Original published: P. Gibbon, M. Hofmann, G. Rünger, and R. Speck. Parallel sorting algorithms for optimizing particle simulations. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing, Workshops and Posters (CLUSTER WORKSHOPS)*, pages 1–8. IEEE, 2010. Online available at http://dx.doi.org/10.1109/CLUSTERWKSP.2010.5613105.

Parallel Sorting Algorithms for Optimizing Particle Simulations

Michael Hofmann* and Gudula Rünger
Department of Computer Science
Chemnitz University of Technology, Germany
Email: {mhofma,ruenger}@cs.tu-chemnitz.de

Paul Gibbon and Robert Speck
Forschungszentrum Jülich
Jülich Supercomputing Centre, Germany
Email: {p.gibbon,r.speck}@fz-juelich.de

***Abstract*—Real world particle simulation codes have to handle a huge number of particles and their interactions. Thus, parallel implementations are required to get suitable production codes. Parallel sorting is often used to organize the set of particles or to redistribute data for locality and load balancing concerns. In this article, the use and design of parallel sorting algorithms for parallel particle simulation codes are discussed. As a typical example, the particle simulation code PEPC is considered and a specific parallel sorting algorithm for this application is presented. The resulting parallel simulation code was implemented on an IBM Blue Gene/P system and corresponding performance results are shown.**

***Keywords*-particle simulations; parallel sorting; load balancing; data redistribution; performance optimization;**

I. Introduction

Particle simulation methods belong to the most commonly used approaches for numerical simulations of complex physical problems [1]. The fields of application include astrophysics, molecular dynamics, plasma physics, and fluid dynamics. Real-world scientific problems are modelled using large particle systems, thus leading to large scale computational problems. Consequently, advanced and efficient simulation methods are required and the use of high performance computer systems is inevitable. Complex particle systems consist of millions or billions of particles and involve large amounts of data that need to be processed efficiently during the simulation. Parallel sorting is often used in parallel particle simulations to distribute the particles across the compute nodes and to prepare the particle data in such a way that the locality of computations is increased. Therefore, efficient parallel sorting methods for complex and high scaling parallel environments are of great importance.

Sorting in general and parallel sorting in particular is a fundamental problem in computer science. The sorting problem has been the subject of both theoretical and practical interest for decades, resulting in numerous contributions on sequential and parallel sorting [2], [3], [4], [5]. Current research activities include sorting in high scaling parallel environments [6] and the use of multi-core processors and GPUs [7], [8], [9], [10]. Most research contributions for parallel sorting focus on generic sorting problems without taking into account the specific demands of the particular applications (e.g., in scientific computing). However, to achieve scalability and efficiency for large, complex particle simulations, the parallel sorting algorithm has to be adapted to the specific needs of the specific particle simulation.

In this article, the need for efficient parallel sorting algorithms in particle simulations is discussed. An overview of the demands on sorting algorithms in parallel particle simulations is given. Important aspects are the structure of the data elements to be sorted, the memory requirements and weighted data elements. Parallel sorting in particle simulations serves different purposes, including the creation of an appropriate order for the computations, locality aspects for memory accesses, or particle distribution in distributed memory machines. In most cases, parallel sorting is used to achieve fast and efficient simulation codes. Thus, the sorting itself should be fast and efficient and should fit into the application code, which leads to the challenge to design specific parallel sorting methods for specific simulation codes. The investigation of parallel sorting in the context of particle simulation codes is part of the ScaFaCoS project[1].

The contribution of this article is to consider the problem of using an appropriate parallel sorting algorithm for the specific needs of a parallel particle simulation code. As a realistic example, the simulation code PEPC [11] is investigated. In this simulation code, sorting is used for locality and load balancing reasons. For this application code, an improved parallel sorting phase is developed and incorporated into the code. The improved sorting is based on partitioning which exploits weights measuring the computational load associated with the particles. This leads to an optimization of the parallel particle simulation code PEPC. Performance results on an IBM Blue Gene/P system [12] using up to 16,384 processes are shown.

The rest of this article is organized as follows. Section II introduces efficient particle simulation methods and their need for parallel sorting. Section III discusses the demands on parallel sorting algorithms in particle simulations. Section IV describes the optimization of parallel sorting in the particle simulation code PEPC and Sect. V presents performance results. Section VI concludes the article.

*Supported by Deutsche Forschungsgemeinschaft (DFG).

[1]ScaFaCoS (*Scalable Fast Coulomb Solvers*) is a research project supported by the German Federal Ministry of Education and Research (BMBF).

978-1-4244-8396-9/10/$26.00

II. Efficient Particle Simulations

Simulating systems consisting of a large number of particles is a common problem in physics [13]. For example, the motion of the particles induced by interactions through long-range forces are studied. Since systems with more than two particles do not have analytical solutions for the equations of motion, numerical solutions are inevitable. The dynamical evolution of such systems can be simulated by calculating the interactions between particles at discrete time steps and modifying their positions and velocities accordingly.

