Electronic Preprint for Journal of Information Processing Vol.27

Regular Paper

Parallelization of Matrix Partitioning in Construction of Hierarchical Matrices using Task Parallel Languages

Zhengyang Bai[1,a)] Tasuku Hiraishi[2] Hiroshi Nakashima[2] Akihiro Ida[3] Masahiro Yasugi[4]

Received: April 5, 2019, Accepted: September 3, 2019

Abstract: A hierarchical matrix ($\mathcal{H}$-matrix) is an approximated form that represents $N \times N$ correlations of N objects. $\mathcal{H}$-matrix construction is achieved by partitioning a matrix into submatrices, followed by calculating the element values of these submatrices. This paper proposes implementations of matrix partitioning using task parallel languages, Cilk Plus and Tascell. Matrix partitioning is divided into two steps: cluster tree construction by dividing objects into clusters hierarchically, and block cluster tree construction by observing all cluster pairs at the same level of the cluster tree that satisfy an admissibility condition. As the two types of trees constructed and traversed in these steps are unpredictably unbalanced, it is expected that we can efficiently parallelize both these steps using task parallel languages. To obtain sufficient parallelism in the cluster tree construction, we not only execute recursive calls in parallel but also parallelize the inside of each recursive step. For the block cluster tree construction, we assigned each worker its own space so that the workers can store the cluster pairs without using locks. As a result, compared to a sequential implementation in C, we achieved up to an 11.5-fold speedup using Cilk Plus and a 12.6-fold speedup by Tascell for the cluster tree construction. For the block cluster tree construction, up to a 37.7-fold speedup by Cilk Plus and a 38.8-fold speedup using Tascell are achieved. In regard to the entire process of matrix partitioning, we achieved up to a 12.2-fold speedup by Cilk Plus and a 14.5-fold speedup using Tascell. All experiments were executed on two 18-core Xeon processors with real datasets to generate coefficient matrices used in the surface charge method.

Keywords: task parallel languages, hierarchical matrix, tree construction

1. Introduction

In the boundary element method (BEM) and N-body simulations, using a coefficient matrix that represents the interaction between physical elements to solve simultaneous linear equations is common. However, as the quantity of all interactions between N elements is N^2, such a matrix is dense, and when N is extremely large, the execution time and memory usage will be unacceptable or even unavailable. Therefore, various approximation techniques have been proposed to reduce execution time and memory usage.

Hierarchical matrices ($\mathcal{H}$-matrices) [1], [2], [3] are used as one such approximation technique. An $\mathcal{H}$-matrix is constructed directly from the interactions between element sets, not from its dense counterpart, to reduce the memory usage from $O(N^2)$ to $O(N \log N)$ by hierarchically dividing the matrix into many submatrices and replacing them (if possible) with their small-size low-rank approximated forms. Though this technique can significantly reduce computation cost and memory usage with reasonable accuracy, the computation cost is still large. Thus, accelerating the computation for $\mathcal{H}$-matrices, including not only calculations such as $\mathcal{H}$-matrix-vector and $\mathcal{H}$-matrix-$\mathcal{H}$-matrix multiplication but also $\mathcal{H}$-matrix construction, using parallel computing is critical.

$\mathcal{H}$-matrix construction is achieved by dividing a matrix into submatrices (partitioning), followed by calculating the element values of these submatrices (filling). We can find many proposals [4], [5], [6], [7], [8], [9] to parallelize the filling operation and they are applied to $\mathcal{H}$-matrix libraries such as Hlib [3] and $\mathcal{H}$ACApK [7], but the partitioning operation still remains sequential. This is partly because the cost of the partitioning operation is much lower compared to the filling operation, approximately one thousandth when $N \simeq 10^6$. However, this cost is becoming considerable because hundreds of speedups have been achieved for the filling operation using MPI, GPU, and SIMD vectorization [4], [5], [9]. For example, Ref. [9] shows that, when constructing a $\mathcal{H}$-matrix derived from 1,188,000 elements, it takes 0.736 s for partitioning and 1,479 s for filling in sequential computation, and the filling operation is accelarated by 214.8 times using 256 cores. We can expect more speedups using more computing resources in the near future. Then the partitioning operation will be a bottleneck if it remains sequential, and it will be significant for larger datasets. Thus, we should also consider parallelizing the partitioning operation. Therefore, in this paper, we propose parallel implementations for matrix partitioning in the construction of $\mathcal{H}$-matrices, based on the sequential version proposed in Ref. [2].

1 Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan
2 Academic Center for Computing and Media Studies, Kyoto University, Kyoto 606–8501, Japan
3 Information Technology Center, the University of Tokyo, Bunkyo, Tokyo 113–8658, Japan
4 Department of Computer Science and Networks, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
a) haku@sys.i.kyoto-u.ac.jp

The matrix partitioning operation is divided into the following two steps: construction of cluster tree (CT) and construction of block cluster tree (BCT) [*1]. As trees constructed and traversed in these steps are unpredictably unbalanced, we employed task parallel languages to parallelize these operations solving the load imbalance problem with reasonable programming cost. Among task parallel languages, we used Cilk Plus [10] and Tascell [11] and evaluated and compared the performance of our implementations using these two languages.

The contributions of this paper are summarized as follows:

- We propose parallel implementations for matrix partitioning in the construction of $\mathcal{H}$-matrix using Cilk Plus and Tascell based on the sequential implementation of the $\mathcal{H}$ACApK library.
- We evaluated the performance of our parallel implementations; the results indicated that we obtained reasonable speedups with both languages.
- We showed that, in CT construction, our implementations using task parallel languages significantly overperform a naïve implementation using OpenMP.

The remainder of this paper is organized as follows. We introduce $\mathcal{H}$-matrices in Section 2 and the sequential algorithm of matrix partitioning in Section 3. In Section 4, we introduce Cilk Plus and Tascell, task parallel languages used in our implementations. In Section 5, we present our proposed parallel algorithms and their implementations using these task parallel languages. We evaluate the performance of our implementations in Section 6 and present related work in Section 7. Finally, we provide our conclusions and describe our future work in Section 8.

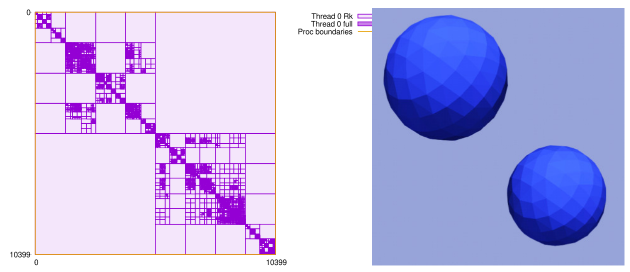
2. Hierarchical Matrices

2.1 Overview

An $\mathcal{H}$-matrix is a collection of submatrices created by matrix partitioning where all submatrices neither overlap with each other nor hold gaps among them. Submatrices are also called leaf matrices. **Figure 1** (a) shows an example of a structure of an $\mathcal{H}$-matrix. This $\mathcal{H}$-matrix is derived from the input data illustrated in Fig. 1 (b) and represents the interactions among the elements distributed on the surface of the two spheres [*2].

A $l \times m$ leaf matrix will be approximated by the product of two low-rank matrices of sizes $l \times k$ and $k \times m$, where k denotes the rank of these two matrices and is far less than l and m. Therefore, restricting the memory usage to $O(NK \log N)$, where K is the upper limit of k, is possible. K can be tuned based on the accuracy requirements, but it is usually far less than l and m. Not all these submatrices can be approximated, depending on judgment results of an *admissibility condition*. Thus, an $\mathcal{H}$-matrix is combined by both approximated submatrices and unmodified submatrices (full submatrices).

For a matrix X, we let $\mathrm{R}(X)$ denote the range of X's indices.



(a) Example of a structure of an $\mathcal{H}$-matrix (b) Example of input data

Fig. 1 Examples of a structure of $\mathcal{H}$-matrix and input data. The number of elements is 10,400.

