Research on Parallel Hierarchical Matrix Construction

by

Zhengyang Bai

Supervised by

Keiichiro Fukazawa

Submitted to the Graduate School of Informatics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Informatics

Department of Systems Science
Graduate School of Informatics, Kyoto University

January 2022

Research on Parallel Hierarchical Matrix Construction

by

Zhengyang Bai

Abstract

Hierarchical matrix ($\mathcal{H}$-matrix), which is widely used in boundary element method (BEM) and N-body simulations, is an approximated form to represent $N \times N$ correlations of N objects. The construction of the hierarchical matrix is achieved by dividing a matrix into submatrices (matrix partitioning), followed by calculating these submatrices' entries (filling). This thesis proposes parallel implementations for both matrix partitioning and filling using task parallel languages in shared memory systems (SMSs) and distributed memory systems (DMSs).

This thesis proposes implementations of matrix partitioning using task parallel languages, Cilk Plus and Tascell, in SMSs. Matrix partitioning is divided into two steps: cluster tree (CT) construction by dividing objects into clusters hierarchically and block cluster tree (BCT) construction by finding out all cluster pairs at the same level of the cluster tree that satisfies an admissibility condition. Because the two kinds of trees constructed and traversed in these steps are unpredictably unbalanced, using task parallel languages to parallelize both steps is expected to be efficient. To obtain sufficient parallelism in the cluster tree construction, this thesis proposes a two-level parallelization that not only executes recursive calls in parallel but also parallelizes the inside of each recursive step. In the block cluster tree construction, each worker has been assigned its own space so that the workers can store the cluster pairs without using locks.

This thesis extended the implementation above to DMSs as well and proposes two parallel implementation methods for matrix partitioning operations: distributed cluster tree construction (DCTC) and redundant cluster tree construction (RCTC). Only the task parallel language Tascell is used because it employs both intra- and inter-node work stealing. In DCTC, both CT and BCT constructions are parallelized using workers in all computing nodes. As mentioned in the last paragraph, for CT construction, both recursive calls and calculations inside recursive steps are parallelized. However, the calculation inside recursive steps is not suitable for stealing work between different computing nodes because the communication cost does not consider the computation cost. Therefore, we add a new feature, called `node_guard`, to Tascell to allow programmers to control a worker to reject inter/intra-node work steal requests for this type of task, and accept only intra/inter-node requests. In RCTC, CT is constructed in every computing node redundantly by employing only intra-node work stealing. The BCT is then constructed in parallel using workers in all computing nodes. RCTC cannot achieve speedup using multiple computing nodes but can eliminate the data exchange cost incurred by DCTC.

For the filling operation, it is possible to apply task parallelism to filling by treating each submatrix as a parallelization unit. In existing implementations, the partitioning is redundantly executed on every computing node, and filling is executed in parallel by assigning a set of tasks to each worker statically. However, it cannot get a good load balance using such implementations because it is difficult to predict the workload of each task precisely. This thesis proposes a new parallel implementation of the hierarchical matrix construction, where a BCT construction is executed in parallel using all workers of all computing nodes by the task parallel language Tascell, which enables easy and efficient parallelization of tree recursive algorithms by employing a dynamic load balancing strategy, and when a worker finds a BCT leaf during BCT construction, it

executes filling for the submatrix corresponding to the leaf to apply dynamic load balancing to the filling operation as well.

All these implementations are evaluated in numerical experiments with 3D electric field analyses using up to 16 computing nodes each of which has 36 CPU cores.

Acknowledgments

This thesis summarizes my research results at the Supercomputing Research Laboratory, Department of Systems Science, Graduate School of Informatics, Kyoto University from 2017 to 2023. This thesis would not be completed without the support of the people and associations below.

I would like to express my appreciation to my supervisor, Prof. Keiichiro Fukazawa of the Academic Center for Computing and Media Studies (ACCMS), Kyoto University. He not only gives me constructive academic advice for my thesis but also cultivate my ability to make a self-managed study to become a researcher after graduation. I want to express my thankfulness to Prof. Tasuku Hiraishi of the Department of Information and Computer Science, Faculty of Engineering, Kyoto Tachibana University, who was the assistant professor at the ACCMS until 2020. He is the person who lead me to this particular research area and he has been always willing to dedicate his time to help me with my research even after he left the ACCMS. I also want to honor the memory of former Prof. Hiroshi Nakashima of the ACCMS, who accepted me as a student and gave me many advice and suggestions at the beginning of my research.

Besides the teachers above, I also want to show my thankfulness to some other teachers from other laboratories. First, I want to thank Prof. Akihiro Ida of the Research Institute for Value-Added-Information Generation (VAiG), Japan Agency for Marine-Earth Science and Technology (JAMSTEC). He has greatly contributed to the research area of parallelization of $\mathcal{H}$-matrix computation and explained the algorithms of matrix partitioning to me patiently. He also provided data sets that I used in this thesis. This research could not complete without his help. Second, I want to thank Prof. Masahiro Yasugi of the Department of Computer Science and Networks, Kyushu Institute of Technology. He is one of the developers of Tascell, which I used in my research and he gave me much helpful advice in my research. Then, I also want to thank Prof. Toshiyuki Tanaka and Prof. Shin Ishii of the Graduate School of Informatics, Kyoto University. They are the vice chair of my doctoral dissertation committee and gave me many useful and inspiring pieces of advice during the dissertation reviewing.

Other than teachers directly related to this thesis, I would like to thank some other teachers who did not help me directly in this research but in other ways. I would like to show my appreciation to Prof. Qiang Ma of the Graduate School of Informatics, Kyoto University, recruit me as a research assistant which broaden my horizons and provided me a considerable income. I would like to express my thankfulness to Prof. Satoshi Matsuoka, Dr. Aleksandr Drozd and Dr. Emil Vatai of the RIKEN Center for Computational Science (R-CCS) for their patience and inspiring talks during my internship. I also want to thank Prof. Liangyu Chen of the Software Engineering Institute, East China Normal University, who lead me to computer science and cultivated my interest.

I want to thank my parents who have given me both financial and spiritual support selflessly. I could not accomplish my research without their love and support. I would also want to thank Miss Jing Xu and Mr. Jingde Zhou, my colleages and friends at the Supercomputing Research

Laboratory, Kyoto University, for their encouragement.

Last but not the least, the Kyoto University Graduate Division Fellow provided by the Kyoto University Graduate Division under the Japan Science and Technology Agency Pioneering Research Initiated by the Next Generation Program of Japan Science and Technology Agency (JST), and the Kawaguchi Shizu Memorial Scholarship from The Asian Foundation for International Scholarship Interchange also helped me so that I have been free from any part-time jobs and concentrated on my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

For the physics experiments that are difficult to observe, such as electromagnetic field simulation, earthquake cycle, quantum mechanics, etc., the computer simulation is one of the most practical and economical way to present the result. When we use the computer simulation, in particular with boundary element method (BEM) [1, 2] and N-body simulations [3], naïvely, a coefficient matrix that represents the interaction between physical elements is commonly used to solve simultaneous linear equations. Because the number of all interactions between N elements is N^2, such a matrix is dense, and for large values of N, the execution time (i.e. $O(N^3)$ in matrix-matrix multiplications) and memory usage may still be unacceptable or even impractical, even when employing large scale distributed memory systems (DMSs). Consequently, various approximation techniques have been proposed to reduce execution time and memory usage.

Various methods to reduce memory usage and execution time by compressing the submatrices of a matrix have been proposed. The mosaic-skeleton approximations [4] was firstly proposed in 1978 and applied to BEM successfully [5–7]. The panel clustering method [5], the fast multipole method (FMM) [7, 8] and the wavelet approximation methods [9] are proposed later and showed their good performance when applied to BEM [2] where they strongly reduced the memory usage. However, none of these methods efficiently performed matrix operations other than matrix-vector multiplication.

Hierarchical matrices ($\mathcal{H}$-matrices) [10–12] are used as approximation techniques as well. They not only reduce memory usage or execution time of matrix-vector multiplications but also offer a major advantage in the results of matrix arithmetic operations like matrix-matrix multiplication, factorization, and inversion. They are widely used in applications like discretizing integral equations [11, 13–15], practical BEM applications [14, 16–18] or solving elliptic partial differential equations [19,20]. An $\mathcal{H}$-matrix is directly constructed from the interactions between element sets, and not from its dense counterpart. The complexity of memory usage is reduced from $O(N^2)$ to $O(N \log N)$ by hierarchically dividing the matrix into submatrices and replacing the submatrices with their small low-rank approximated forms. Although this technique can significantly reduce the computational cost and the memory usage with reasonable accuracy, it is still critical to parallelize and accelerate the computation for $\mathcal{H}$-matrices, including calculations such as $\mathcal{H}$-matrix-vector and $\mathcal{H}$-matrix-$\mathcal{H}$-matrix multiplications, as well as the $\mathcal{H}$-matrix construction.

The $\mathcal{H}$-matrix construction is achieved by *partitioning* a matrix into submatrices, and *filling*

the submatrices with their entries[1]. The matrix partitioning operation consists of two sub-steps: construction of a cluster tree (CT) and construction of a block cluster tree (BCT)[2]. The filling operation calculates entries of submatrices that are represented as leaf nodes of the BCT. The adaptive cross approximation (ACA) [14, 21] and the improved adaptive cross approximation (ACA+) [12] are algorithms widely used for the filling operation.

Numerous studies have been conducted that deal with accelerating the $\mathcal{H}$-matrix construction in shared memory systems (SMSs) and DMSs, such as [22–28]. However, in most of the existing parallel implementations of the $\mathcal{H}$-matrix construction, matrix partitioning and filling operations are executed independently. They parallelized the filling operation in both SMSs and DMSs using conventional data parallelism, by which parallelization is achieved by parallel executing the same instruction on different data. Kriemann parallelized $\mathcal{H}$-matrix arithmetic and proposed a parallel implementation of the filling operation on SMSs [22] and applied it to $\mathcal{H}$-matrix library Hlib [12]. Besides filling, they also parallelized matrix-vector multiplication, matrix multiplication, and matrix inversion. Ida et al. proposed implementations of the filling operation on DMSs both by flat-MPI, OpenMP and hybrid MPI/OpenMP [23,24]. They also packaged these implementations into $\mathcal{H}$ACApK [28], a library for parallel $\mathcal{H}$-matrix computing. However, few of them parallelized the matrix partitioning operation. This is partially because the cost of the filling operation is much larger than the matrix partitioning (about a thousand times when $N \simeq 10^6$) and the difficulty of parallelization of unpredictably unbalanced tree structure construction in partitioning operation using data parallelism. However, since hundreds of speedups have been achieved for the filling operation using MPI, GPU, and SIMD vectorization [26, 27] and thousands of speedups will be realized in the near future, the partitioning operation will be a bottleneck if it remains sequential. Thus, we should also consider parallelizing the matrix partitioning operation.

There is another problem in the filling operation of the existing parallel implementations of $\mathcal{H}$-matrix construction. Although there are several implementations of parallel filling operation which achieved good performance, they are sharing the same static work assignment strategy among computing nodes, which is based on the estimation of computing complexity. However, we cannot get a good load balance using such implementations because it is difficult to predict the workload of each task precisely (the detail reason is described in Section 6.2.3) and it leads to a decline in performance. This problem becomes serious especially when the actual ranks of low-rank matrices are large on average or when using a large number of computing nodes. Thus, we should consider to eliminate the difference in workloads among computing nodes by dynamic load balancing instead of the static work assignment.

It is expected that the problems mentioned above can be solved using task parallelism. Unlike conventional data parallelism, which achieves parallelization by executing the same instructions on different ranges of data, task parallelism treats tasks as a parallelization unit and achieves parallelization by executing tasks with no dependencies at the same time. Because the sizes of tasks are usually different, to avoid the load imbalance problem, a dynamic load balancing method, called work-stealing, is always utilized in task parallelism. In terms of work-stealing, when a worker (thief) has no task to do, it asks for tasks from another worker (victim), so that all workers will work until no task is left and the workload among each worker is basically even. Parallel languages/libraries using task parallelism are called task parallel languages/libraries. Cilk [29]

[1] To distinguish the elements of submatrices from the physical elements, we call elements of matrices as *entries* in this thesis.

[2] Although the implementations presented in this paper only generate a list of leaf nodes in the BCT, we still refer to this operation as the BCT construction according to conventions in this research area.

is the most well-known task parallel language created by MIT based on C. It is later commercialized as Cilk++ [30, 31] and after the acquisition of Intel, which increased compatibility with existing code, calling the result Cilk Plus [32–34]. After Intel stopped supporting Cilk Plus, MIT is again developing Cilk in the form of OpenCilk [35]. There are many other task parallel languages and libraries, such as OpenMP Task [36], Intel Threading Building Blocks (TBB) [37,38], Massive Threads [39] and Tascell [40]. Most of them share the same concept but are different from each other by different work-stealing strategies and implementations, different languages supports and/or DMSs supports.

1.2 Our Proposal

In this thesis, to solve the problems we mentioned in Section 1.1, we propose parallel implementations for the matrix partitioning in the construction of $\mathcal{H}$-matrices, based on the sequential version proposed in [11] in SMSs and DMSs. Because CTs and BCTs constructed and traversed in these steps are unpredictably unbalanced, we employed task parallel languages, which enable easy and efficient parallelization of the tree recursive algorithms by employing a dynamic load balancing strategy, to parallelize these operations solving the load imbalance problem with reasonable programming cost. Among task parallel languages, we used Cilk Plus [32–34] and Tascell [40] in SMS implementation and only employed Tascell in DMS implementation, because Cilk Plus does not support inter-node work-stealing. To achieve a good performance of CT construction in SMSs, we propose a "two-level" parallelization, where not only recursive calls but also the computation inside of each recursive step are parallelized.

We extend our implementation in SMSs to DMSs in pursuit of better performance and larger data capacity. We propose two implementations for matrix partitioning in DMSs, Distributed Cluster Tree Construction (DCTC) and Redundant Cluster Tree Construction (RCTC). In DCTC, both CT and BCT construction are parallelized by all workers in all computing nodes, but this implementation has an inevitable overhead of exchanging CT results among all computing nodes because the CT results are distributed to different computing nodes randomly. In RCTC, CT is built inside of each computing node redundantly. Though, CT construction cannot get benefits from the multi-nodes parallelization, this implementation avoids the overhead of CT result exchanging. To get reasonable performance in CT construction of DCTC, we enhanced Tascell on its node-aware work-stealing strategy.

To the problem of load balance in existing parallel filling implementations, we propose a new parallel implementation for filling operation in $\mathcal{H}$-matrix construction where a BCT construction is executed in parallel using all workers of all computing nodes based on the technique we used on BCT parallelization in DMSs. In addition, a worker executes filling for each submatrix immediately when a worker finds a BCT leaf node corresponding to the submatrix during BCT construction. To get good load balance, we parallelized BCT construction using the task parallel language Tascell, which enables easy and efficient parallelization of the tree recursive algorithms by employing a dynamic load balancing strategy.

1.3 Contributions

This thesis makes the following contributions:

- We propose parallel implementations for matrix partitioning in the construction of $\mathcal{H}$-matrix in SMSs using Cilk Plus and Tascell based on the sequential implementation of the $\mathcal{H}$ACApK library. 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.
- We propose two parallel implementation methods for matrix partitioning in the construction of $\mathcal{H}$-matrices using Tascell in DMSs: Distributed Cluster Tree Construction (DCTC) and Redundant Cluster Tree Construction (RCTC).
  - In DCTC, both CT and BCT are constructed in parallel using workers for all computing nodes. To achieve load balance, we employed the cross-node work-stealing capability of Tascell. Because the result of CT construction is distributed among all computing nodes, we need to exchange the result (CT nodes) among computing nodes prior to BCT construction.
  - In RCTC, CT is constructed in every computing node redundantly by employing only intra-node work stealing. BCT is constructed using workers in all computing nodes, as in DCTC. Although we do not expect any speedup using multiple computing nodes in CT construction, we may achieve higher total performance because we do not need to exchange the result of CT construction among computing nodes. (Although we need to "reorder" CT nodes in each computing node as described later in Section 5.3.2, the cost of this process is expected to be much smaller than the data exchange in DCTC.)
- To make Tascell to work better in DMSs, we add a new feature `tcell-broadcast` to make all computing nodes to do the same instructions at the same time. To achieve better performance for CT construction in DCTC, we enhanced the Tascell language to enable annotations, called `node_guard`, for parallel statements to specify whether a kind of task can be stolen by workers only in internal/external nodes.
- We propose a new parallel implementation of $\mathcal{H}$-matrix construction using Tascell, where BCT construction is parallelized and filling operation for each submatrix begins immediately when a worker finds the corresponding BCT leaf node to make filling operation can be parallelized in the parallel BCT construction using dynamic load balancing provided by Tascell.
- We evaluated the performance of all the implementations in numerical experiments with 3D electric field analyses using BEM with four datasets, which at most has approximately 10^8 elements. For matrix partitioning in SMSs, the results indicated that we obtained remarkable speedups with both task parallel languages, Cilk Plus and Tascell. We showed that, in CT construction, our implementations using task parallel languages significantly overperform a naïve implementation using OpenMP.
- We evaluated the performance of our parallel implementations of matrix partitioning in DMSs with up to 8 nodes $\times$ 36 workers. Our RCTC implementation achieved better performance using multiple computing nodes than the existing implementation in SMSs. Although our DCTC implementation cannot achieve speedup using multiple nodes owing to the data exchange cost of a CT, we did obtain speedup in the CT construction phase, which proves the usefulness of the proposed enhancement to Tascell.

- We evaluated the proposed filling operation using Tascell on DMSs using up to 16 nodes × 36 workers, and achieved better scalability and up to 1.9-fold speedups compared to the implementation using the static task assignment strategy in existing implementations.

1.4 Position and Scope of the Thesis

In this thesis, We parallelized the matrix partitioning process by task parallel languages, which has rarely been parallelized in the existing implementations and we improved the performance of the filling process by solving the load balance problem of the existing implementations using a task parallel language Tascell. Because the construction of $\mathcal{H}$-matrices is accelerated by the proposal in this thesis, all the applications that can use $\mathcal{H}$-matrix can get benefits from this thesis. As far as we know, this thesis is the first parallel implementation of the whole process of $\mathcal{H}$-matrix construction.

To get better performance of CT construction in DMSs, we propose a node-aware work-stealing strategy by which programmers can control whether a task can be accepted or not based on the data locality. There are several proposals for work-stealing strategies that take data locality into consideration, but only a few of them support DMSs and they are different from our strategy. We are looking forward to applying this strategy to other applications.

We would like to clarify the scope of this thesis that this thesis focuses on the improvements of the performance of $\mathcal{H}$-matrix construction using parallelization techniques. We use the implementation of $\mathcal{H}$ACApK as the baseline and construct the same $\mathcal{H}$-matrices when we use the same data inputs and parameters. Therefore, other methods accelerating the $\mathcal{H}$-matrix construction by modifying the $\mathcal{H}$-matrix construction algorithm (including the existing $\mathcal{H}$-matrices variants like $\mathcal{H}^2$-matrices [41] or *lattice* $\mathcal{H}$-matrices [42]), or the approximation error limitation is not discussed in this thesis. We use numerical experiments with 3D electric field analysis using BEM to evaluate the performance of our implementations, but the 3D electric field analysis itself or the BEM is not the focus of this thesis.

1.5 Organization of the Thesis

The organization of this thesis is as follows.

Chapter 2 introduces the hierarchical matrix ($\mathcal{H}$-matrix). We first define the problem targeted in this thesis. Then, we describe the sequential $\mathcal{H}$-matrix construction algorithm, including matrix partitioning and filling.

Chapter 3 introduces what task parallel language is using Cilk Plus and Tascell, which we used in this thesis, as examples and show the differences between them.

Chapter 4 provides the parallel implementation of matrix partitioning in shared memory systems. We implement a two-level parallelization that not only executes recursive calls in parallel but also parallelizes the inside of each recursive step. We evaluate the implementation at the last of the chapter and show the advantage of our implementation compared with OpenMP implementation.

Chapter 5 describes two implementations, Distributed Cluster Tree Construction (DCTC) and Redundant Cluster Tree Construction (RCTC), to extend the parallel implementation in SMSs to DMSs. We also describe the details of the enhancement of Tascell for better performance in DMSs in this chapter. At last, we evaluate the implementations at the last of the chapter.

Chapter 6 proposes a parallel $\mathcal{H}$-matrix construction algorithm where BCT construction is parallelized by all workers in all computing nodes and filling operation for each submatrix begins immediately when a worker finds the corresponding BCT leaf node in SMSs and DMSs. We evaluate the implementations at the last of the chapter and shows better performance of our proposal comparing with the existing implementations.

Chapter 7 summarizes related work that is relevant to the parallelization of $\mathcal{H}$-matrix construction, related applications using similar algorithm and work-stealing strategies of task parallel languages.

Finally, Chapter 8 concludes this thesis and describes future work.

Chapter 2

Hierarchical Matrices

2.1 Overview

An $\mathcal{H}$-matrix is a collection of submatrices created by matrix partitioning, where all submatrices neither overlap nor hold gaps between them. Submatrices are also referred to as leaf matrices. Figure 2.1(a) shows an example of an $\mathcal{H}$-matrix structure. This $\mathcal{H}$-matrix is derived from a BEM analysis using the input data illustrated in Figure 2.1(b), and represents the interactions among elements distributed on the surface of two spheres[1].

