between processors, a set of local ‘branch’ nodes is defined, comprising the minimum number of *complete* twig and leaf nodes covering the whole local domain—Fig. 2a). 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.

By far the most algorithmically demanding part of this code is the tree walk, which in PEPC combines a previous list-based vectorised algorithm [4] with the asynchronous scheme of Warren & Salmon [6] 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 around 1000 particles at a time before actually computing their forces.

In the first pass of the walk (Fig. 2b), traversals are made through the local trees using the familiar divide-and-conquer strategy common to sequential tree codes [4]. 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 (dark-shaded nodes) to be processed in the 2nd ‘exchange’ half of the routine (Fig. 2c) 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 in principle allows some overlap of communication with the creation of new local tree nodes (copies). At the end of subsequent passes (Fig.2d: here just 2 are needed), each processor’s local tree contains all the nodes required to compute the forces on its own particles. With increasing system size, the nodes fetched during the traversals eventually take up most of the space in the local hash-table.

Three further variations of this procedure are currently implemented in PEPC: i) a *prefetch* mode in which lists of the fetched and requested nodes are retained for the subsequent timestep, allowing most of the locally essential tree to be rebuilt via a prune-and-graft procedure; ii) a purely collective exchange replacing the asynchronous SEND/RECEIVE swaps for each pass and iii) a *freeze* mode in which the entire tree structure is held fixed for several timesteps, but where the multipole information is updated and exchanged where necessary.

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 cache-optimised. Also, the physics and algorithm are kept naturally apart, so that additional forces and/or boundary conditions 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. To redress this problem, the batch size N_b for each processor is determined individually, so that the integral $\sum_{p=1}^{N_b} N_{\mathrm{int}}(p)$ is the same, and each processor computes the same number of interaction pairs during each pass.

3. Analysis of algorithm performance and scaling using KOJAK

The KOJAK performance-analysis tool environment [11] provides a complete tracing-based solution for automatic performance analysis of MPI, OpenMP, or hybrid applications running on parallel computers. KOJAK automatically searches execution traces of the application for patterns that indicate inefficient use of the underlying programming model(s). The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data.

The instrumentation inserts extra measurement code to capture begin and end of important phases in the user code (e.g., subroutines or loops), MPI message transfers and collective operations, as well as OpenMP constructs. KOJAK can handle C, C++, and Fortran source code. If necessary, the application can also be linked to the PAPI library [16] for collection of hardware counter metrics as part of the trace file.

Running the instrumented executable generates a trace file in the EPILOG format. After program termination, the trace file is fed into the EXPERT analyzer. EXPERT transforms event traces into a compact representation of performance behavior, which is essentially a mapping of tuples (performance problem, call path, location) onto the time spent on a particular performance problem while the program was executing in a particular call path at a particular location. There are two classes of search patterns, those that collect simple profiling information, such as communication or execution time, and those that identify complex inefficiency situations, such as a receiver waiting for

the wrong message. The former are usually described by pairs of enter and exit events, whereas the latter are described by more complex compound events usually involving more than two events covering multiple locations, a situation which can easily arise in a tree code.

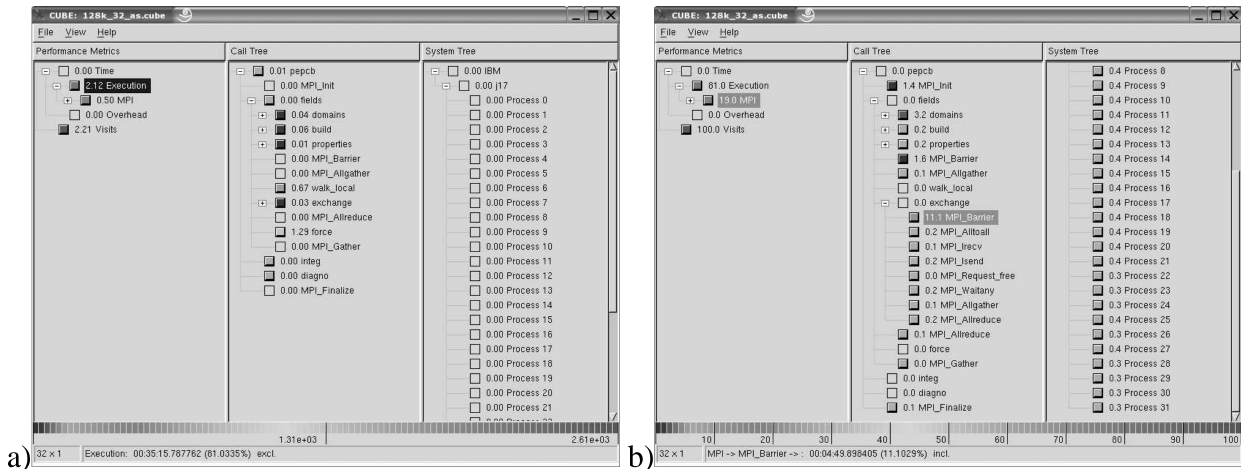

Figure 2. KOJAK EXPERT analysis of a 40-timestep PEPC test run using the asynchronous tree walk algorithm of Fig. 2 showing breakdown in: a) execution time and b) MPI-call time.

