

Parallel Sorting with Weights for Improving Load Balancing in Particle Simulations

Michael Hofmann* and Gudula Rünger

Department of Computer Science
Chemnitz University of Technology, Germany
{mhofma, ruenger}@cs.tu-chemnitz.de

Abstract Real-world applications, such as particle simulations, often use parallel sorting to reorganize the set of particles or to redistribute data for locality and load balancing concerns. This leads to application-specific demands that are usually not supported by general purpose parallel sorting algorithms. In this article, we present a parallel sorting algorithm for sorting weighted data items in such a way that the resulting distribution of data items among processes is balanced according to weights. The algorithm is used to implement the parallel sorting of particles within a Barnes-Hut tree code. The particles are weighted according to their computational load, thus leading to an improved load balancing after the parallel sorting. Performance results are shown for an IBM Blue Gene/P system.

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

1 Introduction

Efficient particle simulation methods, such as the Barnes-Hut algorithm [2], are used for numerical simulations of complex physical problems in many fields of application. Real-world scientific problems are modelled using large particle systems, thus leading to large scale computational problems and making the use of high performance computer systems inevitable. To achieve an efficient processing of the large amounts of data involved, parallel sorting is often used in parallel particle simulations to redistribute the particles across the processes and to prepare the particle data in such a way that the locality of computations is increased. Therefore, efficient parallel sorting methods for parallel environments are of great importance.

In this article, we propose a parallel sorting algorithm for sorting weighted data items such that the resulting distribution of data items among processes is balanced according to these weights. Weighted data items are data items that have a key value for the sorting and an additional weight value that can be set by an user or an application to control the distribution of data items among

* Supported by Deutsche Forschungsgemeinschaft (DFG).

processes for distributed memory parallel programming. The novel part of this parallel sorting algorithm is the exact partitioning algorithm that determines how the data items need to be redistributed among the processes. The sorting algorithm is used to implement the parallel sorting of particle data within a particle simulation code based on the Barnes-Hut algorithm. The particles to be sorted have key values derived from their spatial positions and weight values that approximate their computational load. Sorting the particles in parallel and balancing their distribution among processes according to the weights can improve the locality and the load balancing of the subsequent computations. The contribution of this article is the development and implementation of a parallel sorting algorithm for weighted data items and the usage of this algorithm for improving the performance of parallel particle simulation codes. Performance results for an IBM Blue Gene/P system demonstrate the good performance of the approach.

The rest of this article is organized as follows. Section 2 presents related work. Section 3 introduces the parallel sorting method for sorting weighted data items. Section 4 shows performance results and Section 5 concludes the paper.

2 Related Work

The sorting problem has been the subject of both theoretical and practical interest for decades, resulting in numerous contributions on sequential and parallel sorting [8,1]. Recent works focus on sorting in high scaling parallel environments [13] and the use of multi-core processors and GPUs [3,9]. However, most contributions focus on generic sorting problems without taking into account specific demands such as the resulting distribution of data items among processes for distributed memory parallel programming. *Probabilistic Splitting* [5] and *Regular Sampling* [12] are fast algorithms that can lead to large imbalances in the worst case. *Exact Splitting* [5] and *Histogram Sort* [7] repeatedly refine the partitioning of the data items until a sufficiently balanced partitioning is achieved. The *Exact Partitioning* algorithm proposed in this article allows the distribution of data items among processes with respect to individual weights of the data items, thus making the parallel sorting algorithm adaptable to other load balancing metrics.

3 Partitioning-based Parallel Sorting with Weights

In this article, a partitioning-based sorting algorithm for distributed memory parallel programming is proposed. Let n be the total number of data items to be sorted. These data items are initially distributed among p parallel processes such that each process $q_i, i \in \{1, \dots, p\}$, owns a local sequence of n_i data items ($\sum_{i=1}^p n_i = n$). The overall parallel sorting algorithm consists of the following steps:

1. Sort the local sequence of data items on each process.

2. Divide the local sequence of data items of each process into p (contiguous) local sub-sequences such that for each $j \in \{1, \dots, p-1\}$, the largest data item of the j -th sub-sequences (of all processes) is not larger than the smallest data item of the $(j+1)$ -th sub-sequences. (For simplicity, we assume that none of the sub-sequences is empty.)
3. Redistribute the sub-sequences of data items with an all-to-all communication operation such that each process $q_i, i \in \{1, \dots, p\}$, receives the i -th sub-sequences from all processes (including its own i -th sub-sequence).
4. Merge the (received) sub-sequences of data items on each process.