A. Particle Simulation Methods

Long-range potentials, like the Coulombic or gravitational potential, provide significant contributions even for distant particles. Therefore, all pairwise interactions between particles need to be taken into account. The direct evaluation of these potentials for a system with n particles would require $\mathcal{O}(n^2)$ operations in each time step. Efficient methods like the *Barnes-Hut* algorithm [14] or the *Fast Multipole Method* [15] can reduce these costs down to $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$ operations, respectively. These methods can be used for an efficient simulation of large particle systems. However, the continuing need for simulating an ever increasing number of time steps and larger systems in reasonable time requires efficient implementations of these methods on high performance computer systems.

The Barnes-Hut algorithm and the Fast Multipole Method represent efficient methods that divide the interactions into *near field* and *far field*. A hierarchical partitioning is used to group particles into boxes according to their positions. Interactions of particles that are "close" to each other represent near field interactions, which are calculated directly. Interactions of particles that are "separated" from each other (based on specific criteria of the particular application) represent far field interactions. Their contributions are approximated using truncated expansions (e.g., Taylor expansion or Legendre expansion). This enables a treatment of a set of distant particles as a single massive pseudo-particle and leads to reduced computational costs for the far field calculations.

Partitioning the particles into boxes is usually based on a specific numbering of the boxes. Particles are assigned a box number according to the box they are located in. By sorting the particles and their associated data according to their box numbers, all particles that belong to the same box are contiguously arranged in memory. By using an appropriate numbering of the boxes, it is also possible to arrange particles of neighboring boxes close to each other in memory. This can be used to increase the locality of the later computations. Figure 1 shows a two dimensional example for a numbering of boxes according to a *Z-order* space filling curve [16].

Data parallel implementations of particle simulations can be achieved by distributing the particles across the available processing elements (e.g., compute nodes or processor cores). An efficient and scalable parallel implementation requires a partitioning of the particle data that leads to an appropriate balancing of the computational load and to a minimization of the resulting communication. A common partitioning scheme is based on the linear ordering of the boxes according to a space filling curve. The curve is divided into several parts and the particles and boxes of each part are assigned to a single compute node. The two-dimensional example in Fig. 1 shows a distribution of the particles and boxes across three compute nodes.

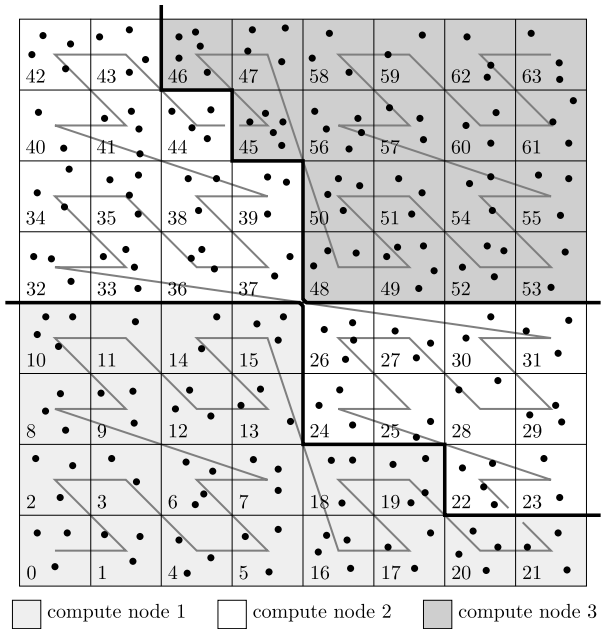

Figure 1. Two dimensional (particle system) example for partitioning and numbering of boxes according to a Z-order space filling curve and distribution them across three compute nodes.

B. A library for long-range interactions

The goal of the ScaFaCoS project is to create a software library that includes various fast solution methods for the efficient calculation of long-range interactions. This includes parallel implementations of algorithms like the Fast Multipole Method (FMM), the Barnes-Hut algorithm (BH), Particle-Particle Particle-Mesh methods (P^3M), Multigrid methods, and fast summations with Fast Fourier transforms. The parallel solution methods are based on efficient and scalable implementations that are suitable for high scaling parallel architectures like the IBM Blue Gene/P platform.

The ScaFaCoS library is intended to provide a uniform interface for all implemented solution methods. Similarly to other successful library approaches for standard operations (e.g., (P)BLAS and (Sca)LAPACK for linear algebra), the library is intended to be a parallel software library that can be easily integrated into various applications. This should allow the application programmer to focus on the needs of

the specific application (e.g., a molecular dynamics code), while using the efficient solution methods of the library for the computationally intensive calculations of the particle interactions. The simulation algorithms of the library exploit parallel sorting of the particles and so parallel sorting algorithms are included.

