Three-Dimensional Spatial Join Count Exploiting CPU Optimized STR R-Tree

Ryuya Mitsuhashi*, Hideyuki Kawashima†, Takahiro Nishimichi‡, Osamu Tatebe†
*Graduate School of Systems and Information Engineering, University of Tsukuba
Email: mitsuhashi@hpcs.cs.tsukuba.ac.jp
†Center for Computational Sciences, University of Tsukuba
Email: {kawasima,tatebe}@cs.tsukuba.ac.jp
‡ Kavli Institute for the Physics and Mathematics of the Universe
Email: takahiro.nishimichi@ipmu.jp

***Abstract*—In this study, we attempt to address the issue regarding the spatial join count, where in the number of particles around a halo is counted only once for a given simulation result. An efficient spatial index is necessary for accelerated counting; therefore, we propose a CPU optimized sort-tile-recursive R-tree that employs a parallel radix sort and node packing with thread pool and single instruction multiple data instructions. In an experiment conducted with astronomical data, the proposed method demonstrates an improvement in performance by 26.8 times compared with that using a conventional CPU optimized R-tree. We also propose a partial materialization approach to handle large amount of data that exceeds the capacity of main memory. To accelerate the approach, we propose a construct-search-destruct pipeline that exploits a thread pool to conceal the latency of the construction and destruction of the index. The pipelining method achieves an improvement in performance by 27.5 times compared with that of a conventional CPU optimized R-tree. All our codes are available on GitHub.**

***Keywords*-Spatial join count, STR R-tree, Periodic boundary condition, SIMD**

I. Introduction

A. Spatial Join Count over Shells

Spatial join [1] is an important query used in a number of fields such as astronomy, neuroscience, and geology. For example, in astronomy the spatial join appears in queries for identifying stars [2], finding close celestial bodies [3], and counting neighboring astronomical objects [4]. For a spatial join on two relations R and S containing spatial objects, if the distance between an object from R and that of another object from S is less than a predefined threshold, such a pair of objects is included in the join result. The cost of the spatial join is $O(MN)$ when R and S contain M and N objects respectively.

In astronomy, a number of simulations are conducted to analyze the state of the universe [5][6] and find an unknown or a rare phenomenon. Spatial join is frequently used in such analysis. In this study, we focus on a variant of the spatial join count in proposed by Taruya et al. [4].

This type of spatial joint count involves counting the number of particles in each shell of a three-dimensional Euclidean space. Each shell has a halo at its center. The halo is a celestial sphere-like object, represented by its center point. Only the object ID and point information are required in this workload.

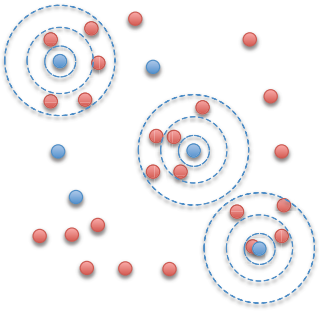

Fig. 1. SJCS. Red and blue points represent particles and halos, respectively. Circles spreading out from the blue points are shells.

```
SELECT SH1.id, SH1.count, SH2.count, SH3.count
FROM (SELECT HALO.id, COUNT(*) FROM HALO, PART
      WHERE distance(HALO.p, PART.p) < THRESHOLD1
      GROUP BY HALO.id) AS SH1,
     (SELECT HALO.id, COUNT(*) FROM HALO, PART
      WHERE distance(HALO.p, PART.p) >= THRESHOLD1
        AND distance(HALO.p, PART.p) < THRESHOLD2
      GROUP BY HALO.id) AS SH2,
     (SELECT HALO.id, COUNT(*) FROM HALO, PART
      WHERE distance(HALO.p, PART.p) >= THRESHOLD2
        AND distance(HALO.p, PART.p) < THRESHOLD3
      GROUP BY HALO.id) AS SH3
```

Fig. 2. SQL query for SJCS

Figure 1 represents an overview of this workload, and Fig. 2 shows a structured query language (SQL) query for this workload, referred to as spatial join count over shells (SJCS) in this study.

B. Problems with SJCS

The query shown in Fig. 2 is inefficient because it contains many relational joins. These joins can be merged into a single join that can be processed naturally in the following three steps. First, we construct an index for particles in a simulation result. Second, for each halo, we find the largest shell using a range search within the index. Finally, we count the number of particles in each shell. In this approach, we need to consider the following two problems.

1) **Expensive index construction**:

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

The index cannot be constructed in advance because SJCS is executed **only once** and is applied immediately after the simulation data are generated. Therefore, we require a fast search index that can be constructed in a computationally inexpensive manner.

2) **Large data exceeding memory capacity**:
Coordinates of only particle data are required to construct the index. However, the number of particles is large (8×10^9), making the size of the index greater than 180 GB.
It is difficult to maintain such a large index in the main memory; hence, multiple storage accesses are required. However, this reduces performance because a number of write and read accesses occur during the index construction phase and search phase, respectively.

C. Contributions

We propose a CPU-optimized sort-tile-recursive R-tree (**CoSTR-R-tree**) that dramatically accelerates the SJCS workload. Note that all codes are available on GitHub [7]. The proposed method solves the above problems as follows.

1) **Parallel index construction**: In the SJCS workload, the number of dimensions for astronomical objects is three. That is, the positions of the halos and the particles are represented by x, y and z coordinates. In such low-dimensional space, we can exploit R-tree variants effectively without encountering the curse of dimensionality. Thus, we are able to accelerate the STR R-tree [8], and achieve an improvement in performance by **26.8 times** with a small SJCS workload (Section V).
The STR R-tree construction consists of a sort and pack phases. We employ an efficient parallel radix sort [9], which is a CPU cache-conscious multi-threaded $O(N)$ sorting technique, to accelerate the sort phase. We parallelize the pack phase with multi-threading and Intel Streaming SIMD Extensions (SSE) intrinsics (`_mm_min` and `_mm_max`). In addition, we employ thread pool pattern construction to reduce the overheads of thread-forking and thread-joining.
2) **Partial Materialization**:
To solve the memory shortage problem, we adopt a partial materialization approach that divides particle data into multiple segments and then applies the SJSC to each segment repeatedly. To accelerate this approach, we propose a construct-search-destruct (CSD) pipeline. The pipeline exploits the thread pool to conceal the latency of the construction and destruction of the index. The pipelining method with a CoSTR-R-tree achieves an improvement in performance by **27.5 times** (Section VI).