In the following subsections, Step 2 is considered in more detail. After summarizing the approach of regular sampling [12] that divides the data items into sub-sequences with the help of splitter keys, we propose an exact partitioning algorithm for weighted data items with integer keys.

3.1 Regular Sampling

For regular sampling, each process selects $p-1$ sample keys from its local data items evenly spaced throughout its local sequence of data items. The sample keys are sent to a designated root process gathering the sample keys of all processes. The root process sorts all $p \times (p-1)$ sample keys and selects $p-1$ splitter keys evenly spaced throughout the sorted sample keys. The splitter keys selected are sent to all processes such that each process can use the splitter keys to divide its local sequence of data items into p local sub-sequences. Using regular sampling for parallel sorting results in an approximately even distribution of data items among processes. Each process will own not more than $2\frac{n}{p}$ data items after the parallel sorting [10].

Gathering and sorting the sample keys at the root process may become a bottleneck for an increasing number of processes. Therefore, we omit the usage of a root process and sort the sample keys in parallel using a parallel sorting method based on sorting networks (see [4] for more details). The resulting sorted sample keys are evenly distributed among p processes (i.e., process q_1 has the lowest $p-1$ sample keys, process q_2 has the next $p-1$ higher sample keys, etc.). Since the $p-1$ splitter keys have to be selected evenly spaced throughout these distributed sample keys, it follows that each process but one has to select one splitter key from its part of the sample keys. Afterwards, all splitters keys determined are sent to all processes.

3.2 Exact Partitioning

The exact partitioning algorithm performs a search for splitting positions (in contrast to splitter keys) that divide the local sequence of data items of each process into sub-sequences. Let $\mathcal{E}_i = \langle x_{i,0}, \dots, x_{i,n_i-1} \rangle$ be the locally sorted sequence of data items of process $q_i, i \in \{1, \dots, p\}$. The division of each local sequence \mathcal{E}_i into p sub-sequences $\mathcal{E}_{i,j}, j = 1, \dots, p$, is then specified by splitting

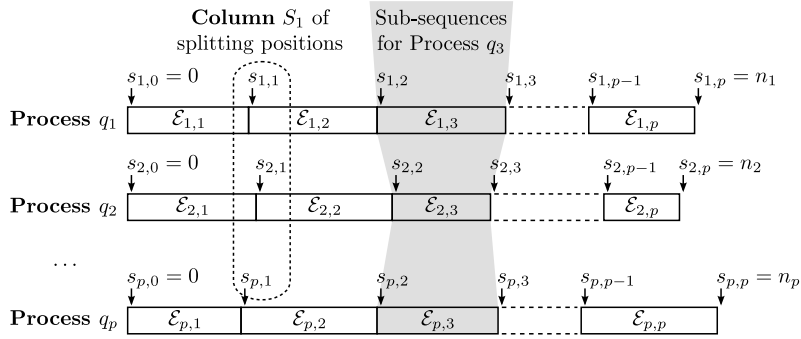


Figure 1. Partitioning of the local sequences of data items into local sub-sequences using splitting positions.

positions $s_{i,j} \in \{0, \dots, n_i\}, j = 0, \dots, p$ (with $s_{i,0} = 0, s_{i,p} = n_i$, and $s_{i,j-1} \leq s_{i,j}$ for $j = 1, \dots, p$), as follows:

$$\mathcal{E}_{i,j} = \begin{cases} \langle x_{i,s_{i,j-1}}, \dots, x_{i,s_{i,j}-1} \rangle & \text{if } s_{i,j-1} < s_{i,j} \\ \text{empty} & \text{otherwise} \end{cases} .$$

Figure 1 illustrates how the splitting positions create a partitioning of the data items. Since the first and last splitting positions are fixed for all process, only $p \times (p - 1)$ splitting positions need to be determined.

