

Construction of Large Zero-Aware Pattern Databases for Sliding Puzzles on Distributed Memory Machines

Tomoya Nagahashi[1] and Daisuke Takahashi[2(✉)]

[1] Graduate School of Science and Technology, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan
nagahashi@hpcs.cs.tsukuba.ac.jp

[2] Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305-8577, Japan
daisuke@cs.tsukuba.ac.jp

Abstract. A pattern database, which is a precomputed lookup table that estimates the cost required to reach the goal state, is used for pruning in heuristic searches. In the search for an optimal solution to a sliding puzzle, a zero-aware pattern database (ZPDB) enables highly effective pruning. Using a database that considers the interactions of more tiles enhances pruning but increases computational time and memory usage. Here, we develop a program for constructing a large ZPDB on a distributed memory system. The program is based on a breadth-first search algorithm and is parallelized using MPI and OpenMP. An evaluation of its parallel performance reveals that increasing the number of MPI processes by 48-fold results in up to a 24x speedup. Furthermore, we construct a large ZPDB for the 24-puzzle on a 5×5 board that stores all patterns formed by nine tiles and the blank (more than 1.7 trillion entries) in 41 min using 2304 compute nodes.

Keywords: Pattern database · Breadth-first search · Parallelization

1 Introduction

The 15-puzzle is a sliding puzzle in which 15 tiles are arranged on a 4×4 board, as shown in Fig. 1. Rearrangement is performed by repeatedly sliding tiles into the blank on the board. A generalization of this puzzle to an $N \times N$ board is commonly referred to as the $(N^2 - 1)$-puzzle. Search-based methods are used to find an optimal solution that minimizes the number of tile moves. Various pruning techniques have been studied to efficiently explore the vast search space.

A pattern database (PDB) is a table that provides a lower bound on the number of moves required for an optimal solution from a given tile configuration. The values in this precomputed table are referenced during the search for pruning. There are several methods for building PDBs, and among them, the

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
O. Gervasi et al. (Eds.): ICCSA 2025, LNCS 15648, pp. 272–284, 2025.
https://doi.org/10.1007/978-3-031-97000-9_17

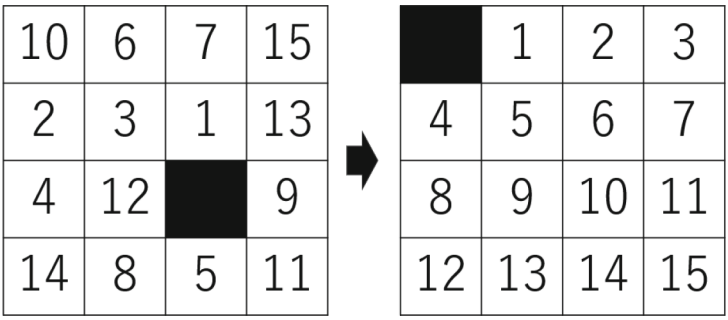

Fig. 1. 15-puzzle

Zero-Aware Pattern Database (ZPDB) [8] enables particularly high-performance pruning.

In the present study, we implement a program that uses MPI [1] and OpenMP [2] for constructing large ZPDBs that can be executed in a large-scale distributed memory environment.

2 Background

The problem of finding an optimal solution for the $(N^2 - 1)$-puzzle is NP-hard [14]. The number of possible board states is proportional to $N^2!$, exceeding 10^{13} for the 15-puzzle and reaching approximately 7×10^{24} for the 24-puzzle. Therefore, to efficiently find an optimal solution, heuristic search methods are used.

A heuristic is said to be admissible if it is guaranteed never to overestimate the true cost. Using an admissible heuristic, search algorithms such as iterative deepening A* (IDA*) [13] can be used to find the optimal solution for the $(N^2 - 1)$-puzzle. Thus, a heuristic that estimates the number of moves as accurately as possible without exceeding the true value is required.

The Manhattan distance is a simple example of an admissible heuristic for the $(N^2 - 1)$-puzzle. Since a tile move reduces the Manhattan distance between the tile's current position and its goal position by at most one, the sum of the Manhattan distances of all tiles from their current positions to their goal positions provides a trivial lower bound on the number of required moves.