D. Organization

The remainder of this paper is organized as follows. Section II describes related work. In Section III, we present the CoSTR-R-tree and describe its concept, design, and implementation. Section IV describes the details of the SJCS workload, including the periodic boundary condition. Section V presents a performance evaluation of the CoSTR-R-tree with an SJCS workload with real data and uniformly randomly generated data compared with a CPU-optimized R-tree (CoR-tree). In Section VI, we present and evaluate partial materialization for large-scale particle data that exceed memory capacity. Conclusions are given in Section VII.

II. Related Work

A. R-Tree and its Variants

An R-tree [10] is a tree shaped index that can be used to search spatial objects efficiently. In this paper, we treat three-dimensional data; therefore, we do not require indices for high-dimensional data, such as an SS-tree [11], an SR-tree [12], an A-tree [13], an X-tree [14], a VA-File [15], or a VA+-File [16]. R-tree variants are sufficient for our purposes.

The R-Tree structure has been studied relative to performance improvement. The node splitting principal of the R-tree minimizes the area of split nodes. It does not consider overlap between nodes; therefore, performance deteriorates depending on the data distribution. The R*-tree [17] was designed to solve this problem. The R*-tree minimizes the overlap between nodes and the area of nodes. The search performance of the R*-tree is more efficient than that of an R-tree.

Typically, the R-tree and the R*-tree are dynamic structure, i.e., they change their shape relative to data insertions or deletions. However, R-trees can also be constructed in a static manner. A static R-tree index loads all data simultaneously.

The time required to construct an index is much less than that required by dynamic variants.

The Packed R-tree [18] is the simplest statically structured R-tree. It takes only the x-coordinate; therefore, it shows poor search performance. The Hilbert Packed R-tree [19] uses the Hilbert value (value of a space-filling curve) of its centroid to sort records. It shows better performance than the Packed R-tree and the R*-tree. The STR R-tree [8] is an improved version of the Packed R-tree. It sorts data from first to last coordinates recursively. It was reported that the STR-R-tree outperforms the Hilbert Packed R-tree in most cases [8].

B. STR R-Tree

STR divides records into tiles (groups of records) and sorts recursively for each dimension. We briefly describe the construction procedure in the following.

1) **Input records**: Let the number of children in a node be C, the number of records be N, and the dimension of data be D. In the STR R-tree, D times sorts are performed recursively.
2) **Sorting tiles**: Records are sorted by the $(D-k)$-th axis, where k is the number of remaining sorts and initialized by D. Note that the axis number starts at 0. If k is equal to 0, proceed to step 4.
3) **Recursive sort**: The sorted records are divided into $\lceil \lceil N/C \rceil^{1/k} \rceil$ tiles. Each tile has $C\lceil \lceil N/C \rceil^{(k-1)/k} \rceil$ records.

Let k be $k-1$. Step 2 is applied to the records in each tile.

4) **Pack nodes**: Records are divided into $\lceil N/C \rceil$ groups. Parent nodes are created from those groups. Let the number of records N be $\lceil N/C \rceil$. If N is greater than C, repeat step 2 onwards with the parent nodes. Otherwise, create the root node and terminate this procedure.

Although a GPU optimized STR R-tree has been proposed by Simin, et al. [20], to the best of our knowledge, a CPU optimized version has not yet been proposed.

III. Three-Dimensional CPU-Optimized STR R-Tree

As described above, construction of the STR R-tree consists of sort and pack phases.

The sort phase sorts records recursively to minimize overlaps between nodes and the pack phase packs records and creates the parent nodes of the records.

R-tree variants variant searches involve tree traversal and determining if an object exists within a query range (Section II-A). These procedures have significant parallelisms. This section explains how we optimize these procedures.

A. Parallel Construction

1) Overview of Construction: The sort phase is the primary process in index construction. For practical applications, sorting must be executed quickly. The pack phase can also be parallelized because each relation between a parent node and its child nodes is independent of each other. To parallelize the pack phase, each thread obtains C nodes from an array of nodes and the destination index of the array and then creates the parent node at the destination. When a parent node is created, its minimum bounding rectangle (MBR), which encloses all its child nodes, is set. In this process, we can use SSE intrinsics, `_mm_min` and `_mm_max`, to obtain the minimum and maximum values for each coordinate.

In the construction of the three-dimensional CoSTR-R-tree, there are iterations of three parallel sorts and one parallel node packing. If we use the fork and join thread model, there are four forks and four joins per iteration. To avoid overhead, we use a thread pool mechanism. In the parallel radix sort, worker threads are symmetric, i.e., all threads do the same work. The same can be said for parallel node packing. Based on these facts, we employ a **uniform thread pool**. With a uniform thread pool, a single request is processed by all threads in the pool, i.e., there is no work request queue. This eliminates the overhead associated with managing a work request queue, thus improving performance.

2) Details of Construction: The CoSTR-R-tree is mapped on a single in-memory array, which facilitates easy parallelization of the construction. To allocate an array that can contain the entire index, it is necessary to calculate the number of nodes required in advance. Here, let the number of records be N, the maximum number of children a node can hold be C and the total number of required nodes be M. M can be calculated as follows.

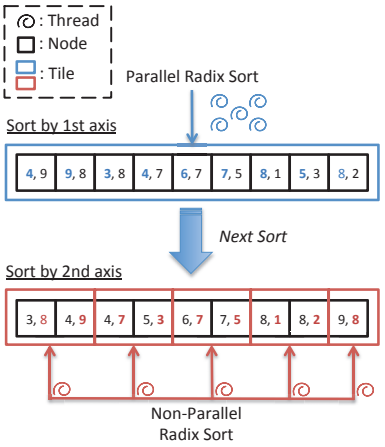

Fig. 3. Recursive sort implementation (numbers in the boxes represent x and y coordinates).

$$P_{n+1} = \lceil P_n/C \rceil \cdots (P_0 = N), M = \sum_{k=0}^{\lceil \log_C N \rceil} P_k \qquad (1)$$

Algorithm 1 shows the CoSTR-R-tree construction procedure.