For each $j \in \{1, \dots, p-1\}$, let $S_j = \langle s_{1,j}, \dots, s_{p,j} \rangle$ be the column of splitting positions that contains the j -th splitting position of all processes $q_i, i = 1, \dots, p$ (Fig. 1 shows column S_1 as an example). On each process $q_i, i \in \{1, \dots, p\}$, the j -th splitting position $s_{i,j}$ separates the data items of the j -th sub-sequence from the data items of the $(j + 1)$ -th sub-sequence. To achieve that the largest data item of all j -th sub-sequences is not larger than the smallest data item of all $(j + 1)$ -th sub-sequences (as requested in Step 2 of Sect. 3), the splitting positions of a single column S_j have to be chosen cooperatively by all processes. The value of $s_{i,j}$ is equal to the number of data items that process q_i has to send to the processes q_z with $z \leq j$. Thus, summing up the values of all j -th splitting positions yields the total number of data items that all the processes q_z with $z \leq j$ will finally own. To control the resulting distribution of data items among processes, the exact partitioning algorithm determines the splitting positions of each column $S_j, j \in \{1, \dots, p-1\}$ such that $\sum_{i=1}^p s_{i,j}$ is within a given interval $[b_{low}^j, b_{high}^j]$. The boundary constraints b_{low}^j and $b_{high}^j, j \in \{1, \dots, p-1\}$, are used to specify the minimum and maximum number of data items all the processes q_z with $z \leq j$ will own. To get a distribution of data items among processes where each process will own at most $\frac{n}{p} + n_{imba}$ data items, the following boundary constraints can be used:

$$b_{low}^j = j \frac{n}{p} - \frac{n_{imba}}{2} \quad \text{and} \quad b_{high}^j = j \frac{n}{p} + \frac{n_{imba}}{2} \quad \text{for } j = 1, \dots, p-1 \quad . \quad (1)$$

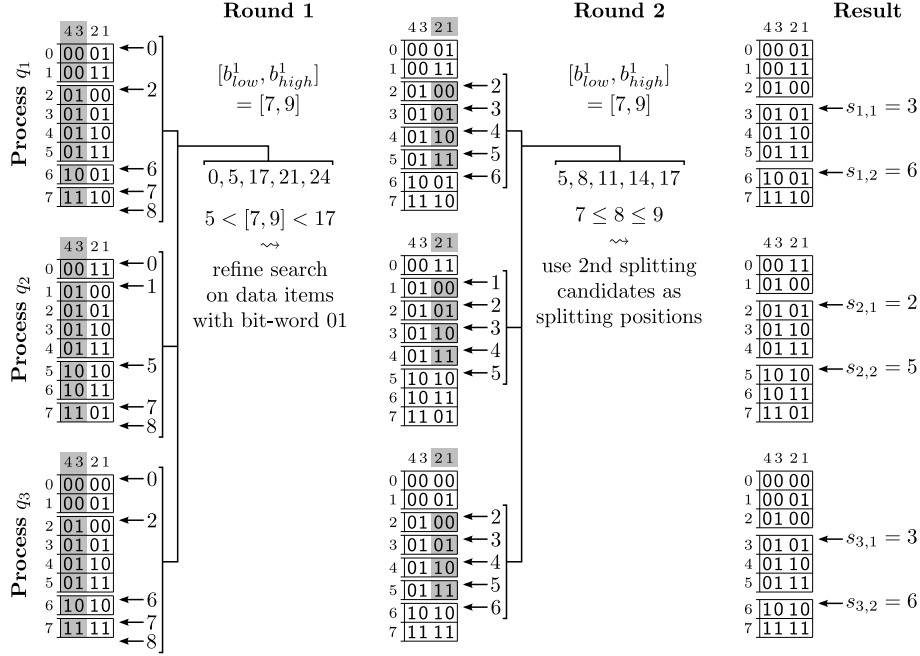


Figure 2. Example for searching splitting positions with the exact partitioning algorithm using a total number of $n = 24$ data items and $p = 3$ processes. Each process has a locally sorted sequence of data items with 4-bit integer keys and $t = 2$ bit positions are used in each search round. Boundary constraints are chosen according to Eq. (1) with $n_{imba} = 2$, i.e., $[b_{low}^1, b_{high}^1] = [7, 9]$. (Details for the search with boundary constraints $[b_{low}^2, b_{high}^2] = [15, 17]$ are not shown.)