3 Pattern Databases

Several types of PDB have been proposed, including non-additive PDB [10], the additive PDB (APDB) [12], and the ZPDB [8].

3.1 Non-additive Pattern Database

The PDB for the $(N^2 - 1)$-puzzle, proposed by Culberson and Schaeffer [10], records the number of moves required to solve a new puzzle. It focuses on the configuration of specific tiles (pattern tiles), while treating all other tiles as identical. Since the positions of tiles not included in the pattern are not considered, the new puzzle is a sub-problem of the original. The number of moves in the optimal solution for the new puzzle does not exceed that of the original problem. To distinguish this PDB from other types of PDB, it is also referred to as the non-additive PDB.

3.2 Additive Pattern Database

Felner et al. proposed the APDB [12], which enables high-performance pruning by dividing tiles into multiple groups and summing the heuristic values for each pattern, as shown in Fig. 2. In the non-additive PDB, even if tiles are divided into multiple groups and a separate database is created for each group, their values could not be summed because counting tile moves redundantly could make the heuristic inadmissible. In the APDB, only the moves of the pattern tiles are counted and thus the same move is not counted redundantly across different groups, allowing the heuristic values to be summed. This enables the inclusion of all tile moves in the heuristic evaluation, leading to more effective pruning.

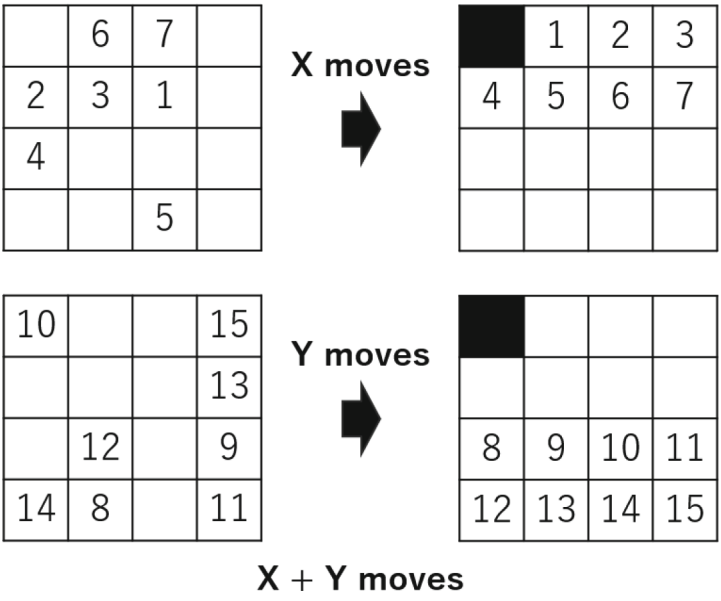

Fig. 2. Heuristic using APDB

Döbbelin et al. proposed an algorithm for constructing an APDB in a distributed memory environment and created an APDB for the 24-puzzle by dividing the tiles into three groups of nine, nine, and six tiles, respectively [11]. The APDB that recorded the patterns of nine tiles had approximately 7.4×10^{11} entries. Even if each entry required only one byte, this APDB would exceed 700 GB.

3.3 Zero-Aware Pattern Database

The ZPDB [8], proposed by Clausecker and Reinefeld, is an extension of the APDB. In the APDB, to reduce memory usage, the position of the blank is not considered. As a result, depending on the position of the blank, the estimated number of moves may differ significantly from the actual value. In the ZPDB, the position of the blank is taken into account. Instead of considering the coordinates of the blank, the values are recorded for each region divided by pattern tiles, thereby reducing the number of states. For example, in the 24-puzzle, considering a pattern of eight tiles, there are 17 possible positions for the blank, but as shown in Fig. 3, the maximum number of regions for the blank is seven, so the values for these seven possibilities are recorded. In practice, cases where the space is divided into seven regions are rare and the number of states is even lower. Since the moves of tiles not included in the pattern are ignored, this does not affect pruning performance.