1) **Input records**: Let the number of input records be N, and the number of child nodes in a node be C (Algorithm 1, lines 2 and 3, respectively).
2) **Create leaf nodes**: We calculate the total number of nodes M and allocate the array of M nodes. Then, we create N leaf nodes from the input records and place them into the array (Algorithm 1, lines 6-8).
3) **Sort by first axis** (**parallel**): Previously generated nodes are sorted by the first axis using parallel radix sort (Algorithm 1, line 12). Note that this single sort is parallelized.
4) **Sort by second axis and third axes** (**parallel**): Sorted nodes are divided into $\lceil N/C \rceil^{1/3}$ tiles (Algorithm 1, line 14). Nodes in each tile are sorted by the second axis using a radix sort. Note that multiple sorts are performed simultaneously. Then, the sorted nodes in each tile are divided into $\lceil N/C \rceil^{1/2}$ sub-tiles, and sorted by the third axis with a radix sort (Algorithm 1, at line 16).
5) **Pack nodes and tree growth** (**parallel**): Sorted nodes are divided into $\lceil N/C \rceil$ groups (Algorithm 1, line 18). Each thread is provided with a group and the destination offset of the array to write the parent node. Then they create the parent node from the group and write it to the destination.
6) **Repeat**: Let the number of records N be $\lceil N/C \rceil$. If N is 1, create the root node, then terminate the construction. Otherwise, let the parent nodes generated in the previous step be the new input. Then, repeat step 3 onwards.

Note that parallel radix sort is used in step 3; however, it is not used in step 4. Figure 3 illustrates how we implement these sorts. In the upper part of Fig. 3, a parallel radix sort is applied to the first sort, while in the lower part, the second

Algorithm 1 CoSTR-R-tree construction

```
1: A[] ← array of data
2: N ← number of data
3: C ← max number of children in a node
4: T ← thread pool
5: function main(A, N, C, T)
6:     M ← calculate total number of nodes from N
7:     allocate array of nodes NA[M]
8:     create leaf nodes from A[] and put them into NA[M − N] to
       NA[M − 1]
9:     NP ← pointer to NA[M − N]
10:    while N > C do
11:        {parallel radix sort by first axis using thread pool T}
12:        ParallelRadixSort(NP, N, 0, T)
13:        {tile-by-tile radix sort by second axis using thread pool T}
14:        RadixSortTileByTile(NP, N, 1, ⌈N/C⌉^{2/3} × C, N, T)
15:        {tile-by-tile radix sort by third axis using thread pool T}
16:        RadixSortTileByTile(NP, N, 2, ⌈N/C⌉^{1/2} × C, ⌈N/C⌉^{2/3} ×
           C, T)
17:        {parallel node packing using thread pool T}
18:        ParallelPackNodes(NP, N, C, T)
19:        NP ← NP − ⌈N/C⌉
20:        N ← ⌈N/C⌉
21:    end while
22:    {pack nodes and create the root node}
23:    PackChildNodes({NA[1], ..., NA[N]}, NA[0])
24: end function
```

Algorithm 2 Tile-by-tile sorting

```
1: NP ← pointer to array of nodes
2: NN ← number of nodes
3: K ← axis number
4: W ← number of elements in a tile
5: M ← number of elements in a tile of K − 1th axis
6: T ← thread pool
7: function RadixSortTileByTile(NP, NN, K, W, M, T)
8:     # begin parallel by T, share: n ← 0
9:     while all tiles have not been sorted do
10:        {get the range of tile for this thread}
11:        s ← n {n is locked here}
12:        if s ≥ NN then
13:            {all tiles have been sorted, unlock n}
14:            return to thread pool T
15:        end if
16:        e ← s + W − 1
17:        {fix the end of tile region}
18:        if e ≥ NN then
19:            e ← NN − 1
20:        else if there is multiple of W in s, ..., e then
21:            e ← (multiple of W that does not exceed e) −1
22:        end if
23:        n ← e + 1 {n is unlocked here}
24:        {radix sort tile by Kth axis}
25:        RadixSort({NP[s], ..., NP[e]}, e − s + 1, K)
26:    end while
27:    # end parallel by T
28: end function
```

sort is performed concurrently using a tile-by-tile radix sort by assigning a thread to each tile. A parallel radix sort is not used for the second and third sort due to thread synchronization overhead. For example, if $N = 10^8$ and $C = 10$, there are $\lceil N/C \rceil^{1/3} = 216$ tiles in the second axis sort. Each tile has $\lceil N/C \rceil^{2/3} \times C = 464160$ nodes[1]. Applying parallel radix sort to each tile would require 216 parallel radix sorts. This is inefficient because parallel radix sort has thread barriers; thus, the total number of thread barriers becomes at least 512.

Algorithm 2 shows tile-by-tile multi-threaded radix sort that corresponds to step 4. Each thread gets the tile on which they perform the sort (Algorithm 2, lines 11-22). Note that the nodes in the tile are sorted (Algorithm 2, line 25).

Algorithm 3 shows multi-threaded node packing that corresponds to step 5. Each thread obtains the destination offset of the array for writing the parent node (Algorithm 3, line 9-10)). Each thread obtains the group to be the children of the parent node (Algorithm 3, line 12-20). This divides the nodes into $\lceil N/C \rceil$ groups. Note that each group has C nodes. Then, we create the parent nodes for these groups (Algorithm 3, line 23).

Algorithm 4 shows node packing used in Algorithms 1 and 3 that creates a parent node. The MBR of the parent node is created (Algorithm 4, lines 5-10). To find the lowest coordinate and the highest coordinate from the child nodes' MBR, `_mm_min` and `_mm_max` are used, respectively (Algorithm 4, lines 8-9). They are SSE intrinsics that obtain the minimum and maximum values for each element from the two vectors, respectively.

[1]The last tile may have fewer nodes than this.

B. Parallel Range Search in Spatial Join

1) Overview of Search: This subsection explains how we apply the CoSTR-R-tree to the spatial join. Since there are many overlap checks and distance calculations in the search, SIMD instructions can be effective. Furthermore, there is search parallelism in spatial joins, one side of the data is indexed and the other side of the data becomes the search keys. These searches are independent of other searches; therefore, it is easy to parallelize searches with multi-threading.

2) Details of Search: Algorithm 5 shows the search procedure of the spatial join with the CoSTR-R-tree.