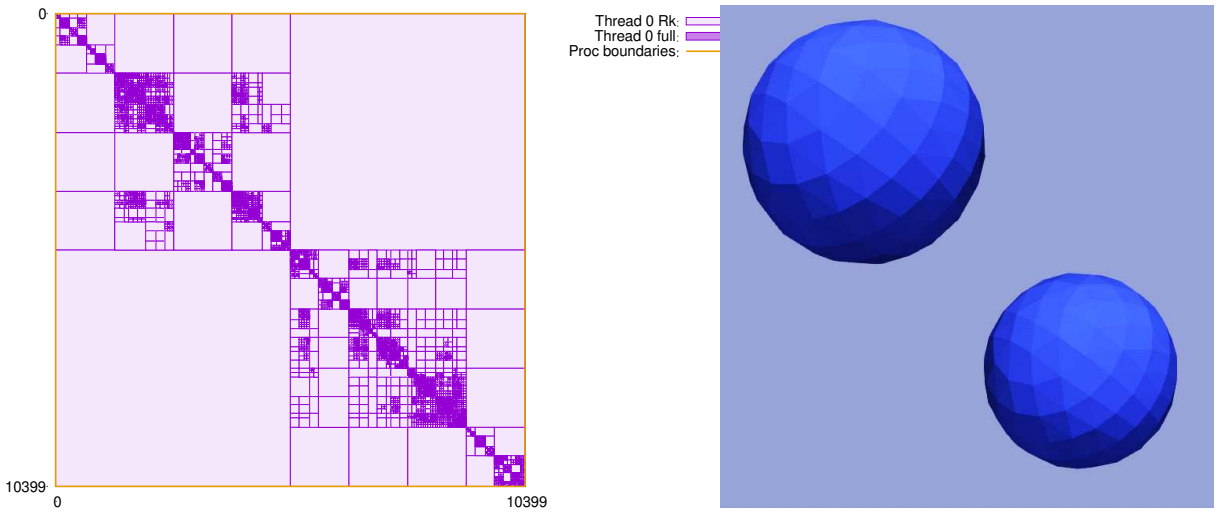
An $l \times m$ leaf matrix is approximated by the product of two low-rank matrices of sizes $l \times k$ and $k \times m$, where k denotes the rank of these matrices. Therefore, restricting memory usage to $O(NK \log N)$, where K is the upper limit of k, is possible. K can be tuned based on accuracy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

For matrix X, we let $\mathrm{R}(X)$ denote the range of X's indices. Moreover, let $[n_1, n_2]$ denote the 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 partition P of $\mathrm{R}(A)$ as a set of subranges that satisfies the following conditions: (1) $\bigcup_{p \in P} p = \mathrm{R}(A)$ (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 that corresponds to the subrange $p \in P$. When a $\mathcal{H}$-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 $V_p \cdot W_p$ as follows:

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

where $\#S$ denotes the number of elements in set S. We then define $k_p \in \mathbb{N}$ as the rank of $\tilde{A}|^p$: Typically, k_p is significantly 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 $p \in P$, $\mathcal{H}$-matrix $\tilde{A}$ is equal to A.

[1]The surface in Figure 2.1(b) is composed of triangles, whose gravity centers are considered elements. This is also true for Figure 4.3, which appears later in Chapter 4.3.



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

(b) Example of input data

Figure 2.1: Example of a structure of $\mathcal{H}$-matrix and input data. The number of elements is 10,400.

2.2 Construction of Hierarchical Matrices

As mentioned in Chapter 1, an $\mathcal{H}$-matrix can be constructed using 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 using an example. We present the matrix partitioning algorithm and filling algorithm in detail in Section 2.3 and Section 2.4.

In general, a dense matrix does not have an obvious structure. However, dense matrices appearing in numerical analyses, such as BEM, usually exhibit an implicit structure that can be represented by approximated submatrices. In $\mathcal{H}$-matrix construction, we should find as many 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 space and the interaction between each element pair decreases quadratically according to the distance between them. Under this condition, if the distance between the two sets of elements E_1 and E_2 is sufficiently large, it is not necessary to strictly consider the interactions between the two sets' individual elements. Instead, we can select representative elements from these sets to approximate the overall interaction between the sets. This relationship 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 E_i ($i \in \{1, 2\}$) into two subsets $E_{i,1}$ and $E_{i,2}$ according to the geometric distribution of its elements.
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 sub-spaces containing the clusters and the distance between clusters.

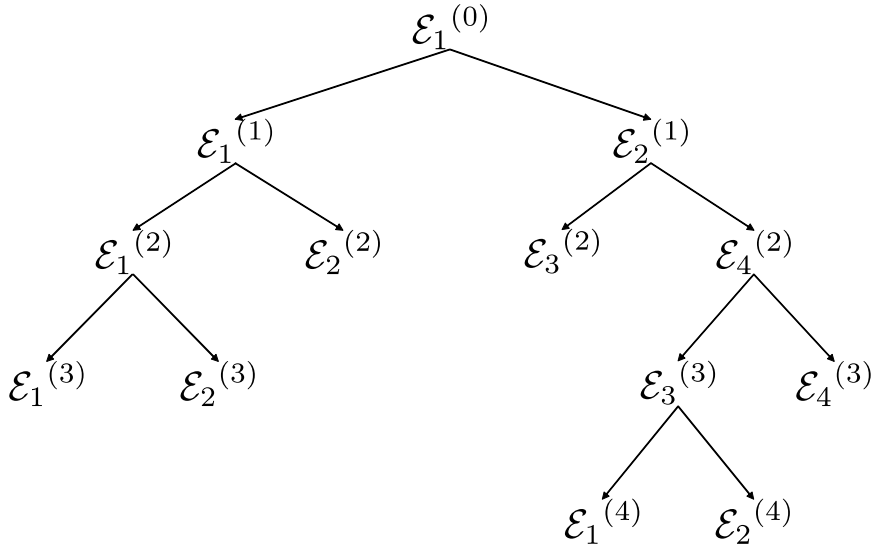


Figure 2.2: Example of a cluster tree structure.

3. If the accuracy is acceptable, we determine 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 acceptable, apply 1–4 to the cluster pair recursively.

The filling operation follows matrix partitioning, and the element values of the leaf matrices are calculated. Adaptive cross approximation (ACA) [14, 21] and ACA+ [12] are standard algorithms used for this operation.

2.3 Matrix Partitioning Algorithm

Although Section 2.2 outlines a single recursive procedure for matrix partitioning, this procedure can be broken into a series of the following processes, such that a cluster is split only 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 determine the matrix structure recursively.

In this section, we present details pertaining to the sequential algorithms and parallel implementations in SMSs by Tascell.

2.3.1 Cluster Tree Construction

First, we present the algorithm used to construct CT. The cluster $\mathcal{E}_1{}^{(0)} = \{e_0, ..., e_{N-1}\}$ containing all input elements, is treated as the root node of the CT. The children of a CT node are created by dividing the cluster into two subclusters. The grandchildren of each node can be created by dividing the corresponding cluster in the same manner. Such division operations are repeated recursively until the cluster size becomes smaller than the threshold $N_{\min}$. N_{min} can control the depth of CT. Practically, it is common to set N_{min} to $2K$, or a specific number like 16. Figure 2.2 illustrates an 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 eright are obtained by reordering e
    // in-place in our actual sequential implementation.
    elem[] eleft = {e | e.x < M};
    elem[] eright = {e | e.x ≥ M};
    t.child[0] = buildClusterTree(eleft);
    t.child[1] = buildClusterTree(eright);
  }
  return t;
}
```

Figure 2.3: Pseudocode of the algorithm for CT construction (for simplicity, we show the case where elements are placed in one-dimensional space).

Each recursion step provides several ways to divide a cluster. In BEM, elements are often divided by pivoting. For example, when elements are placed in a one-dimensional 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 2.3 shows the pseudocode of the CT construction algorithm using this division strategy.

In 3D space, the largest value is obtained 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 correspond to the lengths of the edges of the *bounding box* surrounding the cluster. The corresponding axis was selected as the pivot axis. The pivot value M is set as $(\max_{e\in\mathcal{E}} e.x + \min_{e\in\mathcal{E}} e.x)/2$, $(\max_{e\in\mathcal{E}} e.y + \min_{e\in\mathcal{E}} e.y)/2$, or $(\max_{e\in\mathcal{E}} e.z + \min_{e\in\mathcal{E}} e.z)/2$ based on the pivot axis and the elements are divided based on the coordinate values of the pivot axis.

In each recursion step, the number of elements contained in the two subclusters is generally uneven. Thus, the CT constructed using this algorithm becomes unbalanced. Additionally, because the computational cost of cluster division 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 imbalance 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 in its roots.

2.3.2 Block Cluster Tree Construction

For BCT construction, we used the CT constructed in the previous step. A node in the BCT at an arbitrary level corresponds to a pair of two CT nodes (corresponding to two clusters) at the same level. If a pair of clusters satisfies an *admissibility condition*, the corresponding BCT node does not have child nodes, which means that interaction between the clusters can be approximated by a low-rank submatrix. If the admissibility condition cannot be satisfied and at least one of the corresponding CT nodes is a leaf, we determine that the corresponding submatrix cannot be

```
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]);
  }
}
```

Figure 2.4: Pseudocode of the algorithm for BCT construction.

approximated, and create 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 child node pairs of CT nodes. Because any self-pairs of CT nodes cannot be admissible, the depth of BCT is equal to that of CT. The number of BCT nodes at a level is at most the square of the number of CT nodes in the level corresponding to all possible combinations of CT nodes. However, it is usually much less than that, especially at deeper levels.

The admissibility condition can prevent the approximation error from becoming too large. Thus, it can be tuned based on the accuracy requirements and nature of the problem setting. For example, the common setting of the admissibility condition in BEM or N-body simulation is that the distance between two clusters is longer than a certain multiple of the two subsets' diameter, and can be expressed through the following inequalities:

$$\eta \times diam(\tau) \leq dist(\tau, \sigma) \wedge \ \eta \times diam(\sigma) \leq dist(\tau, \sigma) \tag{2.1}$$

where $diam(\tau)$ is the diameter of the cluster τ, $dist(\tau, \sigma)$ is the distance between clusters τ and σ, and η is a constant. It is noteworthy 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, and $dist(\tau, \sigma)$ is the distance between the two nearest points on the surfaces of the corresponding bounding boxes.

The pseudocode of the algorithm for BCT construction is shown in Figure 2.4. Two walkers, `t` and `s`, traverse the cluster tree from its root node to verify the admissibility condition. Note that the execution tree of this recursive algorithm forms a structure of the BCT; however, rather than creating the tree's structure, we only list its leaf nodes. These nodes correspond to the set of submatrices, and the tree structure is not used in subsequent operations.

Additionally, a BCT constructed using this algorithm is unpredictably unbalanced because of not only the imbalance of the CT, but also because not every BCT node has children depending on the admissibility condition.

```
1  leaf_node[] leaf_node_list;    //result of BCT construction
2  void filling (leaf_node[] leaf_node_list,
3                elem[] e,
4                ε,  //upper limit of the approximation error
5                counter){
6      for( i=0; i<=counter; i++){
7          if(leaf_node_list[i] is "full submatrix"){
8              //calculate interactions without approximation
9              cal_interaction(leaf_node_list[i], e);
10         }
11         if(leaf_node_list[i] is "low-rank submatrix"){
12             //calculate approximated matrices by ACA+
13             ACA_plus(leaf_node_list[i], e, ε);
14         }
15     }
16 }
```

Figure 2.5: Pseudocode of the algorithm for filling operation.

2.4 Filling Algorithm

Figure 2.5 shows the pseudocode of the sequential filling algorithm. As defined in Section 2.1, P is the set of all leaf matrices determined in the matrix partitioning process. That is, P corresponds to the set of BCT leaf nodes. The filling operation calculates all the entries of all $\tilde{A}|^p$ for $p \in P$. When $\tilde{A}|^p$ is a full matrix, we directly fill the leaf matrices by $\tilde{A}|^p = A|^p$ using Equation (2.4). It is represented by function `cal_interaction` in line 9 of Figure 2.5. Otherwise, we use two low-rank matrices to approximate $\tilde{A}|^p$ by cross approximation. There are two well-known algorithms to calculate entries of leaf matrices, the adaptive cross approximation (ACA) [14, 21] and the improved adaptive cross approximation (ACA+) [12]. We only employed the ACA+ in the performance evaluations in this study.

2.4.1 Full Matrices Filling in the Case of Boundary Element Method

$\mathcal{H}$-matrices are different from application to application in order to get a better approximation result based on the particular situation of an application. Here we briefly describe the formulation of a system of linear equations provided in [23, 24], whose coefficient matrices are used in the BEM of 3D electric field analyses to evaluate the performance.

Let H be a Hilbert space of functions on a $(d-1)$-dimensional domain $\Omega \subset \mathbb{R}^d$, and H' be the dual space of H. For $u \in H, f \in H'$ and a kernel function of a convolution operator $g : \mathbb{R}^d \times \Omega \to \mathbb{R}$, we have a typical operator in the form

$$\int_\Omega g(x, y)u(y)\, dy = f. \tag{2.2}$$

We can calculate this equation numerically using Ritz-Galerkin method, where function u can be approximated from a n-dimensional subspace. Given a basis $(\varphi_i)_{i\in\beth}$, where $\beth := \{1, ..., n\}$ is

the set of indices, we can have an approximant u^h to u, which is written by a coefficient vector $\phi = (\phi_i)_{i\in\beth}$ satisfying $u^h = \sum_{i\in\beth} \phi_i\varphi_i$. Equation (2.2) can be reduced to

$$A\phi = b, \tag{2.3}$$

where the entries of A and b can be calculated by

$$A_{ij} = \int_\Omega \varphi_i(x) \int_\Omega g(x,y)\varphi_j(y)\, dydx, \text{for all } i, j \in \beth \tag{2.4}$$

and

$$b_i = \int_\Omega \varphi_i(x) f\, dx, \text{for all } i, j \in \beth. \tag{2.5}$$

In BEM, the coefficient matrix A_{ij} is dense because the kernel function is written in the form $g(x,y) = |x-y|^{-p}, p > 0$.

2.4.2 Low-rank Approximation

An intuitive approximate for submatrices is the singular value decomposition (SVD). However, this method is not widely used because the computation cost is too large. The cross approximation [43] is proposed to do the approximation of this kind of problem with a fixed rank, based on an idea that, for a block $p = I_p \times J_p$, we can choose a small set $s \subset I_p$ of pivot columns and small set $t \subset J_p$ of pivot rows so that for a matrix $S \in \mathbb{R}^{s\times t}$,

$$\tilde{A}|^p := [A|^p]_{J_p\times s} \cdot S \cdot [A|^p]_{t\times I_p}, \tag{2.6}$$

where the rank of $\tilde{A}|^p$ is $\min\{\#s, \#t\}$. There is a naïve way, called *full pivoting*, to determine the pivot index pair (i, j) by the maximal entry $|[A|^p]_{i,j}|$. However, it is necessary for cross approximation with full pivoting to calculate all the entries in $A|^p$ in advance. *Partial pivoting* can solve the problem. The idea of partial pivoting is to maximize $|[A|^p]_{i,j}|$ only for one of the two indices i or j and keep the other one fixed. When we fix an arbitrary row i^*, we can calculate the maximizer of the corresponding column by

$$j^* := \arg\max_{j\in J_p} |[A|^p]_{i^*,j}|, \tag{2.7}$$

to get the pivot pair (i^*, j^*), which is not a maximizer for all pairs of indices but for one row.

Adaptive Cross Approximation

The adaptive cross approximation (ACA) [14, 21] is an improved version of the cross approximation with partial pivoting. The ACA determines the rank adaptively instead of a fixed one and eliminates the matrix S in Equation (2.6), so that we can simply use the production of two low-rank matrices V_p and W_p to represent $\tilde{A}|^p$. The pivot column-vector $[A|^p]_{*,j}$ and pivot row-vector $[A|^p]_{i,*}$ are chosen in the same way as cross approximation with partial pivoting, and append the following vectors respectively to V_p and W_p:

$$[V_p]_{*,k} = \left([A|^p]_{*,j} - \textstyle\sum_{l=1}^{k-1}[W_p]_{i,l}[V_p]_{*,l}\right)/\delta_V(i,j) \tag{2.8}$$

$$[W_p]_{k,*} = \left([A|^p]_{i,*} - \textstyle\sum_{l=1}^{k-1}[V_p]_{l,j}[W_p]_{l,*}\right)/\delta_W(i,j) \tag{2.9}$$

where $\delta_V(i,j) = 1$, $\delta_W(i,k) = [A|^p]_{i,j}$. This operation is repeated until V_p and W_p sufficiently approximate $A|^p$.

The important point is that we do not need the entire $A|^p$ to calculate V_p and W_p, we only need some column and row vectors of it. The number of the vectors r_p is the rank of the approximated matrix $\tilde{A}|^p$. When we define $\|\cdot\|_F$ is the Frobenius norm of a matrix and the upper limit of the approximation error is ε, the condition

$$\frac{\|A - \tilde{A}\|_F}{\|A\|_F} \leq \varepsilon \tag{2.10}$$

should be satisfied to obtain a sufficient approximation. This condition is approximately equivalent to:

$$\frac{\|[V_p]_{*,r_p}\|_2 \|[W_p]_{r_p,*}\|_2}{\sqrt{\sum_{k=1}^{r_p} (\|[V_p]_{*,k}\|_2 \|[W_p]_{k,*}\|_2)^2}} \leq \varepsilon \tag{2.11}$$

Thus, we can find r_p that satisfies the condition above without referring to the entire dense matrix $A|^p$.

Improved Adaptive Cross Approximation

In some cases [12], the ACA cannot do the approximation due to pivoting strategy. Therefore, an improvement of ACA, called improved adaptive cross approximation (ACA+) [12], is proposed. The difference between ACA and ACA+ is in the way of choosing the pivot column j and pivot row i. In the ACA+, a reference column of $A|^p$ is defined by

$$(a^{\text{ref}})_i := [A|^p]_{i,j_{\text{ref}}}, i \in \{1, ..., n\}, \tag{2.12}$$

where the choice of the index j_{ref} is arbitrary. Then, we can determine the reference row by

$$i_{\text{ref}} := \arg\min_{i \in \{1,...,n\}} |(a^{\text{ref}})_i|, \tag{2.13}$$

$$(b^{\text{ref}})_j := [A|^p]_{i_{\text{ref}},j}, i \in \{1, ..., m\}. \tag{2.14}$$

Although the choice of i_{ref} is counterintuitive, because i_{ref} corresponds to a row that produces a small entry in the column j_{ref}, it can avoid the same problem that approximation fails in some cases in the ACA. Then, the pivoting strategy of the ACA+ is listed below, based on the knowledge of a^{ref} and b^{ref}:

step 1-1: Determine the index i^* of the largest entry in modulus in a^{ref}, $i^* := \arg\max_{i=1,...,n} |a_i^{\text{ref}}|$.

step 1-2: Determine the index j^* of the largest entry in modulus in b^{ref}, $j^* := \arg\max_{i=1,...,m} |b_j^{\text{ref}}|$.

step 1-3: If $|a_{i^*}^{\text{ref}}| > |b_{j^*}^{\text{ref}}|$ then

1. compute the vector $b \in \mathbb{R}^m$ with entries $b_j := [A|^p]_{i^*,j}$;
2. redefine the column pivot index $j^* := \arg\max_{i=1,...,m} |b_j|$;
3. compute the vector $a \in \mathbb{R}^n$ with entries $a_i := [A|^p]_{i,j^*} / [A|^p]_{i^*,j^*}$;

step 1-4: otherwise

1. compute the vector $a \in \mathbb{R}^n$ with entries $a_i := [A|^p]_{i,j^*}$;
2. redefine the column pivot index $i^* := \arg\max_{i=1,\dots,n} |a_i|$;
3. compute the vector $b \in \mathbb{R}^m$ with entries $b_j := [A|^p]_{i^*,j}/[A|^p]_{i^*,j^*}$;

step 1-5: The matrix ab^T is the cross approximation to $A|^p$.

In the next step, we can omit determining a new reference row and column by updating the already existing ones, except for the case that the reference row/column is the pivot row/column. In this case, we need to define a new reference row or column. The rows and columns we used as pivots before are denoted by $\mathcal{P}_{rows}$ and $\mathcal{P}_{cols}$.

step 2-1: Update the reference row and column:

$$a^{\text{ref}} := a^{\text{ref}} - a \cdot b_{j_{\text{ref}}}, \tag{2.15}$$

$$b^{\text{ref}} := b^{\text{ref}} - a_{i_{\text{ref}}} \cdot b, \tag{2.16}$$

step 2-2: Determine both i^* and j^* as previous step above if $i^* = i_{\text{ref}}$ and $j^* = j_{\text{ref}}$, because the possible indices is restricted to non-pivot ones.

step 2-3: If only $i^* = i_{\text{ref}}$ then we choose a new reference row b^{ref} corresponding to a reference index i_{ref} that has not yet been a pivot index and that is consistent to j_{ref}:

$$i_{\text{ref}} := \underset{i \in \{1,\dots,n\} \setminus \mathcal{P}_{rows}}{\arg\min} |a_i^{\text{ref}}|, \tag{2.17}$$

step 2-4: If only $j^* = j_{\text{ref}}$ then we choose a new reference column a^{ref} corresponding to a reference index j_{ref} that has not yet been a pivot index and that is consistent to i_{ref}:

$$j_{\text{ref}} := \underset{j \in \{1,\dots,m\} \setminus \mathcal{P}_{cols}}{\arg\min} |b_j^{\text{ref}}|, \tag{2.18}$$

Note that the ACA+ is different from ACA only in the strategy of pivot choosing, they share the same formulations to calculate the entries as Equation (2.8) and (2.9), where appropriate $(\delta_V(i,j), \delta_W(i,j)) \in \{(1, [A|^p]_{i,j}), ([A|^p]_{i,j}.1)\}$ is used depending on the selected pivot vectors in ACA+.

Chapter 3

Task Parallel Languages

3.1 Introduction

In terms of parallelization, the most common way is to assign data to different processes or threads using MPI and/or OpenMP. This kind of parallelism is called data parallelism which treats data as a parallelization unit and the parallelization is implemented by doing the same kind of calculation on different groups of data. When the amount of data has a certain correlation with computing complexity, the data parallelism usually gets good performance because the workload among processes and threads is basically even. However, when we cannot predict the computing cost by the data amount precisely, using data parallelism may lead to a bad workload balance and have a bad effect on the total performance.

Task parallelism is expected to solve this kind of problem. Task parallelism, which is different from data parallelism, treats an independent logical task as a parallelization unit, which allows workers to do totally different kinds of calculations at the same time. To deal with the workload balance problem, task parallelism usually has a work-stealing strategy, which is a dynamic scheduling strategy to spawn and assign tasks among workers. By work-stealing, a worker (thief) will randomly choose another worker (victim) and asks for a task. If the victim does not have a task that can be stolen, the thief will choose another victim until the thief steals a task successfully or there is not task left. Therefore, task parallelism can achieve dynamic load balancing among workers.

As mentioned in Section 2.3.1 and Section 2.3.2, it is difficult to estimate the computational cost of construction of each sub-CT and sub-BCT in advance. Thus, we cannot get a good load balance when we statically assign computational cores to subtrees. Therefore, we parallelized CT and BCT construction using task parallelism, by which we can parallelize such tree recursive algorithms easily and efficiently employing the dynamic load balancing strategy.

Languages that employ task parallelism are called task parallel languages. There are many task parallel languages, with different work-stealing strategies and implementations, such as OpenMP Task [36], Cilk Plus [32, 33], Intel Threading Building Blocks (TBB) [37, 38] and Tascell [40].

In this chapter, we provide a brief introduction to the task parallel languages used for our parallel implementations, Cilk Plus and Tascell, and discribe the difference between them.