C. PEPC: A Multi-Purpose Parallel Tree-Code

PEPC (*Pretty Efficient Parallel Coulomb-solver*) is a particle simulation code for the computation of long-range forces based on the Barnes-Hut algorithm [14]. The kernel routines of PEPC are used by several "front-end" applications that are built on top of PEPC. The front-end applications implement the dynamic evolution of particle systems and include problem specific aspects of the particular physical domain. Examples for front-end applications are PEPC-E for molecular dynamics simulations, PEPC-B for simulating laser- or beam-plasma interactions, or PEPC-G for astrophysical simulations.

The Barnes-Hut algorithm uses a hierarchical partitioning of the particles into boxes until every particle is located in a separate box. The parallel implementation of PEPC is based on the *Hashed Oct Tree* scheme of Warren et al. [17]. In this scheme, the position of a particle is encoded in a specific key such that the oct-tree of boxes is not stored as hierarchical data structure but as a hash table. During the creation of the data structures in PEPC, all particles are assigned a box number according to the box they are located in. The box numbers are recursively created based on their spatial coordinates. The resulting linear ordering of the boxes corresponds to a Morton ordering. For inhomogeneous distributions of particles, this construction leads to non-uniform distributions of the box numbers of the particles. The chosen linear ordering retains the spatial locality of the boxes. By sorting all particles according to their box numbers, particles that are close to each other (in the particle system) become also close to each other in memory. In the same way, the parallel implementation uses parallel sorting to redistribute the particles among the compute nodes, such that particles that are close to each other are distributed to the same compute node (with high probability).

PEPC is also capable to simulate the dynamic evolution of the particle system (e.g., the motion of the particles) across many time steps. In every time step, the Barnes-Hut algorithm is used to calculate the interactions of the particles. Based on information collected in the previous time step, every particle is assigned a computational load value that approximates the costs for calculating the interactions of that particle. These load values are used in the sorting algorithm to determine a balanced distribution of particles across the compute nodes. In PEPC, the parallel sorting algorithm is also responsible for creating this balanced distribution during the redistribution of the particles.

III. Parallel Sorting for Particle Simulations

In practice, parallel sorting is used in parallel particle simulations mainly for preparing the data for later computations. The costs for sorting are rather small ($< 10\,\%$) in comparison to the total runtime. However, especially in parallel applications, even these small program parts can have a significant influence on the overall performance. Using sequential sorting or other centralized (non-parallel) solutions can lead to a significant reduction of the achievable parallel performance. On the other hand, also parallel sorting solutions can have considerable drawbacks, for example, if the scalability of the parallel sorting algorithm is not as good as the scalability of the rest of the application or if the output distribution (of the data) produced by the parallel sorting algorithm leads to load imbalances for the later computations. Thus, parallel sorting in particle simulations has specific demands rarely covered by existing parallel sorting algorithms. In the following, we discuss several aspects that arise from parallel particle simulations.

A. Sorting Data Elements

Sorting data elements in practice involves data elements consisting of a key value and additional information associated with this key. In particle simulations, the additional information usually includes information about properties of the particles, like their positions or their masses. These properties represent data components that need to be rearranged in memory or send to other compute nodes together with the keys during the sorting process. Handling these additional data components is often done by the parallel sorting method and can have a significant influence on its performance.

The size of the additional data has a strong effect on their efficient handling. In particle simulations, the key values are often box numbers that can be represented by single integer values. The size of the additional data depends on the specific application, but is usually larger than the key value. For example, methods for calculating long-range interactions (e.g., FMM or Barnes-Hut) require at least positions (three floating point values in three-dimensional space) and masses or charges (one floating point value) of the particles. Applications like PEPC also require properties like the velocities and the computational load values of the particles.

Besides the size of the data elements, also the data format used to store the data in memory plays an important role for the implementations of parallel sorting methods. The following three different data formats are commonly used:

1) The key value and all other data components of a particle are stored within a structured data type. A single array of this data type is used to store the particles.
2) Separate arrays are used for the key value and for all other data components. For data components that

consist of several values (e.g., three floating point values for positions in three-dimensional space), these values are stored consecutively.

3) Arrays with single basic data type entries are used. Data components that consist of several values are stored in several separate arrays (e.g., the x, y and z values of the positions are stored in three separate arrays).

The particle simulation code PEPC implements the third data format.

B. Memory Requirements

The memory requirements of sorting algorithms in particle simulations depend on the number of particles of the system to be simulated. *Out-of-place* sorting algorithms use separate output arrays for storing the sorted data. Depending on the data format (see Sect. III-A), these memory requirements can include both, the key values and the additional data. The maximum number of particles that can be simulated by a given parallel environment may be limited by the memory requirements of the parallel sorting step. To avoid this limitations, *in-place* sorting algorithms are highly required for sequential and parallel sorting. In-place sorting algorithms sort the input data itself and require only an amount of additional memory that is independent from the data to be sorted. However, in-place sorting algorithms usually achieve these reduced memory requirements at the expense of an increased runtime [18].