1) **Initialization**: Let the query radius be R (Algorithm 5, line 3).
2) **Create query MBR** (**parallel**): To search the certain range from an object with the CoSTR-R-tree, the query MBR is created (Algorithm 5, line 9).
3) **Issue query** (**parallel**): We traverse the CoSTR-R-tree from the root node with the query MBR, and get the objects in the specified range (Algorithm 5, line 12).

Algorithm 6 query MBR creation in step 2. Here, let the query radius be R and the query point be $\boldsymbol{P}$. Then, a query MBR M is created as follows. Set the lower point of the MBR to $\boldsymbol{P} - R$ and the upper point to $\boldsymbol{P} + R$ (Algorithm 6, lines 5-7). This calculation is performed by `_mm_sub` and `_mm_add` which are the intrinsics for vector subtraction and the addition, respectively.

Algorithm 7 shows range search in step 3. If the node overlaps the query MBR, we descend the node recursively. When a leaf node is reached, children of the leaf node are

Algorithm 3 Parallel packing

```
1: NP ← pointer to array of nodes
2: NN ← number of nodes
3: C ← max number of children in node
4: T ← thread pool
5: function ParallelPackNodes(NP, NN, C, T)
6:     # begin parallel by T, share: n ← 0, p ← 1
7:     while all nodes have not been packed do
8:         {get offset of parent node}
9:         q ← p {p is locked here}
10:        p ← p + 1 {p is unlocked here}
11:        {get nodes to pack}
12:        s ← n {n is locked here}
13:        if s ≥ NN then
14:            {all nodes have been packed, unlock n}
15:            return to thread pool T
16:        end if
17:        e ← s + C − 1
18:        if e ≥ NN then
19:            e ← NN − 1
20:        end if
21:        n ← e + 1 {n is unlocked here}
22:        {pack nodes and create the parent node}
23:        PackChildNodes({NP[s], ..., NP[e]}, NP[−q])
24:    end while
25:    # end parallel by T
26: end function
```

Algorithm 4 Node Packing

```
1: NC[] ← array of child nodes
2: DP ← destination of parent node
3: function PackChildNodes(NC, DP)
4:     {create MBR of parent node}
5:     M ← MBR of NC[0]
6:     for all nc: nodes in NC[] do
7:         m ← MBR of nc
8:         lower point of M ← _mm_min(lower point of M, lower
           point of m)
9:         upper point of M ← _mm_max(upper point of M, upper
           point of m)
10:    end for
11:    set NC[] for the children of DP, M for the MBR of DP
12: end function
```

checked to filter false alarms. If the distance is less than the query radius R, the object is added to the results.

Algorithm 8 shows how we check for MBR overlaps in Algorithm 7. Here, let the query MBR be $M1$, and the MBR of a node be $M2$. It checks the following condition to determine that there are no overlaps between those MBRs.

- If the lower point of $M1$ ($M2$) is not less than the upper point of $M2$ ($M1$), then there are no overlaps.

If the above condition is not satisfied, they overlap. The SSE intrinsics can also handle vector comparison and branch by `_mm_cmpnlt` and `_mm_test_all_ones`. The `_mm_cmpnlt` intrinsic performs the "not less than" operation for each element of two vectors and returns the vector of the comparison result. We must know whether there is an element that satisfies one of the above conditions to prove that there are no overlaps.

To determine this, we use the `_mm_test_all_ones` compound intrinsic, which returns true if all elements in the vector are 0. If it returns false, a value of 1 exists in the vector and we know that the non-overlapping condition is satisfied.

Algorithm 5 CoSTR-R-tree spatial join

```
1: ST ← STR-R-tree
2: A[] ← array of data point
3: R ← search range
4: T ← thread pool
5: function main(ST, A, R, T)
6:     # begin parallel by T, assume there is exclusive control to
       fetch p
7:     for all p: data point in A[] do
8:         NR ← get root node of ST
9:         M ← create a MBR
10:        CreateQueryMBR(M, p, R)
11:        {we do not want to calculate the square root of the distance
               for every object, so take the square of R in advance}
12:        RangeSearch(NR, M, p, R^2)
13:    end for
14:    # end parallel by T
15: end function
```

Algorithm 6 Query MBR creation

```
1: M ← query MBR
2: P ← query point
3: R ← search range
4: function CreateQueryMBR(M, P, R)
5:     VR ← 3-dimensional vector of R
6:     lower point of M ← _mm_sub(P, VR)
7:     upper point of M ← _mm_add(P, VR)
8: end function
```

Algorithm 9 shows the distance calculation in Algorithm 7. The difference of two vectors is calculated using `_mm_sub` (Algorithm 9, line 5). Then, the square of the difference for each element is summed with `_mm_dp` (Algorithm 9 , line 6), which is the dot product intrinsic. Here `_mm_dp` returns the vector of the distance. Thus, we must convert it to a scalar

Algorithm 7 Range search

```
1: NN ← node of STR-R-tree
2: M ← query MBR
3: P ← query point
4: R ← search range
5: function RangeSearch(NN, M, P, R)
6:     if NN is not leaf node then
7:         for all nc: child nodes of NN do
8:             m ← MBR of nc
9:             if CheckOverlap(M, m) returns true then
10:                {descend the child node}
11:                RangeSearch(nc, M, P, R)
12:            end if
13:        end for
14:    else
15:        p ← lower point of MBR of NN
16:        if CalculateDistance(P, p) < R then
17:            {this object is truly included in range R}
18:            add object pointed to by NN to the result
19:        end if
20:    end if
21: end function
```

Algorithm 8 Overlap checking

```
1: M1 ← a MBR
2: M2 ← a MBR
3: function CheckOverlap(M1, M2)
4:     F1 ← _mm_cmpnlt(upper point of M1, lower point of M2)
5:     if _mm_test_all_ones(F1) returns false then
6:         return false
7:     end if
8:     F2 ← _mm_cmpnlt(upper point of M2, lower point of M1)
9:     if _mm_test_all_ones(F2) returns false then
10:        return false
11:    end if
12:    return true
13: end function
```

Algorithm 9 Distance calculation

```
1: P1 ← a point
2: P2 ← a point
3: {this function returns the square of the distance}
4: function CalculateDistance(P1, P2)
5:     VS ← _mm_sub(P1, P2)
6:     VD ← _mm_dp(VS, VS)
7:     {distance is in VD as vector so extract it}
8:     return _mm_cvt(VD)
9: end function
```

value. `_mm_cvt` (Algorithm 9, line 8) executes the vector to scalar conversion.