Moreover, we let $[n_1, n_2]$ denote a set of successive natural numbers $\{i \in \mathbb{N} \mid n_1 \leq i \leq n_2\}$. Let A be a square matrix of order N. Thus, we have $\mathrm{R}(A) = [1, N] \times [1, N]$ and a subrange p of $\mathrm{R}(A)$ can be defined as $p = I_p \times J_p$ where $I_p = [i_1, i_2]$ and $J_p = [j_1, j_2]$ for some $i_1, i_2, j_1, j_2 \in \mathbb{N}$ such that $1 \leq i_1 < i_2 \leq N$ and $1 \leq j_1 < j_2 \leq N$. We define a partition P of $\mathrm{R}(A)$ as a set of subranges that satisfies the following two conditions: (1) $\bigcup_{p \in P} p = \mathrm{R}(A)$ and (2) $p_1 \cap p_2 = \emptyset$ for all $p_1, p_2 \in P$ such that $p_1 \neq p_2$. We let $A|^p$ denote a part of A corresponding to the subrange $p \in P$. When the $\mathcal{H}$-matrix $\tilde{A}$ approximates A, it comprises leaf matrices corresponding to $A|^p$ for all $p \in P$, each of which is denoted by $\tilde{A}|^p$. If a submatrix $A|^p$ can be approximated, the leaf matrix $\tilde{A}|^p$ is represented by the product of two low-rank matrices $V_p \cdot W_p$ shown below:

$$V_p \in \mathbb{R}^{\#I_p \times k_p},\ W_p \in \mathbb{R}^{k_p \times \#J_p},\ k_p \leq \min(\#I_p, \#J_p)$$

where $\#S$ denotes the number of elements in the set S. Then, we define $k_p \in \mathbb{N}$ as the rank of $\tilde{A}|^p$. Usually, k_p is far less than $\#I_p$ and $\#J_p$. When $\tilde{A}|^p$ is a full submatrix, we have $\tilde{A}|^p = A|^p$. Thus, if $\tilde{A}|^p = A|^p$ for any $p \in P$, the $\mathcal{H}$-matrix $\tilde{A}$ is equal to A.

2.2 Construction of $\mathcal{H}$-matrix

As mentioned in Section 1, an $\mathcal{H}$-matrix can be constructed by the following two steps:

(1) Partitioning the matrix into submatrices.

(2) Filling in element values of the submatrices.

In this section, we first explain the principle of matrix partitioning with an example. We will present the partitioning algorithms in detail later in Section 3.

Generally, a dense matrix does not have an obvious structure. However, in dense matrices appearing in numerical analyses such as BEM, we can usually observe an implicit structure that can be represented as approximated submatrices. In $\mathcal{H}$-matrix construction, we should find as large and as many of such submatrices as possible to obtain better compressibility. $\mathcal{H}$-matrices achieve this using the underlying implicit hierarchy in the interactions among N elements representing the matrix A. For instance, in an astronomical simulation, N elements are distributed in the space and the interaction between each element pair quadratically decreases according to the distance between them. In this condition, if the distance between two sets of elements E_1 and E_2 are far enough, we do not need to strictly consider the interactions between all elements in E_1 and E_2. Instead, we can choose some representatives from these sets and approximate the interaction between the

[*1] Though our implementations presented in this paper do not create the whole tree structure but only the list of the leaf nodes of the BCT, we still call this operation BCT construction according to convention in this research area.

[*2] The surface in Fig. 1 (b) is composed of triangles and the gravity centers of the triangles are treated as elements in this example. This is also for Fig. 12 appearing later in Section 6.

sets as the interactions between the representatives of one set and all elements in the other set, which can be represented as an approximated submatrix. Thus, we can find a good partition by the following recursive procedure for a pair of element sets, or *clusters* in short, (E_1, E_2), starting from the self-pair (E, E) where E is the set of all elements.

(1) Split each of E_i ($i \in \{1, 2\}$) into two subsets $E_{i,1}$ and $E_{i,2}$ according to the geometric distribution of elements in E_i.
(2) For each cluster pair $(E_{1,j}, E_{2,k})$ ($j, k \in \{1, 2\}$), we estimate the approximation accuracy depending on the size of the subspaces containing the clusters and the distance between clusters.
(3) If the accuracy is admissible, we decide the range of the low-rank matrix for the cluster pair based on the elements contained in the pair.
(4) If the accuracy is not admissible, apply (1)–(4) to the cluster pair recursively.

The filling operation follows the matrix partitioning and calculates the element values of the leaf matrices. ACA (Adaptive Cross Approximation) [12], [13] and ACA+ [3] are well-known algorithms for this operation. See these papers for more details because this is beyond the scope of this paper.

3. Matrix Partitioning Algorithm

Though Section 2.2 outlined one single recursive procedure for the matrix partitioning, it can be broken into the series of the following two procedures so that a cluster is split just once:

(1) Cluster tree (CT) construction to split clusters recursively.
(2) Block cluster tree (BCT) construction to examine the admissibility of the cluster pair and to determine the matrix structure recursively.

In this section, we provide details about the algorithms for constructing CT and BCT.

3.1 Cluster Tree Construction

First, we show the algorithm to construct a CT. The cluster ${\mathcal{E}_1}^{(0)} = \{e_0, ..., e_{N-1}\}$ containing all input elements is treated as the root node of CT. The children of a CT node are created by dividing the cluster into two sub-clusters. We can create the children of each child node by dividing the corresponding cluster in the same manner. Such division operations are repeated recursively until the size of the cluster becomes less than the threshold $N_{\min}$. **Figure 2** shows an example of a cluster tree structure.

In each recursive step, there are many ways to divide a cluster. In BEM, elements are often divided by pivoting. For example, in the case where elements are placed in the 1D space, $\mathcal{E} = \{e_0, \ldots, e_{n-1}\}$ can be divided into $\mathcal{E}_{\mathrm{L}} = \{e \in \mathcal{E} \mid e.x < M\}$ and $\mathcal{E}_{\mathrm{R}} = \{e \in \mathcal{E} \mid e.x \geq M\}$ where $e.x$ is the x-coordinate of e and the pivot value M is $(\max_{e \in \mathcal{E}} e.x + \min_{e \in \mathcal{E}} e.x)/2$. **Figure 3** shows the pseudocode of the CT creation algorithm using this division strategy.

In the 3D space, the largest value is taken from $x_d = (\max_{e \in \mathcal{E}} e.x - \min_{e \in \mathcal{E}} e.x)$, $y_d = (\max_{e \in \mathcal{E}} e.y - \min_{e \in \mathcal{E}} e.y)$, and $z_d = (\max_{e \in \mathcal{E}} e.z - \min_{e \in \mathcal{E}} e.z)$, which corresponds to the length of edges of the *bounding box* surrounding the cluster. Then, the corresponding axis is chosen as the pivot axis. The pivot value M is

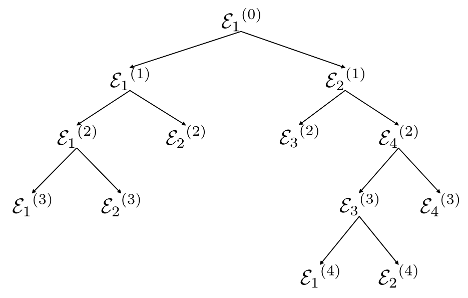


Fig. 2 Example of a cluster tree structure.

```
cluster buildClusterTree (elem[] e){
  cluster t = createCTnode(e);
  if(#e ≥ $N_{min}$){
    // $e$ ranges over e
    $M$ = $(\max(e.x) + \min(e.x))/2$;
    // eleft and elight are obtained by reordering e
    // in-place in our actual sequential implementation.
    elem[] eleft = $\{e \mid e.x < M\}$;
    elem[] eright = $\{e \mid e.x \geq M\}$;
    t.child[0] = buildClusterTree(eleft);
    t.child[1] = buildClusterTree(eright);
  }
  return t;
}
```

Fig. 3 Pseudocode of the algorithm for CT construction (for simplicity, we show the case where elements are placed in the 1D space).

set to $(\max_{e \in \mathcal{E}} e.x + \min_{e \in \mathcal{E}} e.x)/2$, $(\max_{e \in \mathcal{E}} e.y + \min_{e \in \mathcal{E}} e.y)/2$, or $(\max_{e \in \mathcal{E}} e.z + \min_{e \in \mathcal{E}} e.z)/2$ based on the pivot axis, and elements are divided based on the coordinate values of the pivot axis.

In each recursive step, the numbers of elements contained in two sub-clusters are generally uneven. Thus, a CT constructed using this algorithm is unbalanced. In addition, because the computational cost of dividing a cluster is proportional to the number of elements, the computational cost of constructing a CT whose root node contains N elements varies from $O(N \log N)$ to $O(N^2)$ depending on the unbalancedness of the CT (as is the case in the Quicksort algorithm). Therefore, we cannot estimate the computational cost of constructing each sub-CT simply from the number of elements contained in its root.

3.2 Block Cluster Tree Construction

In BCT construction, we use the CT constructed in the previous step. A node of BCT in an arbitrary level corresponds to a pair of two nodes of CT (corresponding to two clusters) in the same level. If a pair of clusters satisfies an *admissibility condition*, the corresponding BCT node does not have its child nodes as it means that the interaction between the clusters can be approximated by a low-rank submatrix. If the admissibility condition cannot be satisfied and one of both CT nodes are leaves, we determine the corresponding submatrix cannot be approximated and make the BCT node leaf for a full submatrix. Otherwise, i.e., if the non-leaf cluster pair is not admissible, the BCT node has four children corresponding to all pairs of two children of the CT nodes. As any self-pairs of CT nodes cannot be admissible, the depth of BCT is equal to that of CT. The number of BCT nodes in a level is at most the square of the number of CT nodes in the level corresponding to all possible combinations of CT nodes but is usually much less than that, especially in deep levels.