3.3 Searching Splitting Positions in Data Items with Integer Keys

First, it is described how all processes cooperatively search for the splitting positions of a single column $S_j, j \in \{1, \dots, p-1\}$. The search proceeds in several rounds and uses the binary digit representation of the integer keys of the data items. Let the data items have r -bit integer keys with bit positions $1, \dots, r$ enumerated from the lowest (rightmost) to the highest (leftmost) bit. Each round of the search uses t bits of the keys ($1 \leq t \leq r$), starting with the highest t bits in the first round. For simplicity, it is assumed that t is a multiple of r and thus $\frac{r}{t}$ search rounds need to be performed at most.

Figure 2 shows an example for searching the splitting positions of column S_1 with $p = 3$ processes and a total number of $n = 24$ data items. The boundary constraints are chosen according to Eq. (1) to get an even distribution of data items among processes with a maximum imbalance of $n_{imba} = 2$ data items, i.e., $[b_{low}^1, b_{high}^1] = [7, 9]$. The data items have 4-bit integer keys and $t = 2$ bit positions are used in each search round.

In the first search round, each process determines $2^t + 1 = 5$ candidates for splitting positions by considering only the two highest bit positions of its keys. The splitting candidates are those positions that cut the sorted sequence of 2-bit words into 4 sub-sequences, each of them containing only equal bit-words. The values of the splitting candidates are summed up separately over all processes (i.e., the sum over all first splitting candidates is created, the sum over all second splitting candidates is created, etc.) and the resulting 5 sum values are compared to the boundary constraints $[b_{low}^1, b_{high}^1]$. Since $[7, 9]$ is within the interval of the second sum of splitting candidates with value 5 and the third sum of splitting candidates with value 17, the search is refined in the next round using the data items with bit-word 01.

In the second round, new splitting candidates and the corresponding sums are determined, now using the two lowest bit positions of the keys and only sub-sequences of the data items. The first sum of splitting candidates has value 8 and is within the given interval $[7, 9]$. Therefore, the search is finished and each process uses its first splitting candidate as its splitting position for column S_1 (i.e., $s_{1,1} = 3, s_{2,1} = 2, s_{3,1} = 3$). The splitting positions determined for column S_2 (i.e., $s_{1,2} = 6, s_{2,2} = 5, s_{3,2} = 6$) are shown only in the final result.

3.4 Weighted Data Items

For weighted data items, the algorithm is adapted as follows: Boundary weight constraints w_{low}^j and $w_{high}^j, j \in \{1, \dots, p-1\}$, are introduced analogously to Eq. (1), but using the total weight of all data items instead of n and the maximum weight imbalance allowed instead of n_{imba} . For each splitting candidate, a corresponding weight value is determined by summing up the weights of all data items from the beginning of the local sequence of data items up to the (potential) splitting position described by the splitting candidate. (If all data items have a constant weight of 1, then the weight value of a splitting candidate is equal to the value of the splitting candidate itself.) The weight values of the splitting candidates are summed up over all processes such that they can be compared to the boundary weight constraints to decide whether the search is finished or need to be refined in the next search round.

Figure 3 shows pseudo-code for the exact partitioning algorithm, that performs the search rounds for all $p-1$ columns of splitting positions together in an interleaved way (i.e., the search rounds using the highest t bit positions are performed for all columns first, then the search rounds using the next lower t bit positions are performed, etc.). Interleaving the search rounds of several columns in this way allows a reuse of splitting candidates and weights that have already been determined. Especially in the first round in which all data items are considered, the same splitting candidates and weights are used for all columns.