```
1 int fib(int n){
2   if(n ≤ 2){
3     return n;
4   }
5   int a = cilk_spawn fib(n-1);
6   int b = fib(n-2);
7   cilk_sync;
8   return a+b;
9 }
```

Figure 3.1: Doubly recursive Fibonacci in Cilk Plus.

3.2 Cilk Plus

3.2.1 Overview

Cilk Plus [32, 33] is a commercial version of Cilk [29] that was 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 *reducers* to help programmers parallelize computations with reduction and `cilkrts_get_nworkers()` to obtain the total number of workers. We provide two sample programs written in Cilk Plus in Figure 3.1 and Figure 3.2. In Figure 3.1, `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 3.2 shows how to find the maximum element in a list using `cilk_for` with the help of reducers.

Although, Intel has stopped supporting Cilk Plus, MIT is again developing Cilk in the form of OpenCilk [35], which basically shares the same implementation methods. It is reasonable to believe that our implementation in Cilk Plus can be ported to OpenCilk with little deterioration in performance.

3.2.2 Work-stealing Strategy

Cilk Plus has the same work-stealing strategy, called Lazy Task Creation (LTC) [44], as Cilk, which keeps all workers busy by creating plenty of logical threads and schedules them internally by task queues. The logical threads are used as tasks that can be allocated to idle workers. When a logical thread recursively spawns offspring logical threads, the adoption of the oldest-first work-stealing strategy is effective in making tasks larger. This strategy becomes a widely used strategy that most task parallel libraries, such as Intel Threading Building Blocks (TBB) [37, 38] and Massive Threads [39].

```
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;
}
```

Figure 3.2: Finding the maximum element in a list in Cilk Plus.

3.3 Tascell

3.3.1 Overview

Tascell [40] is a task parallel language that employs the *backtracking-based work stealing strategy*, which is totally different from Clik Plus. 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. Tascell provides the `do_two` (for parallel execution of the following two statements) and `do_many` (for parallel loops) constructs. Figure 3.3 and Figure 3.4 show examples of Tascell programs[1]. Figure 3.5 shows the manner in which backtracking-based task spawning occurs when a Tascell worker performs the program in Figure 3.3.

3.3.2 Work-Stealing Strategy

A Tascell worker executes its own task sequentially and does not spawn a task until it receives a work-stealing request (task request) from another worker. That is, when the worker reaches a statement for which a task can be spawned (e.g., a parallel loop), it simply remembers the possibility at this point, and then executes the statement *as if choosing a completely sequential execution*. Each worker has its own workspace containing the data required for the search, and the search data are updated at each step.

When a worker (victim) receives a task request from another worker (thief), it backtracks to the oldest point among the parallelizable (task-spawnable) points, that is, the point at which the largest task can be spawned, and then spawns a task *as if changing the choice of execution to parallel from sequential*. The victim then allocates and initializes a new workspace for the task by making a copy of its workspace after backtracking. While the cost for each work steal in Tascell using backtrack is higher than Cilk Plus (which only does a dequeue operation), the parallelization overhead is lower because it does not have to manage task queues.

[1]The actual Tascell language has an S-expression-based syntax [45], but we write programs with a C-like syntax here for readers' convenience.

```
1  // The definition of a task named tfib
2  task tfib{
3    in:    int n;        //input
4    out:   int r;        //output
5  };
6  //The entry point of tfib.
7  //The task object this is declared implicitly
8  task_exec tfib
9  {this.r = fib (this.n);}
10 worker int fib(int n){
11   if(n ≤ 2) return 1;
12   {
13     int s1, s2;
14     do_two              //construct in Tacell
15       s1 = fib(n-1);
16       s2 = fib(n-2);
17     handles tfib {
18       //put part (performed before sending a task)
19       {this.n = n-2;}
20       //get part (performed after receiving the result)
21       {s2 = this.r}
22     }                   //end do_two
23     return s1 + s2;
24   }
25 }
```

Figure 3.3: Doubly recursive Fibonacci in Tascell.

3.3.3 DMS support in Tascell

In Tascell, both the do_two and do_many constructs support work stealing among workers in different computing nodes. To allow a worker to send input and receive an output of a task to/from an external node, programmers can add in: and out: annotations to the member definitions of task objects, as in lines 3–4 of Figure 3.3. To send/receive complex data structures such as arrays, programmers need to define the task_sender and task_receiver methods, which explicitly specify how to transmit data. Lines 1–3 in Figure 3.4 show examples of the definition of the task_sender method. Two types of implementations have been proposed for internode communication in Tascell. One employs a Tascell-server, to which all computing nodes are connected via TCP/IP [46], whereas the other uses the MPI library [47]. We employed the MPI-based implementation for the experiments in this thesis.

Although there are many other task parallel languages that feature dynamic load balancing, few supports work stealing among different computing nodes. This is the main reason we chose Tascell for implementations in DMSs.

```
task_sender find_max{
    send_doubles(list, this.i2 - this.i1 + 1)
}
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;
}
```

Figure 3.4: Finding the maximum element in a list in Tascell.

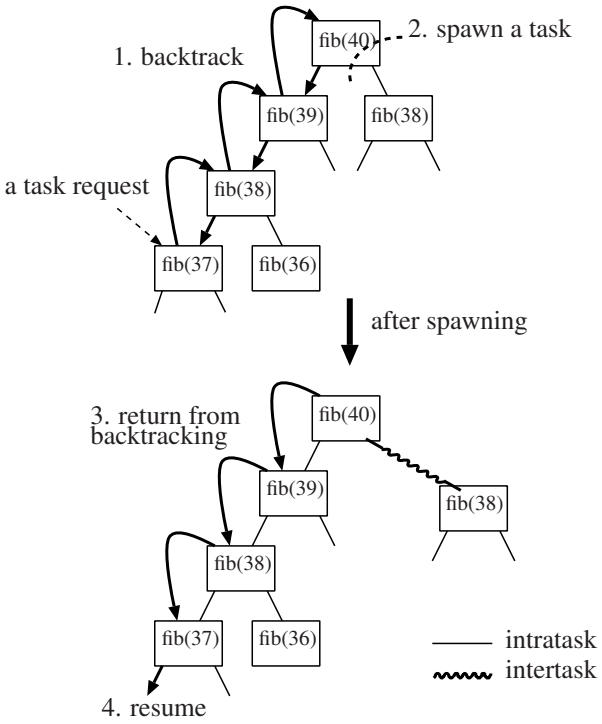


Figure 3.5: Spawning a task lazily while computing fib(40). When a Tascell worker detects a task request (at fib(37)), it (1) backtracks to the oldest task-spawnable point, (2) spawns a task for fib(38), (3) returns from backtracking, and (4) resumes its own computation.

Chapter 4

Parallel Matrix Partitioning in Shared Memory Systems

4.1 Introduction

As mentioned in Section 2.2, unlike the filling operation, matrix partitioning is not parallelized in existing implementations. This is partly because it is difficult to parallelize it using conventional parallel methods and 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 [25, 26, 42]. For example, [25] shows that, when constructing a $\mathcal{H}$-matrix derived from 1,188,000 elements, it takes 0.736s for partitioning and 1479s for filling in sequential computation, and the filling operation is accelerated 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 will be significant for larger datasets. Thus, we should also consider parallelizing the matrix partitioning operation.

In this chapter, we explain our proposed parallel algorithm and implementation of matrix partitioning operation in shared memory systems (SMSs) and evaluate the performance of our implementation.

4.2 Parallel Implementation

4.2.1 Cluster Tree Construction

As explained in Chapter 2.3, a CT is constructed using a recursive algorithm. It is obvious that the two recursive calls in Figure 2.3 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 Figure 2.3) 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.

Figure 4.1 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 Figure 3.2 and 3.4.

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 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 [48], 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 in CT construction. Therefore, we employed the sequential implementation for the prefix-sum computation in the evaluations in Section 4.3.

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.

```
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 | e.x < M};
20       elem[] eright = {e | e.x ≥ M};
21     }
22     if(#e > T_N){
23       spawn {
24         if(#eleft > T_S){
25           // swap L1 and L2
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 L1
33           t.child[0] = buildClusterTreePar(eleft);
34         }
35       }
36       spawn {
37         if(#eright > T_S){
38           // swap L1 and L2
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 L1
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 }
```

Figure 4.1: Pseudocode of the parallel algorithm for CT construction (for simplicity, we show the case in which elements are placed in the 1D space).