IV. Applying CoSTR-R-tree to SJCS Workload

A. Periodic Boundary Condition

The periodic boundary condition is required for the SJCS workload. This is the idea that space consists of repetitions of the same cells. This condition is widely used in scientific simulations, such as fluid dynamics [21] and astronomy [22], because it is very difficult or nearly impossible to handle the whole data space.

There are two approaches to count the appropriate number of particles with this condition.

1) (A1) Hold the extra cells that surround the original cell (excluding the original cell). Then construct the index for this chunk of cells.
2) (A2) Hold only the original cell and construct its index. When the query range sticks out of the cell, some queries that are equivalent to those for adjacent cells are issued.

Figure 4 illustrates these two approaches. With the first principle, the number of extra cells that surround the center is $2\Sigma_{k=1}^{D} 3^{k-1}$, where D is the dimension of the data. This requires large amount of memory. However, we cannot tolerate this drawback because the data size is huge in our problem. The second principle requires no extra cells. However, it requires 2^D times searches at most [2].

In the second principle, there are no objects in the out-of-bound region for each query (Fig. 4, right); therefore, the cost of each query is relatively small. Thus, the reduced search performance caused by the multiple query issue is permissible. We employ the second principle to apply the periodic boundary condition to the search with the CoSTR-R-tree. To check of the out-of-bound condition faster, we use `_mm_cmplt` and `_mm_cmpgt` to determine if query MBRs are beyond the lower and upper bounds in each dimension.

[2]We assume the query MBR is smaller than the cell.

Algorithm 10 CoSTR-R-tree spatial join count over shells

```
1: ST ← STR-R-tree
2: D ← dimension of data
3: A[] ← array of data point
4: S[] ← shells
5: T ← thread pool
6: function main(ST, A, S, T)
7:     # begin parallel by T, assume there is exclusive control to
       fetch p
8:     for all p: data point in A[] do
9:         NR ← get root node of ST
10:        M ← create a MBR
11:        CreateQueryMBR(M, p, radius of outer most S[])
12:        LF[] ← lower out of bound flags
13:        UF[] ← upper out of bound flags
14:        CheckOutOfBound(LF, UF, M)
15:        for i ← 0 to D − 1 do
16:            q ← copy of p
17:            if LF[i] is true then
18:                q[i] + coordinate of upper bound
19:            else if UF[i] is true then
20:                q[i] − coordinate of upper bound
21:            else
22:                continue this loop
23:            end if
24:            CreateQueryMBR(M, q, radius of outer most S[])
25:            RangeCountOverShells(NR, M, q, S)
26:            for j ← i + 1 to D − 1 do
27:                r ← copy of q
28:                if LF[i] is true then
29:                    r[i] + coordinate of upper bound
30:                else if UF[i] is true then
31:                    r[i] − coordinate of upper bound
32:                else
33:                    continue this loop
34:                end if
35:                CreateQueryMBR(M, r, radius of outer most S[])
36:                RangeCountOverShells(NR, M, r, S)
37:                for k ← j + 1 to D − 1 do
38:                    s ← copy of r
39:                    if LF[i] is true then
40:                        s[i] + coordinate of upper bound
41:                    else if UF[i] is true then
42:                        s[i] − coordinate of upper bound
43:                    else
44:                        continue this loop
45:                    end if
46:                    CreateQueryMBR(M, s, radius of outer most S[])
47:                    RangeCountOverShells(NR, M, s, S)
48:                end for
49:            end for
50:        end for
51:    end for
52:    # end parallel by T
53: end function
```

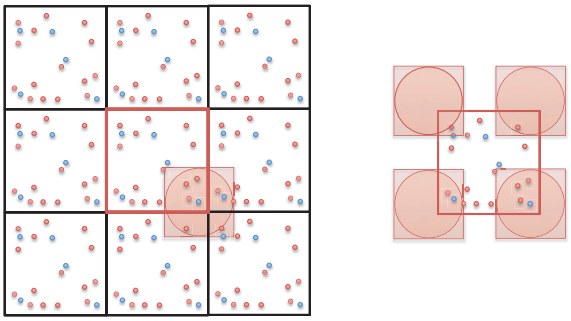

Fig. 4. Query with periodic boundary condition: The left holds the extra cells and issues only one query (A1), while the right holds only one cell and issues multiple queries (A2).

Algorithm 11 Out of bound checking

```
1: LF[] ← lower out of bound flags
2: UF[] ← upper out of bound flags
3: M ← query MBR
4: function CheckOutOfBound(LF, UF, M)
5:     VL ← 3-dimensional vector of lower bound coordinate
6:     VU ← 3-dimensional vector of upper bound coordinate
7:     LF[] ← _mm_cmplt(lower point of M, VL)
8:     UF[] ← _mm_cmpgt(upper point of M, VU)
9: end function
```

B. Details of Applying Periodic Boundary Condition

To adopt the CoSTR-R-tree for the SJCS, we need to modify the search algorithm described in Alg. 5. Alg. 10 shows the algorithm of the SJCS. In line 11 to 14, it checks whether the query MBR is out of periodic bound. In line 15 to 43, queries that are applied periodic bound condition are issued.

Alg. 11 shows the algorithm of checking out of bound query. At line 7 and 8, it checks whether the query MBR is being out of bound with `_mm_cmplt` and `_mm_cmpgt`, which are the vector comparison intrinsics.

Alg. 12 shows the algorithm of counting objects in the shells. It is similar to the Alg. 7 except for the leaf node handling. At line 16, it calculates the distance between the halo and the particle. Then, it checks which shell the particle pointed to by the leaf node belongs to and counts up the number of particles for the shell.

V. Evaluation of CoSTR-R-tree

A. Baseline and Data

In this section, we discuss the difference in performance between the CoSTR-R-tree and a **CoR-tree**) that is accelerated with SIMD and multi-threads. In the construction of the CoR-tree, the area calculation that occurs during node splitting is accelerated by SIMD.