The particle simulation code PEPC makes excessive use of additional memory during the parallel sorting of the particle data. Especially for large numbers of compute nodes, the memory requirements become hardly predictable and can often only be resolved by an exaggerated usage of memory.

C. Weighted Data Elements

For parallel sorting of data elements of equal size, each of these data elements imposes the same computational load. Therefore, in homogeneous parallel environments a balanced distribution of data elements across compute nodes is preferred by the vast majority of parallel sorting algorithms. Moreover, for specific algorithms a balanced distribution can be strictly required [19]. However, parallel applications like particle simulations often rely on their own (application-specific) modelling of the computational load. From the perspective of parallel sorting algorithms, this kind of application-specific property can be seen as a *weight* value associated with each data element.

The computational load values assigned to the particles in PEPC can be treated as weights. A balanced distribution of the particle data according to these weights after the parallel sorting is essential for achieving good load balancing in the following computations.

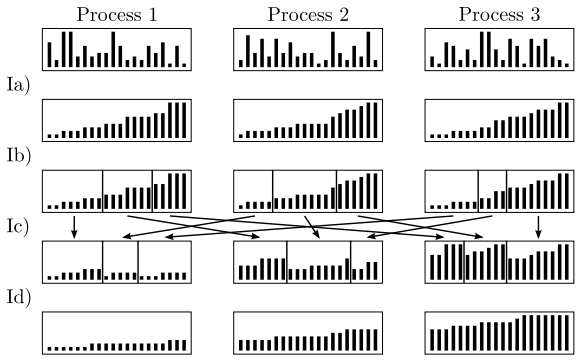

Figure 2. Example for parallel sorting the keys in Step I (keys are represented by individually sized vertical bars) with $p = 3$ processes.

IV. Optimizing Parallel Sorting in PEPC

As an example for a successful adaptation of parallel sorting to the requirements of a particle code, we present an improved parallel sorting step in the parallel particle simulation code PEPC. Parallel sorting in PEPC is part of the domain decomposition method that is used for a data parallel implementation of the Barnes-Hut algorithm. As described in Sect. II-C, each particle is assigned a box number represented by an integer key. By sorting the particles in parallel with respect to these key values, the particle data are redistributed among the processes. This parallel sorting step is also responsible for creating a balanced distribution of particles to processes according to the computational load associated with each particle.

Besides the key value, each particle has about 100 Bytes of additional data that needs to be redistributed together with the key value. The different components of the additional data are stored in separate arrays as described as data format 3 in Sect. III-A.

A. Original Parallel Sorting in PEPC

The original parallel sorting method in PEPC sorts the particle data in two steps:

I. *Sorting keys*: Sort the key values in parallel to create a *plan* that captures the reordering and redistribution to be performed during parallel sorting of the entire data set.

II. *Redistributing particle data*: Reorder and redistribute the particle data according to the plan from Step I.

Sorting the key values in Step I uses an adapted version of *Parallel Sorting by Regular Sampling* [20]. Figure 2 shows an example for this parallel sorting strategy with $p = 3$ processes. Step I has the following substeps:

Ia) Sort the local sequences of keys on each process.

Ib) Determine $p - 1$ sample keys by *adaptive sampling*. These sample keys are used by each process to divide its local sequence of keys into p sub-sequences.

Ic) Redistribute the sub-sequences of keys of all processes with an all-to-all communication operation.
Id) Merge the (received) sub-sequences of keys on each process.

The adaptive sampling in Step Ib is a variation of *regular sampling* [20]. For regular sampling, each process determines $p-1$ local sample keys and sends them to a root process. The root process selects $p-1$ global sample keys and sends them to all processes. The adaptive sampling of the original parallel sorting method in PEPC increases the number of local sample keys adaptively in several rounds. The additional local sample keys are selected from sub-sequences that contain too many keys. Additionally, the choice of the global sample keys is performed with respect to the computational load values of the particles. This ensures that after redistributing the particles, the parallel Barnes-Hut algorithm achieves a good load balancing.

The redistribution plan created during the parallel sorting of the keys in Step I consists of the parameters used for the all-to-all communication operation and two permutations that describe the local reordering of the keys before and after the redistribution. Redistributing the particle data itself according to this plan in Step II proceeds as follows:

IIa) Permute the particles according to the permutation created during the local sorting and store the reordered particle data in a single intermediate array.
IIb) Redistribute the particle data with an all-to-all communication operation.
IIc) Permute the received particles according to the permutation created during the local merge and store the reordered particle data in the target arrays.