Function PARTITION in Fig. 3 is executed by all processes in parallel in the SPMD programming model. Each process $q_i, i \in \{1, \dots, p\}$, uses its locally sorted sequence \mathcal{E}_i of data items as input and returns splitting positions $s_{i,j}, j = 1, \dots, p-1$, as result. All processes use the same boundary weight constraints $[w_{low}^j, w_{high}^j], j = 1, \dots, p-1$, and the same parameter values r and t . The

```

1: function PARTITION( $\mathcal{E}_i$ ) /* executed by all process  $q_i, i = 1, \dots, p$ , in parallel */
2:   let  $i$  be the process number and  $p$  the number of parallel processes
3:   let  $\mathcal{E}_i = \langle x_{i,0}, \dots, x_{i,n_i-1} \rangle$  be the sorted sequence of  $n_i$  data items of process  $q_i$ 
4:   let  $[w_{low}^j, w_{high}^j], j = 1, \dots, p-1$ , be the boundary weight constraints
5:   set  $l_{i,j} = 0$  and  $h_{i,j} = n_i$  for  $j = 1, \dots, p-1$  /* initialization */
6:   for  $l = 1, \dots, \frac{r}{t}$  do /* loop over bit positions */
7:     for  $j = 1, \dots, p-1$  and column  $S_j$  not finished do /* loop over columns */
8:       determine  $2^l + 1$  splitting candidates that cut sequence  $\langle x_{i,l_{i,j}}, \dots, x_{i,h_{i,j}-1} \rangle$ 
9:         into  $2^l$  sub-sequences using the currently considered bit positions
10:      determine the weights of the splitting candidates
11:      sum up the weights over all processes with an ALL-REDUCE-SUM comm. op.
12:      if  $k$  exists with  $[w_{low}^j, w_{high}^j]$  is within the  $k$ -th and  $(k+1)$ -th weight then
13:        /* prepare  $l_{i,j}$  and  $h_{i,j}$  to refine the search in next round */
14:        set  $l_{i,j}$  and  $h_{i,j}$  to the  $k$ -th and  $(k+1)$ -th splitting candidate, respectively
15:      else
16:        /* search for splitting position  $s_{i,j}$  finished */
17:        determine smallest  $k$  such that the  $k$ -th weight is within  $[w_{low}^j, w_{high}^j]$ 
18:        set  $s_{i,j}$  to the  $k$ -th splitting candidate
19:        mark column  $S_j$  as finished
20:      end if
21:      /* failure handling in the last search round */
22:      if  $l = \frac{r}{t}$  and column  $S_j$  not finished then set  $s_{i,j} = \lfloor \frac{1}{2}(l_{i,j} + h_{i,j}) \rfloor$ 
23:    end for
24:  end for
25:  return  $s_{i,j}, j = 1, \dots, p-1$ 
26: end function

```

Figure 3. The function PARTITION implements the exact partitioning algorithm and is executed by p processes in parallel in an SPMD way. Each process $q_i, i \in \{1, \dots, p\}$, uses its locally sorted sequence \mathcal{E}_i of n_i data items as input and returns the splitting positions $s_{i,j}, j = 1, \dots, p-1$, determined.

local variables $l_{i,j}$ and $h_{i,j}$ store the positions of the lower and the upper border of the sub-sequence of data items considered for each column $S_j, j \in \{1, \dots, p-1\}$. The initialization (line 5) ensures that at the beginning of the search all data items are considered. After the initialization, the algorithm performs a loop over the bit positions (l -loop in line 6). In each iteration of the l -loop, a loop over all columns of splitting positions that are not already marked as finished is performed (j -loop in line 7). During the j -loop, one search round is performed for each column S_j using the currently considered bit positions. This includes determining the splitting candidates (line 8) and their weight values (line 9) as well as summing them up over all processes using a global reduction operation called ALL-REDUCE-SUM (line 10). This communication operation leads to a synchronisation of all processes and ensures that the resulting sums of the weights are available on all processes. After comparing the sums of the weights to the boundary weight constraints, either the values of $l_{i,j}$ and $h_{i,j}$ are modified to refine the search in the next round (lines 11–13), or splitting position $s_{i,j}$ is found and

column S_j is marked as finished (lines 15–18). If a splitting position is not found in the final search round, then a failure handling is performed to use the mid position between $l_{i,j}$ and $h_{i,j}$ as splitting position $s_{i,j}$ (in this case, it is possible that the boundary weight constraints are not fulfilled).

The values of all parameters of the algorithm (i.e., r , t , and $[w_{low}^j, w_{high}^j]$, $j = 1, \dots, p-1$) are identical on all processes. Therefore, the conditional statements of the algorithm depend only on the sums of the weights. Since these are available on all processes after the global reduction operation, it follows that all processes perform the algorithm in an identical way. The l -loop and the j -loop perform at most $\frac{r}{t}$ and $p-1$ iterations, respectively. Determining the splitting candidates involves all data items of a process in the worst case. Let $n_{max} = \max_{i=1, \dots, p} \{n_i\}$ be the maximum number of data items of all processes. A splitting candidate can be determined with a binary search in the sorted sequence of data items in time $\mathcal{O}(\log n_{max})$, while determining the weight of a splitting candidate requires time $\mathcal{O}(n_{max})$ to sum up the weights of the data items. Let $T_{ALL-REDUCE-SUM}(p)$ be the time for summing up a single value over p processes using the global reduction operation. Assuming r and t to be constant, the exact partitioning algorithm in Fig. 3 for weighted data items with integer keys requires time