Figure 2 shows a screen-dump of the result presentation component (CUBE) of the EXPERT automatic event trace analyzer for a PEPC simulation of a sphere comprising 128000 charges. Using the color scale shown on the bottom, the severity of performance problems found (left pane) and their distribution over the program's call tree (middle pane) and machine locations (right pane) is displayed. The severity is expressed in percentage of execution time lost due to this problem. By expanding or collapsing nodes in each of the three trees, the analysis can be performed on different levels of granularity. We refer to [11] for a detailed description of KOJAK and EXPERT.

In this example the run took 43 minutes (2600 s) on aggregate (or 80 s per CPU): in the left-hand column we see how this is divided up in 'useful' execution- (81%) and MPI- (19%) time. Clicking on the Execution button we obtain the breakdown by routine in the 2nd column of Fig. 2a), where we find values consistent with those given in Table 1. The MPI breakdown in Fig. 2b) reveals a large imbalance, which is almost entirely due to nonlocal multipole fetches in the 'exchange' routine.

KOJAK also supports analysis of the *difference* between two measurements (e.g., using different input data sets, executing on different processor numbers, or comparing diferent implementations of the same code) by providing an utility that "subtracts" one CUBE result file from another one, resulting in an CUBE file which contains the differences of the severity for each problem for each call path on each loaction. This file can also be analyzed using the CUBE result browser.

This feature is exploited to compare the asynchronous tree-walk against the 'frozen tree' variation described previously for the same test problem as before – Fig. 3. Negative numbers (aggregate runtime seconds scaled according to bottom ruler) indicate an improvement over the asynchronous algorithm; positive numbers a deterioration. For the execution time, we see that there are substantial savings in the tree-building overhead (domains, build), which is what we expect by freezing the data structure and rebuilding only every 10th timestep. More significantly, communication time is also saved on the exchange part of the tree-walk, which is just what was intended. This is paid for by

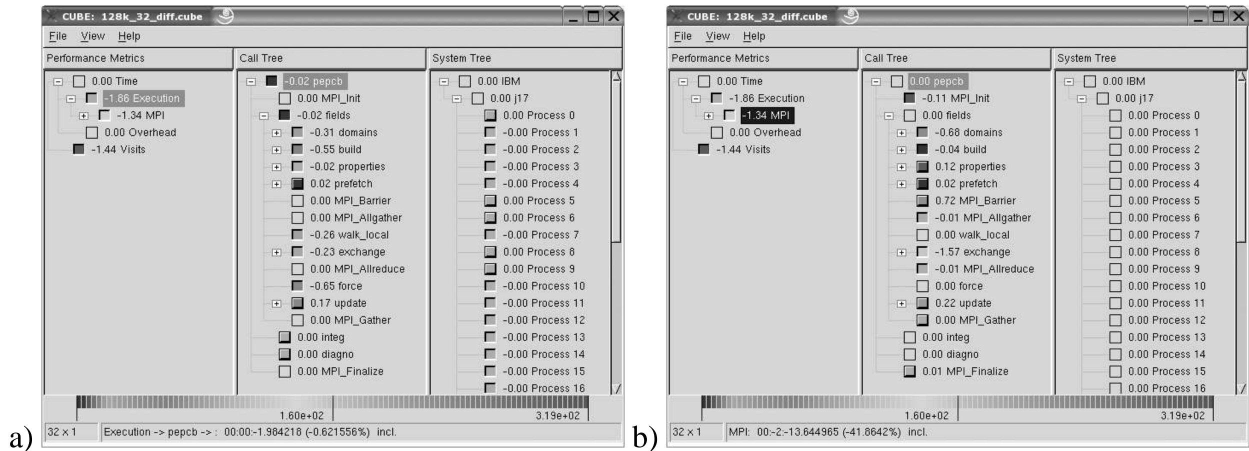

Figure 3. Differential analysis of PEPC sphere simulations comparing the asynchronous tree walk in Fig. 2 to the 'freeze' mode described in Sec. 2): a) execution breakdown; b) MPI breakdown.

the update routine, and, somewhat mysteriously, by increased barrier time – a side-effect which is presently not fully understood.