```
1  counter-list = {0, ..., 0}; // counter for each worker
2  leaf-node[][] leaf-node-list;
3  void buildBlockClusterTreePar (cluster t, cluster s){
4    // w_id: the ID of the worker executing the task
5    if( combination(t, s) is admissible){
6      leaf-node-list[w_id][counter-list[w_id]] =
7          createBTLeafNode(t, s, "low-rank submatrix");
8      counter-list[w_id]++;
9    }else if(t or s has no child){
10     leaf-node-list[w_id][counter-list[w_id]] =
11         createBTLeafNode(t, s, "full submatrix");
12     counter-list[w_id]++;
13     }else{
14       if(t.nelems > T_N and s.nelems > T_N){
15         for (i=0; i<=1; i++)
16           for (j=0; j<=1; j++)
17             spawn buildBlockClusterTreePar(t.child[i],
18                                            s.child[j]);
19         sync;
20       }else{
21         for (i=0; i<=1; i++)
22           for (j=0; j<=1; j++)
23             buildBlockClusterTree(t.child[i],s.child[j]);
24       }
25     }
26   }
27 }
```

Figure 4.2: Pseudocode of the parallel algorithm for BCT construction.

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 smaller 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.

4.2.2 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 Figure 2.4). Figure 4.2 shows the pseudocode of the parallel implementation of BCT construction.

The only concern is about the space in which leaf nodes of BCT are stored. In the sequential implementation, they are stored in 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 enable 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[1] of 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 the given two clusters are greater than T_{N}. As discussed in Section 2.3.2, we cannot accurately estimate the depth of (a subtree of) BCT using the number of elements because it depends on admissibility conditions, but the upper bound of the depth can be estimated in this way.

4.3 Performance Evaluation

4.3.1 Evaluation Setup

We evaluate our proposed parallel implementations with the following four datasets from which coefficient matrices of the surface element method are generated [49].

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

SphereCube: $10 \times 10 \times 10$ 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.

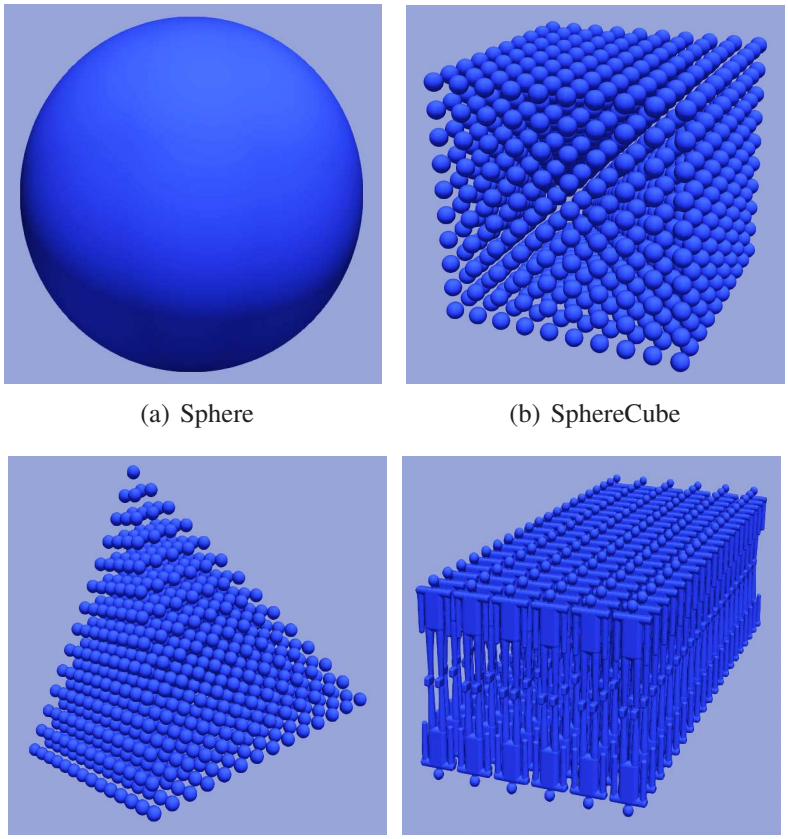
We choose these four datasets as the benchmark to evaluate the performance because these datasets can cover most scenarios. The Sphere is the simplest scenario. The SphereCube and the SpherePyramid are scenarios where the CT and BCT are most likely to be balanced and imbalanced. The Humans is a relatively real scenario that simulates the scene where people stand in an array.

These four datasets are illustrated in Figure 4.3[2] and their characteristics are summarized in Table 4.1. 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 4.2.

[1] 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.

[2] Figure 4.3(d) illustrates only 6×10 pairs to make it easy to recognize.



(a) Sphere (b) SphereCube

(c) SpherePyramid (d) Humans

Figure 4.3: Input datasets used in the evaluations.

Table 4.1: 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

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 4.3.

4.3.2 Cluster Tree Construction

Performance Parameter Tuning

First, we need to tune the three parameters presented in Section 4.2.1 that can significantly affect the performance.

[3] `https://bitbucket.org/tasuku/sc-tascell/commits/158c691ce4059afbbca6c437a6714e8b911756be`

Table 4.2: 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 [3] + GCC version 4.8.5 with -O3 option + Trampoline-based implementation of nested functions in GCC.

Table 4.3: 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

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

$\boldsymbol{T_{\mathrm{S}}}$ 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 4.2.1, 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 $T_{\mathrm{S}} = 10^4$ and $C = 10^4$.
2. Find the optimal value of T_{N} within the range $T_{\mathrm{N}} \in \{10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 5 \times 10^7\}$ while fixing T_{S} and C.
3. Find the optimal value of T_{S} within the range $T_{\mathrm{S}} \in \{10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 5 \times 10^7\}$ while fixing T_{N} and C.
4. Find the optimal value of C within the range $C \in \{1, 2, 4, 8, 16, 32, 64, 128, 10^3, 10^4, 10^5, 10^6, 10^7\}$ while fixing 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 $(T_N, T_S, C) = (10^2, 10^4, 32)$ for Cilk Plus and $(T_N, T_S, C) = (10, 10^4, 10^3)$ for Tascell.

The effects of these parameters on the performance are shown in Figure 4.4. From Figure 4.4(a) we can see that the parameter T_N does not affect the performance for CT construction very much when T_N is not extremely large. Parameter T_S and C, which are parameters controlling the parallelization of calculation inside of each recursive step, intuitively, should not vary much among different datasets. For evidence, we do the same tuning for all four datasets on a single node of supercomputer ITO subsystem A of Kyushu University, whose specification is listed in Table 6.4. The result shows that using parameters tuning for each dataset respectively can only improve the performance by 8% at most in the case of SphereCube and the parameter for SpherePyramid is exactly the same as what we get in Sphere. Therefore, we used these parameter settings for all other performance measurements. We can see from Figure 4.4(b) 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.

Performance Evaluation

The performance of our parallel implementations for CT construction using Cilk Plus and Tascell is shown in Figure 4.5. Table 4.4 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 4.2.1 and memory bandwidth saturation.

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

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 larger 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. When the number of elements of a current CT node falls below $\overline{T_N}$, skip constructing the subtree of the CT whose root is the current node and add a task for this construction into the task list.

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)`. The inside of each recursive step is executed sequentially.

Performance comparison among the implementations using OpenMP and the task parallel languages is shown in Figure 4.6. Figure 4.7 shows the execution time for each task in the task list of the OpenMP implementation. Figure 4.8 shows the cumulative execution time of each worker for constructing the deeper part of CTs in the OpenMP implementation. When $\overline{T_N}$ is large, we cannot get a good load balance because coarse-grained tasks are not parallelized, especially for Sphere, where the variation of the task sizes is relatively large. When $\overline{T_N}$ is small, we can achieve good load balance but the overhead for managing plenty of tasks get large. In both cases, the OpenMP implementation cannot overperform the implementations using task parallel languages.

4.3.3 Block Cluster Tree Construction

Performance Parameter Tuning

As presented in Section 4.2.2, 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 Figure 4.9.

Performance Evaluation

The performance of our parallel implementations of BCT construction using Cilk Plus and Tascell is shown in Figure 4.10. Table 4.5 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 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, 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 that 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.

4.3.4 Total Performance of Matrix Partitioning

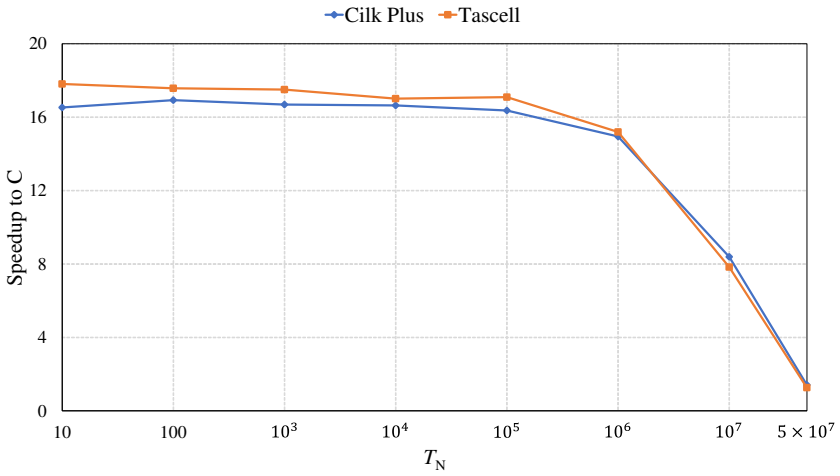
Finally, we demonstrate the total performance of matrix partitioning, i.e., both CT and BCT construction in Figure 4.11. Table 4.6 shows the best total performance achieved by these parallel implementations. As seen in Table 4.3, the elapsed time to construct CT is longer than that for BCT construction. Thus, the total performance of matrix partitioning is mainly impacted by the performance of CT construction.

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 achieved 11.5–14.5 times speedups compared to the sequential implementation for matrix partitioning.

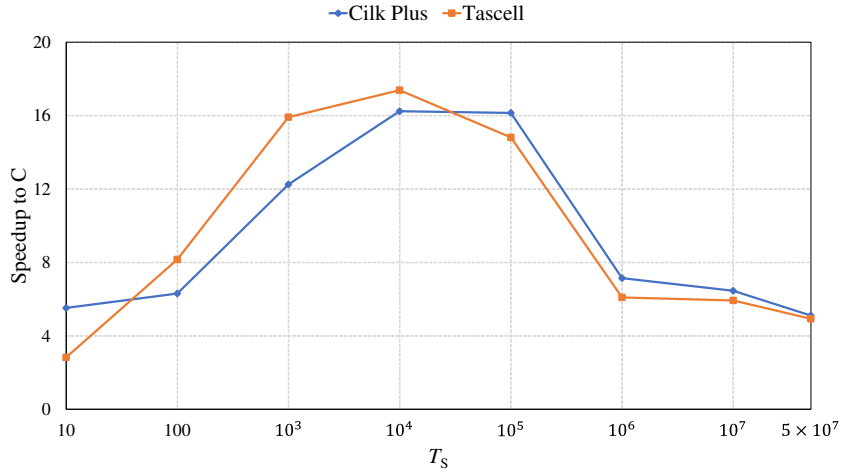
4.4 Conclusion

In this chapter, we proposed a parallel implementation 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.

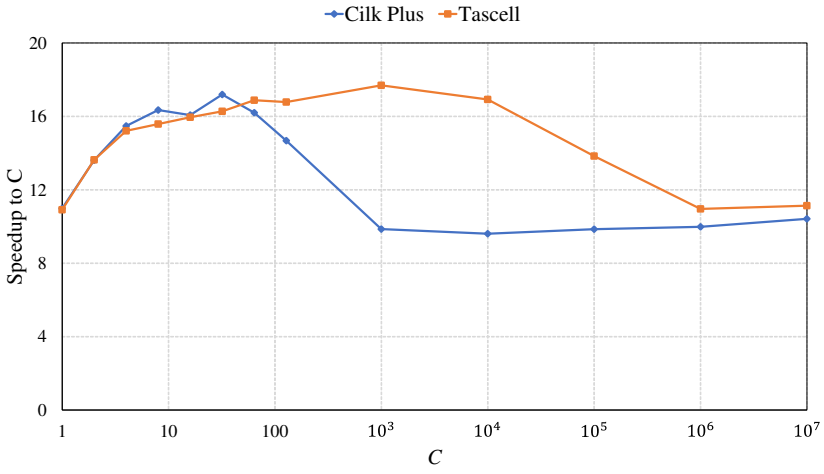
As a result, compared to a sequential implementation in C, we achieved 15.6–16.9 times speedups by Cilk Plus and 17.7–19.1 times speedups by Tascell for the CT construction. For the BCT construction, speedups using Cilk Plus are 18.9–37.7 times, and those using Tascell are 22.7–38.8 times. In regard to the whole process of matrix partitioning, we achieved 15.7–17.7 times speedups by Cilk Plus and 17.8–21.3 times speedups by Tascell.



(a) The effect of T_N. (T_S,C) is (10^4,32) for Cilk Plus and (10^4,1000) for Tascell.

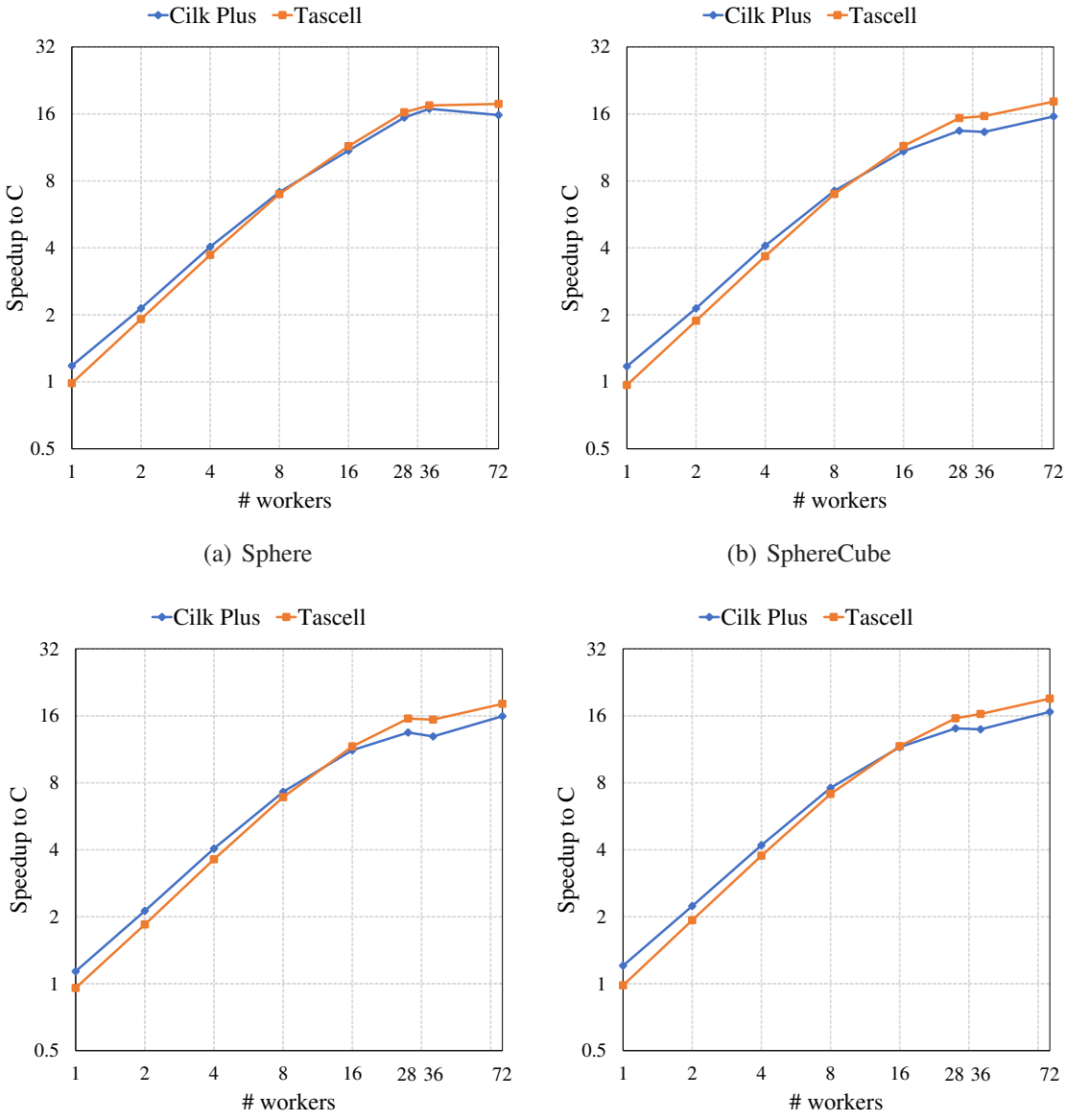


(b) The effect of T_S. (T_N,C) is (100,32) for Cilk Plus and (10,1000) for Tascell.



(c) The effect of C. (T_N,T_S) is (100,10^4) for Cilk Plus and (10,10^4) for Tascell.

Figure 4.4: Effect of the three parameters on the performance of CT construction (36-worker executions, Sphere).



(a) Sphere

(b) SphereCube

(c) SpherePyramid

(d) Humans

Figure 4.5: Performance of Cilk Plus and Tascell in CT construction.

Table 4.4: Best performance (speedup to C) achieved by the Cilk Plus and Tascell implementations of CT construction.

	Sphere	SphereCube	SpherePyramid	Humans
Cilk Plus	16.9 (36 workers)	15.6 (72 workers)	15.9 (72 workers)	16.7 (72 workers)
Tascell	17.7 (72 workers)	18.2 (72 workers)	18.1 (72 workers)	19.1 (72 workers)

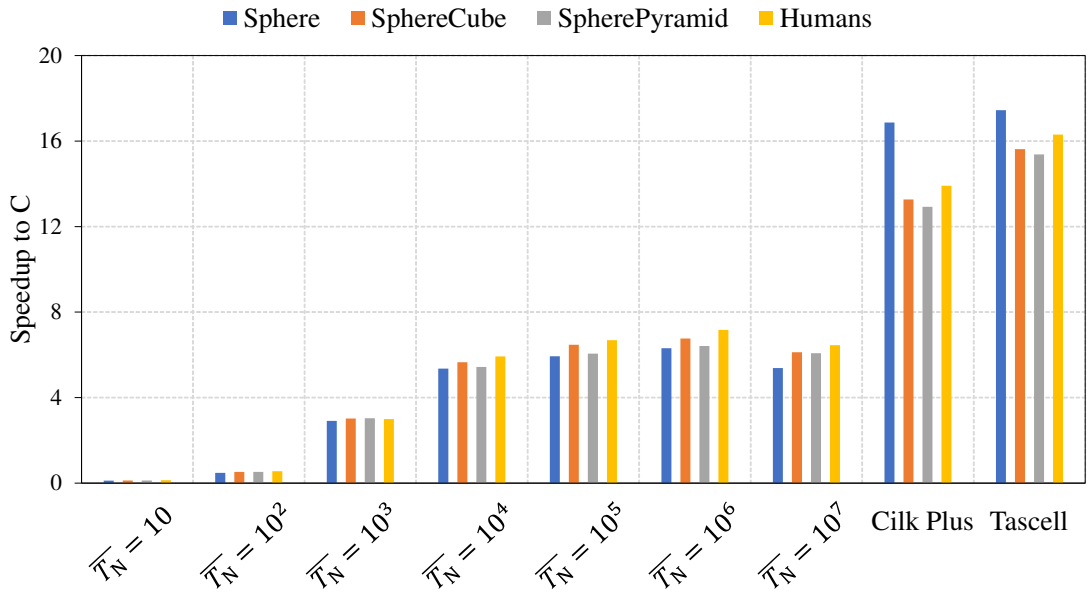


Figure 4.6: The performance for CT construction of the OpenMP, Cilk Plus and Tascell implementations (36-thread/worker executions)

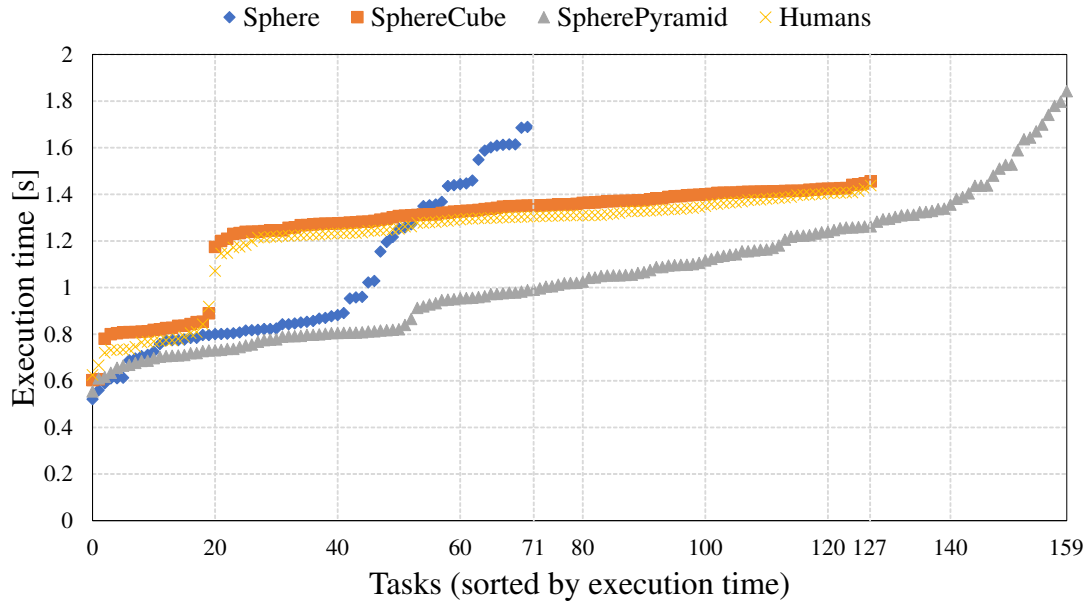
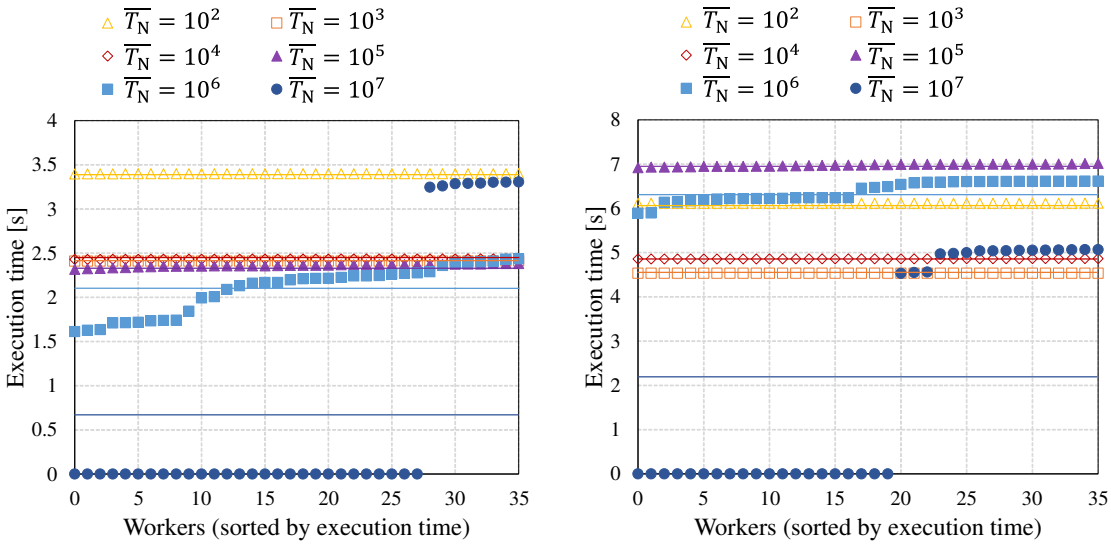


Figure 4.7: Execution time for each iteration of the parallel for loops for constructing the deeper part of CTs in the OpenMP implementation ($\overline{T_N} = 10^6$).



(a) Sphere

(b) SphereCube

(c) SpherePyramid

(d) Humans

Figure 4.8: Cumulative execution time of each worker for constructing the deeper part of CTs in the OpenMP implementation (36-worker executions). The horizontal lines indicate the average execution times among workers.

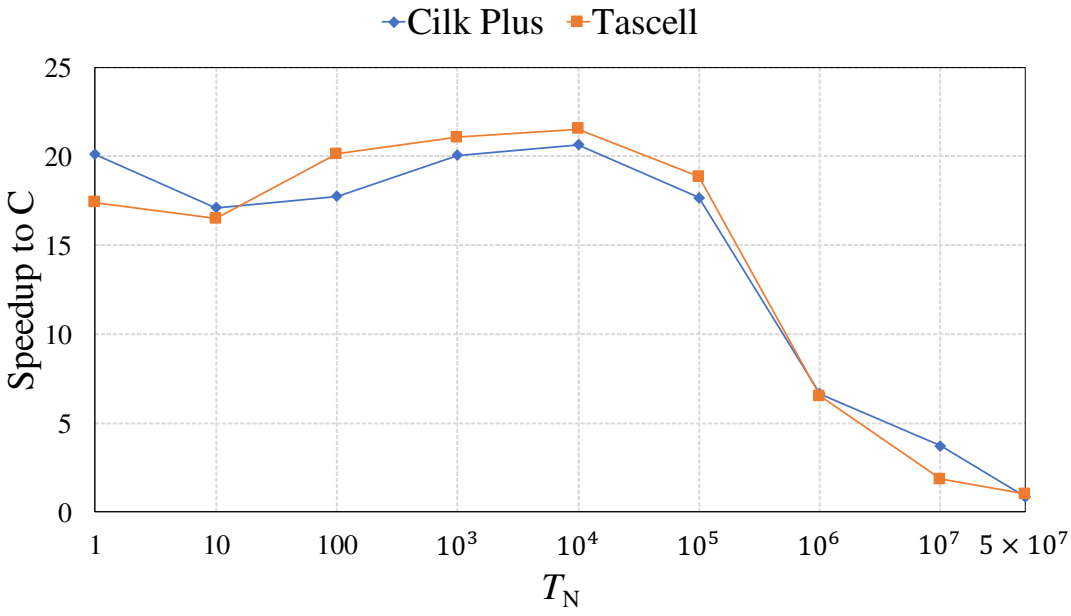
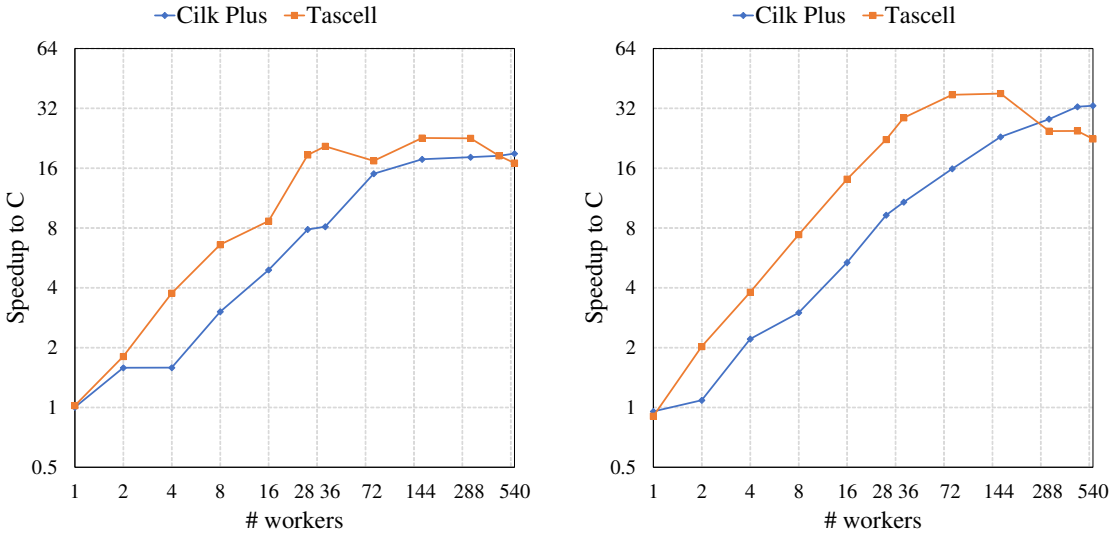
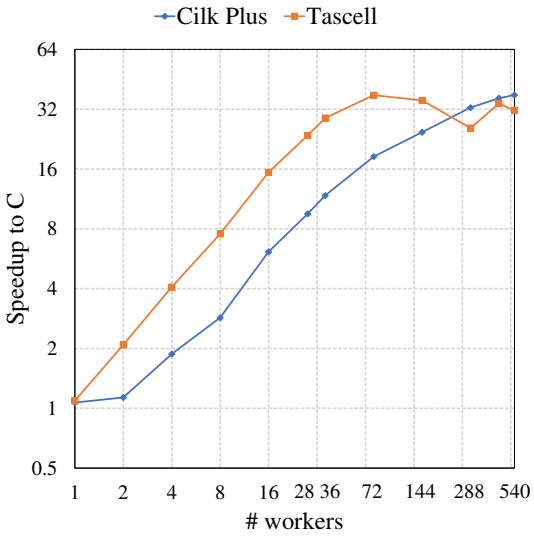


Figure 4.9: The effect of T_N on the performance of BCT construction (288-workers, Sphere).

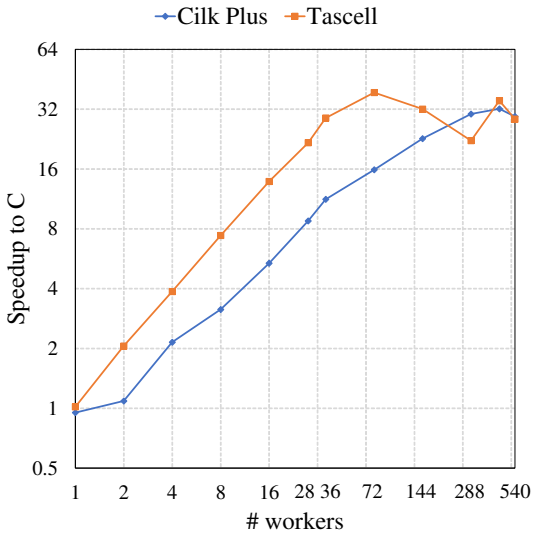


(a) Sphere

(b) SphereCube



(c) SpherePyramid

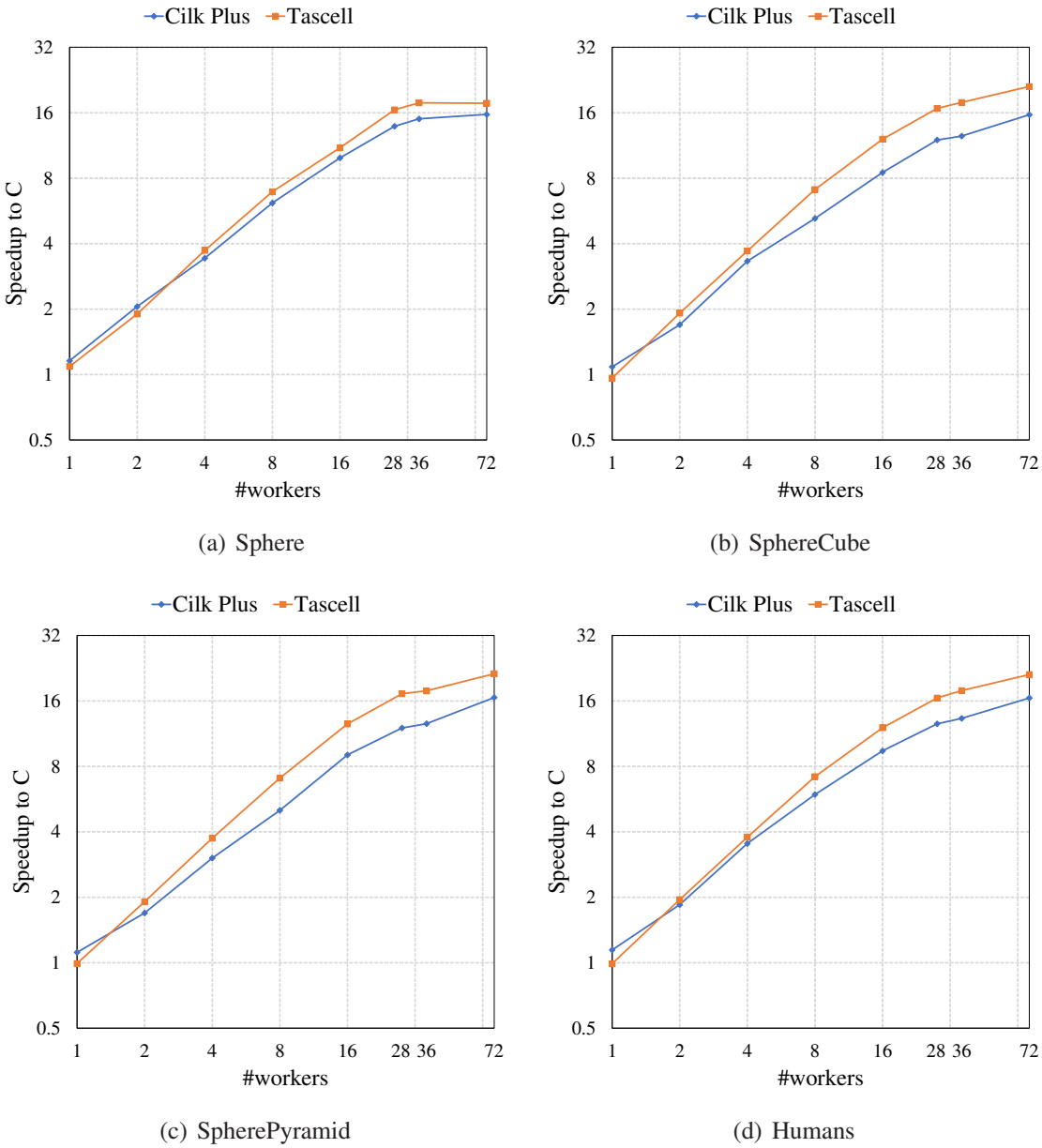


(d) Humans

Figure 4.10: The performance of Cilk Plus and Tascell in BCT construction.

Table 4.5: 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)



(a) Sphere

(b) SphereCube

(c) SpherePyramid

(d) Humans

Figure 4.11: Total performance of the Cilk Plus and Tascell implementations of matrix partitioning.

Table 4.6: Best total performance (speedup to C) achieved by the Cilk Plus and Tascell implementations of matrix partitioning.

	Sphere	SphereCube	SpherePyramid	Humans
Cilk Plus	17.7 (72 workers)	15.7 (72 workers)	16.6 (72 workers)	16.5 (72 workers)
Tascell	17.8 (36 workers)	21.1 (72 workers)	21.3 (72 workers)	21.1 (72 workers)

Chapter 5

Parallel Matrix Partitioning in Distributed Memory Systems

5.1 Introduction

With the exponential growth of data science, requirements for memory capacity and computational power in simulations have exceeded the capabilities of shared memory systems (SMSs). In contrast, distributed memory systems (DMSs), which consist of multiple computing nodes connected by a high-speed communication network, can satisfy these requirements. Therefore, DMSs are widely used in large-scale simulations.

In the last chapter, we proposed a parallel matrix partitioning implementation using two task parallel languages Cilk Plus and Tascell, and achieved a good performance. However, this implementation is only designed for SMSs, which means the upper limit of the data size that the implementation support is restricted by the memory of the computing node. Thus, it cannot deal with datasets with data sizes larger than the memory capability. In addition, other calculations, such as filling operation in $\mathcal{H}$-matrix construction, and $\mathcal{H}$-matrix arithmetics have already been parallelized in DMSs in existing parallel implementations. Therefore, in order to support larger datasets and achieve a better adaption to existing parallel implementations in DMSs, it is worth extending our parallel implementation of matrix partitioning to DMSs. Because Cilk Plus does not support DMSs, we only use Tascell to implement our proposal.

In this chapter, we propose two parallel implementations for matrix partitioning in DMSs:

- Distributed Cluster Tree Construction (DCTC), where CT and BCT are both parallelized by all workers in all computing nodes.

- Redundant Cluster Tree Construction (RCTC), where CT is constructed in every computing node redundantly by employing only intra-node work stealing.

To make these two implementations effective, we enhance Tascell by adding two new features, *synchronous instructions* (`tcell-broadcast`) and *node-aware work-stealing* (`node-guard`) to the language processing system of Tascell.

We describe the enhancement of Tascell first and show the details of the implementations in DMSs later. We evaluate the performance of our implementations at the last of this chapter.

5.2 Enhancement of Tascell

5.2.1 Synchronous Instructions

For task parallel languages, it is natural that a worker spawns sub-tasks to let other workers steal. These tasks usually have a hierarchical relation, such as task and sub-task. However, when we extend our implementation to DMSs, some tasks, such as data loading and MPI collective communication, do not have such a relationship but need to be executed in all computing nodes. Unfortunately, Tascell does not support this innately because when a Tascell program starts, except for worker 0 in rank 0 who has the initial task, the other workers will be busy at sending task requests to ask for tasks and do these tasks. Therefore, to implement synchronous instructions in Tascell, we add a new annotation `tcell-broadcast`, by which worker 0 in rank 0 can broadcast a series of instructions to all computing nodes. The syntax of `tcell-broadcast` is as follows.

`tcell-broadcast` *task_name* {//put part} {$expression_{victim}$}

Programmers using `tcell-broadcast` have to define an empty task and need to add a $expression_{thief}$ in the `task-receiver` function. Because the task we defined here is empty, the instructions really executed are in the *expression* block. Note that the $expression_{victim}$ in `tcell-broadcast` can be different from $expression_{thief}$ in `task-receiver`, which creates a possibility to let victim and thieves do different instructions. Programmers can also send data to other computing nodes by the "put part" in the same way as `do_two` and `do_many` when necessary.

5.2.2 Node-aware Work-stealing

As we know, work-stealing in DMSs may cause data communication. Therefore, we should trade off whether the benefit of parallelization can pay the overhead of data transformation caused by it in advance. In pursuing the best performance in total, we need to encourage some kinds of tasks to be stolen across computing nodes but restrict other kinds of tasks which should only be stolen by workers in the same computing node. We call this work-stealing strategy with consideration of computing node information Node-aware Work-stealing.

To the best of our knowledge, existing Tascell and other task parallel languages do not support node-aware work-stealing mechanisms. We enhanced Tascell such that programmers can write a `node_guard` annotation to `do_two` and `do_many` statements, which specifies whether a part of the loop can be stolen only by workers in internal/external nodes. The syntax of `node_guard` is as follows.

`node_guard` $expression_{\text{flag}}$

Programmers can append this annotation to `do_two` or `do_many` parallel statements, as in line 14 in Figure 5.1. When the value of $expression_{\text{flag}}$ is 0, a worker does not accept any task request for the parallel statement, regardless of the origin point. When the value is 1, the worker rejects task requests from external nodes but accepts requests from internal nodes. When the value is 2, the worker rejects requests from internal nodes but accepts requests from external nodes. When the value is 3, the worker accepts all task requests regardless of origin, which is the same behavior as in the case where the `node_guard` annotation is not provided. Note that when a task request is refused because of a `node_guard` annotation, it still has the chance to be accepted in another `do_two` or `do_many` statement.