$$\mathcal{O}(p[n_{max} + T_{ALL-REDUCE-SUM}(p)]) \quad .$$

4 Performance Results for Particle Data Sorting

Performance results have been obtained on an IBM Blue Gene/P system [6]. 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. All parallel algorithms were implemented using the *Message Passing Interface* (MPI) library available on the BG/P platform. In the following, results are shown for comparing regular sampling and the exact partitioning algorithm, both in isolation and as part of the parallel sorting step in the parallel particle simulation code PEPC [11]. The boundary (weight) constraints were chosen such that an even distribution of data items among processes with a maximum imbalance of 1% is achieved (e.g., $n_{imba} = 0.01 \frac{n}{p}$).

4.1 Regular Sampling and Exact Partitioning in Isolation

For comparing regular sampling and the exact partitioning algorithm in isolation, a total number of 8 mil. data items with 64-bit integer keys and uniformly distributed random key values have been used. Figure 4 (left) shows runtimes of the exact partitioning algorithm depending on the number t of bits used in each search round with 128 and 1024 processes. The results show that parameter t has a strong influence on the performance. Increasing t reduces the number of search rounds ($\frac{r}{t}$), but increases the number of splitting candidates ($2^t + 1$) and the amount of communication required for the global reduction operations. For

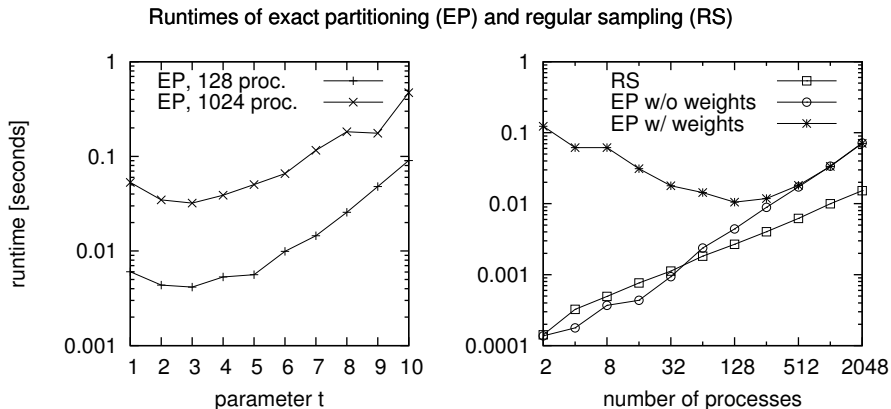


Figure 4. (left) Runtime of the exact partitioning algorithm depending on the number t of bits used in each search round with 128 and 1024 processes. (right) Runtimes of regular sampling (RS) and of the exact partitioning algorithm (EP) with (w/) and without (w/o) weights depending on the number of processes.

larger values of t , the exponential growth of the amount of communication dominates the overall runtime. An optimal value for t can depend on the parallel platform that is used. For the following measurements, we have used $t = 3$.

Figure 4 (right) shows runtimes of regular sampling and of the exact partitioning algorithm with and without weighted data items depending on the number of processes. All weighted data items are considered to have a constant weight of 1, thus adding the overhead for handling weighted data items but without altering the result. The results show that the runtimes of both regular sampling and exact partitioning (w/o weights) increase for increasing numbers of processes. This is the expected behavior, because increasing the number of processes increases the number of splitter keys or splitting positions that need to be determined. For large numbers of processes, regular sampling is significantly faster. The exact partitioning algorithm with weights has significantly higher runtimes for small numbers of processes, because determining the weights of the splitting candidates requires to sum up the weights of all local data items of a process. Hence, there is a benefit when the number of data items per process decreases. However, the runtime decreases only up to about 128 processes and for large numbers of processes, the exact partitioning algorithm with and without weights has almost the same performance.