We also apply SIMD to the creation of a parent node's MBR. In addition, the CoR-tree employs parallel search (Section III-B1). Thus, the CoSTR-R-tree and CoR-tree use the same search algorithm in these experiments. We used this as the baseline and compared it to the CoSTR-R-tree to demonstrate construction, destruction, and search performance. The parallelization applied to each index are shown in Table I.

Algorithm 12 Range count over shells with periodic boundary condition

```
1: NN ← node of STR-R-tree
2: M ← query MBR
3: P ← query point
4: S[] ← shells
5: function RangeCountOverShells(NN, M, P, S)
6:     if NN is not leaf node then
7:         for all nc: child nodes of NN do
8:             m ← MBR of nc
9:             if CheckOverlap(M, m) returns true then
10:                {descend the child node}
11:                RangeCountOverShells(nc, M, P, S)
12:            end if
13:        end for
14:    else
15:        p ← lower point of MBR of NN
16:        d ← CalculateDistance(P, p)
17:        for all s: shell in S[] do
18:            if d < radius of s then
19:                count up the number of particles for s
20:                end this loop
21:            end if
22:        end for
23:    end if
24: end function
```

TABLE I
Index Acceleration

	CoSTR-R-tree		CoR-tree	
	Construct	Search	Construct	Search
Multi-threads	Yes	Yes	No	Yes
SIMD	Yes	Yes	Yes	Yes

TABLE II
Experimental environment

CPU	Intel(R) Xeon(R) CPU E5-2650 v3 2.30 GHz × 2
Cores	20 (10 per socket)
Memory	64 GB
Compiler	gcc 6.1.0
Dimension:	3
CoR-tree/CoSTR-R-tree fanout:	10
#Particles:	10^8
#Halos:	10^7
#Shells:	40
Minimum range:	0.001
Maximum range:	5

We apply the small SJCS workload to the real dataset and the uniformly randomly generated dataset. Here, the number of particles is 10^8, which is $1/80$ of the real workload. The number of halos is 10^7, which is the same size as that from the real workload. The experimental environment is described in Table II.

B. Construction

The construction of the CoSTR-R-tree is parallelized, as described in Section III. We conducted this experiment to observe how performance changes depending on the number of threads. Figure 5 shows the relationship between the number of construction threads and the construction time for both the

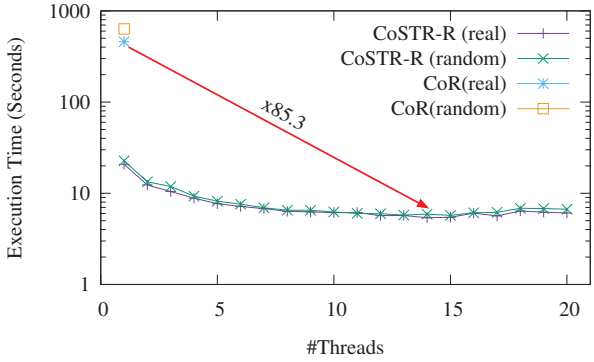

Fig. 5. Construction time

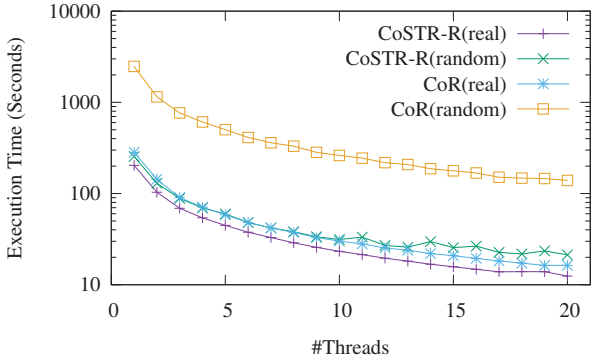

Fig. 6. Search Time

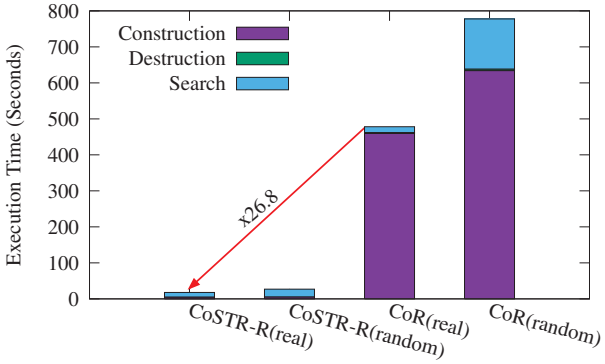

Fig. 7. Breakdown

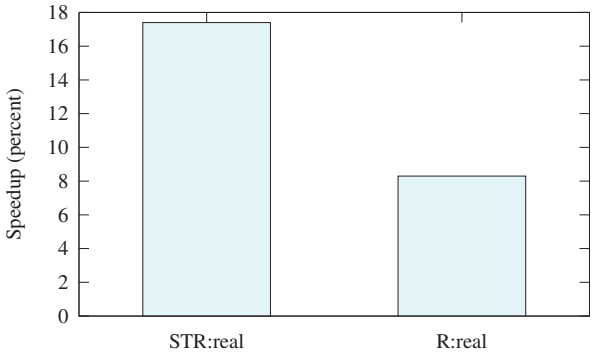

Fig. 8. Speedup with SIMD in total execution time

real dataset and random datasets. As can be seen, CoSTR-R-tree outperforms CoR-tree. With the real dataset, 14 threads demonstrate the best performance for the CoSTR-R-tree (85.3 times faster than the CoR-tree).

C. Search

There are 40 shells in the real workload. The radius of the minimum shell is 0.001 and that of the maximum shell is 5 where the range of the data space is 0 to 1000 for each dimension. The interval between shells is equal in log space. The experiments were conducted based on those conditions.

Figure 6 shows the time required to search with the CoR-tree and the CoSTR-R-tree for the real and random datasets. The result for CoR-tree with the random dataset shows poor performance due to the splitting policy of the R-tree. Because this policy minimizes the area of nodes (i.e., MBRs), the overlapping area between nodes is not considered. Consequently, many overlaps occur in the tree, and many node traversals will occur during the search. In contrast, there are few node overlaps in the CoSTR-R-tree because the construction process contains sorts. Therefore, the reduction in performance is small with random data distribution.