The admissibility condition can protect the error of approxi-

```
counter = 0;
leaf-node[] leaf-node-list;
void buildBlockClusterTree (cluster t, cluster s){
  if( combination(t,s) is admissible){
    leaf-node-list[counter] =
        createBTLeafNode(t, s, "low-rank submatrix");
    counter++;
  }else if(t or s has no child){
    leaf-node-list[counter] =
        createBTLeafNode(t, s, "full submatrix");
    counter++;
  }else{
    for (i=0; i<=1; i++)
      for (j=0; j<=1; j++)
        buildBlockClusterTree(t.child[i],s.child[j]);
  }
}
```

Fig. 4 Pseudocode of the algorithm for BCT construction.

mation from getting too large. Thus, it can be tuned based on accuracy requirements and the nature of problem setting. For example, the common setting of the admissibility condition in BEM or N-body simulation is that the distance between the two clusters is longer than a certain multiple of the diameter of these two subsets and can be expressed as the inequalities below:

$$\eta \times diam(\tau) \leq dist(\tau, \sigma) \wedge \ \eta \times diam(\sigma) \leq dist(\tau, \sigma)$$

where $diam(\tau)$ is the diameter of cluster τ, $dist(\tau, \sigma)$ is the distance between the clusters τ and σ, and η is a constant. Note that $diam(\tau)$ and $dist(\tau, \sigma)$ can be defined in various ways. In our implementation, $diam(\tau)$ is the length of the diagonal of τ's bounding box, while $dist(\tau, \sigma)$ is the distance between two nearest points in the surfaces of corresponding bounding boxes.

The pseudocode of the algorithm for BCT construction is shown in **Fig. 4**. The two walkers t and s traverse the cluster tree from its root node, checking the admissibility condition. Note that the execution tree of this recursive algorithm forms a structure of the BCT; however, we do not create the structure of the tree but only the list of the leaf nodes of the BCT, which corresponds to the set of submatrices, because the tree structure is not used in subsequent operations such as the filling operation and $\mathcal{H}$-matrix arithmetic.

Note that a BCT constructed using this algorithm is unpredictably unbalanced due to not only the unbalancedness of the CT but also the fact that each BCT node may not have its children depending on the admissibility condition.

4. Task Parallel Languages

As mentioned in Sections 3.1 and 3.2, it is difficult to estimate the computational cost of construction of each sub-CT and sub-BCT in advance. Thus, we cannot get good load balance when we statically assign computational cores to subtrees. Therefore, we parallelized CT and BCT constructrion using task parallel languages, by which we can parallelize such tree recursive algorithms easily and efficiently employing the dynamic load balancing strategy.

In this section, we provide a brief introduction of the task parallel languages used for our parallel implementations of CT and BCT construction.

4.1 Cilk Plus

Cilk Plus [14] is a commercial version of Cilk [15] that was

```
int fib(int n){
  if(n ≤ 2){
    return n;
  }
  int a = cilk_spawn fib(n-1);
  int b = fib(n-2);
  cilk_sync;
  return a+b;
}
```

Fig. 5 Doubly recursive Fibonacci in Cilk Plus.

```
double find_max(double* list, int length){
    double max;
    CILK_C_REDUCER_MAX(max,double,-DBL_MAX);
    CILK_C_REGISTER_REDUCER(max);
    cilk_for(int id=0;id<length;id++){
        CILK_C_REDUCER_MAX_CALC(max,list[id]);
    }
    max = REDUCER_VIEW(max);
    CILK_C_UNREGISTER_REDUCER(max);
    return max;
}
```

Fig. 6 Finding the maximum element in a list in Cilk Plus.

first created in CSAIL, MIT. It became a default part of the Intel C++ Compiler since version 12 and works with both C++ and C. Cilk Plus employs the oldest-first work stealing strategy: a Cilk Plus worker, usually corresponding to an OS thread, can push (parent) tasks that can be stolen by other workers into its own queue. When a worker is idle, it randomly chooses another worker as a victim and steals the oldest task from the victim's task queue. A task parallel program can be easily implemented using the following three Cilk Plus constructs: *cilk_spawn* (for parallel execution of function calls), *cilk_for* (for parallel for-loops) and *cilk_sync* (for synchronization). Cilk Plus also provides useful APIs such as *cilkrts_get_nworkers()* to obtain the total number of workers and *reducers* to help programmers parallelize computations with reduction. We provide two sample programs written in Cilk Plus in **Fig. 5** and **Fig. 6**. In Fig. 5, *cilk_spawn* and *cilk_sync* are used to parallelize the calculation of the $(n-1)$-th and $(n-2)$-th fibonacci numbers at each recursive step. Figure 6 shows how to find the maximum element in a list using *cilk_for* with the help of reducers.

4.2 Tascell

Tascell [11] is a task parallel language that employs the *backtracking-based work stealing strategy*. A Tascell worker always chooses not to spawn a task at first and performs sequential computations. When a worker is chosen as a victim and receives a task request, it temporarily rewinds the execution of its task to backtrack to the oldest point of task spawning and then spawns a task. After that, the victim returns from the backtracking and resumes its own task. While the cost for each work-steal in Tascell is higher than Cilk Plus, the parallelization overhead is lower because it does not have to manage task queues. Tascell provides the *do-two* (for parallel execution of the following two statements) and *do-many* (for parallel loops) constructs. **Figures 7** and **8** show examples of Tascell programs [*3]. **Figure 9** shows the manner in which backtracking-based task spawning occurs when a Tascell worker performs the program in Fig. 7.

[*3] The actual Tascell language has an S-expression-based syntax [16], but we write programs with a C-like syntax here for readers' convenience.

```
// The definitioin of a task named tfib
task tfib{
  in:    int n;        //input
  out:   int r;        //output
};
//The entry point of tfib.
//The task object this is declared implicitly
task_exec tfib
{this.r = fib (this.n);}
worker int fib(int n){
  if(n ≤ 2) return 1;
  {
    int s1, s2;
    do_two              //construct in Tacell
      s1 = fib(n-1);
      s2 = fib(n-2);
    handles tfib {
      //put part (performed before sending a task)
      {this.n = n-2;}
      //get part (performed after receiving the result)
      {s2 = this.r}
    }                   //end do_two
    return s1 + s2;
  }
}
```

Fig. 7 Doubly recursive Fibonacci in Tascell.

```
task find_max{
  in:    int i1;         //from
  in:    int i2;         //to
  in:    double* list;   //input
  out:   double r;       //output
};
task_exec find_max
{this.r = find (this.i1, this.i2, this.list);}
worker double find(int i1, int i2, double* list){
  double max = -DBL_MAX;
  do_many for i from i1 to i2
    if(max < list[i]) max = list[i];
  handles find_max from j1 to j2 {
    //put part
    this.i1 = j1;
    this.i2 = j2;
  } {
    //get part
    if(max < this.r) max = this.r;
  }
  return max;
}
```

Fig. 8 Finding the maximum element in a list in Tascell.

5. Parallel Implementation

In this section, we explain our proposed parallel algorithms and implementations of CT and BCT construction.

5.1 Cluster Tree Construction

As explained in Section 3, a CT is constructed using a recursive algorithm. It is obvious that the two recursive calls in Fig. 3 can be executed in parallel and thus these can be parallelized using the cilk_spawn or do_two constructs.

However, after preliminary evaluations, we found that the parallel performance is far below our expectations and that we can only achieve 5.3 times speedup at most using two 18-core processors. This is because the computation cost of each recursion step (lines 4–9 in Fig. 3) is proportional to the size of e and the critical path thus cannot be shortened when only recursive calls are executed in parallel. To obtain better performance, we also need to parallelize inside the recursion step. The costly operations in the recursion step are two-fold: 1) finding the maximum and minimum x, y, and z-coordinate values to decide the pivot value and axis and 2) the pivoting operation, i.e., reordering elements based on the coordinate values of them.

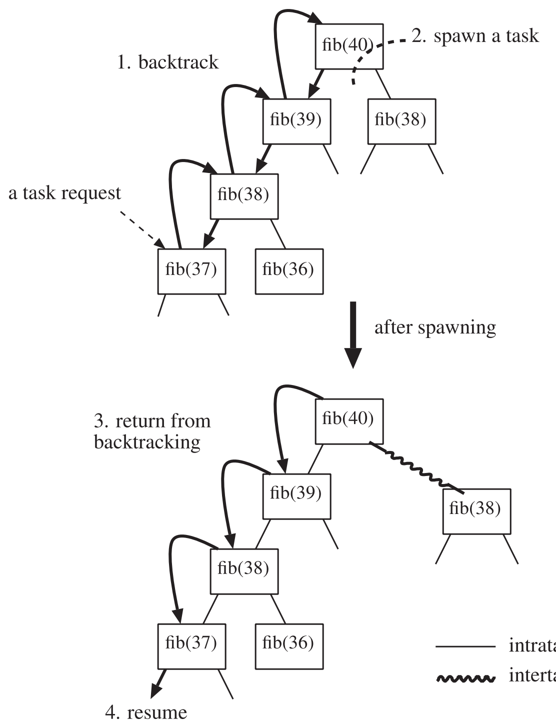


Fig. 9 Spawning a task lazily while computing fib(40). When a Tascell worker detects a task request (at fib(37)), it (1) backtracks to the oldest task-spawnable point, (2) spawns a task for fib(38), (3) returns from backtracking, and (4) resumes its own computation.