4.2 Particle Data Sorting

As an example for an application that requires parallel sorting of weighted data items, we use an implementation of the Barnes-Hut algorithm for the calculation of long-range interactions in particle simulations [2]. The parallel implementation used is part of the freely available parallel tree code PEPC. PEPC can be used to simulate the dynamical evolution of a system of particles by calculating

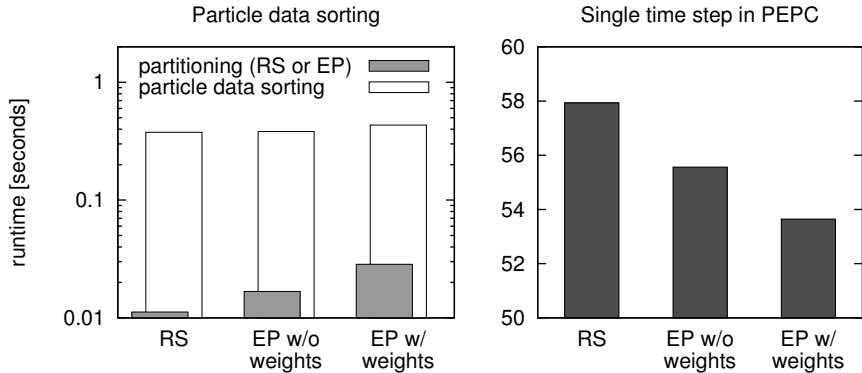


Figure 5. Runtime of particle data sorting (left) and the overall computations (right) for a single time step in PEPC depending on the partitioning algorithm used (regular sampling (RS) or exact partitioning (EP) with (w/) or without (w/o) weights) with 1024 processes and 51.2 mil. particles.

the interactions between particles at discrete time steps (using the Barnes-Hut algorithm) and modifying their positions and velocities accordingly.

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 is based on the *Hashed Oct-Tree* scheme of Warren et al. [14], where the position of a box is encoded in their box number such that the oct-tree of boxes is not stored as hierarchical data structure but as a hash table. The box numbers are recursively created based on their spatial coordinates. The resulting linear ordering of the boxes corresponds to a Morton ordering, which retains the spatial locality of the boxes. All particles are assigned a key according to the number of the box they are located in. By sorting all particles according to their keys, particles that are close to each other (in the particle system) become close to each other in memory. The distributed memory parallel implementation uses parallel sorting to redistribute the particles among the processes such that particles that are close to each other are distributed to the same process (with high probability). In PEPC, each particle is assigned a weight value that approximates the computational load induced by that particle. The overall parallel sorting of the particles is performed as described in Sect. 3, either using regular sampling or the exact partitioning algorithm. The weights of the particles are used to distribute the particles among the processes such that the load balancing is increased.

Figure 5 (left) shows runtimes for sorting the particle data during a single time step of PEPC with 1024 processes and 51.2 mil. particles. Results are shown for regular sampling and for the exact partitioning algorithm with and without using the weights of the particles. It can be seen that partitioning with regular sampling is faster than with the exact partitioning algorithm and using the weights of the particles leads to an additional increase of the runtime. However, the partitioning step itself requires only a small part of the overall runtime for

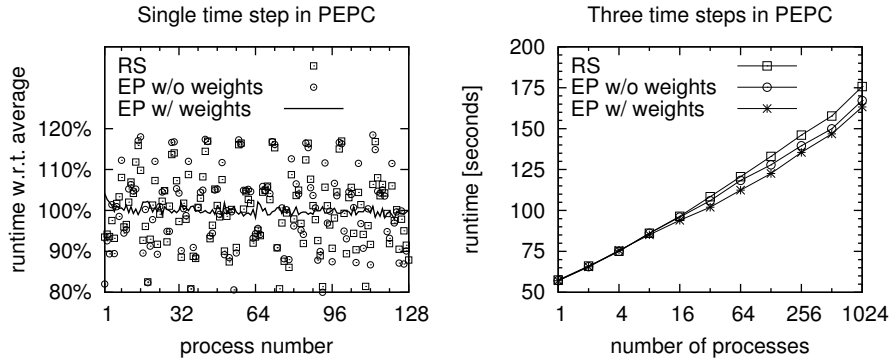


Figure 6. (left) Runtimes of individual processes during a single time step in PEPC for different partitioning algorithms (regular sampling (RS) or exact partitioning (EP) with (w/) or without (w/o) weights) with 128 processes and 6.4 mil. particles. (right) Overall runtime of three time steps in PEPC depending on the number of processes for different partitioning algorithms with 50.000 particles per process.

sorting the particle data. The major part of the runtime is spent for the other steps of the parallel sorting algorithm (local sorting, all-to-all communication, local merging). Figure 5 (right) shows the corresponding runtimes of the overall computations in PEPC for a single time step. The results show that, even though the exact partitioning algorithm itself is more expensive, the overall runtime of the computations decreases.