The best performance was observed with 20 threads. There is nearly no reduction in performance as the number of threads increases due to the independence of searches (Section III-B1). In the 20 threads search with the real dataset, the CoSTR-R-tree shows 1.31 times better performance than the CoR-tree.

D. Destruction

The destruction time for both CoR-tree and CoSTR-tree are negligible compared to search and construction (Fig. 7). Destruction of CoSTR-R-tree is 249 times faster than that of CoR-tree because CoSTR-tree destruction requires freeing of only a single array.

E. Performance Summary

Figure 7 shows the total execution time breakdown of the SJCS workload for each pair of index and dataset. Note that we varied the number of worker threads in each stage for optimization. The numbers for CoSTR-R-tree were 14, 20, and 1 for construction, search, and destruction, while those for CoR-tree were 1, 20, and 1, respectively.

The CoR-tree consumes most time for construction. In contrast, CoSTR-R-tree construction is considerably fast. In total, the CoSTR-R-tree demonstrates **26.8 times** performance improvement compared to the CoR-tree.

We also examined how much performance is gained by SIMD. Fig. 8 shows the performance improvement of the CoSTR-R-tree and the CoR-tree by SIMD with the real data set. Our CoSTR-R-tree achieves 17.4% improvement in total execution time.

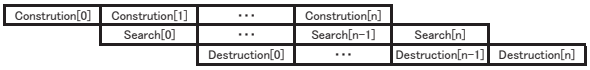

Fig. 9. Construct-search-destruct pipeline (CSD)

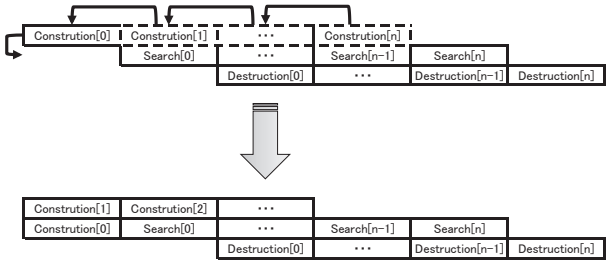

Fig. 10. Construct-search-destruct pipeline++ (CSD++)

VI. Applying CoSTR-R-Tree to a Large Dataset

A. Partial Materialization

When the number of particles increases, its index no longer fits in main memory. To solve the memory shortage problem, we adopt a partial materialization approach. This approach divides the particle data into multiple segments, and then it conducts a sequence of SJCS tasks. A task consists of construction, search, and destruction to avoid memory overflow. We can obtain the result by naively repeating the sequences. We refer to this naive method as **non-pipelined (NP)**.

To accelerate partial materialization, we propose **CSD pipelining**. This provides pipeline parallelism. Both the construction and destruction overlap while searching for another segment. Figure 9 illustrates the proposed CSD method.

Figure 10 shows that it is possible to overlap the first index construction with the second index construction. The pipeline bubble under the first index construction in Fig. 9 is filled with the second index construction. We refer to this method as **CSD pipeline++ (CSD++)**.

The CoSTR-R-tree is suitable for this approach because its execution time is faster (26.8 times) than that of the CoR-tree. If we apply this approach to the CoR-tree, then it requires approximately 26.8 times more time than that of the proposed method, which would not be suitable for astronomical researchers.

B. Thread Pool for Partial Materialization

We employ a non-uniform thread pool with a work request queue so that each thread can execute its tasks respectively.

This allows us to overlap index construction, search, and index destruction simultaneously. In addition, threads that have completed index construction or destruction can help the incomplete searches; thus we can maintain a high CPU usage ratio. However, the non-uniform thread pool conflicts with the uniform thread pool in the CoSTR-R-tree construction (Section III-A1). To address this problem, we introduce an inter-thread pool movement mechanism, which is described as follows.

1) **Initialize**: We create a non-uniform thread pool and empty the uniform thread pool.

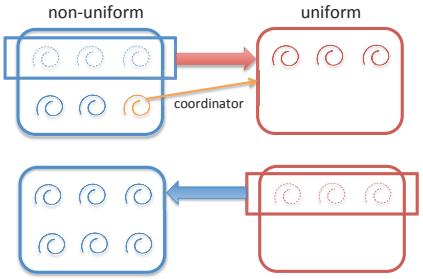

Fig. 11. Inter-thread pool movement. Blue: non-uniform thread pool. Red: uniform thread pool.

TABLE III
Experimental condition

CoSTR-R-tree fanout:	10
Data type:	Real data
#Particles:	1.3×10^9
#Halos:	10^7
#Searches:	10
Size of segment:	1.3×10^8

2) **Move**: With the uniform thread pool, we move the threads to use from the non-uniform thread pool to the uniform thread pool. Then, an additional thread acts as a coordinator of the uniform thread pool for synchronization.
3) **Execute**: The threads in the uniform thread pool execute index construction. Concurrently, the remaining threads in the non-uniform thread pool execute search.
4) **Return**: After all threads in the uniform thread pool have finished their work, they return to the non-uniform thread pool and help with search.

Figure 11 illustrates the above procedure. When we use the uniform thread pool, some threads are moved from the non-uniform thread pool. Then, the coordinator thread (represented by the orange spiral) is assigned to the uniform thread pool. The coordinator places a barrier among the threads or instructs them to return to the original thread pool. When the uniform thread pool ends its work, its threads return to the non-uniform thread pool. This enables parallel execution of both multi-threaded search and multi-threaded index construction.

C. Evaluation

1) Condition: We conducted experiments to determine how performance improves with the CSD pipeline. Table III shows the conditions of the experiments. We divided 1.3×10^9 particles into 10 segments, then, we performed 10 SJCS tasks for 1.3×10^8 particles and 10^7 halos in both NP and pipelined manners.

Note that the real SJCS includes 8×10^9 particles, which requires approximately 100 GB for particle data and 200 GB for CoSTR-R-tree, respectively. Our method, CSD pipelining, can be applied to such a task size, but requires significant time to experiment. In this paper, to investigate whether the proposed method is feasible for use with large-scale data, we used 1.3×10^9 particles. If we increase the repetition number from 10 to 50, our method works for 8×10^9 particles.

