Implementation of Partitioning of Hierarchical Matrices using Task Parallel Languages

Zhengyang Bai*
haku@sys.i.kyoto-u.ac.jp
Graduate School of Informatics, Kyoto University
Kyoto

Tasuku Hiraishi
Hiroshi Nakashima
Academic Center for Computing and Media Studies, Kyoto University
Kyoto

Akihiro Ida
Information Technology Center, the University of Tokyo
Tokyo

Masahiro Yasugi
Department of Computer Science and Networks, Kyushu Institute of Technology
Iizuka, Fukuoka

1 INTRODUCTION

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

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

$\mathcal{H}$-matrix construction is achieved by dividing a matrix into submatrices (partitioning), followed by calculating the element values of these submatrices (filling). We can find many proposals [5–8, 11, 12] to parallelize the filling operation and they are applied to $\mathcal{H}$-matrix libraries such as Hlib [1] and $\mathcal{H}$ACApK [7], but the partitioning operation still remains sequential. This is partly because the cost of the partitioning operation is much lower compared to the filling operation. However, as hundreds of speedups have been achieved for the filling operation using MPI, GPU, and SIMD vectorization [5, 6, 12]. We can expect more speedups using more computing resources in the near future. Then the partitioning operation will be a bottleneck if it remains sequential and it will be significant for larger datasets. Thus, we should also consider parallelizing the partitioning operation. Therefore, in this presentation, we present parallel implementations for matrix partitioning in the construction of $\mathcal{H}$-matrices, based on the sequential version proposed in [3].

The matrix partitioning operation is divided into the following two steps: construction of a cluster tree (CT) to split clusters recursively and construction of a block cluster tree (BCT) [1] to examine the admissibility of the cluster pair and to determine the matrix structure recursively. As trees constructed and traversed in these steps are unpredictably unbalanced, we employed task parallel languages, Cilk Plus [9] and Tascell [4], to parallelize these operations solving the load imbalance problem with reasonable programming cost.

2 MATRIX PARTITIONING ALGORITHM

2.1 Cluster Tree Construction

First, we show the algorithm to construct a CT. The cluster $\mathcal{E}_1{}^{(0)} = \{e_0, ..., e_{N-1}\}$ containing all input elements is treated as the root node of CT. The children of a CT node are created by dividing the cluster into two sub-clusters. We can create the children of each child node by dividing the corresponding cluster in the same manner. Such division operations are repeated recursively until the size of the cluster becomes less than the threshold N_{min}. In each recursive step, there are many ways to divide a cluster. In BEM, elements are often divided by pivoting based on their coordinate.

2.2 Block Cluster Tree Construction

In BCT construction, we use the CT constructed in the previous step. A node of BCT in an arbitrary level corresponds to a pair of two nodes of CT (corresponding to two clusters) in the same level. If a pair of clusters satisfies an *admissibility condition*, the corresponding BCT node does not have its child nodes as it means that the interaction between the clusters can be approximated by a low-rank submatrix. If the admissibility condition cannot be satisfied and one of both CT nodes are leaves, we determine the corresponding submatrix cannot be approximated and make the BCT node leaf for a full submatrix. Otherwise, i.e., if the non-leaf cluster pair

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

is not admissible, the BCT node has four children corresponding to all pairs of two children of the CT nodes.

3 PARALLEL IMPLEMENTATION

3.1 Cluster Tree Construction

It is obvious that the recursive calls in the CT constrction can be executed in parallel. However, after preliminary evaluations, we found that the parallel performance is far below our expectations. This is because the computation cost of each recursion step is proportional to the number of elements and the critical path thus cannot be shortened when only recursive calls are executed in parallel. To obtain better performance, we also parallelized inside the recursion step using work stealing based parallel loops provided by Cilk Plus and Tascell.

The costly operations in the recursion step are two-fold: 1) finding the maximum and minimum 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. Parallelizing 1) is relatively easy, but parallelizing 2) is more difficult. In sequential implementations, we can easily reorder the elements in-place using the commonly used algorithm for Quicksort. 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.

3.2 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.

The only concern is about the space to which leaf nodes of BCT are stored. In the sequential implementation, they are stored to the global array. However, sharing such a single array controlled by a lock among workers brings large overheads. Therefore, we allocated space for each worker.