```
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
        node_guard 1 // accepts only intra-node work steal requests{
    //put part
    this.i1 = j1;
    this.i2 = j2;
  } {
    //get part
    if(max < this.r) max = this.r;
  }
  return max;
}
```

Figure 5.1: Example of a node_guard annotation in finding maximum from list.

Generally, tasks with light computation cost but large amount of data communication are not suitable for inter-node work-stealing. However, we do not have metrics to determine what is light or what is large, because they are relative concepts and effect by many factors, such as problem size and system specifications. Therefore, it is difficult to find a perfect $expression_{\text{flag}}$ by compiler automatically. Programmers have to choose which $expression_{\text{flag}}$ is appropriate by themselves based on the their knowledge of particular situation.

5.3 Parallel Implementation

5.3.1 Cluster Tree Construction in DCTC

In DCTC, a CT is constructed in parallel using workers in all computing nodes. As explained in Section 4.2.1, we parallelized the CT construction at two levels. First-level parallelization is performed by expressing two parallel recursive calls at each step, whereas second-level parallelization is performed by parallelizing the operations inside each recursion step. Second-level parallelization is not suitable for stealing work between different computing nodes because the communication cost for sending an input array does not consider the computation cost for finding the maximum and minimum elements or the pivoting operation. For example, suppose a task is executed to find the maximum element in an array using two workers in different computing

nodes employing work stealing. The victim worker needs to send half of the array to the thief, which incurs a comparable cost to that of finding the maximum element in the overall sub-array. Thus, the operation does not produce a speedup. Therefore, we should allow a worker to refuse inter-node work steal requests for this type of task, and accept only intra-node requests. We added the `node_guard 1` annotation to `do_many` statements that appear in the `parallel_minmax` and `parallel_pivoting` functions in lines 6 and 9 of Figure 4.1.

We implemented first-level parallelization on DMSs using only the existing features of Tascell. That is, for the `do_two` statement that corresponds to **spawn**s in lines 23 and 36 of Figure 4.1, a worker accepts work steal requests from both internal and external nodes. When the (victim) worker accepts a request from an external node, it sends an array of elements (a sub-cluster) as input of a spawned task to the node, and the (thief) worker that requests the task constructs a (sub-)CT using the received array. Then, the thief sends a local index of its root node as the task output. From this index and the process ID of the thief worker, the victim can determine the location of the sub-CT.

To clarify the effect of the `node_guard` annotation in this application, suppose a worker completed the construction of the CT node ${\mathcal{E}_1}^{(0)}$ and initiated the construction of ${\mathcal{E}_1}^{(1)}$ in Figure 2.2. At this time, the worker can spawn a task to construct a sub-CT whose root is ${\mathcal{E}_2}^{(1)}$ (t_{T}), or a task to help the pivoting operation create ${\mathcal{E}_1}^{(1)}$ (t_{P}). When a worker accepts a task request, it spawns task t_{T}, which is near the root of the task tree. Subsequently, when the worker receives an inter-node task request, it can only spawn t_{P}; however, this kind of task should not be sent to an external node, as explained above. Using the `node_guard 1` annotation, such a task request can be refused (and the worker that requested the task can send a request to another worker). If the task request is an intra-node request, it can be accepted.

As explained in Section 2.3.2, BCT construction uses the results of CT construction. However, we implemented CT construction for DCTC so that a computing node that created a sub-CT owns all resultant CT nodes. There are two possible implementations for sharing CT nodes among computing nodes: transferring CT nodes on demand and exchanging all CT nodes before BCT construction. We employed the second implementation because it is difficult to manage information regarding which computing node has specified CT nodes, and an increase in the number of communications required to exchange this information would degrade performance.

In the existing implementation of SMSs, the result of CT construction is stored as a linked tree, where each CT node object has pointers to its children. Each CT node is allocated using `malloc`. In our implementation for DCTC, we allocated a single array prior to CT construction, and let workers allocate CT nodes from this pre-allocated array, whose mutual exclusion was controlled by compare-and-swap (CAS) operations[1]. We employed this representation so that computing nodes could exchange their CT nodes efficiently after CT construction using MPI functions. The performance of CT construction with this allocation strategy is expected to be higher when the number of workers in each computing node is small; however, it degrades as the number of workers increases owing to the overhead of CAS operations. To reduce the number of CAS operations, and thus the overhead, we let a worker allocate a worker-local chunk of size c CT node objects at each CAS operation.

After CT construction, worker 0 in the rank 0 process sends requests to other processes to exchange CT nodes. Subsequently, worker 0 in each process calls the `MPI_allgather` function

[1] In our current implementation, the size of a pre-allocated array is estimated based on problem size. If the size is insufficient at runtime, the program will stop abnormally. We can enhance our implementation to dynamically resize the space; however, we avoided this for the sake of simplicity and to reduce overhead.

to exchange the number of CT nodes created in the computing node. Workers can then exchange CT nodes by calling the `MPI_allgatherv` function. We can use `tcell-broadcast` to make MPI collective communication function call executed by all computing nodes.

5.3.2 Cluster Tree Construction in RCTC

In RCTC, a CT is constructed recursively in every computing node. In our implementation, we let worker 0 in the rank 0 MPI process send tasks to other processes to construct their own CTs.

For RCTC, we used the existing parallel implementation of CT construction on SMSs, except that we employed a pre-allocated array to store the CT nodes, as in DCTC. In order to construct CT on each computing node redundantly, we banned the inter-node work stealing by adding `node_guard 1` annotation in the `do_two` statements in CT construction, so that the tree construction process in each computing node will not affect each other.

Because a CT is constructed in parallel in each node, its node objects are stored in different orders among the computing nodes, which is inconvenient in the subsequent BCT construction phase. Therefore, after CT construction, each computing node reorders the resulting CT nodes in the pre-order. We parallelized this reordering operation for each computing node. This parallelization can be achieved easily using the `do_two` construct of Tascell when the number of descendants of the left subtree of each CT node is stored during CT construction. We also applied `node_guard 1` in the parallel reorder process to avoid inter-node work stealing.

5.3.3 Block Cluster Tree Construction

Because the CT nodes are stored in the same order in every computing node after the exchange (in DCTC) or reordering (in RCTC), parallelization of BCT construction can be achieved easily even on DMSs owing to the cross-node work-stealing capability of Tascell. The data that must be transmitted at each inter-node work steal are two integers that represent the indices of the corresponding CT nodes. We can implement BCT construction on DMSs by only slightly modifying the program for SMSs.

We used the same implementation of BCT construction in DCTC and RCTC. It should be noted that in RCTC, computing nodes do not need to synchronize before BCT construction. Therefore, we let worker 0 in the rank 0 process initiate BCT construction immediately following CT construction (and reordering) in the process without synchronization. Workers in other processes join BCT construction after performing CT construction in their processes.

The BCT leaf nodes obtained by our implementation were scattered across all computing nodes in random order. As mentioned in Section 2.2, the filling operation is subsequently applied to these leaf nodes, and we can then obtain $\mathcal{H}$-matrix. The leaf nodes should be sorted by their indices on the matrix before $\mathcal{H}$-matrix is used for calculation, so that the calculation can be performed efficiently. Because relocating leaf nodes after the filling operation incurs a significant cost, we should consider relocating them before or during the filling operation in future research.

[2] `https://github.com/simon2/sc-tascell/commit/6c8481feed28c9ff57a266a0c31b1b057c1f6ace`

Table 5.1: Characteristics of input datasets used in the evaluations.

	Sphere	SphereCube	SpherePyramid	Humans
depth of CT, BCT	29	29	30	29
# elements	50,000,000	101,250,000	102,768,750	98,320,000
# nodes in CT	9,221,951	18,969,663	19,432,285	19,486,207
# leaf nodes in BCT	159,866,368	481,577,248	493,697,509	645,121,588

Table 5.2: Evaluation environment.

	CRAY CS400 2820XT (Laurel 2) (up to 8 nodes)
CPU	Intel Xeon Broadwell × 2 sockets (2.1GHz, 18 cores/socket)
Memory	DDR4-2400 128 GB (154 GB/s)
Network	Intel Omni-Path (12 GB/s), fat tree, half-bisection
OS	Red Hat Enterprise Linux Server release 7.9 (Maipo)
Compiler	C: Intel C++ Compiler version 2021.3.0 with -xavx2 -O3 option
	Tascell: Tascell Compiler version of May. 15, 2022 [2]
	+ Intel C++ Compiler version 2021.3.0
	(gcc version 4.8.5 compatibility) with -O3 option
	+ Trampoline-based implementation of nested functions in GCC.
	+ Intel MPI version 2021.3.0

5.4 Performance Evaluation

5.4.1 Evaluation Setup

We evaluated our proposed parallel implementations using the following four datasets from which coefficient matrices of the surface element method are generated [49]. Note that this evaluation does not include the filling operation, but only the matrix partitioning operation.

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

SphereCube: $10 \times 10 \times 10$ 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 Figure 4.3 [3] and their characteristics are summarized in Table 5.1. We set $N_{\min}$ to 15 and η to 2 for all measurements.

We measured performance using up to eight nodes 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 5.2.

[3] Figure 4.3(d) depicts only 6×10 pairs, making it easy to recognize.

Table 5.3: Performance of sequential implementation in C (linked-tree and pre-allocated array implementations, elapsed time in seconds).

		Sphere	SphereCube	SpherePyramid	Humans
linked-tree Impl.	CT	24.9	52.5	54.0	53.6
	BCT	12.9	44.9	46.0	61.9
	Total	37.8	97.4	100	116
pre-allocated array Impl.	CT	17.8	38.2	39.2	39.9
	BCT	11.9	40.7	41.7	56.8
	Total	29.7	78.9	80.9	96.7

Table 5.4: Evaluation of the effect of CAS operations (execution time in seconds).

	Sphere	SphereCube	SpherePyramid	Humans
multiple nodes DCTC Impl.	1.53 (8 nodes)	3.02 (8 nodes)	3.42 (2 nodes)	2.97 (8 nodes)
single node linked-tree Impl.	1.63 (36 workers)	3.58 (36 workers)	3.75 (36 workers)	3.53 (36 workers)

When executing CT construction on a single node, we can choose the linked-tree implementation or the pre-allocated array implementation to store the cluster tree. Therefore, we executed both implementations and compared their performances.

The performance of the sequential implementation using C for CT and BCT constructions is shown in Table 5.3. It can be observed that the performance of the pre-allocated array implementation was approximately 25% higher than that of the linked-tree implementation in CT construction, and approximately 8% higher in BCT construction. Therefore, when evaluating the performance of parallel implementations, we used the performance of the sequential pre-allocated array implementation as the baseline for speedup.

5.4.2 Cluster Tree Construction

Performance Parameter Tuning

As mentioned in Section 4.2.1, three parameters (T_N, T_S, C) affect the total performance. We determined the best parameter combination $(T_N, T_S, C) = (10^4, 15, 10^3)$ using the parameter search algorithm employed in Section 4.2 by 36-worker executions of the linked tree implementation in a single node for the sphere. We extended this parameter combination to all experimental conditions.

Additionally, the chunk size C in Section 5.3 affects the performance of the preallocated array implementation. We set C to 1000 based on a parameter survey of 36-worker executions in a single computing node for the sphere.

Table 5.5: Performance comparison between the best multiple-node executions of DCTC and existing implementations on SMSs (execution time in seconds).

	Sphere	SphereCube	SpherePyramid	Humans
linked-tree Impl. (1 worker)	18.2	38.7	40.1	39.5
pre-allocated array Impl. (1 worker)	16.3	34.6	36.2	35.3
linked-tree Impl. (36 worker)	1.63	3.58	3.75	3.53
pre-allocated array Impl. (36 worker)	1.61	3.29	3.72	3.27

Performance Evaluation

First, we compared the performance of the pre-allocated array implementation with that of the linked-tree implementation using 1-worker and 36-worker executions in a single node. The measurement results are shown in Table 5.4. We can see that the pre-allocated array implementation performed better for all evaluation settings.

We then evaluated the performance using multiple computing nodes. The performance of our parallel implementations for CT construction with DCTC and RCTC is presented in Figure 5.2. Note that because MPI processes do not synchronize after CT construction, as mentioned in Section 5.3.3, the RCTC performance shown in the figures are based on execution times only for the rank 0 process.

Table 5.5 details the best performance with DCTC and its corresponding number of nodes, as well as the performance of a linked-tree implementation that is listed for comparison. We can observe that speedups tend to increase with the number of computing nodes. When compared to C, we obtained at most 13.4-fold speedups (for Humans with eight computing nodes). We can also observe that the performance using eight computing nodes is 6.5%–19% higher than that of the single-node executions of the existing linked-tree implementation. Although, the speedup of using 8 nodes to the pre-allocated array implementation using 1 node is not obvious in Figure 5.2(a), the shortening of execution time of DCTC (despite of CT exchange time) in Figure 5.2(b) did shows that our node-aware work-stealing strategy can achieve some improvement in performance for CT construction.

There are two limiting factors for the performance of multiple-node executions with DCTC. First, from the breakdown of execution times illustrated in Figure 5.2(b), regardless of the number of computing nodes used, the execution time for creating the CT root node cannot be reduced. This is natural because we let workers reject all inter-node work steal requests during the creation of the root CT node ($\mathcal{E}_1{}^{(0)}$ in Figure 2.2), which occupies approximately 25% of the total time in 1-worker executions. The other computing nodes have nothing to do until the root CT node is created. When we check the speedups excluding the root node creation time, as shown in Figure 5.2(c), we can see that up to 24.1-fold speedups (for SphereCube with four nodes) can be obtained compared with C.

Second, the amount of data transferred at each inter-node work steal is relatively large. For example, it required approximately 0.5 s to send an array as input to the construction task of a sub-CT corresponding to $\mathcal{E}_2{}^{(1)}$ in Figure 2.2. We speculate that this is the main reason why we could not obtain significant speedup for the Sphere and SpherePyramid. For SpherePyramid, we could not obtain good speedup even though its problem size was approximately the same as those of SphereCube and Humans. Further analyses are required to determine the detailed causes of this.