Figure 10 shows the pseudocode of the parallel algorithm for CT construction. Finding the maximum and minimum numbers can be easily implemented using the cilk_for construct and reducers in Cilk Plus, and the do_many construct in Tascell, as shown in Figs. 6 and 8.

Parallelizing the pivoting operation is more difficult. In sequential implementation, we can easily reorder the elements in-place using the commonly used algorithm for Quicksort, wherein two pointers scan the array from both the left and right sides and swap the elements when the left/right pointer finds the value to be more/less than the pivot. However, this in-place algorithm is difficult to be parallelized.

Therefore, we employed two arrays L_1 and L_2. Initially, the element data are stored in L_1, the result of reordering at the first level of CT is stored in L_2. Similarly, at the second level, elements in L_2 are reordered and the result is stored in L_1. This operation is repeated recursively until the number of elements is less than a threshold T_S. After that, pivoting is sequentially performed using the in-place algorithm.

We parallelized the pivoting operation using the following steps (elements in L_1 are reordered and stored in L_2).

step 1: Divide the array L_1 into N_c chunks.

step 2: For each chunk, count the number of elements whose value is less than and not less than the pivot. The counts for the i-th chunk are stored in $less[i]$ and $more[i]$, respectively.

step 3: Calculate $N_{less} = \sum_{k=0}^{N_c-1} less[k]$.

step 4: Calculate $Iless[i] = \mathrm{PS}(less[i])$ and $Imore[i] = \mathrm{PS}(more[i]) + N_{less}$, where $\mathrm{PS}(A[i])$ is the i-th prefix-sum of A, i.e., $\mathrm{PS}(A[i]) = A[0] + A[1] + \ldots + A[i-1]$.

step 5: For each i-th chunk, copy the elements whose values

```
1  cluster buildClusterTreePar (elem[] e, elem[] temp_e){
2    elem[] eleft, eright, temp_eleft, temp_eright;
3    cluster t = createCluster(e);
4    if(#e ≥ $N_{min}$){
5      if(#e > $T_S$){
6        min_x, max_x = parallel_minmax(e);
7        $M$ = (max_x + min_x)/2;
8        // nl is the number of elements less than $M$
9        temp_e, nl = parallel_pivoting(e, $M$);
10       // The following four (sub)arrays are obtained by pointing the subspace
11       // of e and temp_e without copying.
12       eleft = e[0..nl-1];
13       eright = e[nl..#e];
14       temp_eleft = temp_e[0..nl-1];
15       temp_eright = temp_e[nl..#temp_e];
16     }else{
17       // same with the sequential algorithm
18       $M$ = $(\max(e.x) + \min(e.x))/2$;
19       elem[] eleft = $\{e \mid e.x < M\}$;
20       elem[] eright = $\{e \mid e.x \geq M\}$;
21     }
22     if(#e > $T_N$){
23       spawn {
24         if(#eleft > $T_S$){
25           // swap $L_1$ and $L_2$
26           t.child[0] =
27               buildClusterTreePar(temp_eleft,eleft);
28         }else{
29           if (depth%2 == 0 and #e > $T_S$) {
30               memcpy(eleft, temp_eleft, size of eleft);
31           }
32           // sequential (in-place) pivoting on $L_1$
33           t.child[0] = buildClusterTreePar(eleft);
34         }
35       }
36       spawn {
37         if(#eright > $T_S$){
38           // swap $L_1$ and $L_2$
39           t.child[1] =
40               buildClusterTreePar(temp_eright,eright);
41         }else{
42           if (depth%2 == 0 and #e > $T_S$) {
43             memcpy(eright, temp_eright, size of eright);
44           }
45           // sequential (in-place) pivoting on $L_1$
46           t.child[1] = buildClusterTreePar(eright);
47         }
48       }
49     }else{
50       Same to lines 23–48 but without spawn.
51     }
52     sync;
53   }
54   return t;
55 }
```

Fig. 10 Pseudocode of the parallel algorithm for CT construction (for simplicity, we show the case in which elements are placed in the 1D space).

are less than the pivot into the subspace of L_2 starting from $L_2[Iless[i]]$. Similarly, copy the elements whose values are not less than the pivot into the subspace of L_2 starting from $L_2[Imore[i]]$.

In these steps, **steps 2**, **3** and **5** can be easily parallelized over chunks. In addition, we parallelized **step 4** in two ways. One used the parallel prefix-sum algorithm proposed in Ref. [17], and the other implements a more simple algorithm as follows: 1) we divide the array into chunks and calculate the prefix-sum of each chunk, 2) calculate the prefix-sums of $C[\,]$ where $C[j]$ is the total sum of the j-th chunk, and 3) add $\mathrm{PS}(C[i])$ to all array elements in the $(i+1)$-th chunk. Here, 1) and 3) are executed in parallel over chunks and 2) is sequentially executed.

However, in both implementations, we obtained only slight speedups when we measured the performance of prefix-sum computation independently, and the performance degraded when the parallel prefix-sum implementation is embedded to CT construction. Therefore, we employed the sequential implementation for

```
counter-list = {0,...,0}; // counter for each worker
leaf-node[][] leaf-node-list;
void buildBlockClusterTreePar (cluster t, cluster s){
  // w_id: the ID of the worker executing the task
  if( combination(t, s) is admissible){
    leaf-node-list[w_id][counter-list[w_id]] =
        createBTLeafNode(t, s, "low-rank submatrix");
    counter-list[w_id]++;
  }else if(t or s has no child){
    leaf-node-list[w_id][counter-list[w_id]] =
        createBTLeafNode(t, s, "full submatrix");
    counter-list[w_id]++;
    }else{
      if(t.nelems > $T_N$ and s.nelems > $T_N$){
        for (i=0; i<=1; i++)
          for (j=0; j<=1; j++)
            spawn buildBlockClusterTreePar(t.child[i],
                                           s.child[j]);
        sync;
      }else{
        for (i=0; i<=1; i++)
          for (j=0; j<=1; j++)
            buildBlockClusterTree(t.child[i],s.child[j]);
      }
    }
  }
}
```

Fig. 11 Pseudocode of the parallel algorithm for BCT construction.

the prefix-sum computation in the evaluations in Section 6.

One of the obvious drawbacks of this parallel algorithm is that there are additional costs to scan the element list multiple times and to compute prefix-sums. When the number of elements is small, the parallelization overhead would be larger than the parallel speedups. Therefore, we apply the sequential pivoting algorithm when the number of elements is not greater than the threshold T_S.

In addition, we employed another parameter T_N to achieve better performance. A worker calls the recursive function sequentially when the number of elements is lower than T_N. This parameter should be set considering the tradeoff between overheads for parallelization (e.g., cilk_spawn and do_two) and load imbalance. Note that the execution will fall to sequential when both T_S and T_N are greater than N.

5.2 Block Cluster Tree Construction

Compared to CT construction, our parallel implementation of BCT construction is relatively simple. As the computation cost for each recursion step is small, we can obtain sufficient speedups only by parallelizing recursive calls (line 15 in Fig. 4). **Figure 11** shows the pseudocode of the parallel implementation of BCT construction.

The only concern is about the space to which leaf nodes of BCT are stored. In the sequential implementation, they are stored to the global array. However, sharing such a single array controlled by a lock among workers brings large overheads. Therefore, we allocated space for each worker. We can implement such allocation naturally in Tascell as it provides features that enables workers to have their own storage. As Cilk Plus does not provide such features, we allocated a two-dimensional array of the size $n_w \times s$, where n_w is the number of workers and s is an arbitrary size [*4] of

[*4] In our current implementation, s is estimated based on the number of CT nodes. If s is not enough at run time, the program will abnormally stop. We can enhance our implementation to dynamically resize the space but we did not do that for the sake of simplicity of the implementation and for reducing overheads.

the space assigned to each worker.

Cilk Plus for C++ provides *list-reducers* through which we can build an array in parallel with very simple code. However, as list-reducers are implemented for creating linked lists, they lead to worse performance compared to the implementation using the two-dimensional array. Therefore, we did not use this feature.