The original parallel sorting method in PEPC has several drawbacks. The adaptive sampling strategy creates large numbers of sample keys (among which only $p-1$ are finally chosen). This results in high memory requirements as well as large amounts of communication for coordinating the selection of the sample keys between all processes. Balancing the computational load of the particles evenly depends on the chosen sample keys. Improving the load balancing can only be achieved by further increasing the number of sample keys, thus worsening the high memory and communication requirements. Separating the parallel sorting in two steps involves two all-to-all communication operations (one for the keys and one for the particle data) and is another drawback.

B. Optimized Sorting by Partitioning

To improve the parallel sorting method used in PEPC, we have replaced the adaptive sampling in Step Ib with a more efficient and exact partitioning algorithm. The adaptive sampling determines $p-1$ sample keys that are used to divide each local sequence of keys into p sub-sequences. In contrast, the partitioning algorithm searches directly for the positions of the boundaries of the sub-sequences. The input of the partitioning algorithm is a sequence of keys locally sorted on each process. The output consists of $p-1$ position values on each process that divide each local sequence into p sub-sequences. The keys of the i-th sub-sequences of all processes are smaller than the keys of the $(i+1)$-th sub-sequences for $i = 1, \ldots, p-1$. The sub-sequences are then redistributed, such that process i receives the i-th sub-sequences of all processes for $i = 1, \ldots, p$.

The partitioning algorithm performs a search for the local boundary positions of the sub-sequences. For ease of description, we give a simplified algorithm that divides each local sequence K of keys into two sub-sequences only. The division is performed such that the total weight of the first sub-sequences of all processes is between w_{min} and w_{max}. The second sub-sequences (implicitly) get the remaining weight. The weights of the keys approximate the computational load of the corresponding particles in PEPC and are given as input array W. The result of the simplified partitioning algorithm is a local boundary position pos on each process that divides the local sequence of keys into two sub-sequences. To create all $p-1$ local boundary positions, the simplified partitioning is performed $p-1$ times using different values w_{min} and w_{max} each time. The values of w_{min} and w_{max} are chosen depending on the total weight of all keys. The difference between w_{min} and w_{max} controls the imbalance allowed. The partitioning algorithm uses the binary digit representation of the integer keys to search for boundary positions. The search starts with the most significant bits and proceeds towards the least significant bits (as in *most-significant-digit-first* radix sort [2]).

The simplified partitioning algorithm is given in Fig. 3. The function PARTITION is executed by all processes in parallel (according to the SPMD model) and every process contributes the keys and weights of its local particles. Variables pos and $size$ describe the local range of keys that are currently considered and are initialized to contain all local keys. The while loop is used to iterate over the bits of the keys and performs an adaptive refinement in each iteration. The while loop stops if there are no bits left for further refinement or if the boundary position is found. The refinement proceeds as follows: First, the number of keys in the current range (given by pos and $size$) with a zero in the r-th bit is determined (line 12). Then the local weight of these keys is determined (line 13) and summed up with a global reduction operation (ALL-REDUCE-SUM) over all processes to determine their global weight w_s (line 14). By comparing the global weight w_s to w_{max} and w_{min}, either the keys with a zero bit are selected for further refinement (line 15), or the other keys are selected for further refinement (line 17). Otherwise, the current weight is between w_{min} and w_{max} and the current position after the keys with a zero bit is chosen as boundary position. All memory requirements of the algorithm are exactly known in advance. The amount

```
1: function PARTITION(n_local, K, W, w_min, w_max)
2:     let: n_local be the number of local keys
3:     let: K be the local array of locally sorted keys
4:     let: W be the local array of weights
5:     let: w_min be the minimum weight for the boundary
6:     let: w_max be the maximum weight for the boundary
7:     /* initialization */
8:     pos = 1 ; size = n_local ; weight = 0 ; done = 0
9:     r = highest bit position
10:    /* search for the boundary position */
11:    while r > 0 and done = 0 do
12:        n_0 = size − Σ_{i=0}^{size−1} GETBIT(K(pos + i), r)
13:        w_0 = Σ_{i=0}^{n_0−1} W(pos + i)
14:        w_s = ALL-REDUCE-SUM(w_0) /* comm. op. */
15:        if weight + w_s > w_max then
16:            size = n_0
17:        else if weight + w_s < w_min then
18:            pos = pos + n_0
19:            size = size − n_0
20:            weight = weight + w_s
21:        else
22:            pos = pos + n_0
23:            done = 1
24:        end if
25:        r = r − 1
26:    end while
27:    return pos
28: end function
```

Figure 3. Simplified partitioning algorithm that divides the local sequence of keys K of each process in two sub-sequences such that the total weight of the first sub-sequences of all processes is between w_{min} and w_{max}. Weights of the keys are given in W. GETBIT(k, r) retrieves the r-th bit of key k and ALL-REDUCE-SUM(w) performs a global reduction operation returning the sum of all values w over all process.

of communication required for the global summation of the weights is constant in every iteration of the while loop. In general, this leads a to lower memory and communication requirement in comparison to the adaptive sampling strategy of the original parallel sorting method in PEPC.