Overall however, the 'tree-freezing' concept shows promising scalability improvements over the asynchronous mode, as single-timestep benchmarks in Fig. 4 demonstrate. For large systems, the standard algorithm performs well on both the Regatta and BlueGene/L systems, scaling up to 1024 CPUs on the latter.

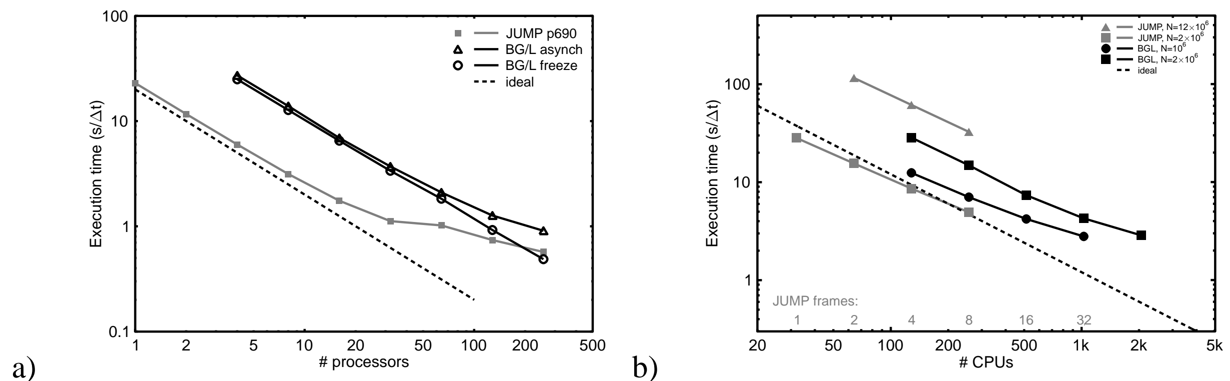

Figure 4. Timings on IBM-p690 cluster and BG/L for a) 128k and b) multi-million charge spheres.

4. Performance visualization using VISIT and XNBODY

Online visualization and steering (OVS) is normally used to track and adjust the dynamic development of simulations where the outcome depends on a large set of parameters. Here we use the OVS system developed at ZAM [13] to provide insight into the *algorithmic* behaviour of PEPC, and in particular to investigate the load-balancing characteristics for a dynamically evolving problem.

As noted before, this can be adjusted on the fly during the domain decomposition by appropriate weighting of the particle key-list segments allocated to each CPU.

To illustrate this in action, we consider an example taken from Ref. [8], in which a thin ionized wire comprising 1 million electrons and ions is irradiated by a high-intensity laser pulse. Physically, what happens here is that the laser begins to strip electrons from the target surface, accelerating them in all directions to form a rapidly expanding, negatively charged plume around the wire – Fig. 5(a–c). Despite the initially uniform density distribution, the target's geometry already poses problems for an unbalanced parallel tree code, as illustrated in the middle sequence (d–f), in which the particles are equally divided amoung the processors. The boxes represent local tree domains coloured according to the total amount of work performed by each CPU in the force calculation, which here varies by as much as 30%.