As in CT construction, we employed a parameter T_N to control the granularity of parallel tasks. A worker calls the sequential version of the recursive function when both of the numbers of elements of given two clusters are greater than T_N. As discussed in Section 3.2, we cannot accurately estimate the depth of (a subtree of) BCT using the number of elements because it depends on admissiblity conditions, but the upper bound of the depth can be estimated in this way.

6. Performance Evaluation

6.1 Evaluation Setup

We evaluate our proposed parallel implementations with the following four datasets from which coefficient matrices of the surface element method are generated [18].

Sphere: a sphere having 5×10^7 elements in its surface.

SphereCube: 10×10×10 spheres placed cubically. Each spherical surface is composed of 101,250 elements.

SpherePyramid: $1^2 + 2^2 + \ldots + 14^2 = 1015$ spheres placed pyramidally. Each spherical surface is composed of 101,250 elements.

Humans: 50 × 100 pairs of human-shaped objects. The surface of each object pair is composed of 19,664 elements.

These four datasets are illustrated in **Fig. 12** [*5] and their characteristics are summarized in **Table 1**. We set $N_{\min}$ to 10 and η to 2 for all measurements.

We measured the performance using a single node of Laurel 2, a supercomputer at the Academic Center for Computing and Media Studies, Kyoto University. The details of the evaluation environment are summarized in **Table 2**.

When evaluating the performance of the parallel implementations, we use the performance of the sequential implementations using C as the baseline of speedups. The performance of the sequential implementation for CT and BCT construction is shown in **Table 3**.

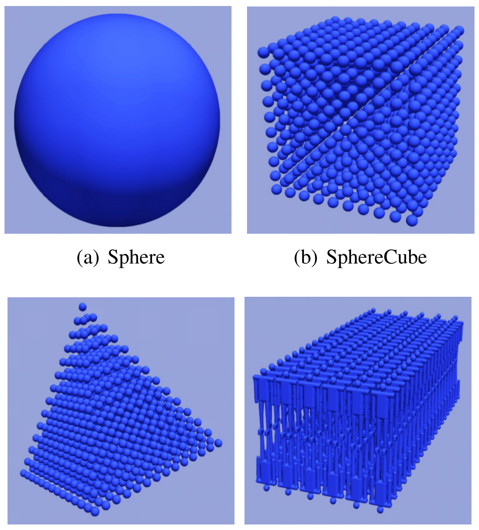

(a) Sphere (b) SphereCube (c) SpherePyramid (d) Humans

Fig. 12 Input datasets used in the evaluations.

Table 1 Characteristics of the input datasets used in the evaluations.

	Sphere	SphereCube	SpherePyramid	Humans
depth of CT, BCT	30	30	31	30
# elements	50,000,000	101,250,000	102,768,750	98,320,000
# nodes in CT	14,489,439	28,648,159	28,969,539	27,043,359
# leaf nodes in BCT	36,696,448	189,114,616	199,092,703	123,061,030

Table 2 Evaluation environment.

	CRAY CS400 2820XT (Laurel 2) (1 node)
CPU	Intel Xeon Broadwell × 2 sockets (2.1GHz, 18 cores/socket) with Hyper-Threading enabled.
Memory	DDR4-2400 128 GB (154 GB/s)
OS	Red Hat Enterprise Linux Server release 7.4 (Maipo)
Compiler	Cilk Plus: Intel C++ Compiler version 17.0.6 with -xavx2 -O3 option
	Tascell: Tascell Compiler version of Jan. 21, 2019 [*6]
	+ GCC version 4.8.5 with -O3 option
	+ Trampoline-based implementation of nested functions in GCC.

Table 3 Performance of the sequential implementation in C (elapsed time in seconds)

	Sphere	SphereCube	SpherePyramid	Humans
CT	19.8	41.9	43.0	42.9
BCT	2.6	15.6	17.6	10.5
Total	22.4	57.5	60.6	53.4

6.2 Cluster Tree Construction

6.2.1 Performance Parameter Tuning

First, we need to tune the three parameters presented in Section 5.1 that can significantly affect the performance.

$\boldsymbol{T_N}$ denotes the threshold of the number of elements that decides whether recursive function calls are executed in parallel in CT construction.

$\boldsymbol{T_S}$ denotes the threshold of the number of elements for deciding whether computations inside a recursive step are parallelized in CT construction. As presented in Section 5.1, we parallelized finding the maximum and minimum elements and the pivoting operation. Workers perform the parallel algorithm when the number of elements to be divided is more than T_S and performs the sequential (in-place) algorithm otherwise.

$\boldsymbol{C}$ is the chunk size used in parallel executions of the pivoting operation in CT construction. When C decreases (the number of the chunks increases), the degree of parallelism increases but the computation cost for prefix-sums also increases.

We tuned these parameters by measuring the performance of 36-worker executions of the Cilk Plus and Tascell implementations using the Sphere dataset by the following parameter search.

(1) Set $T_S = 10^4$ and $C = 10^4$.

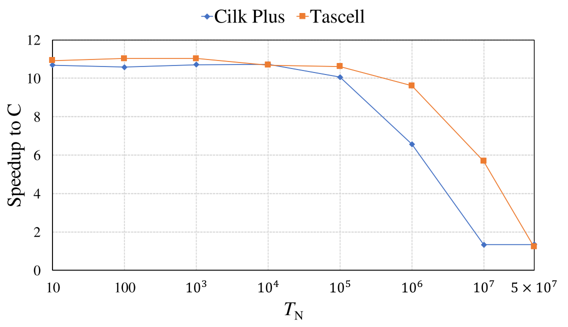
(2) Find the optimal value of T_N within the range $T_N \in \{10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 5 \times 10^7\}$ while fixing T_S and C.

(3) Find the optimal value of T_S within the range $T_S \in \{10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 5 \times 10^7\}$ while fixing T_N and C.

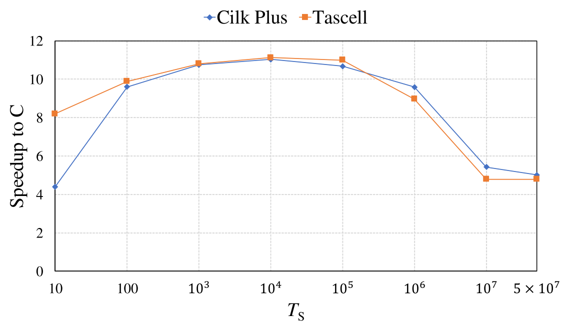
(4) Find the optimal value of C within the range $C \in \{1, 2, 4, 8, 16, 32, 64, 128, 10^3, 10^4, 10^5, 10^6, 10^7, 5 \times 10^7\}$ while fixing

[*5] Figure 12 (d) illustrates only 6 × 10 pairs to make it easy to recognize.

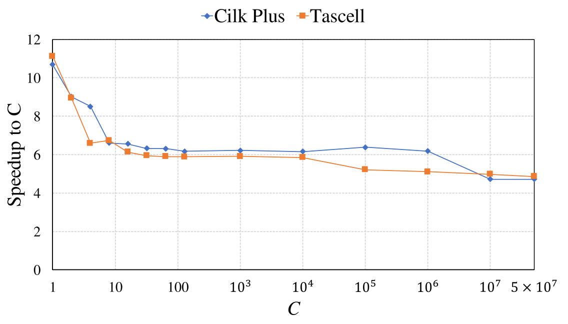
[*6] https://bitbucket.org/tasuku/sc-tascell/commits/158c691ce4059afbbca6c437a6714e8b911756be



(a) The effect of T_N. (T_S,C) is (10000,1) for Cilk Plus and Tascell.



(b) The effect of T_S. (T_N,C) is (100,1) for Cilk Plus and (10000,1) for Tascell.



(c) The effect of C. (T_N,T_S) is (100,10000) for Cilk Plus and (10000,10000) for Tascell.

Fig. 13 Effect of the three parameters on the performance of CT construction (36-worker executions, Sphere).

T_N and T_S.

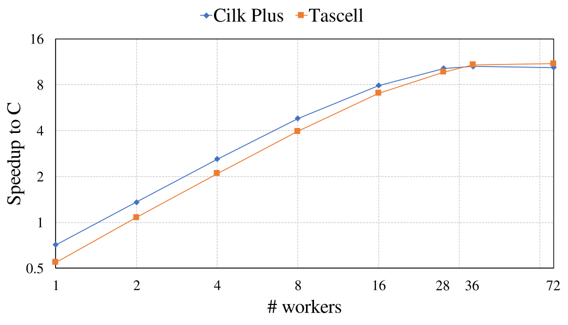
(5) Repeat (2)–(4) until the optimal parameter settings do not change.

As a result, we found the optimal parameter settings $(T_N, T_S, C) = (10^2, 10^4, 1)$ for Cilk Plus and $(T_N, T_S, C) = (10^4, 10^4, 1)$ for Tascell.