The algorithm in Fig. 3 illustrates the basic idea of the partitioning algorithm. The entire implementation of the partitioning algorithm searches all $p-1$ boundaries at once (instead of using the simplified algorithm $p-1$ times). To reduce the number of iterations of the while loop (and therefore the number of communication operations), several bits of the keys are used at once for refining the search.

C. Optimized Parallel Sorting in PEPC

The optimized parallel sorting method of PEPC uses the partitioning algorithm of Sect. IV-B and combines the key sorting of Step I and the particle data redistribution of Step II into one particle data sorting step. This step requires only one single all-to-all redistribution and proceeds as follows:

1. Sort the local sequence of keys to creating the sort permutation (corresponds to Step Ia).
2. Permute the particles according to the sort permutation and store them in a single intermediate array (corresponds to Step IIa).
3. Use the partitioning algorithm of Sect. IV-B to determine the boundaries (corresponds to Step Ib).
4. Redistribute the particles with an all-to-all communication operation (combines Steps Ic and IIb).
5. Merge the keys to create the merge permutation (corresponds to Step Id).
6. Permute the received particles according to the merge permutation and store them in the target arrays (corresponds to Step IIc).

These steps are performed by all processes in parallel. Only Steps 3 and 4 include communication between the processes.

V. Performance Results

Performance results of the optimized parallel sorting have been obtained on an IBM Blue Gene/P system. A single compute node of the system consists of a 4-way SMP processor with 2 GiB main memory. The *virtual node mode* was used, leading to four processes on each compute node.

Figure 4 shows runtimes of the partitioning algorithm of Sect. IV-B depending on the number of processes for different total numbers of keys. The experiments use random keys with constant weights. The keys are partitioned equally allowing an imbalance of less than 1 %. The runtime results of the partitioning algorithm can be divided into two phases. Starting with a low number of processes, the runtime decreases as the number of processes is increased. This is caused by the decreasing number of keys per process that lead to fewer local operations. However, with increasing numbers of processes, also the number of boundaries that need to be found by the partitioning algorithm increases. This increases the work of the partitioning algorithm and leads to increasing runtimes for large numbers of processes. With lower numbers of keys this increase in runtime starts earlier, but the runtime behavior of the partitioning algorithm is independent from the numbers of keys. With more than 4096 processes the benefits due to lower numbers of local keys on each processes vanish and the differences in runtime for different numbers of keys become rather small.

The optimized particle sorting with partitioning as described in Sect. IV-C was integrated in PEPC. The partitioning algorithm uses the computational load values given by PEPC as weights and partitions the particle data equally allowing an imbalance of less than 1 %. The following results were obtained using the molecular dynamics program version of PEPC (PEPC-E) for benchmark simulations with three time steps and with 6.4 and 25.6 million particles.

Figure 5 shows runtimes for particle data sorting in the 3rd time step depending on the number of processes using the original and the optimized parallel sorting, respectively.

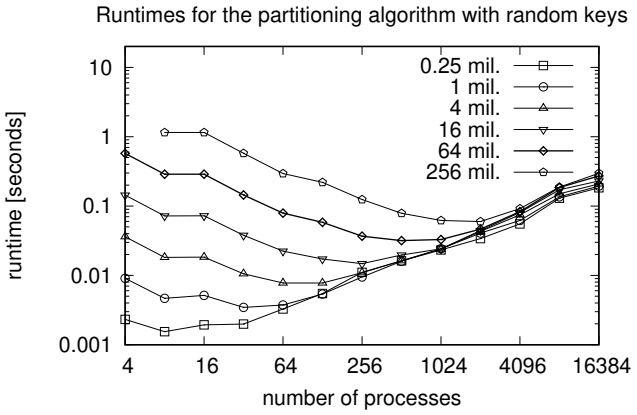

Figure 4. Runtime results of the partitioning algorithm depending on the number of processes with random keys and different total numbers of keys. Results for 4 processes with 256 million keys are missing, because the memory requirements exceeded the memory of the compute nodes.