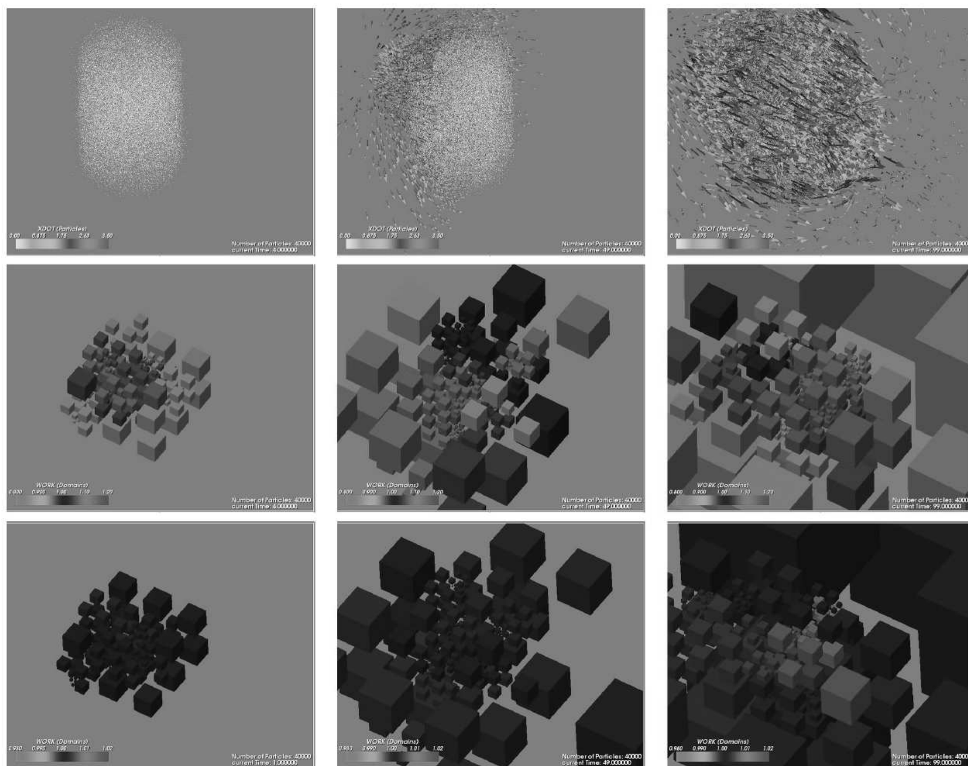

Figure 5. Visualization of dyanmic load balancing using VISIT and XNBODY toolkits. The top row shows the particle positions at times $t = 0, 50$ and 100 during the formation of the laser-induced charge cloud. The centre and bottom rows show tree domains, coloured according to the number of force computations per CPU for equal numbers of particles (middle) and load-balanced (bottom).

In the final sequence (g–i), the simulation has been repeated with dynamic load balancing switched on. In this case the imbalance is stays below 1% for most of the simulation. Towards the end however, we see that the workload apparently becomes uneven again near the centre of the target. Because of the expanding electron cloud, the system effectively becomes more clustered with time, leaving a high concentration of particle keys near the centre. This eventually leads to undersampling in the sort routine in this region, which in turn causes incorrect balancing — a feature which would

be difficult to trace without the direct visual relationship between work load and spatial location provided by this technique.

Summary

A new parallel tree code – PEPC – for rapid computation of long-range interactions has been presented in which various implementations of the tree traversal routine have been compared and the overall scaling of the code with up to 1024 processors of BlueGene/L demonstrated. Load balancing issues have also been investigated with the help of visual techniques, enabling potential pitfalls in the parallel sort routine to be properly addressed. As a result of these algorithmic improvements we expect PEPC to scale well beyond a single BG/L rack in the near future, paving the way for Coulomb/Newtonian gravity simulations with $10^8 - 10^9$ particles.

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] S. Pfalzner and P. Gibbon. A hierarchical tree code for dense plasma simulation. *Comp. Phys. Commun.*, 79:24–38, 1994.

[5] U. Becciani, V. Antonuccio-Delogu, and M. Gambera. A modifi ed parallel tree code for n-body simulation of the large-scale structure of the universe. *J. Comp. Phys.*, 163:118–132, 2000.

[6] M. S. Warren and J. K. Salmon. A portable parallel particle program. *Comp. Phys. Commun.*, 87(266–290), 1995.

[7] J. Dubinski. A parallel tree code. *New Astronomy*, 1:133–147, 1996.

[8] P. Gibbon, F. N. Beg, R. G. Evans, E. L. Clark, , and M. Zepf. Tree code simulations of proton acceleration from laser-irradiated wire targets. *Phys. Plasmas*, 11:4032–4040, 2004.

[9] S. Pfalzner, S. Umbreit, and Th. Henning. Mass and angular momentum transfer in disc-disc interactions. *Ap. J.*, 629: 526, 2005, submitted.

[10] 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.

[11] Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing'*, 49(10–11):421–439, November 2003.

[12] Th. Eickermann, W. Frings, P. Gibbon, L. Kirchakova, D. Mallmann, and A. Visser. Steering UNICORE applications with VISIT. *Phil. Trans. Roy. Soc.*, 363: 1855–1865, 2005.

[13] S. Dominiczak. Development of online visualization for NBODY6++ using the VISIT library. Technical Report FZJ-ZAM-IB-2005-02, Research Centre Jülich, 2005. http://www.fz-juelich.de/zam/docs/printable/ib/ib-05/ib-2005-02.pdf

[14] P. Gibbon. PEPC: Pretty Effi cient Parallel Coulomb-solver. ZAM Technical Report FZJ-ZAM-IB-2003-05, Research Centre Jülich, 2003. http://www.fz-juelich.de/zam/docs/printable/ib/ib-03/ib-2003-05.pdf

[15] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *J. Par. Dist. Comp.*, 14:361–372, 1992.

[16] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.