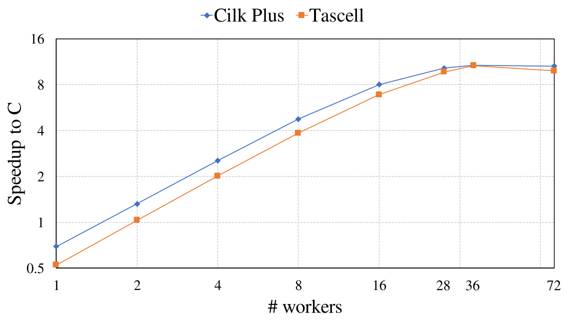
We used these parameter settings for all other performance measurements. The effects of these parameters on the performance are shown in **Fig. 13**. We can see that the performance drops to a 5.3-fold speedup compared to C when $T_S = 5 \times 10^7$. This proves the necessity of parallelizing operations inside the recursive step.

6.2.2 Performance Evaluation

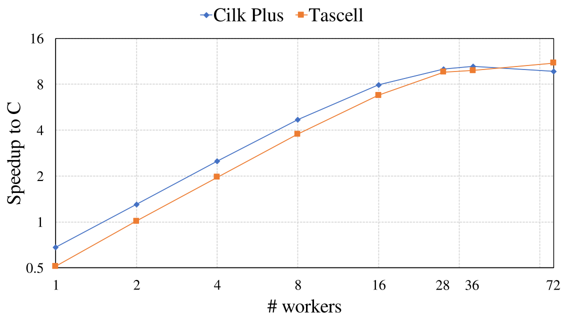
The performance of our parallel implementations for CT construction using Cilk Plus and Tascell is shown in **Fig. 14**. **Table 4** shows the best performance and the number of workers with which the performance is obtained. We can see that for both Cilk Plus and Tascell, the speedups increase proportionally until the number of workers goes up to approximately 28, but the improvement is saturated, or even becomes negative, with a larger number of workers. The possible reasons for this are the additional costs for the parallel pivoting algorithm discussed in Section 5.1 and memory bandwidth saturation.



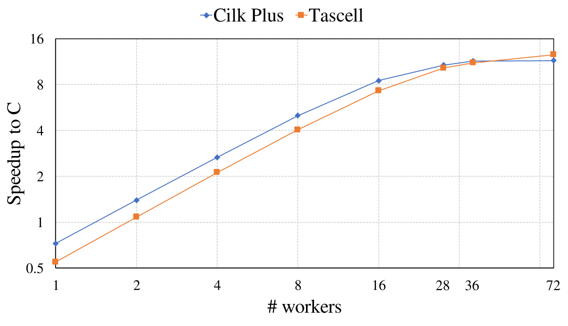
(a) Sphere



(b) SphereCube



(c) SpherePyramid



(d) Humans

Fig. 14 Performance of Cilk Plus and Tascell in CT construction.

In addition, we can see that Cilk Plus demonstrated better performance than Tascell until the number of workers went up to 28, probably due to the more powerful optimization by the Intel Compiler. Cilk Plus and Tascell showed similar performances for 28 to 72 workers. Tascell achieved slightly better performance than Cilk Plus in 72-worker executions for three out of

Table 4 Best performance (speedup to C) achieved by the Cilk Plus and Tascell implementations of CT construction.

	Sphere	SphereCube	SpherePyramid	Humans
Cilk Plus	10.6 (36 workers)	10.7 (36 workers)	10.5 (36 workers)	11.5 (72 workers)
Tascell	11.0 (72 workers)	10.6 (36 workers)	11.0 (72 workers)	12.6 (72 workers)

four datasets. One possible reason for this is because the memory pressure caused by the Cilk Plus runtime is larger than Tascell due to its management of task queues.

6.2.3 Comparison to Implementation without Lightweight Task Parallelism

We also compared our implementations using the task parallel languages to the implementation using OpenMP, which does not have features for lightweight task parallelism. The algorithm of this implementation is summarized as follows.

(1) Construct the shallower part of the CT, the part of the CT where the number of elements is less than $\overline{T_N}$. During this process, all recursive calls are executed sequentially and the inside of each recursive step is executed in parallel using `omp parallel for` loops until the first time that the number of elements is less than T_N. At the same time, add two tasks into the task list for constructing the two subtrees of the CT whose roots are the two children of the node.

(2) Construct the deeper part of the CT by executing the tasks created in (1) in parallel using an `omp parallel for` loop with the option `schedule(dynamic,1)`, we use the sequential version in the parallel for loop.

Performance comparison among the implementations using OpenMP and the task parallel languages is shown in **Fig. 15**. When $\overline{T_N}$ is large, we cannot get good load balance because coarse-grained tasks are not parallelized. When $\overline{T_N}$ is small, we can expect good load balance but the overhead for creating plenty of tasks would be large. In both cases, the OpenMP implementation cannot overperform the implementations using task parallel languages.

6.3 Block Cluster Tree Construction

6.3.1 Performance Parameter Tuning

As presented in Section 5.2, BCT construction has only one execution parameter T_N, which is the threshold that decides whether recursive function calls are executed in parallel.

We tuned T_N by measuring the performance of 288-worker executions using the Sphere dataset. As a result, we obtained the best parameter settings T_N = 10000 both for Cilk Plus and Tascell, as shown in **Fig. 16**.

6.3.2 Performance Evaluation

The performance of our parallel implementations of BCT construction using Cilk Plus and Tascell is shown in **Fig. 17**. **Table 5** shows the best performance and the number of workers with which the performance is obtained. In BCT construction, both the Cilk Plus and Tascell implementations achieved good parallel performance. Tascell achieved its best performance with fewer number of workers than Cilk Plus. Tascell achieved good speedups until the number of workers goes up to 72 and its performance drops down with a larger number of workers. In contrast,

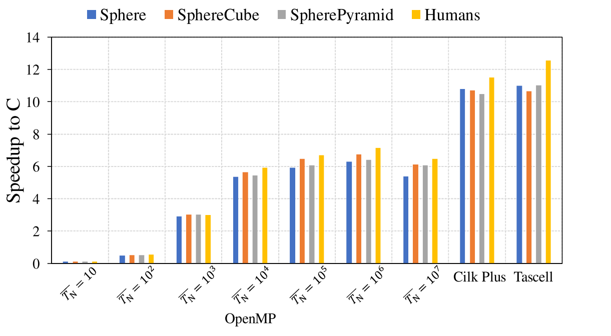


Fig. 15 The performance for CT construction of the OpenMP, Cilk Plus and Tascell implementations (36-thread/worker executions).

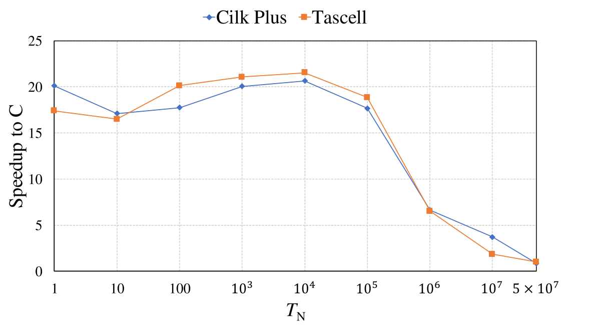


Fig. 16 The effect of T_N on the performance of BCT construction (288-workers, Sphere).

Table 5 Best performance (speedup to C) achieved by the Cilk Plus and Tascell implementations of BCT construction.

	Sphere	SphereCube	SpherePyramid	Humans
Cilk Plus	18.9 (540 workers)	33.0 (540 workers)	37.7 (540 workers)	32.1 (432 workers)
Tascell	22.7 (144 workers)	38.0 (144 workers)	37.6 (72 workers)	38.8 (72 workers)

Table 6 Best total performance (speedup to C) achieved by the Cilk Plus and Tascell implementations of matrix partitioning.

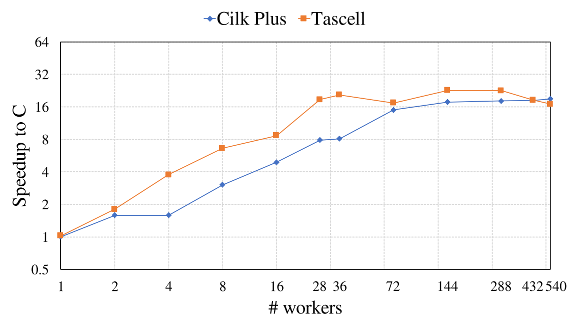
	Sphere	SphereCube	SpherePyramid	Humans
Cilk Plus	10.7 (72 workers)	11.6 (72 workers)	11.3 (72 workers)	12.2 (72 workers)
Tascell	11.5 (72 workers)	12.8 (36 workers)	13.9 (72 workers)	14.5 (72 workers)

Cilk Plus achieved its best performance with a much larger number of workers than the number of cores and its performance does not drop even when the number of workers goes up to 540.