Table 5.6: Best performance achieved by Tascell implementations of BCT construction (execution time in seconds).

	Sphere	SphereCube	SpherePyramid	Humans
multiple nodes (DCTC Impl.)	0.123 (8 nodes)	0.366 (8 nodes)	0.347 (8 nodes)	0.368 (8 nodes)
multiple nodes (RCTC Impl.)	0.254 (2 nodes)	0.450 (8 nodes)	0.491 (8 nodes)	0.623 (8 nodes)
single node (linked-tree Impl.)	0.441 (36 workers)	1.20 (36 workers)	1.28 (36 workers)	1.60 (36 workers)

When comparing the performance between DCTC and RCTC, we can see that RCTC's performance is worse when one node is used. A possible reason is that workers in an MPI process that completes its CT construction earlier send task requests to external computing nodes to obtain tasks for BCT construction, which can interfere with CT construction in target nodes.

We can see from Figure 5.2(b) that the cost of the reordering operation in RCTC is much lower than that of the CT node exchange in DCTC. Considering the reordering and exchange costs, the performance of RCTC is better than that of DCTC with multiple computing nodes.

5.4.3 Block Cluster Tree Construction

Performance Parameter Tuning

As presented in Section 4.2.2, BCT construction has only one execution parameter, T_{N}, which is the threshold that determines whether recursive function calls are executed in parallel. We found the optimal parameter setting to be $T_{\mathrm{N}} = 10^4$ for Sphere by 36-worker executions on a single node, and used this setting for all experimental conditions.

Performance Evaluation

Figure 5.3 shows the performance of BCT construction in both DCTC and RCTC in terms of speedup to C and execution times. Table 5.6 shows the best performance for each dataset in the DCTC, RCTC, and linked-tree implementations. Note that in RCTC, even when a worker in the rank 0 process has started BCT construction, workers in other processes cannot join the construction until CT construction in their processes is complete.

Compared to the speedups of CT construction, those of BCT construction are expected to be more significant because BCT construction does not require heavy calculations inside the recursion steps, and the cost for inter-node work steals is considerably smaller. Indeed, we obtained up to 154.4-fold speedups (for Humans) using 288 workers (8 nodes $\times$ 36 workers) with DCTC compared with C.

The performance of RCTC shown in the corresponding figures and table appears to be worse than that of DCTC. This is not necessarily a negative result because, as mentioned earlier, workers in MPI processes other than rank 0 join BCT construction only after completing CT construction. This is actually an advantage in the sense that the rank 0 process can start BCT construction without waiting for the completion of CT construction in other processes. In the case of Sphere

Table 5.7: Best total performance of matrix partitioning achieved using Tascell (execution time in seconds).

	Sphere	SphereCube	SpherePyramid	Humans
multiple nodes (DCTC Impl.)	1.95 (1 node)	4.38 (1 node)	4.83 (1 node)	4.72 (1 node)
multiple nodes (RCTC Impl.)	1.87 (2 nodes)	4.11 (8 nodes)	4.44 (8 nodes)	4.27 (4 nodes)
single node (linked-tree Impl.)	2.07 (36 workers)	4.77 (36 workers)	5.03 (36 workers)	5.13 (36 workers)

using 8 computing nodes, the performance of DCTC is more than 2 times of RCTC. As we mentioned above, this is caused by the load imbalance of BCT construction as workers in rank 0 always starts earlier than workers in other computing nodes. This imbalance can be significant, for an extreme instance, in our experiments we once found a computing node did not create any BCT leaf-node until the BCT construction is completed, because the difference of execution time of CT construction between that node and the others is longer than the execution time of BCT construction using 8 computing nodes.

5.4.4 Total Performance of Matrix Partitioning

Finally, an evaluation of the overall performance of matrix partitioning, that is, both CT and BCT constructions, is presented in Figure 5.4. Table 5.7 lists the best overall performance achieved by these parallel implementations.

The results show that we achieved 15.3–20.5-fold speedups with DCTC, and 15.8–22.6-fold speedups with RCTC, compared to the sequential implementation. The RCTC implementation achieved 1.11-1.20-fold speedups compared to the existing implementation on SMSs, using up to eight computing nodes, each of which has 36 cores. The use of multiple nodes in DCTC, however, does not contribute to the total performance of matrix partitioning. We can determine the reason for the poor performance from Figure 5.4(b), which makes it apparent that the time required to exchange CT nodes limits the speedup using multiple computing nodes in DCTC. Because the cost of reordering in RCTC is much lower, it can achieve better performance than DCTC when using multiple computing nodes.

5.5 Conclusion

In this chapter, we propose an implementation of matrix partitioning in the construction of an $\mathcal{H}$-matrix using Tascell on DMSs.

We propose two implementation methods: distributed cluster tree construction (DCTC) and redundant cluster tree construction (RCTC). In DCTC, both CT and BCT constructions are parallelized using workers in all computing nodes. In RCTC, CT is constructed in every computing node redundantly by employing only intra-node work stealing. The BCT was then constructed in parallel using workers in all computing nodes. RCTC cannot achieve any speedups using multiple computing nodes, but can eliminate the data exchange cost that arises in DCTC.

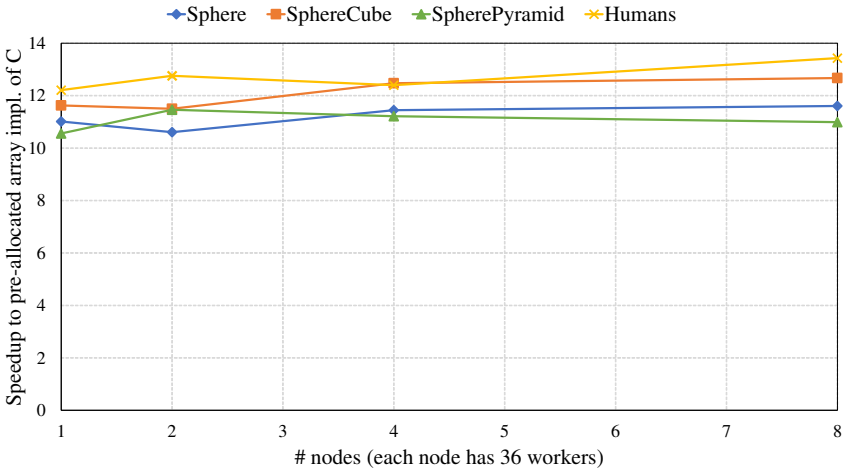
To achieve better performance for CT construction in DCTC, we enhanced the Tascell language to enable annotations for parallel loop statements that specify whether part of the loop can be stolen by workers only in internal/external nodes.

Our parallel implementation of the BCT construction is relatively simple compared to that of the CT construction, except that CT nodes need to be exchanged among all computing nodes (in DCTC) or reordered (in RCTC) before BCT construction.

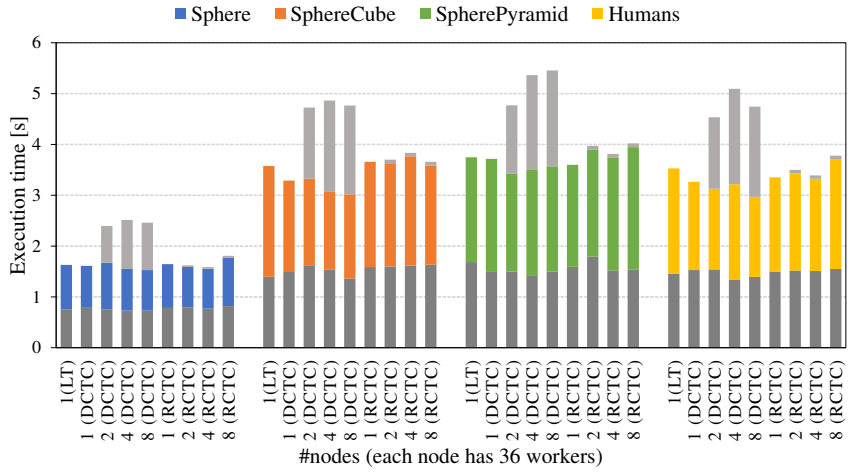
When we evaluated the performance of CT construction, our DCTC implementation achieved speedups using multiple computing nodes compared with the existing implementation on SMSs. We also confirmed that BCT construction yields good speedups using multiple nodes owing to the dynamic load-balancing mechanism of Tascell.

With regard to the entire process of matrix partitioning, our RCTC implementation achieved 1.11–1.20-fold speedups using up to eight nodes, each of which has 36 cores, compared with the single-node performance of the existing implementation of SMSs. Our DCTC implementation could not achieve speedups using multiple computing nodes. However, by evaluating the DCTC implementation, we confirmed that the node-aware work-stealing mechanism is useful for the parallel implementation of CT construction on DMSs. This mechanism should also be useful for other applications, such as k-D tree applications.

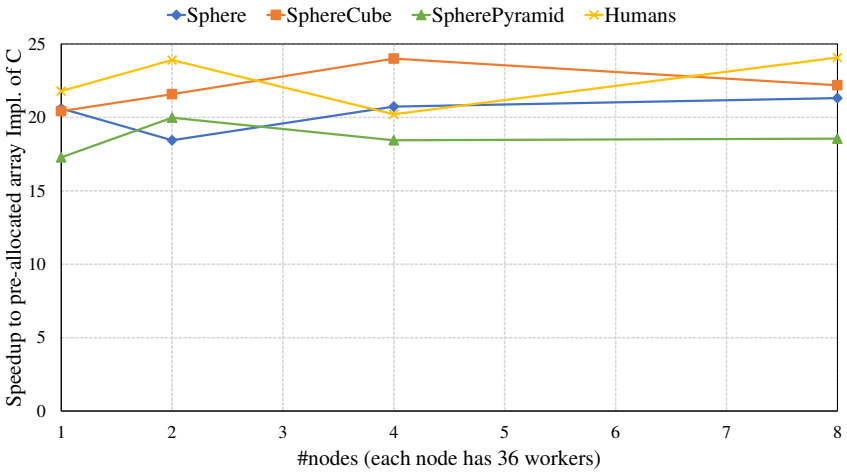
Finally, our implementations on DMSs also have the advantage of handling large input data such that the number of resultant BCT nodes is too large to be stored in a single node.



(a) Speedup of CT construction using multiple computing nodes in DCTC.

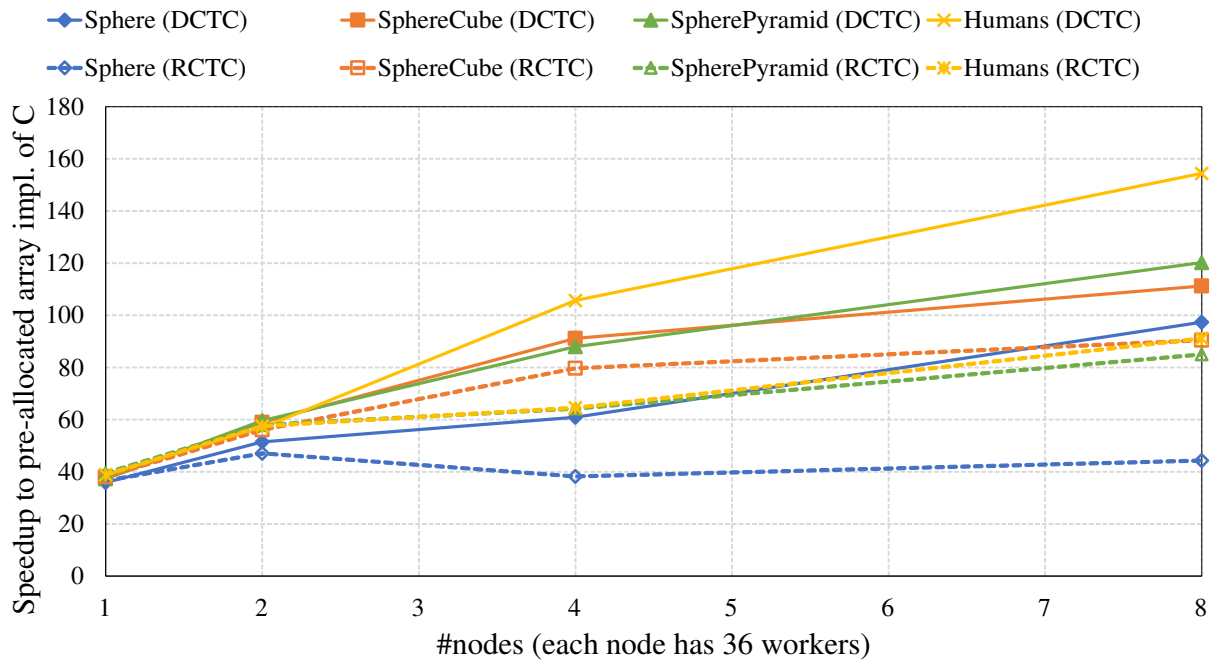


(b) Breakdown of CT construction. Dark gray indicates the execution time of the first tree-level calculation, and light gray indicates the time for the exchange of CT nodes (for DCTC) or reordering (for RCTC). 1(LT) indicates single-node executions of the linked-tree implementation.

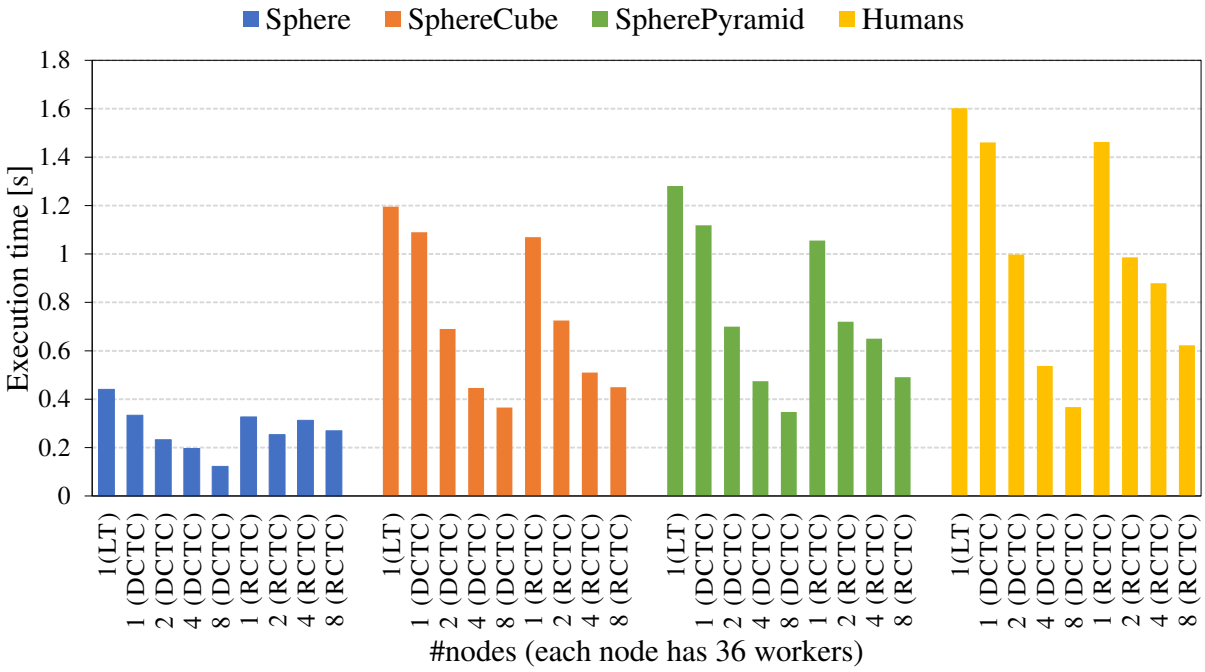


(c) Speedup of CT construction excluding the root CT node creation.

Figure 5.2: Performance of Tascell in CT construction (DCTC).

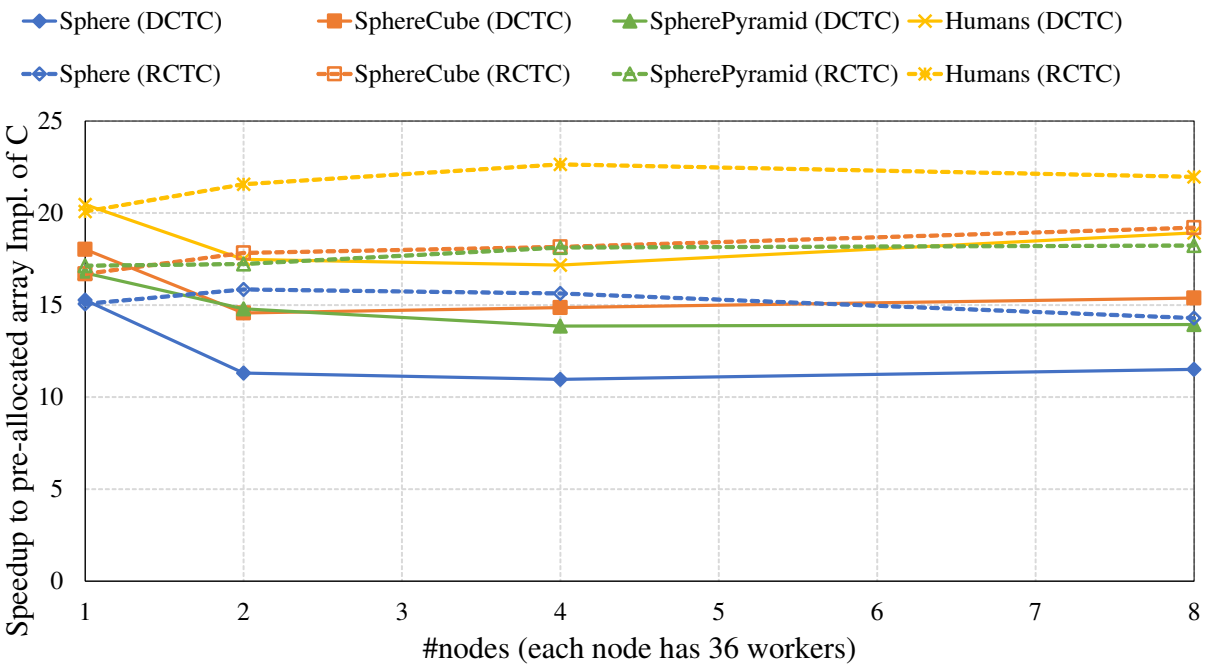


(a) Speedup of BCT construction

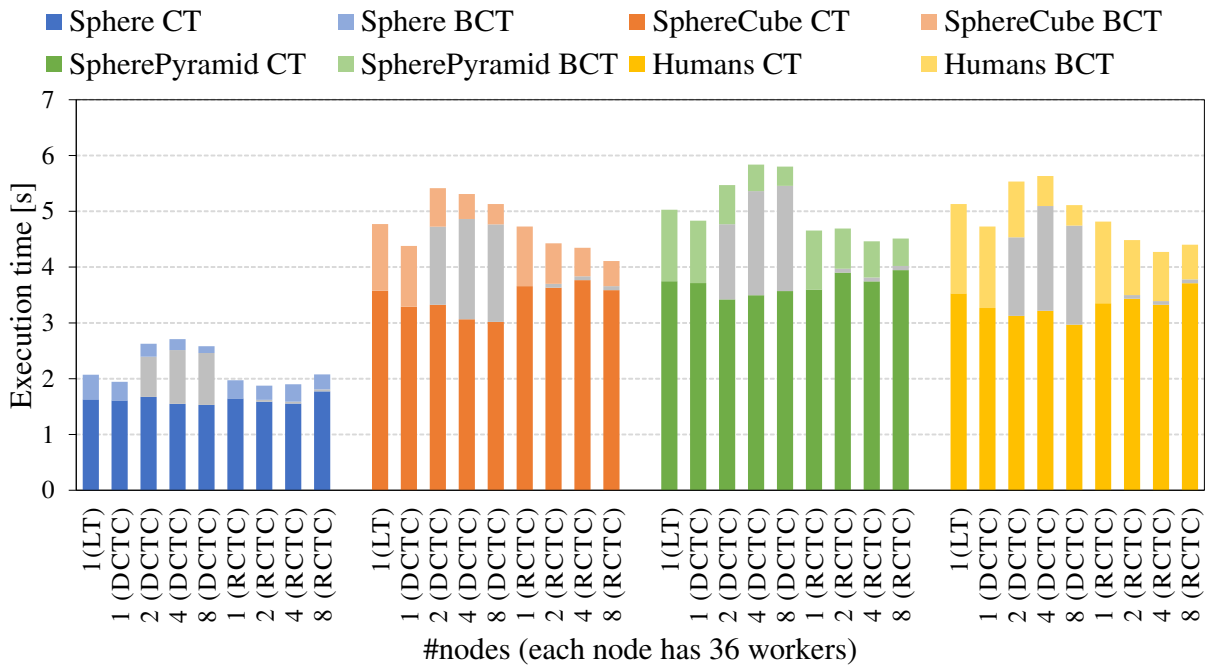


(b) Execution time of BCT construction. 1(LT) indicates single-node executions of the linked-tree implementation.

Figure 5.3: Performance of Tascell in BCT construction.



(a) Speedup of matrix partitioning



(b) Breakdown of matrix partitioning. The time for exchanging CT nodes (in DCTC) and reordering (in RCTC) is indicated in gray. 1(LT) indicates single-node executions of the linked-tree implementation.

Figure 5.4: Performance of Tascell in matrix partitioning.

Chapter 6

Parallel Hierarchical Matrices Construction using Task Parallel Language

6.1 Introduction

In existing parallel implementations of the $\mathcal{H}$-matrix construction, partitioning and filling operations are executed and parallelized independently. In previous works mentioned in Section 4.2 and Section 5.3, we parallelized matrix partitioning operation in SMSs and DMSs. However, the leaf matrices created by our proposal above are scattered across all computing nodes in random order, which is not convenient for the consequent parallel filling operation which requires BCT results laid in order if we use the existing parallel filling approaches. In addition, the filling operation in existing parallel filling implementations represented by $\mathcal{H}$ACApK [23, 24, 28], which is one of the well-known $\mathcal{H}$-matrix libraries, filling operation is executed in parallel by assigning a set of tasks to each worker statically by estimating the workloads based on empirical rules, which is apparently not precise.

To solve the problems above we propose our parallel $\mathcal{H}$-matrix construction using task parallel language Tascell, where we combined BCT construction and Filling operation by treating each filling operation of submatrices as a parallelization unit.

In this chapter, we first explain the implementation of $\mathcal{H}$ACApK and then introduce our proposal using task parallel language Tascell.

6.2 Existing Parallel Implementation: $\mathcal{H}$ACApK

In this section, we introduce the parallelization strategy for partitioning employed in the $\mathcal{H}$ACApK library [28].

The filling operation is parallelized on DMSs using hybrid MPI/OpenMP programming [23]. It has been improved in [24] to avoid the trouble that ranks of approximated matrices increases rapidly as the matrix size increases when the conventional $\mathcal{H}$-matrices with ACA are employed to an integral equation whose kernel function has high-order singularities. However, both methods basically share the same parallelization strategy for $\mathcal{H}$-matrix construction. These implementations are integrated into the $\mathcal{H}$ACApK library.

6.2.1 Matrix Partitioning

$\mathcal{H}$ACApK is implemented using hybrid MPI/OpenMP programming. Partitioning (CT and BCT construction) is executed on every MPI process redundantly. In each process, partitioning is done sequentially based on the algorithm explained in Section 2.2.

After the completion of the partitioning operation, all processes have an identical list of BCT leaf nodes, that is, leaf matrices.

6.2.2 Parallelization of Filling

$\mathcal{H}$ACApK first reorders the result of BCT by QuickSort and then parallelizes the filling operation based on the following static task assignment strategy.

step 1: Estimate the computing cost of filling for each leaf matrix (on every computing node).

step 2: Assign exclusive sets of leaf matrices to OpenMP threads of all MPI processes so that all threads approximately have the same load based on the estimation in step 1.

step 3: Each thread executes filling for leaf matrices assigned to it (calculates all entries for full matrices and row- and column-vectors using ACA+ for low-rank matrices, as explained in Section 2.4).

In step 1, the computing cost of the filling operation is considered to be approximately proportional to the number of the entries $N_P = \sum_{p \in P} N_p$, where N_p is the number of entries in $\tilde{A}|^p$, $p \in P$. When $\tilde{A}|^p$ can be approximated (satisfies the admissibility condition), N_p can be calculated by:

$$N_p = r_p \cdot (\#S_p + \#T_p) \tag{6.1}$$

Otherwise, that is, when p is a full leaf matrix, $\tilde{A}|^p = A|^p$ and N_p can be calculated by:

$$N_p = \#S_p \cdot \#T_p \tag{6.2}$$

In the equation (6.1), N_p cannot be calculated accurately until the filling operation is completed because the rank of the two low-rank matrices r_p is unknown. Therefore, instead of the real rank r_p, a prediction value r_{est} is used here to predict the number of entries of $\tilde{A}|^p$. Thus, as an estimation of N_p, N_p^{est} can be calculated as follows:

$$N_p^{\text{est}} = r_{\text{est}} \cdot (\#S_p + \#T_p) \tag{6.3}$$

The total estimated cost of the filling operation is $N_P^{\text{est}} = \sum_{p \in P} N_p^{\text{est}}$, where the ranks of all leaf matrices are estimated as the single value of r_{est}. The value of r_{est} is set based on empirical rules.

After getting the result of step 1, the rest steps should be easy to implement. Therefore, we will not go into details about the rest steps.

6.2.3 Problems in Existing Parallel Implementations

In the static task assignment strategy introduced above, we cannot get a good load balance among threads because of the following reasons:

- In practice, the ranks of leaf matrices are usually different from each other and the difference between the estimated and actual number of entries $|N_p^{\mathrm{est}} - N_p|$ is considerably large. It leads to inaccuracy in the estimation of the computing cost of filling for leaf matrices and load imbalance. This problem becomes serious especially when the actual ranks of low-rank matrices are large on average. Although, the largest ratio of largest number of entries and average number of entries is about 1.1 when using the four datasets in this experiment, we have found larger ratio, about 1.4, in other case.

- This strategy assumes that the computing cost of the filling operation is approximately proportional to the number of entries of leaf matrices. However, the average computing cost of calculating an entry in low-rank leaf matrices using ACA+ ($\simeq 3.78 \times 10^{-7}$ [s]) is larger than that in full leaf matrices ($\simeq 3.30 \times 10^{-7}$ [s]) due to the pivoting strategy of ACA+. This difference is not significant when the ranks of low-rank leaf matrices are small but becomes considerable when the ranks are large. Therefore, the static work assignment strategy that aims to assign leaf matrices to threads so that the number of entries to be calculated per thread becomes the same is not necessarily a good solution.

6.3 Proposed Implementation

To solve the load imbalance problem pointed out in Section 6.2.3, we propose a new parallel implementation of $\mathcal{H}$-matrix construction, where BCT construction is executed in parallel using all workers of all computing nodes. When a worker finds a BCT leaf during BCT construction, it immediately executes filling for the corresponding leaf matrix. To get good load balance, we parallelized BCT construction using a dynamic load balancing mechanism offered by the Tascell task parallel language [40]. Tascell enables easy and efficient parallelization of the tree recursive algorithms.

In this section, we describe the details of our implementation.

6.3.1 Cluster Tree Construction

Although parallel implementations of CT construction are proposed in Section 4.2 and Section 5.3, we did not employ them in this study because the datasets used in the evaluations are much smaller than those in Section 4.2 and Section 5.3 and CT construction time is negligible compared to the time for filling (as shown in Table 6.1 in Section 6.4). As in $\mathcal{H}$ACApK, we also executed CT construction on every computing node redundantly. In each node, a single worker executes CT construction sequentially.

6.3.2 Parallel Block Cluster Tree Construction and Filling

Parallel implementations of BCT construction are also proposed in Section 4.2 and Section 5.3, which uses Tascell. In particular, Section 5.3 proposes parallel implementation on DMSs. The parallelization of BCT construction is relatively simple: we can obtain sufficient speedup only by executing recursive calls (Line 15 in Fig.2.4) in parallel.

Parallelization of BCT construction can be achieved easily even on DMSs owing to the cross-node work-stealing capability of Tascell. The data that must be transmitted at each inter-node