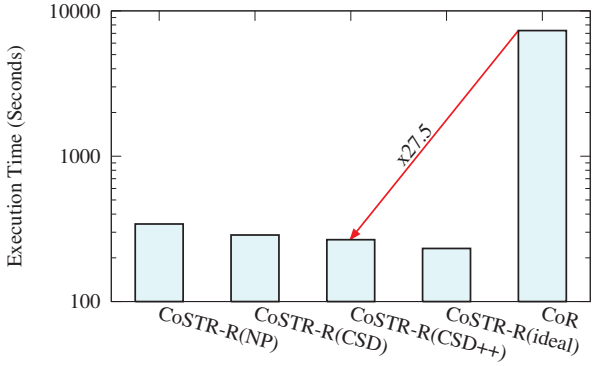

Fig. 12. Execution time for large SJCS workload

2) Results: We show the results in Fig. 12. The NP method, which is just an iteration of construction, search, and destruction, required a total of 341.6 sec. CoSTR-R(CSD) and CoSTR-R(CSD++) required 287.1 sec and 266.5 sec, respectively. CoSTR-R(CSD++) achieved a 28 % improvement in performance (over the NP). Here, constructions that overlap with searches were executed using three threads. CoSTR-R(CSD++) shows **27.5 times** faster performance than CoR.

Note that these results contain particle data reading time, while the results in Section V do not. The I/Os are included in part of the index construction. We also show a reference value that does not include I/O as CoSTR-R(ideal) in Fig. 12. The value is thirteen-fold value of the value shown for CoSTR-R(real) in Fig. 7, which is a simply reference to ideal (i.e., zero I/O cost) case.

VII. Conclusions

In astronomy, a number of simulations are conducted to analyze the state of the universe and find an unknown or a rare phenomenon. Spatial join is frequently used in such analysis. In this study, we focused on a variant of the spatial join count in proposed by Taruya et al. [4].

We first proposed a CoSTR-R-tree with multi-threading and SIMD instructions. We applied the proposed CoSTR-R-tree to the spatial join count workload in astronomy, and we evaluated performance by comparing it to a CPU optimized R-tree (CoR-tree). The results show that the proposed method outperformed CoR-tree by 26.8 times with a size-limited SJCS workload when applied to a real dataset.

We then proposed a construct-search-destruct (CSD++) pipelining method for partial materialization to handle a large dataset that exceeds the capacity of main memory. It succeeded to conceal I/O accesses, index constructions, and index destructions with pipelining. Experimental results show that the CSD++ method performed appropriately for a large SJCS workload, where the size of the dataset exceeds the capacity of main memory. In addition, CSD++ is 27.5 times faster than the CoR-tree based method as a reference.

All of our codes are available on GitHub [7] for reproducibility and for use by astronomy researchers.

Acknowledgements

This work was supported in part by the JST CREST "Development of System Software Technologies for post-Peta Scale High Performance Computing", "Extreme Big Data (EBD): Next Generation Big Data Infrastructure Technologies Towards Yottabyte/Year", "Statistical Computational Cosmology with Big Astronomical Imaging Data" and KAKENHI(#16K00150).

References

[1] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Trans. Database Syst.*, vol. 32, no. 1, 2007.
[2] A. S. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, D. Slutz, and R. J. Brunner, "Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey," in *SIGMOD Conference*, 2000, pp. 451–462.
[3] A. S. Szalay, J. Gray, A. R. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg, "The sdss skyserver - public access to the sloan digital sky server data," in *SIGMOD Conference*, 2002, pp. 570–581.
[4] A. Taruya, T. Nishimichi, S. Saito, and T. Hiramatsu, "Non-linear evolution of baryon acoustic oscillations from improved perturbation theory in real and redshift spaces," *Physical Review D*, vol. 80, 2009.
[5] N. Yoshida, K. Omukai, and L. Hernquist, "Protostar formation in the early universe," *Science*, vol. 321, no. 5889, pp. 669–671, 2008.
[6] T. Nishimichi and P. Valageas, "Testing the equal-time angular-averaged consistency relation of the gravitational dynamics in n-body simulations," *Phys. Rev. D*, vol. 90, p. 023546, 2014.
[7] R. Mitsuhashi, "An implementation of the spatial join count over shells (sjcs) workload," [Accessed Sept. 24, 2016]. [Online]. Available: *https://github.com/ryumt/SpatialJoinCountOverShells*
[8] S. T. Leutenegger, M. A. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for r-tree packing," in *ICDE*, 1997, pp. 497–506.
[9] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort," in *SIGMOD Conference*, 2010, pp. 351–362.
[10] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD Conference*, 1984, pp. 47–57.
[11] D. A. White and R. Jain, "Similarity indexing with the ss-tree," in *ICDE*, 1996, pp. 516–523.
[12] N. Katayama and S. Satoh, "The sr-tree: An index structure for high-dimensional nearest neighbor queries," in *SIGMOD Conference*, 1997, pp. 369–380.
[13] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima, "The a-tree: An index structure for high-dimensional spaces using relative approximation," in *VLDB*, 2000, pp. 516–526.
[14] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The x-tree: An index structure for high-dimensional data," in *VLDB*, 1996, pp. 28–39.
[15] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, 1998, pp. 194–205.
[16] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi, "Vector approximation based indexing for non-uniform high dimensional data sets," in *CIKM*, 2000, pp. 202–209.
[17] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD Conference*, 1990, pp. 322–331.
[18] N. Roussopoulos and D. Leifker, "Direct spatial search on pictorial databases using packed r-trees," in *SIGMOD Conference*, 1985, pp. 17–31.
[19] I. Kamel and C. Faloutsos, "On packing r-trees," in *CIKM*, 1993, pp. 490–499.
[20] S. You, J. Zhang, and L. Gruenwald, "Parallel spatial query processing on gpus using r-trees," in *Workshop on Analytics for Big Geospatial Data*, 2013, pp. 23–31.
[21] T. A. Hunt, "Periodic boundary conditions for the simulation of uniaxial extensional flow," *Molecular Simulation*, vol. 42, pp. 347–352, 2016.
[22] T. Fukushige, J. Makino, T. Ito, S. K. Okumura, T. Ebisuzaki, and D. Sugimoto, "Wine-1: Special-purpose computer for n-body simulations with a periodic boundary condition," *Astronomical Society of Japan*, vol. 45, pp. 361–375, 1993.