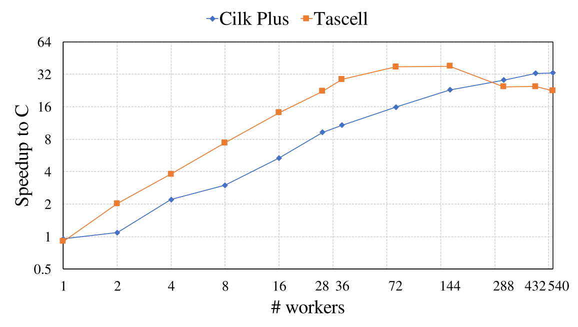
One possible reason that we can obtain better performance with more workers than CPU cores is because we can hide the memory access latency for storing BCT leaf nodes. We need a significantly large number of workers in Cilk Plus to obtain better performance. The reason for this is uncertain, and we will investigate it in a future study.

6.4 Total Performance of Matrix Partitioning

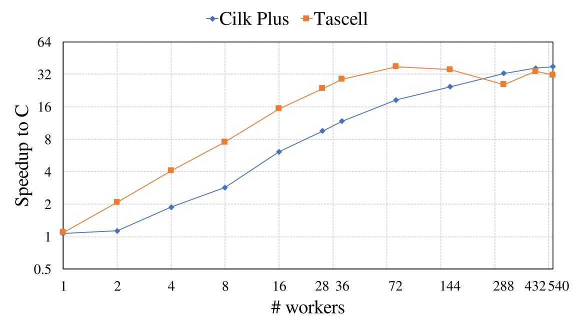
Finally, we demonstrate the total performance of matrix partitioning, i.e., both CT and BCT construction in **Fig. 18**. **Table 6** shows the best total performance achieved by these parallel implementations. As seen in Table 3, the elapsed time to construct CT is longer than that for BCT construction. Thus, the total performance of matrix partitioning is mainly impacted by the performance of CT construction.



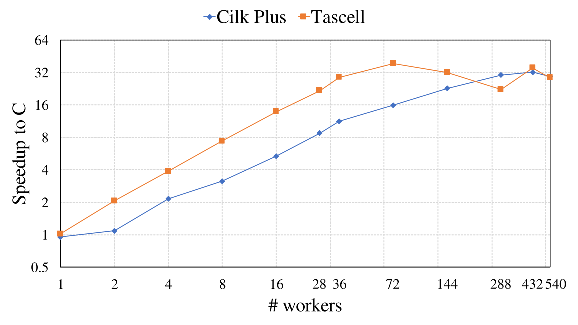
(a) Sphere



(b) SphereCube

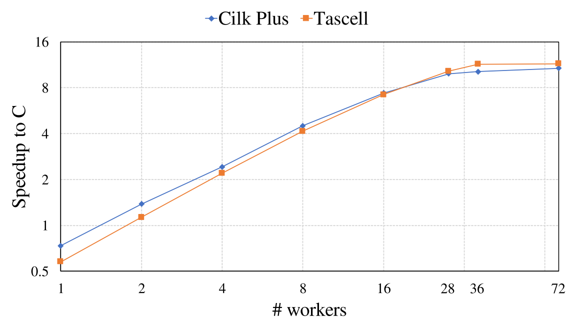


(c) SpherePyramid

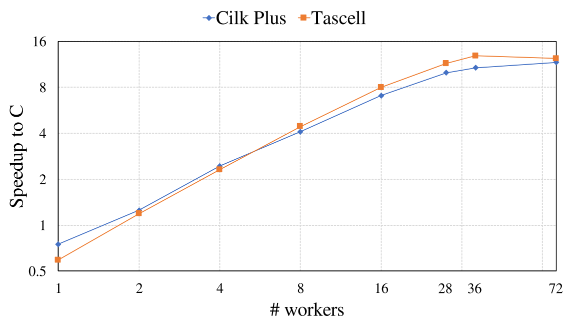


(d) Humans

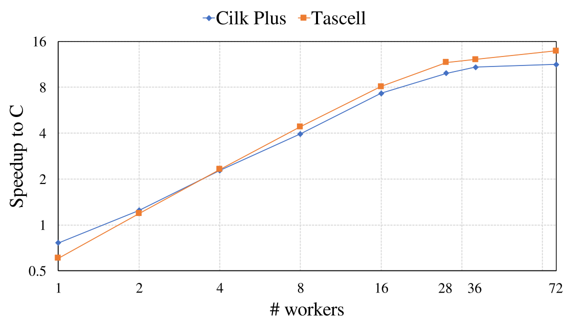
Fig. 17 The performance of Cilk Plus and Tascell in BCT construction.



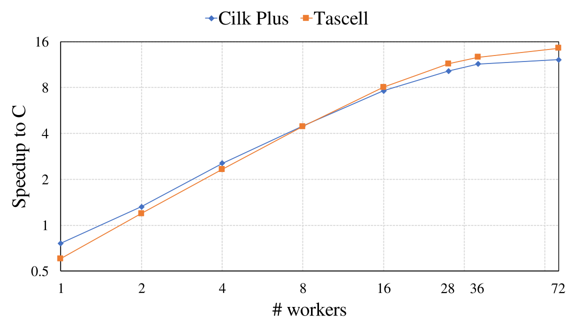
(a) Sphere



(b) SphereCube



(c) SpherePyramid



(d) Humans

Fig. 18 Total performance of the Cilk Plus and Tascell implementations of matrix partitioning.

The results showed that we achieved reasonable and stable speedups for all datasets. Specifically, Cilk Plus achieved 10.7–12.3 times speedups and Tascell achived 11.5–14.5 times speedups compared to the sequential implementation for matrix partitioning.

7. Related Work

As mentioned in Section 1, numerous studies have been conducted that deal with the parallelization of the filling operation in $\mathcal{H}$-matrix construction. Kriemann parallelized $\mathcal{H}$-matrix arithmetic and proposed a parallel implementation of the filling operation on shared memory system in Ref. [6]. Besides filling, they also parallelized matrix-vector multiplication, matrix multiplication, and matrix inversion. Ida et al. proposed implementations of the filling operation on distributed memory systems both by flat-MPI and MPI+OpenMP hybrid parallelization in Ref. [7]. They also packaged these implementations into $\mathcal{H}$ACApK, a library for parallel $\mathcal{H}$-matrix computing. In Ref. [9], Munakata et al. applied dynamic load balancing to hybrid MPI+OpenMP implementations of the filling operation and matrix-vector multiplication.

CT construction in our research can be considered as a special instance of k-D tree construction for k = 3. There are many studies dealing with parallelization of the construction of k-D trees using GPU [19], [20] and multicore CPUs [21], [22], [23], [24]. Especially in Ref. [21], Byn Choi et al. proposed a parallel k-D tree construction algorithm that parallelizes both inside of and across recursive steps. However, their algorithm is different from ours in that they parallelize inside of recursive steps only in the shallower part of the k-D tree and parallelize recursive calls only in the deeper part. We combine both levels of parallelism across the entire tree. Besides, they also propose an in-place parallel algorithm for pivoting, which we can take into our implementations.

In terms of parallel tree construction for other approximation methods for N-body simulations, Taura et al. implemented the tree construction of Fast Multiple Methods (FMM) using task parallelism [25]. In Refs. [26], [27], parallel implementations of tree construction for the Barnes–Hut algorithm are proposed.

In Ref. [28], Saleem et al. parallelized Quicksort using Cilk Plus, which includes a similar pivoting operation with our CT construction. However, they did not realize very good speedups because they did not parallelize the pivoting operation.

8. Conclusion and Future Work

In this paper, we proposed parallel implementations of matrix partitioning in the construction of $\mathcal{H}$-matrix, using Cilk Plus and Tascell. Matrix partitioning is done in two steps: cluster tree (CT) construction and block cluster tree (BCT) construction. In CT construction, we parallelized not only the recursive function call but also the computation inside of recursive steps. Our parallel implementations of BCT construction are relatively simple compared to CT creation, but we needed to assign a private space for each worker to store the BCT leaf nodes.

As a result, compared to a sequential implementation in C, we achieved 10.5–11.5 times speedups by Cilk Plus and 10.6–12.6 times speedup by Tascell for the CT construction. For the BCT construction, speedups using Cilk Plus are 18.9–37.7 times and those using Tascell are 22.7–38.8 times. In regard to the whole process of matrix partitioning, we achieved 10.7–12.2 times speedups by Cilk Plus and 11.5–14.5 times speedups by Tascell.

In future work, we will improve the performance of CT construction by improving the locality of memory accesses by enhancing the work stealing strategy. We will also extend our implementations for distributed memory environments.

Acknowledgments This work was supported in part by JSPS KAKENHI Grant Numbers JP17H01749, JP17K00099 and JP19H04087.

References

[1] Hackbusch, W.: A sparse matrix arithmetic based on $\mathcal{H}$-matrices, part I: Introduction to $\mathcal{H}$-matrices, *Computing*, Vol.62, No.2, pp.89–108 (1999).