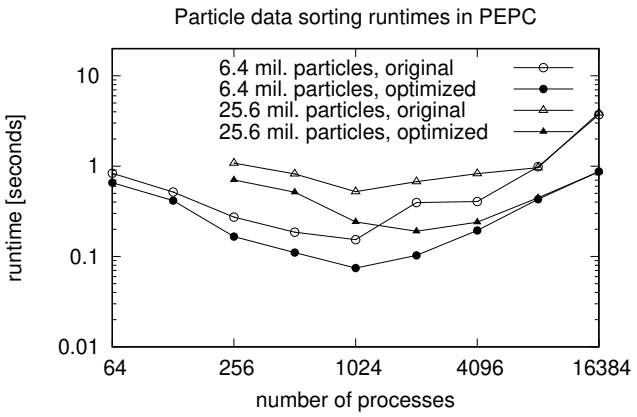

Figure 5. Runtime results of particle data sorting in PEPC (3rd time step) depending on the number of processes. Runtimes are shown for the original parallel sorting (Steps I and II) and the optimized parallel sorting (Steps 1-6) with 6.4 and 25.6 million particles. Results for 64 and 128 processes with 25.6 million particles are missing, because the memory requirements exceeded the memory of the compute nodes.

The optimized method achieves a significant improvement in comparison to the original method for all numbers of processes. With more than 1024 processes, the runtimes of both methods start to increase. The influence of the number of particles vanishes for increasing numbers processes.

Figure 6 shows runtimes of the adaptive sampling (original parallel sorting) and of the partitioning (optimized parallel sorting) for particle data sorting in PEPC (3rd time step, 6.4 million particles) depending on the number of processes. The results show that for low numbers of processes, both adaptive sampling and partitioning require only a small part of the runtime. In these cases, the major part of the runtime of particle data sorting is spend in other steps (e.g., local sort, local merge, and all-to-all redistribution). However, for increasing numbers of processes, the shares of adaptive

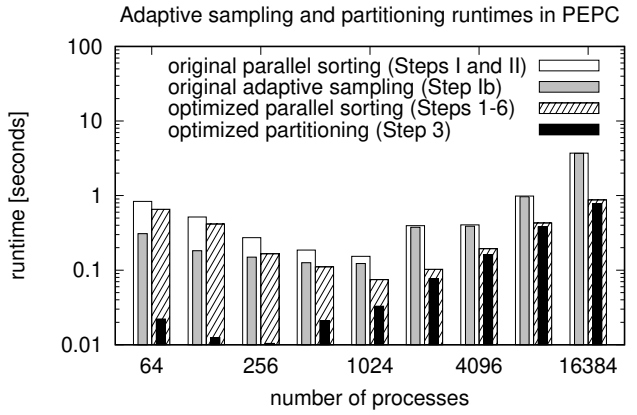

Figure 6. Runtime results of particle data sorting in PEPC (3rd time step, 6.4 million particles) depending on the number of processes. Runtimes are shown for the original parallel sorting (Steps I and II) with the original adaptive sampling and the original adaptive sampling (Step Ib) in isolation as well as the optimized parallel sorting (Steps 1-6) with the optimized partitioning and the optimized partitioning (Step 3) in isolation.

sampling and partitioning increase and they become the major reason of the increasing runtimes in particle data sorting in PEPC. Additionally, the results show that the differences in runtime between the original and the optimized parallel sorting mainly result from the lower runtimes of the partitioning algorithm. This can be attributed to lower costs for searching the boundaries during the partitioning algorithm in comparison to the costs for managing large numbers of sample keys during the adaptive sampling.

Figure 7 shows total runtimes of PEPC simulating three time steps depending on the number of processes using the original and the optimized parallel sorting, respectively. The results show that PEPC itself scales well up to about 4096 processes. Additionally, the optimized parallel sorting leads to a significant improvement in runtime especially for high numbers of processes. The biggest reductions in runtime of about 30 % (with 6.4 million particles) and 26 % (with 25.6 million particles) are achieved with 16,384 processes. However, also with fewer processes a reduction of the total runtime is achieved with the optimized parallel sorting.

Figure 8 shows the reductions in the total runtime of PEPC (simulating three time steps) and in the particle data sorting runtime depending on the number of processes. The results show that in most cases, the reduction of the total runtime exceeds the reduction in the particle data sorting runtime. This shows that, besides the improved particle data sorting, further improvements in the computations of PEPC are achieved. This is caused by an improved load balancing due to a more balanced distribution of the particles with respect to their computational loads. The partitioning algorithm was instructed to allow only 1 % of imbalance. The adaptive sampling of the original particle sorting misses a feasible control of the imbalance and leads to higher load imbalance.

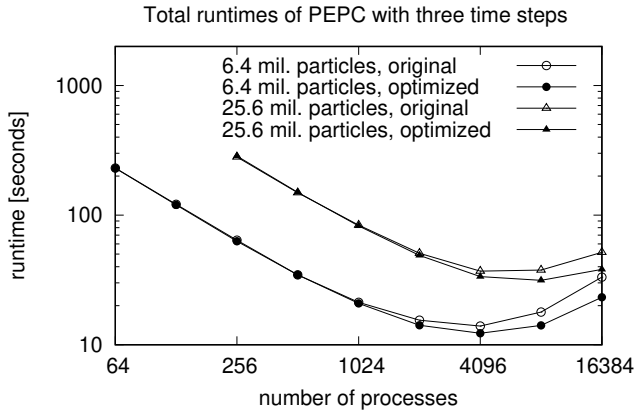

Figure 7. Total runtimes of PEPC simulating three time steps depending on the number of processes using the original and the optimized parallel sorting.