```
1  cnt_list = {0, ..., 0}; // counter for each worker
2  leaf_node[][] leaf_node_list;
3  void buildBlockClusterTreePar (cluster t, cluster s){
4    // w_id: the ID of the worker executing the task
5    if( combination(t, s) is admissible){
6      leaf_node_list[w_id][cnt_list[w_id]] =
7          createBCTLeafNode(t, s, "low-rank leaf matrix");
8      filling(&leaf_node_list[w_id][cnt_list[w_id]]);
9      cnt_list[w_id]++;
10   }else if(t or s has no child){
11     leaf_node_list[w_id][cnt_list[w_id]] =
12         createBCTLeafNode(t, s, "full leaf matrix");
13     filling(&leaf_node_list[w_id][cnt_list[w_id]]);
14     cnt_list[w_id]++;
15   }else{
16     if(t.nelems > T_N and s.nelems > T_N){
17       for (i=0; i<=1; i++)
18         for (j=0; j<=1; j++)
19           spawn buildBlockClusterTreePar(t.child[i],
20                                          s.child[j]);
21       sync;
22     }else{
23       for (i=0; i<=1; i++)
24         for (j=0; j<=1; j++)
25           buildBlockClusterTree(t.child[i],s.child[j]);
26     }
27   }
28 }
```

Figure 6.1: Pseudocode of the parallel algorithm of our proposal.

work steal are two integers that represent the indices of the corresponding CT nodes. We can implement BCT construction on DMSs after some slight modifications.

Implementation of filling is not discussed in Section 4.2 and Section 5.3 because the filling operation can be done after BCT construction using the existing implementations. Our proposal is to enhance their parallel BCT construction so that workers execute filling during the construction. More specifically, we modified the implementation of BCT construction so that, when a worker finds a BCT leaf node, it immediately executes filling for the corresponding leaf matrix. In addition, we parallelized the modified BCT construction using the dynamic load balancing mechanism offered by Tascell as done in Section 5.3.

Figure 6.1 shows the pseudocode of our implementation. The `filling` function in the figure executes filling for a given leaf matrix. An element of `leaf_node_list[][]` contains the size of a leaf matrix and its position in the whole $\mathcal{H}$-matrix, which are set in the `createBCTLeafNode` function. Note that `filling` is called immediately after `createBCTLeafNode` in line 8 (for low-rank leaf matrix) and line 13 (for full leaf matrix). The **spawn** construct in line 19 is used for

Table 6.1: Characteristics of input datasets used in the evaluations.

	Sphere	SphereCube	SpherePyramid	Humans
depth of CT, BCT	14	16	16	18
# elements	101,250	374,400	312,000	1,080,000
# nodes in CT	6,183	21,455	17,879	59,423
# leaf nodes in BCT	94,300	295,792	246,640	784,348
# entries	3.99×10^8	1.12×10^9	9.33×10^8	5.79×10^9
Execution time of C Impl. [s]				
CT construction	0.00883	0.0318	0.0272	0.124
BCT construction w/o filling	0.00364	0.0111	0.00964	0.0464
BCT construction + filling	151	412	342	—
BCT construction w/ filling	148	407	341	—

executing the subsequent function call (for constructing a descendant subtree) in parallel, which can be written using the `do_many` construct in Tascell.

We employed the performance parameter T_N to control the granularity of the parallel tasks. A worker calls the sequential version of the recursive function when the number of elements of two given clusters is less than T_N.

This implementation is expected to solve the load imbalance problem mentioned in Section 6.2.3 because leaf matrices are dynamically assigned to workers.

Note that computing cost for BCT construction without filling is negligible at least for the datasets used in the evaluations in this paper, as shown in Table 6.1 in Section 6.4. However, as discussed in Section 4.2 and Section 5.3, it is expected that we can obtain considerable performance improvement by parallelizing BCT construction for larger datasets and with larger degrees of parallelization.

6.4 Performance Evaluation

In this section, we evaluate the performance of the proposed implementation by comparing it to an implementation using the static task assignment strategy introduced in Section 6.2.2. We executed the program five times for each measurement setting and present the result whose execution time is the median.

6.4.1 Evaluation Setup

We used the following four datasets from which coefficient matrices of the surface element method are generated [49].

Sphere: a sphere with 101,250 elements on its surface.

SphereCube: $3 \times 3 \times 4$ spheres placed cubically. Each spherical surface is composed of 10,400 elements.

SpherePyramid: $1^2 + 2^2 + 3^2 + 4^2 = 30$ spheres placed pyramidally. Each spherical surface is composed of 10,400 elements.

Table 6.2: Evaluation environment.

	Fujitsu PRIMERGY CX2570 M4 (ITO subsystem A) (up to 16 nodes)
CPU	Intel Xeon Gold 6154 (Skylake-SP) × 2 sockets (3.0 GHz, 18 cores/socket)
Memory	DDR4 192 GB (255.9 GB/s)
Network	InfiniBand EDR 4x (100Gbps)
OS	Red Hat Enterprise Linux Server release 7.3 (Maipo)
Compiler	C: Intel C++ Compiler version 2021.3.0 with -xavx2 -O3 -mkl=sequential options Tascell: Tascell Compiler version of Sep. 2, 2022 + Intel C++ Compiler version 2021.3.0 (gcc version 4.8.5 compatibility) with -xavx2 -O3 -mkl=sequential options + Trampoline-based implementation of nested functions in GCC. + Intel MPI version 2021.3.0

Humans: 5×10 pairs of human-shaped objects. The surface of each object pair is composed of 21,600 elements.

The characteristics of these four datasets are summarized in Table 6.1. In this table, we show the execution times of sequential programs implemented in the C language. "BCT construction w/o filling" indicates the execution time of sequential BCT construction. "BCT construction + filling" indicates the total execution time of the BCT construction and filling. "BCT construction w/ filling" indicates the execution time of modified BCT construction where a worker executes filling for a leaf matrix immediately when the worker finds the corresponding BCT leaf node. Execution times involving filling are not shown for Humans because the memory usage exceeded the memory limitation of a single computing node.

We set $N_{\min}$ and η in the inequalities (2.1) respectively to 50 and 0.5, and ε in the inequality (2.11) to 10^{-8} for all measurements. We use 11 as the r_{est} in the equation (6.3) based on our empirical rules. The kernel function of Equation (2.2) we use in this experiment is $g(x, y) = |x - y|^{-1}/4\pi\epsilon$. For performance parameter T_{N} (in line 16 of Fig. 6.1), we get the best performance when T_{N} is 10 in the experiments using Sphere. We used this value for all the datasets.

We measured performance using up to 16 nodes of ITO subsystem A, a supercomputer at the Research Institute for Information Technology, Kyushu University. The details of the evaluation environment are summarized in Table 6.2.

6.4.2 Evaluation in Shared Memory System

We evaluate the performance of our proposed implementation by comparing it with a parallel implementation using the static work assignment strategy introduced in Section 6.2. Although the original $\mathcal{H}$ACApK is implemented in Fortran (using hybrid MPI/OpenMP programming), we ported $\mathcal{H}$ACApK to C and used it for the comparison to minimize the difference in performance arose from the difference in programming languages. We call the ported implementation the MPI/OMP implementation. We do not evaluate the performance of Humans in this section because the memory usage exceeded the memory limitation of a single computing node.

The execution times on a single computing node are illustrated in Fig. 6.2. We can see that, in both implementations, we can get speedups by increasing the number of workers until 16 workers

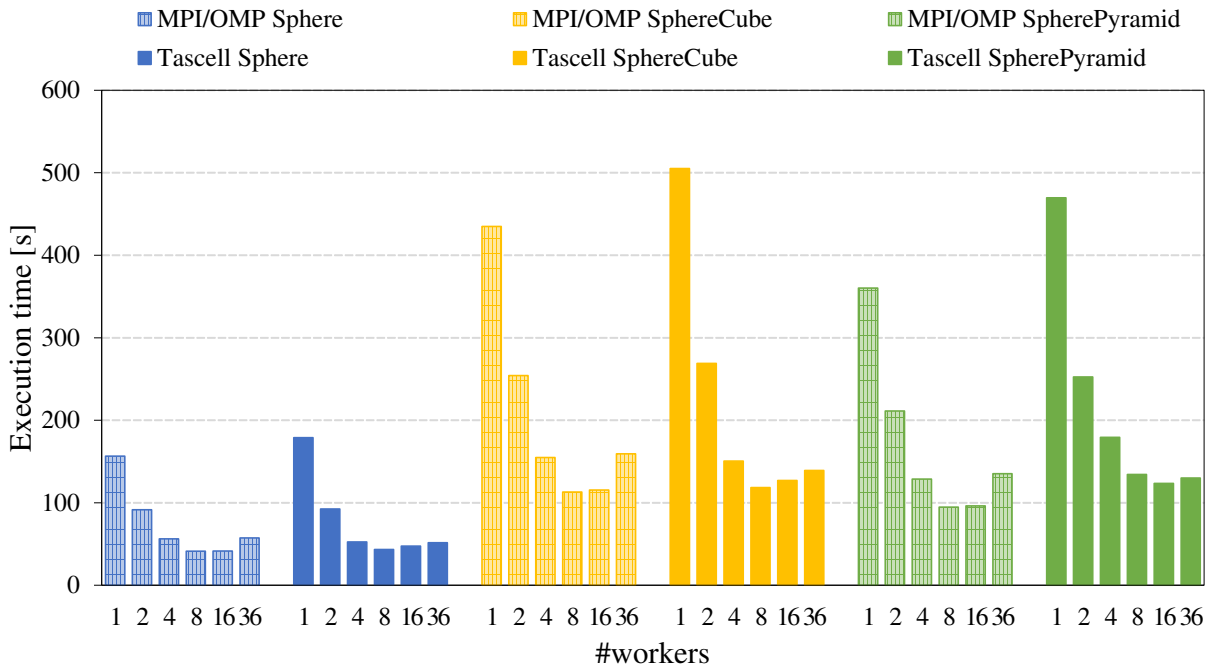


Figure 6.2: Total execution time of BCT construction and filling on a single computing node.

Table 6.3: Best execution time of BCT construction and filling and speedup of Tascell to MPI/OMP.

	Sphere	SphereCube	SpherePyramid	Humans
MPI/OMP Impl. [s]	4.47	13.2	14.2	62.7
Tascell Impl. [s]	3.60	13.4	7.49	61.1
Speedup	1.24	0.99	1.90	1.03

but cannot get further speedups when using 36 workers. This is probably due to heavy memory access. The performance degradation in 36-worker executions is less significant in the Tascell implementation. It is probably because memory accesses are less frequent because workers in Tascell concurrently execute BCT construction and filling.

The MPI/OMP implementation showed better overall best performance on a single node. This is because the load imbalance is not serious with a smaller number of workers and the effect of dynamic load balancing cannot pay for the additional costs in Tascell such as the cost of work stealing and parallel for statements.

6.4.3 Evaluation in Distributed Memory System

Because the best performance on a single node was obtained when using around 16 workers in both implementations and for all the datasets, we used 16 workers in each node for the experiments on DMS. Figure 6.3 shows the total performance of BCT construction and filling on DMS.

From Fig. 6.3(a), we can see that we can get speedups using multiple computing nodes in both implementations. However, the speedups in MPI/OMP are limited when the number of nodes

Table 6.4: Average and maximum loaded time among all threads in the executions of the MPI/OMP implementation using 16 nodes × 16 workers.

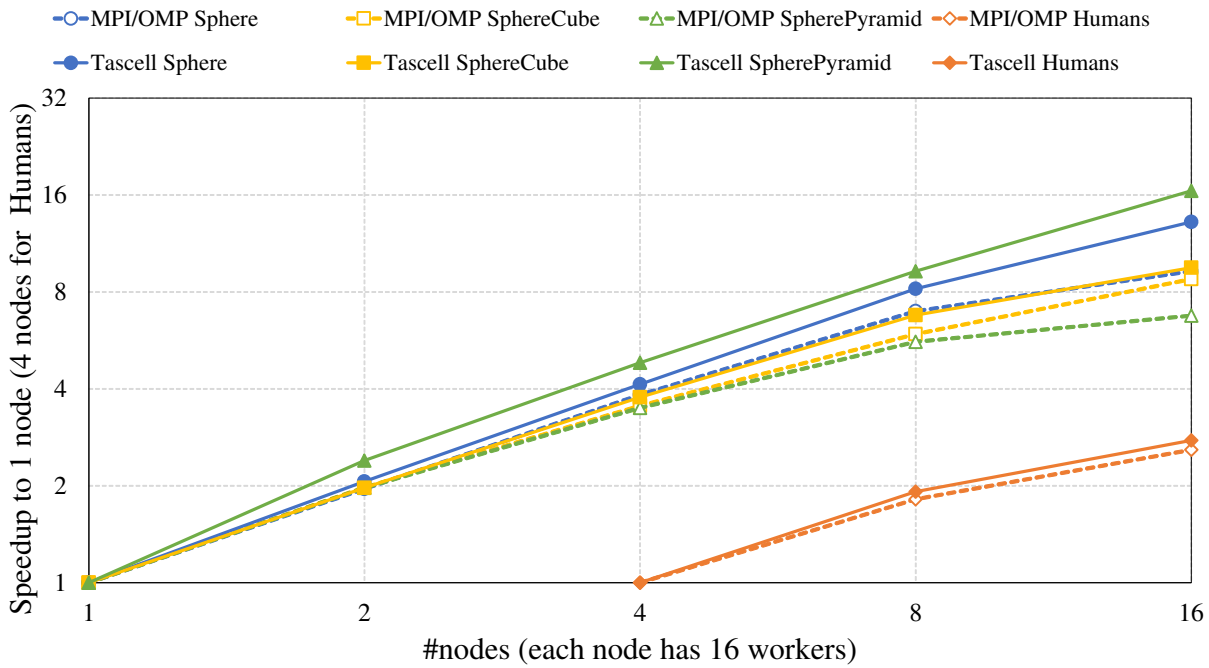
	Sphere	SphereCube	SpherePyramid	Humans
average execution time [s]	3.82	9.95	8.96	52.9
maximum execution time [s]	4.47	13.2	14.2	62.7
maximum / average	1.17	1.32	1.59	1.19

increases. This is because the load imbalance becomes serious with the larger number of workers. From Table 6.4, we can confirm that the load imbalance in the MPI/OMP implementation is considerable. Consequently, in terms of speedups to 1 or 4 nodes, the proposed implementation achieved better performance compared to the MPI/OMP implementation for all the datasets.

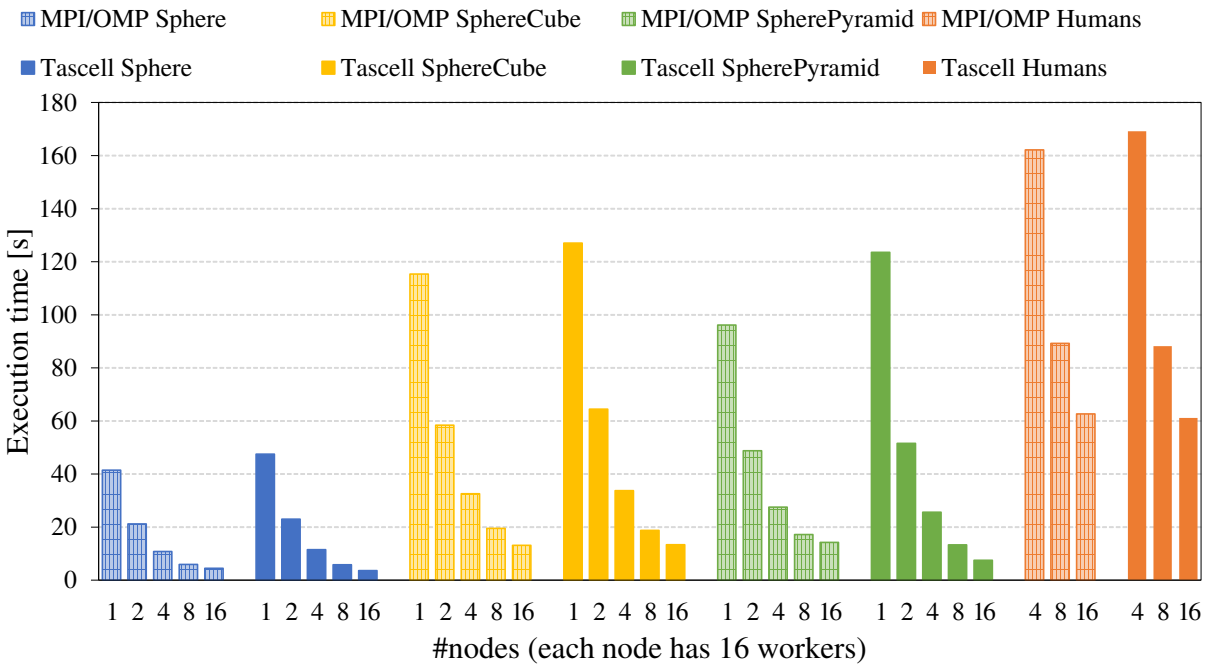
In terms of execution time, we can see from Fig. 6.3(b) and Table 6.3 that the proposed implementation outperformed the MPI/OMP implementation for Sphere and SpherePyramid. In particular, the proposed implementation achieved a 1.9-fold speedup compared to the MPI/OMP implementation for SpherePyramid, in which the degree of load imbalance in MPI/OMP is the largest. The execution times of the proposed implementation are almost the same as those of the MPI/OMP implementation for SphereCube and Humans, because the effect of dynamic load balancing did not outweigh the overhead of Tascell. However, we can expect that Tascell can outperform MPI/OMP with a larger number of workers, where the impact of load imbalance would become more serious.

6.5 Conclusion

In this chapter, we proposed a new parallel implementation of $\mathcal{H}$-matrix construction, where BCT construction is executed in parallel using all workers of all computing nodes. When a worker finds a BCT leaf during BCT construction, it immediately executes filling for the corresponding leaf matrix. To get a good load balance, we parallelized BCT construction using the task parallel language Tascell, which enables easy and efficient parallelization of the tree recursive algorithms by employing a dynamic load balancing strategy. In numerical experiments with 3D electric field analyses using up to 16 computing nodes each of which has 36 cores, our implementation achieved up to 1.9-fold speedups compared to an implementation with the static task assignment strategy.



(a) Speedups to 1-node executions (to 4-node executions for Humans).



(b) Execution time.

Figure 6.3: Total performance of BCT construction and filling on multiple computing nodes.

Chapter 7

Related Work

7.1 Parallel Hierarchical Matrix Construction

As mentioned in Chapter 2, 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 [22]. Besides filling, they also parallelized matrix-vector multiplication, matrix multiplication, and matrix inversion. They summarize their implementations into library Hlib [12]. Ida et al. proposed implementations of the filling operation on distributed memory systems by flat-MPI, OpenMP, and hybrid MPI/OpenMP parallelization [23]. It has been improved in [24] to avoid the trouble that ranks of approximated matrices increase rapidly as the matrix size increases when the conventional $\mathcal{H}$-matrices with ACA are employed to an integral equation whose kernel function has high-order singularities. They also packaged these implementations into $\mathcal{H}$ACApK [28], a library for parallel $\mathcal{H}$-matrix computing. Hoshino et al. achieved even better performance of filling operation and $\mathcal{H}$-matrices arithmetics by GPU, and SIMD vectorization in [26, 27].

However, none of the implementations mentioned above parallelized the matrix partitioning part and most of implementations in DMSs apply a static work assignment strategy based on the estimation of the computing cost of each submatrix, which is not precise. Hoshino et al. proposed a dynamic load balancing implementation of filling and $\mathcal{H}$-matrices-vector multiplication for GPU [27]. However, the dynamic load balancing only works on the thread level and does not support DMSs. Munakata et al. applied dynamic load balancing to hybrid MPI/OpenMP implementations of the filling operation, $\mathcal{H}$-matrix-vector multiplication, and $\mathcal{H}$-matrix-$\mathcal{H}$-matrix multiplication [25]. In their implementations, there is a global task queue only the master process can access and each MPI process has its local task queue. Inside each MPI process, threads get a chunk of tasks from the local queue until the local queue is empty. If a local queue is empty, the corresponding MPI process will ask for tasks from the global task queue. This is different from our implementations using task parallel languages in that we do not need the master process that only maintenance the global task queue.

There are some variant implementations of $\mathcal{H}$-matrices, such as $\mathcal{H}^2$-matrices [41], block low-rank representation (BLR) matrices [50] and *lattice* $\mathcal{H}$-matrices [42]. The $\mathcal{H}^2$-matrices has been parallelized on DMSs [51] using flat-MPI, and the scalability was also evaluated on a small cluster system. The parallelization of BLR matrices is discussed [52] and the *lattice* $\mathcal{H}$-matrices has been parallelized on DMSs using hybrid MPI/OpenMP programming [42].

7.2 Related Applications using Similar Algorithms

7.2.1 K-D Tree Construction

CT construction in our research can be considered as a special instance of k-D tree construction for k = 3. The k-D tree is a space-partitioning data structure for organizing points in a k-dimensional space, widely used in ray tracing. There are many studies dealing with parallelization of the construction of k-D trees using GPU [53, 54] and multicore CPUs [55–58].

Especially, Choi et al. proposed a parallel k-D tree construction algorithm that parallelized both inside of and across recursive steps [55]. However, their algorithm is different from our parallel CT construction 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.

Fatta et al. proposed a parallel k-D tree k-means method using the dynamic load balance on DMSs [56], . However, k-means is a type of clustering method with an iterative refinement process, in which the clustering assignments are updated at each iteration, consequently changing the clusters' definitions. The k-D tree is constructed in every iteration of the k-means process, and the proposed dynamic load balancing distributes the data based on the execution time of each processor in the last iteration. This is completely different from our CT construction because we have only one chance to construct the CT without any evidence of the tree structure.

7.2.2 Fast Multipole Method

The fast multipole method (FMM) [7, 8], widely used in N-body simulations, is also an approximation technique with tree structure construction in its algorithm. It has been parallelized in many proposals in SMSs [59–61], DMSs [62, 63] and GPUs [64–67].

Among them, Taura et al. [61] implemented the tree construction of Fast Multiple Methods (FMM) using a task parallel library, Massive Thread [39]. They showed a remarkable result in absolute performance with small overheads. Amer et al. [68] parallelized FMM by OpenMP Task [36] with awareness of data locality which is called data-driven task parallelism. However, both of them use task parallelism only in SMSs. Note that Taura et al. point out the difficulties to parallelize FMM in DMSs in [61]. One of the difficulties is the problem of the big overheads caused by the random work-stealing across the computing nodes. Our node-aware work-stealing may help with this problem.

7.2.3 Barnes–Hut Simulation

The Barnes–Hut simulation is an approximation algorithm for performing an N-body simulation, which has a similar tree structure to FMM. In [69–71], parallel implementations of tree construction for the Barnes–Hut algorithm are proposed. Especially, Matsui et al. parallelized the Barnes–Hut simulation by Tascell in both SMSs and DMSs in [70, 71]. For better performance, they enhanced the Tascell with space-stealing and probabilistic guards [72]. However, they could not achieve a good speedup as we do in this thesis because the space-stealing caused a large overhead.

7.2.4 Parallel Sorting Implementations

QuickSort is similar to the CT construction algorithm that we employed in the sense that both of them are recursive algorithms with the same pivoting process inside of each recursive step. However, in conventional parallel QuickSort implementations [73, 74], most of them focus on the parallelization of pivoting. In [75], 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.

Bitonic sort [76] is a parallel sorting algorithm with good parallelism. Nakazawa et al. improved the bitonic sort to saw sort [77] using task parallel language, Massive Threads. The scalability of their saw sort and the bitonic sort implementations are impressive that they almost achieved linear speedups when sorting a random array in SMSs.

7.3 Work-stealing Strategy

Many studies have attempted to improve performance by optimizing the work-stealing strategy.

For Tascell, Yoritaka et al. proposed probabilistic guards [72] that prevent thief workers from stealing small tasks from victim workers, thus reducing the total task division cost. Nakashima et al. [78] proposed a work-stealing strategy that considers the amount of work and hierarchy by which a programmer can let each worker estimate and declare the amount of remaining work as priorities or weights. These strategies are different from our node-aware work-stealing strategy in the sense that a thief worker uses information shared by other workers to select a victim worker, whereas, in our strategy, a victim worker accepts or rejects task requests depending on task type.

For other task parallel languages and libraries, Shiina et al. proposed almost deterministic work stealing [79] for MassiveThreads [39] on SMSs, where work stealing is restricted to a certain range to improve data locality based on information declared by the programmer. They evaluate the work-stealing strategy by matrix multiplication and particle interaction calculations. They also applied the strategy to nested parallel programs like recursive repeated map (RRM), QuickSort and k-D tree construction [80].

Chapter 8

Conclusion and Future Work

In this thesis, 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 15.6–16.9 times speedups by Cilk Plus and 17.7–19.1 times speedups 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 15.7–17.7 times speedups by Cilk Plus and 17.8–21.3 times speedups by Tascell.