4 PERFORMANCE EVALUATION

We evaluate our parallel implementations with four datasets from which coefficient matrices of the surface element method are generated [10]. We measured the performance using one computing node having two 18-core Xeon processors.

We tuned the following three parameters: 1) T_N denotes the threshold of the number of elements that decides whether recursive function calls are executed in parallel in CT and BCT construction. 2) T_S denotes the threshold of the number of elements for deciding whether computations inside a recursive step are parallelized in CT construction. 3) C is the chunk size used in parallel executions of the pivoting operation in CT construction.

As a result, compared to a sequential implementation in C, we achieved 10.5–11.5 times speedups by Cilk Plus and 10.6–12.6 times speedup by Tascell for the CT construction. For the BCT construction, speedups using Cilk Plus are 18.9–37.7 times and those using Tascell are 22.7–38.8 times. In regard to the whole process of matrix partitioning, we achieved 10.7–12.2 times speedups by Cilk Plus and 11.5–14.5 times speedups by Tascell. Figure 1 shows the total performance of matrix partioning, i.e., both CT and BCT construction, for the Humans data set, which has 98,320,000 elements.

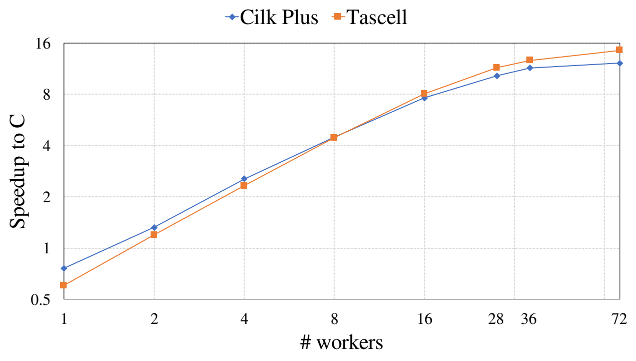

Figure 1: Total performance of the Cilk Plus and Tascell implementations of matrix partitioning (for Humans).

REFERENCES

[1] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. 2005. Hierarchical Matrices. lecture note.

[2] Wolfgang Hackbusch. 1999. A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing* 62, 2 (1999), 89–108.

[3] W. Hackbusch and B.N Khoromskij. 2000. A Sparse $\mathcal{H}$-matrix Arithmetic. Part II: Application to Multi-dimensional Problems. *Computing* 64, 1 (2000), 21–47.

[4] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. 2009. Backtracking-based Load Balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*. 55–64.

[5] Tetsuya Hoshino, Akihiro Ida, Toshihiro Hanawa, and Kengo Nakajima. 2018. Design of Parallel BEM Analyses Framework for SIMD Processors. In *Computational Science – ICCS 2018*, Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot (Eds.). Springer International Publishing, Cham, 601–613.

[6] Akihiro Ida. 2018. Lattice $\mathcal{H}$-Matrices on Distributed-Memory Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 389–398. https://doi.org/10.1109/IPDPS.2018.00049

[7] Akihiro Ida, Takeshi Iwashita, Takeshi Mifune, and Yasuhito Takahashi. 2014. Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters. *Journal of information processing* 22, 4 (2014), 642–650.

[8] Akihiro Ida, Takeshi Iwashita, Makiko Ohtani, and Kazuro Hirahara. 2015. Improvement of hierarchical matrices with adaptive cross approximation for large-scale simulation. *Journal of Information Processing* 32, 3 (2015), 366–372.

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

[10] Takeshi Iwashita, Akihiro Ida, Takeshi Mifune, and Yasuhito Takahashi. 2017. Software Framework for Parallel BEM Analyses with $\mathcal{H}$-matrices Using MPI and OpenMP. *Procedia Computer Science* 108 (2017), 2200 – 2209. https://doi.org/10.1016/j.procs.2017.05.263 International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[11] Ronald Kriemann. 2005. Parallel-matrix arithmetics on shared memory systems. *Computing* 74, 3 (2005), 273–297.

[12] Katsumi Munakata, Tasuku Hiraishi, Akihiro Ida, Takeshi Iwashita, and Hiroshi Nakashima. 2015. Parallel Hierarchical Matrix Arithmetics using Dynamic Load Balancing. In *Research Report in High-Performance Computing (HPC)*, Vol. 2015. 1–15. (in Japanese).