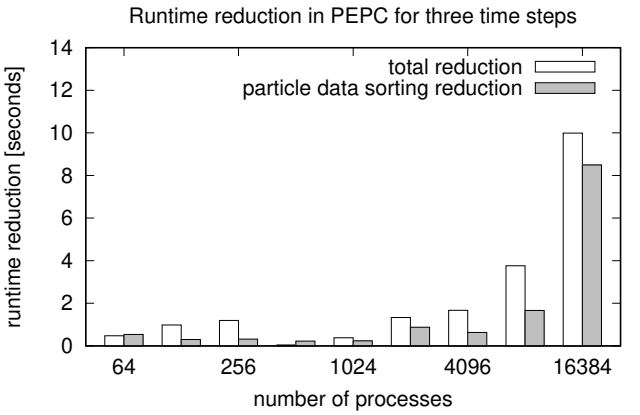

Figure 8. Reductions in the total runtime of PEPC (simulating three time steps) and in the particle data sorting runtime depending on the number of processes. Results represent the differences between using the original and the optimized particle data sorting method.

VI. Summary

In this article, the use of parallel sorting within particle simulations has been considered. Various application-specific demands on parallel sorting algorithms were discussed. Using parallel sorting in particle simulations is a challenging task but provides various opportunities for adaptations and optimizations. As an example for a successful optimization, we presented an improved parallel sorting method based on partitioning with weights for the PEPC application. The results showed that the improved parallel sorting method achieved a performance increase for the parallel sorting step itself as well as for the rest of the application due to an improved load balancing. Especially with large numbers of processes, there was an increasing difference between the original and the optimized parallel sorting method.

Acknowledgment

The measurements were performed at the John von Neumann Institute for Computing, Forschungszentrum Jülich, Germany. http://www.fz-juelich.de/nic

References

[1] R. W. Hockney and J. W. Eastwood, *Computer simulation using particles*. Taylor & Francis, 1988.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.)*. Addison-Wesley, 1998.

[3] S. Akl, *Parallel Sorting Algorithms*. Academic Press, 1990.

[4] D. Bitton, D. J. DeWitt, D. K. Hsaio, and J. Menon, "A taxonomy of parallel sorting," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 287–318, 1984.

[5] D. Richards, "Parallel Sorting - A Bibliography," *SIGACT News*, vol. 18, pp. 28–46, 1986.

[6] E. Solomonik and L. V. Kale, "Highly Scalable Parallel Sorting," in *Proc. of the IPDPS 2010*. IEEE, 2010.

[7] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors," in *PACT '07: Proc. of the 16th Int. Conf. on Parallel Architecture and Compilation Techniques*. IEEE, 2007, pp. 189–198.

[8] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1313–1324, 2008.

[9] Y. Xiaochun, F. Dongrui, L. Wei, Y. Nan, and P. Ienne, "High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs," in *Proc. of the IPDPS 2010*. IEEE, 2010.

[10] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," in *Proc. of the IPDPS 2010*. IEEE, 2010.

[11] "PEPC: A Multi-Purpose Parallel Tree-Code," http://www.fz-juelich.de/jsc/pepc.

[12] IBM Blue Gene Team, "Overview of the IBM Blue Gene/P Project," *IBM J. Res. Dev.*, vol. 52, no. 1-2, pp. 199–220, 2008.

[13] S. Pfalzner and P. Gibbon, *Many-Body Tree Methods in Physics*. Cambridge University Press, 1996.

[14] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[15] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. of Comput. Phys.*, vol. 73, pp. 325–348, 1987.

[16] M. F. Mokbel, W. G. Aref, and I. Kamel, "Analysis of Multi-Dimensional Space-Filling Curves," *Geoinformatica*, vol. 7, no. 3, pp. 179–209, 2003.

[17] M. S. Warren and J. K. Salmon, "A portable parallel particle program," *Comput. Phys. Commun.*, vol. 87, no. 1-2, pp. 266–290, 1995.

[18] H. Dachsel, M. Hofmann, and G. Rünger, "Library Support for Parallel Sorting in Scientific Computations," in *Proc. of the 13th Int. Euro-Par Conf.* Springer, 2007, pp. 695–704.

[19] A. Tridgell and R. Brent, "A General-Purpose Parallel Sorting Algorithm," *Int. J. High Speed Com.*, vol. 7, no. 2, pp. 285–301, 1995.

[20] H. Shi and J. Schaeffer, "Parallel Sorting by Regular Sampling," *J. Parallel Distrib. Comput.*, vol. 14, no. 4, pp. 361–372, 1992.