Figure 6 compares the runtimes of individual processes during a single time step in PEPC with 128 processes and 6.4 mil. particles. Results are shown with respect to the average runtime of all processes. It can be seen that regular sampling and the exact partitioning algorithm without weights lead to an almost identical imbalance in runtime of about $\pm 20\%$. This shows that the improvements between regular sampling and the exact partitioning algorithm without weights are not caused by a better load balancing, but by a more advantageous partitioning of the oct-tree data structure. Using the exact partitioning algorithm with weights decreases the imbalance in runtime down to about $\pm 4\%$ and is therefore the cause of the improvements between with and without weights. Figure 6 (right) shows runtimes for three time steps in PEPC depending on the number of processes with 50.000 particles per process. Especially for increasing numbers of processes, there is a difference in runtime of up to 8% between regular sampling and the exact partitioning algorithm with weights. However, there is a general increase in runtime for increasing numbers of processes, which leads to a parallel efficiency of about 33-35% with 1024 processes.

5 Summary

In this article, we have proposed a parallel sorting algorithm for sorting weighted data items based on a novel exact partitioning algorithm for data items with

integer keys. In comparison to parallel sorting based on regular sampling, the proposed exact partitioning algorithm provides a better way of controlling the resulting distribution of data items among processes, at the cost of a higher runtime for the parallel sorting. As an example application that requires parallel sorting of weighted data items, a parallel particle simulation application was used. The results have shown that the parallel sorting itself requires only a small part of the runtime of the specific application. The additional costs for the exact partitioning algorithm were overcompensated by an improved load balancing, thus leading to a reduction of the overall runtime of the application, especially for large numbers of processes.

Acknowledgment

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

References

1. Akl, S.: Parallel Sorting Algorithms. Academic Press, Inc. (1990)
2. Barnes, J., Hut, P.: A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature* 324, 446–449 (Dec 1986)
3. Chhugani, J., Nguyen, A.D., Lee, V.W., Macy, W., Hagog, M., Chen, Y.K., Baransi, A., Kumar, S., Dubey, P.: Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proc. VLDB Endow.* 1(2), 1313–1324 (2008)
4. Dachselt, H., Hofmann, M., Rünger, G.: Library Support for Parallel Sorting in Scientific Computations. In: *Proc. of the 13th Int. Euro-Par Conf.* pp. 695–704. Springer (2007)
5. DeWitt, D., Naughton, J., Schneider, D.: Parallel sorting on a shared-nothing architecture using probabilistic splitting. In: *PDIS '91: Proc. of the 1st Int. Conf. on Parallel and Distributed Information Systems.* pp. 280–291. IEEE (1991)
6. IBM Blue Gene Team: Overview of the IBM Blue Gene/P Project. *IBM Journal of Research and Development* 52(1-2), 199–220 (2008)
7. Kale, L., Krishnan, S.: A comparison based parallel sorting algorithm. In: *ICPP '93: Proc. of the 1993 Int. Conf. on Parallel Processing.* pp. 196–200. IEEE (1993)
8. Knuth, D.E.: *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley (1998)
9. Leischner, N., Osipov, V., Sanders, P.: GPU sample sort. In: *Proc. of the IPDPS 2010.* IEEE (2010)
10. Li, X., Lu, P., Schaeffer, J., Shillington, J., Wong, P.S., Shi, H.: On the versatility of parallel sorting by regular sampling. *Parallel Comput.* 19(10), 1079–1103 (1993)
11. PEPC: A Multi-Purpose Parallel Tree-Code (benchmark version 1.4), <http://www.fz-juelich.de/jsc/pepc>
12. Shi, H., Schaeffer, J.: Parallel sorting by regular sampling. *J. Parallel Distrib. Comput.* 14(4), 361–372 (1992)
13. Solomonik, E., Kale, L.V.: Highly Scalable Parallel Sorting. In: *Proc. of the IPDPS 2010.* IEEE (2010)
14. Warren, M.S., Salmon, J.K.: A parallel hashed Oct-Tree N-body algorithm. In: *Proc. of the 1993 ACM/IEEE Conf. on Supercomputing.* pp. 12–21. ACM (1993)