We extend the implementation of matrix partitioning in the construction of an $\mathcal{H}$-matrix using Tascell in SMSs to DMSs. We propose two implementation methods: distributed cluster tree construction (DCTC) and redundant cluster tree construction (RCTC). In DCTC, both CT and BCT constructions are parallelized using workers in all computing nodes. In RCTC, CT is constructed in every computing node redundantly by employing only intra-node work stealing. The BCT was then constructed in parallel using workers in all computing nodes. RCTC cannot achieve any speedups using multiple computing nodes but can eliminate the data exchange cost that arises in DCTC. To achieve better performance for CT construction in DCTC, we enhanced the Tascell language to enable annotations for parallel loop statements that specify whether part of the loop can be stolen by workers only in internal/external nodes.

Our parallel implementation of the BCT construction is relatively simple compared to that of the CT construction, except that CT nodes need to be exchanged among all computing nodes (in DCTC) or reordered (in RCTC) before BCT construction.

When we evaluated the performance of CT construction, our DCTC implementation achieved speedups using multiple computing nodes compared with the existing implementation on SMSs. We also confirmed that BCT construction yields good speedups using multiple nodes owing to the dynamic load-balancing mechanism of Tascell.

With regard to the entire process of matrix partitioning, our RCTC implementation achieved 1.11–1.20-fold speedups using up to eight nodes, each of which has 36 cores, compared with the single-node performance of the existing implementation of SMSs. Our DCTC implementation could not achieve speedups using multiple computing nodes. However, by evaluating the DCTC implementation, we confirmed that the node-aware work-stealing mechanism is useful for the

parallel implementation of CT construction on DMSs. This mechanism should also be useful for other applications, such as k-D tree applications.

Our implementations on DMSs also have the advantage of handling large input data such that the number of resultant BCT nodes is too large to be stored in a single node.

Finally, we proposed a new parallel implementation of hierarchical matrix construction, where BCT construction is executed in parallel using all workers of all computing nodes. When a worker finds a BCT leaf during BCT construction, it immediately executes filling for the corresponding leaf matrix. To get a good load balance, we parallelized BCT construction using the task parallel language Tascell, which enables easy and efficient parallelization of the tree recursive algorithms by employing a dynamic load balancing strategy. In numerical experiments with 3D electric field analyses using up to 16 computing nodes each of which has 36 cores, our implementation achieved up to 1.9-fold speedups compared to an implementation with the static task assignment strategy.

In future studies, we aim to improve the performance of our implementations by further optimizing the work-stealing strategy. We may also enhance our implementations to construct other low-rank approximation of matrices, such as block low-rank representation (BLR) [50] and *lattice* $\mathcal{H}$-matrices [42], which are proposed as a variant of $\mathcal{H}$-matrices, to obtain a better load balance and construct more efficient communication patterns for arithmetic operations on distributed memory systems. We can expect that better performance can be achieved in CT and BCT construction for lattice $\mathcal{H}$-matrices, utilizing the fact that the number of elements in CT nodes is balanced near the root. We will also apply our node-aware work-stealing strategy to other applications such as k-D tree construction.

Leaf matrices obtained by our implementation using dynamic load balancing are scattered across all computing nodes in random order. This is not convenient for the arithmetic operations of $\mathcal{H}$-matrix after construction, because the cost of data communication in arithmetic operations will be extremely large. In our future studies, we need to consider implementing arithmetic operations intended for such situations or developing a task assignment strategy that alleviates the randomness. To solve this problem, as a preliminary idea, we are planning to implement a new task scheduler with consideration of data placement. When a leaf-node of BCT is built, the position of the corresponding submatrix represented by the leaf-node is determined. Therefore, when doing the filling task of a submatrix, we can give a specified computing node a high priority to steal this task based on the position information to make the result of BCT leaf-nodes that are near to each other distributed to the same computing node. Note that we still reserve the random work-stealing at some level to keep the load balance of BCT construction.

In addition, we will apply our implementation for larger datasets using a larger number of computing nodes. For larger datasets, we can employ parallel CT construction in Section 4.2 and Section 5.3 to pursue better total performance.

Bibliography

[1] C. A. Brebbia and J. Dominguez, *Boundary elements: an introductory course*. WIT press, 1994.

[2] G. Chen and J. Zhou, *Boundary element methods*, vol. 92. Academic press London, 1992.

[3] S. J. Aarseth and S. J. Aarseth, *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003.

[4] V. Voevodin, "On a method of reducing the matrix order while solving integral equations," *Numerical Analysis on Fortran (Mosk. Gos. Univ., Moscow, 1979)*, pp. 21–26, 1979.

[5] W. Hackbusch and Z. P. Nowak, "On the fast matrix multiplication in the boundary element method by panel clustering," *Numerische Mathematik*, vol. 54, no. 4, pp. 463–491, 1989.

[6] S. Myagchilov and E. Tyrtyshnikov, "A fast matrix-vector multiplier in discrete vortex method," *Russian Journal of Numerical Analysis and Mathematical Modelling*, 1992.

[7] V. Rokhlin, "Rapid solution of integral equations of classical potential theory," *Journal of Computational Physics*, vol. 60, no. 2, pp. 187–207, 1985.

[8] L. Greengard and V. Rokhlin, "A new version of the fast multipole method for the laplace equation in three dimensions," *Acta numerica*, vol. 6, pp. 229–269, 1997.

[9] W. Dahmen, S. Prößdorf, and R. Schneider, "Wavelet approximation methods for pseudodifferential equations ii: Matrix compression and fast solution," *Advances in computational Mathematics*, vol. 1, no. 3, pp. 259–335, 1993.

[10] W. Hackbusch, "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.

[11] W. Hackbusch and B. Khoromskij, "A sparse $\mathcal{H}$-matrix arithmetic. part II: Application to multi-dimensional problems," *Computing*, vol. 64, no. 1, p. 21–47, 2000.

[12] S. Börm, L. Grasedyck, and W. Hackbusch, "Hierarchical matrices. lecture note," 2005.

[13] M. Bebendorf, "Approximation of boundary element matrices," *Numerische Mathematik*, vol. 86, no. 4, pp. 565–589, 2000.

[14] M. Bebendorf and S. Rjasanow, "Adaptive low-rank approximation of collocation matrices," *Computing*, vol. 70, no. 1, p. 1–24, 2003.

[15] S. Börm and L. Grasedyck, "Hybrid cross approximation of integral operators," *Numerische Mathematik*, vol. 101, no. 2, pp. 221–249, 2005.

[16] M. Faustmann, J. Melenk, and D. Praetorius, "Existence of $\mathcal{H}$-matrix approximants to the inverses of bem matrices: The simple-layer operator," *Mathematics of Computation*, vol. 85, no. 297, pp. 119–152, 2016.

[17] M. Ohtani, K. Hirahara, Y. Takahashi, T. Hori, M. Hyodo, H. Nakashima, and T. Iwashita, "Fast computation of quasi-dynamic earthquake cycle simulation with hierarchical matrices," *Procedia Computer Science*, vol. 4, pp. 1456–1465, 2011.

[18] J. Ostrowski, Z. Andjelic, M. Bebendorf, B. Cranganu-Cretu, and J. Smajic, "Fast bem-solution of laplace problems with h-matrices and aca," *IEEE Transactions on Magnetics*, vol. 42, no. 4, pp. 627–630, 2006.

[19] M. Bebendorf and W. Hackbusch, "Existence of $\mathcal{H}$-matrix approximants to the inverse fe-matrix of elliptic operators with L^∞-coefficients," *Numerische Mathematik*, vol. 95, no. 1, pp. 1–28, 2003.

[20] S. Börm, "Approximation of solution operators of elliptic partial differential equations by $\mathcal{H}$- and $\mathcal{H}^2$-matrices," *Numerische Mathematik*, vol. 115, no. 2, pp. 165–193, 2010.

[21] S. Kurz, O. Rain, and S. Rjasanow, "The adaptive cross-approximation technique for the 3D boundary-element method," *IEEE transactions on Magnetics*, vol. 38, no. 2, p. 421–424, 2002.

[22] R. Kriemann, "Parallel-matrix arithmetics on shared memory systems," *Computing*, vol. 74, no. 3, p. 273–297, 2005.

[23] A. Ida, T. Iwashita, T. Mifune, and Y. Takahashi, "Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters," *Journal of information processing*, vol. 22, no. 4, p. 642–650, 2014.

[24] A. Ida, T. Iwashita, M. Ohtani, and K. Hirahara, "Improvement of hierarchical matrices with adaptive cross approximation for large-scale simulation," *Journal of Information Processing*, vol. 32, no. 3, p. 366–372, 2015.

[25] K. Munakata, T. Hiraishi, A. Ida, T. Iwashita, and H. Nakashima, "Parallel hierarchical matrix arithmetics using dynamic load balancing," tech. rep., Research Report in High-Performance Computing (HPC), 2015. (in Japanese).

[26] T. Hoshino, A. Ida, T. Hanawa, and K. Nakajima, "Design of parallel BEM analyses framework for SIMD processors," in *Computational Science – ICCS 2018*, 2018.

[27] T. Hoshino, A. Ida, T. Hanawa, and K. Nakajima, "Load-balancing-aware parallel algorithms of H-matrices with adaptive cross approximation for GPUs," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 35–45, IEEE, 2018.

[28] A. Ida and T. Iwashita, "Hacapk." `https://github.com/Post-Peta-Crest/ppOpenHPC/tree/MATH/HACApK` (accessed 2022-9-10).

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

[30] C. E. Leiserson, "The cilk++ concurrency platform," in *Proceedings of the 46th Annual Design Automation Conference*, pp. 522–527, 2009.

[31] H. Vandierendonck, "The cilk and cilk++ programming languages," *Multicore Computing: Algorithms, Architectures, and Applications*, p. 91, 2013.

[32] A. D. Robison, "Composable parallel patterns with intel cilk plus," *Computing in Science & Engineering*, vol. 15, no. 02, pp. 66–71, 2013.

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

[34] Intel Corporation, "A quick, easy and reliable way to improve threaded performance — Intel Cilk Plus." `https://software.intel.com/en-us/intel-cilk-plus`.

[35] T. B. Schardl, I.-T. A. Lee, and C. E. Leiserson, "Brief announcement: Open cilk," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pp. 351–353, 2018.

[36] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*.

[37] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[38] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in intel threading building blocks.," *Intel Technology Journal*, vol. 11, no. 4, 2007.

[39] J. Nakashima and K. Taura, "Massivethreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond: Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday* (G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, eds.), pp. 222–238, Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

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

[41] W. Hackbusch, B. Khoromskij, and S. Sauter, "On H^2-matrices: Lectures on applied mathematics," 2000.

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

[43] S. Goreinov, E. Tyrtyshnikov, and N. Zamarashkin, "A theory of pseudoskeleton approximations," *Linear Algebra and its Applications*, vol. 261, no. 1, pp. 1–21, 1997.

[44] E. Mohr, D. A. Kranz, and R. H. Halstead, "Lazy task creation: A technique for increasing the granularity of parallel programs," in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, (New York, NY, USA), p. 185–197, Association for Computing Machinery, 1990.

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

[46] T. Hiraishi, M. Yasugi, and S. Umatani, "Evaluation of the tascell dynamic load balancing framework in widely distributed and many-core environments," in *Proc. Symposium on Advanced Computing Systems and Infrastructures 2011, SACSIS 2011*, pp. 55–63, 2011.

[47] D. Muraoka, M. Yasugi, T. Hiraishi, and S. Umatani, "Evaluation of an MPI-based implementation of the Tascell task-parallel language on massively parallel systems," in *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pp. 161–170, IEEE, 2016.

[48] G. E. Blelloch, "Prefix sum and their applications," technical report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

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

[50] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. l'Excellent, and C. Weisbecker, "Improving multifrontal methods by means of block low-rank representations," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. A1451–A1474, 2015.

[51] S. Börm and J. Bendoraityte, "Distributed h2-matrices for non-local operators," *Computing and Visualization in Science*, vol. 11, pp. 237–249, 07 2008.

[52] A. Ida, H. Nakashima, and M. Kawai, "Parallel hierarchical matrices with block low-rank representation on distributed memory computer systems," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pp. 232–240, 2018.

[53] W. Zhefeng, Z. Fukai, and L. Xinguo, "SAH kD-tree construction on GPU," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pp. 71–78, 2011.

[54] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kD-tree construction on graphics hardware," *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, p. 126, 2008.

[55] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, "Parallel SAH k-D tree construction," in *Proceedings of the Conference on High Performance Graphics. Eurographics Association*, pp. 77–86, 2010.

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

[57] S. Popov, J. Gunther, H.-p. Seidel, and P. Slusallek, "Experiences with streaming construction of sah kd-trees," in *2006 IEEE Symposium on Interactive Ray Tracing*, pp. 89–94, 2006.

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

[59] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–12, IEEE, 2010.

[60] L. Ying, G. Biros, and D. Zorin, "A kernel-independent adaptive fast multipole algorithm in two and three dimensions," *Journal of Computational Physics*, vol. 196, no. 2, pp. 591–626, 2004.

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

[62] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta, "A parallel adaptive fast multipole method," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pp. 54–65, 1993.

[63] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pp. 12–21, 1993.

[64] J. Bédorf, E. Gaburov, and S. P. Zwart, "A sparse octree gravitational n-body code that runs entirely on the gpu processor," *Journal of Computational Physics*, vol. 231, no. 7, pp. 2825–2839, 2012.

[65] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, *et al.*, "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2010.

[66] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "Scaling hierarchical n-body simulations on gpu clusters," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2010.

[67] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, IEEE, 2009.

[68] A. Amer, S. Matsuoka, M. Pericàs, N. Maruyama, K. Taura, R. Yokota, and P. Balaji, "Scaling fmm with data-driven openmp tasks on multicore architectures," in *OpenMP: Memory, Devices, and Tasks* (N. Maruyama, B. R. de Supinski, and M. Wahib, eds.), (Cham), pp. 156–170, Springer International Publishing, 2016.

[69] H. Shan and J. P. Singh, “Parallel tree building on a range of shared address space multiprocessors: algorithms and application performance,” in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pp. 475–484, March 1998.

[70] K. Matsui, H. Tasuku, M. Yasugi, and S. Umatani, “A parallel implementation of a fast variant of the barnes-hut simulation,” *Symposium on Advanced Computing Systems and Infrustractures*, vol. 2012, pp. 298–306, 2012. (in Japanese).

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

[72] H. Yoritaka, K. Matsui, M. Yasugi, T. Hiraishi, and S. Umatani, “Extending a work-stealing framework with probabilistic guards,” in *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, pp. 171–180, 2016.

[73] P. Tsigas and Y. Zhang, “A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000,” in *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003. Proceedings.*, pp. 372–381, 2003.

[74] P. Sanders and T. Hansch, “Efficient massively parallel quicksort,” in *Solving Irregularly Structured Problems in Parallel* (G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, eds.), (Berlin, Heidelberg), pp. 13–24, Springer Berlin Heidelberg, 1997.

[75] S. Saleem, M. I. Lali, M. S. Nawaz, and A. B. Nauman, “Multi-core program optimization: Parallel sorting algorithms in Intel Cilk Plus,” *International Journal of Hybrid Information Technology*, vol. 7, no. 2, p. 151–164, 2014.

[76] K. E. Batcher, “Sorting networks and their applications,” in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pp. 307–314, 1968.

[77] T. Nakazawa and K. Taura, “high-speed parallelization of bitonic sort,” *IPSJ SIG Technical Report*, vol. 2012, no. 12, pp. 1–7, 2012. (in Japanese).

[78] R. Nakashima, M. Yasugi, H. Yoritaka, T. Hiraishi, and S. Umatani, “Work-stealing strategies that consider work amount and hierarchy,” *Journal of Information Processing*, vol. 29, pp. 478—489, 2021.

[79] S. Shiina and K. Taura, “Almost deterministic work stealing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’19, (New York, NY, USA), Association for Computing Machinery, 2019.

[80] S. Shiina and K. Taura, “Improving cache utilization of nested parallel programs by almost deterministic work stealing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4530–4546, 2022.

List of Publications

Journal Papers

1. Z. Bai, T. Hiraishi, H. Nakashima, A. Ida, and M. Yasugi, "Parallelization of Matrix Partitioning in Construction of Hierarchical Matrices using Task Parallel Languages," *Journal of Information Processing*, Vol. 27, pp. 840–851, 2019.

2. Z. Bai, T. Hiraishi, A. Ida, and M. Yasugi, "Parallelization of Matrix Partitioning in Hierarchical Matrix Construction on Distributed Memory Systems," *Journal of Information Processing*, Vol. 30, pp. 742–754, 2022.

International Conference Papers

1. Z. Bai, T. Hiraishi, A. Ida, M. Yasugi, and K. Fukazawa, "Construction of Hierarchical Matrix on Distributed Memory Systems using a Task Parallel Language," In *Proc. 2022 The Tenth International Symposium on Computing and Networking Workshops*, *CANDARW 2022*, Himeji, Japan, pp. 48–54, November 2022. (to appear)

International Conference Posters

1. Z. Bai, T. Hiraishi, H. Nakashima, A. Ida, and M. Yasugi, "Implementation of Partitioning of Hierarchical Matrices using Task Parallel Languages," *The 48th International Conference on Parallel Processing*, *ICPP2019*, Kyoto, Japan, August 2019.

Awards

1. Z. Bai, T. Hiraishi, H. Nakashima, A. Ida, and M. Yasugi, "Outstanding Research Award," *The 3rd cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming*, *xSIG2019*, May 2019.

2. Z. Bai, T. Hiraishi, H. Nakashima, A. Ida, and M. Yasugi, "Best Poster Award," *The 48th International Conference on Parallel Processing*, *ICPP2019*, August 2019.

Publications Used in the Thesis

Part of this thesis are based on authors's publications as follows.

Chapter 2 & 3 & 4

Z. Bai, T. Hiraishi, H. Nakashima, A. Ida, and M. Yasugi, "Parallelization of Matrix Partitioning in Construction of Hierarchical Matrices using Task Parallel Languages," *Journal of Information Processing*, Vol. 27, pp. 840–851, 2019.
doi: https://doi.org/10.2197/ipsjjip.27.840

Chapter 3 & 5

Z. Bai, T. Hiraishi, A. Ida, and M. Yasugi, "Parallelization of Matrix Partitioning in Hierarchical Matrix Construction on Distributed Memory Systems," *Journal of Information Processing*, Vol. 30, pp. 742–754, 2022.
doi: https://doi.org/10.2197/ipsjjip.30.742

Chapter 2 & 6

Z. Bai, T. Hiraishi, A. Ida, M. Yasugi, and K. Fukazawa, "Construction of Hierarchical Matrix on Distributed Memory Systems using a Task Parallel Language," In *Proc. 2022 The Tenth International Symposium on Computing and Networking Workshops*, *CANDARW 2022*, Himeji, Japan, pp. 48–54, November 2022.
doi: (to appear)