[2] Hackbusch, W. and Khoromskij, B.N.: A sparse $\mathcal{H}$-matrix arithmetic, part II: Application to multi-dimensional problems, *Computing*, Vol.64, No.1, pp.21–47 (2000).

[3] Börm, S., Grasedyck, L. and Hackbusch, W.: Hierarchical matrices, lecture note (2005).

[4] Ida, A.: Lattice $\mathcal{H}$-matrices on distributed-memory systems, *2018 IEEE International Parallel and Distributed Processing Symposium* (*IPDPS*), pp.389–398 (May 2018).

[5] Hoshino, T., Ida, A., Hanawa, T. and Nakajima, K.: Design of parallel BEM analyses framework for SIMD processors, Shi, Y., Fu, H., Tian, Y., Krzhizhanovskaya, V.V., Lees, M.H., Dongarra, J. and Sloot, P.M.A. (Eds.), *Computational Science – ICCS 2018*, pp.601–613, Springer International Publishing (2018).

[6] Kriemann, R.: Parallel-matrix arithmetics on shared memory systems, *Computing*, Vol.74, No.3, pp.273–297 (2005).

[7] Ida, A., Iwashita, T., Mifune, T. and Takahashi, Y.: Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters, *Journal of Information Processing*, Vol.22, No.4, pp.642–650 (2014).

[8] Ida, A., Iwashita, T., Ohtani, M. and Hirahara, K.: Improvement of hierarchical matrices with adaptive cross approximation for large-scale simulation, *Journal of Information Processing*, Vol.32, No.3, pp.366–372 (2015).

[9] Munakata, K., Hiraishi, T., Ida, A., Iwashita, T. and Nakashima, H.: Parallel hierarchical matrix arithmetics using dynamic load balancing, *Research Report in High-Performance Computing* (*HPC*), Vol.2015, pp.1–15 (2015). (in Japanese).

[10] Intel Corporation: A quick, easy and reliable way to improve threaded performance—Intel Cilk Plus, available from ⟨https://software.intel.com/en-us/intel-cilk-plus⟩.

[11] Hiraishi, T., Yasugi, M., Umatani, S. and Yuasa, T.: Backtracking-based load balancing, *Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP 2009*), pp.55–64 (Feb. 2009).

[12] Kurz, S., Rain, O. and Rjasanow, S.: The adaptive cross-approximation technique for the 3D boundary-element method, *IEEE Trans. Magnetics*, Vol.38, No.2, pp.421–424 (2002).

[13] Bebendorf, M. and Rjasanow, S.: Adaptive low-rank approximation of collocation matrices, *Computing*, Vol.70, No.1, pp.1–24 (2003).

[14] Intel Corporation: Intel(R) Cilk(TM) Plus, Intel C++ Compiler 17.0 Developer Guide and Reference.

[15] Frigo, M., Leiserson, C.E. and Randall, K.H.: The implementation of the Cilk-5 multithreaded language, *ACM SIGPLAN Notices* (*PLDI '98*), Vol.33, No.5, pp.212–223 (1998).

[16] Hiraishi, T., Yasugi, M. and Yuasa, T.: Experience with SC: Transformation-based implementation of various language extensions to C, *Proc. International Lisp Conference*, pp.103–113, Clare College, Cambridge, U.K. (2007).

[17] Blelloch, G.E.: Prefix sum and their applications, Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (1990).

[18] Iwashita, T., Ida, A., Mifune, T. and Takahashi, Y.: Software framework for parallel BEM analyses with $\mathcal{H}$-matrices using MPI and OpenMP, *Procedia Computer Science*, *International Conference on Computational Science*, *ICCS 2017*, Vol.108, pp.2200–2209 (2017).

[19] Zhefeng, W., Fukai, Z. and Xinguo, L.: SAH kD-tree construction on GPU, *Proc. ACM SIGGRAPH Symposium on High Performance Graphics*, pp.71–78 (2011).

[20] Zhou, K., Hou, Q., Wang, R. and Guo, B.: Real-time kD-tree construction on graphics hardware, *ACM Trans. Graphics* (*TOG*), Vol.27, No.5, pp.126 (2008).

[21] Choi, B., Komuravelli, R., Lu, V., Sung, H., Bocchino, R.L., Adve, S.V. and Hart, J.C.: Parallel SAH k-D tree construction, *Proc. Conference on High Performance Graphics. Eurographics Association*, pp.77–86 (2010).

[22] Di Fatta, G. and Pettinger, D.: Dynamic load balancing in parallel kD-tree k-means, *Proc. 2010 IEEE 10th International Conference on Computer and Information Technology* (*CIT*), pp.2478–2485, IEEE (2010).

[23] Popov, S., Günther, J., Seidel, H.-P. and Slusallek, P.: Experiences with streaming construction of SAH kD-trees, *2006 IEEE Symposium on Interactive Ray Tracing, IEEE*, Vol.26, pp.395–404 (2006).

[24] Shevtsov, M., Soupikov, A. and Kapustin, A.: Highly parallel fast kD-tree construction for interactive ray tracing of dynamic scenes, *Computer Graphics Forum, Oxford, UK: Blackwell Publishing LtdEE*, pp.89–94 (2007).

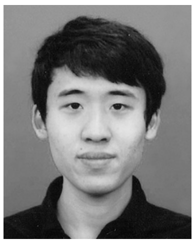
[25] Taura, K., Nakashima, J., Yokota, R. and Maruyama, N.: A task parallel implementation of fast multipole methods, *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp.617–625 (2012).

[26] Shan, H. and Singh, J.P.: Parallel tree building on a range of shared address space multiprocessors: Algorithms and application performance, *Proc. 1st Merged International Parallel Processing Symposium and*

Symposium on Parallel and Distributed Processing, pp.475–484 (Mar. 1998).

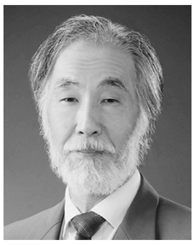
[27] Matsui, K.: Enhancement for task parallel language Tascell and its application to N-body simulation, Master's thesis, Kyoto University (2013). (in Japanese).

[28] Saleem, S., Lali, M.I., Nawaz, M.S. and Nauman, A.B.: Multi-core program optimization: Parallel sorting algorithms in Intel Cilk Plus, *International Journal of Hybrid Information Technology*, Vol.7, No.2, pp.151–164 (2014).



Zhengyang Bai received his B.E. in Software Engineering from East China Normal University in 2015, and Master of Informatics from Kyoto University in 2019. He is currently a doctoral student at the Supercomputing Research Laboratory, Graduate School of Informatics, Kyoto University. His research interest is high performance computing.

Tasuku Hiraishi was born in 1981. He received his B.E. in Information Science in 2003, an Master of informatics in 2005, and his Ph.D. in informatics in 2008, all from Kyoto University. In 2007–2008, he was a fellow of the JSPS (at Kyoto University). Since 2008, he has been working at Kyoto University as an assistant professor at Supercomputing Research Laboratory, Academic Center for Computing and Media Studies, Kyoto University. His research interests include parallel programming languages and high performance computing. He won the IPSJ Best Paper Award in 2010. He is a member of Information Processing Society of Japan (IPSJ), Japan Society for Software Science and Technology (JSSST), and ACM.



Hiroshi Nakashima received his M.E. and Ph.D. from Kyoto University in 1981 and 1991 respectively, and was engaged in research on inference systems with Mitsubishi Electric Corporation from 1981. He became an associate professor at Kyoto University in 1992, a professor at Toyohashi University of Technology in 1997, and a professor at Kyoto University in 2006. His current research interests are in high-performance computing systems and programming on them. He received the Motooka award in 1988 and the Sakai award in 1993. He is a Fellow of IPSJ, and a member of IEEE-CS, ACM, ALP and TUG.

Akihiro Ida was born in Japan in 1971. He received B.Math and M.E. degrees from Nagoya University in 1994 and 1996, respectively. In 2008, Chuo University awarded him a Ph.D. degree in mathematics. In 2000–2012, he researched and developed linear solvers at VINAS Co., Ltd. In 2012–2015, he worked as an assistant professor in the Academic Center for Computing and Media Studies, Kyoto University. He currently works as an associated professor in the Information Technology Center, The University of Tokyo. His research interests include discretization methods for integro-differential equations, numerical linear algebra and high performance computing.

Masahiro Yasugi was born in 1967. He received his B.E. in Electronic Engineering, his M.E. in Electrical Engineering, and his Ph.D. in Information Science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. In 1998–2012, he was an associate professor at Kyoto University. Since 2012, he is a professor at Kyushu Institute of Technology. In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of IPSJ, ACM, and the Japan Society for Software Science and Technology. He was awarded the 2009 IPSJ Transactions on Programming Outstanding Paper Award.