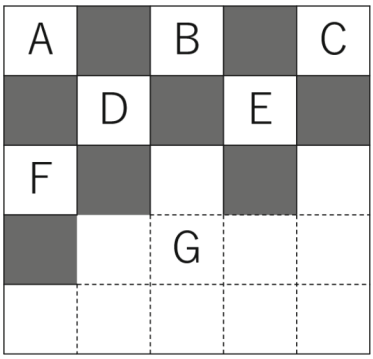

Fig. 3. Example of configuration with maximum of seven blank regions formed by eight tiles in 24-puzzle

In addition, the ZPDB can reduce memory usage by compressing the data per entry to one bit. Consider a graph where tile configuration patterns are treated as vertices and an edge is drawn between vertices if one pattern can be obtained from another by moving a tile within the same group exactly once. Since tile moves are reversible, this graph is undirected. Moreover, since the parity of the sum of the Manhattan distances from each tile's coordinates to the origin differs

between adjacent vertices, the graph is bipartite. The values stored in a ZPDB can be interpreted as the shortest distances from the vertex corresponding to the goal pattern to each vertex. They can be computed using a breadth-first search (BFS) starting from the goal pattern's vertex. When compressing the data per entry to one bit, instead of storing the exact shortest path length, only the upper bits of the value obtained by dividing it by four are stored. The shortest path length is then recovered during lookup by reconstructing the shortest path. In general graphs, if the remainder when dividing each vertex's shortest path length by three is stored, one of the shortest paths to the start vertex can be obtained by repeatedly moving to a vertex closer to the start [9]. In the ZPDB, the remainder is taken modulo 4 and the lower bits, which can be reconstructed using the pattern parity, are discarded.

Clausecker developed a program for constructing ZPDBs in a shared memory environment [3]. It is written in C and can be executed using multiple threads.

4 Implementation

4.1 Breadth-First Search

A ZPDB can be constructed using a BFS. One of the BFS implementations suitable for parallelization is the level-synchronized BFS [7]. The algorithm is shown in Algorithm 1. In this method, a BFS is performed by repeatedly identifying the set of unvisited vertices that can be reached by traversing an edge from the set of visited vertices with the maximum shortest distance.

Algorithm 1. Parallel level-synchronized BFS [7]

Require: $G(V, E)$: The graph, v: Source vertex
Ensure: Array $d[1..n]$ with $d[v]$ holding the length of the shortest path from s to $v \in V$, assuming unit-weight edges

```
 1: for all v ∈ V in parallel do
 2:     d[v] ← −1
 3: end for
 4: d[s] ← 0
 5: Q ← φ
 6: Enqueue s ← Q
 7: while Q ≠ φ do
 8:     for all u ∈ Q in parallel do
 9:         Delete u ← Q
10:         for each v adjacent to u in parallel do
11:             if d[v] = −1 then
12:                 d[v] ← d[v] + 1
13:                 Enqueue v ← Q
14:             end if
15:         end for
16:     end for
17: end while
```

4.2 Parallelization with MPI

When executing a BFS in a distributed memory environment, each node has partial information about the vertex set and distances. There are various methods for partitioning graph data. Here, we implement a simple one-dimensional partitioning method that evenly distributes arrays across all MPI processes.

Each MPI process maintains two arrays for managing vertices: one array is used to store the upper bits of the remainder when the shortest distance is divided by four and one array is used to track whether each vertex has been visited during the BFS. Both arrays consist of 8-bit integers, each of which stores information corresponding to all blank space regions in a single tile pattern.

In each BFS step, each node computes the set of reachable vertices by traversing one edge from the vertices it manages. It then sends these vertices to the nodes responsible for managing them. Upon receiving a vertex set, each node updates its array accordingly.

For exchanging newly discovered vertices, we utilize the collective communication supported by MPI. Specifically, we use MPI_Alltoall() to share the number of data elements to be sent and received in advance and MPI_Alltoallv() to perform the actual data exchange.

In addition, we use MPI_Allreduce() to share the number of newly explored vertices among all nodes and determine when the search should terminate.

4.3 Parallelization with OpenMP

The process of enumerating neighboring vertices from explored vertices using edge information, as well as updating data based on the information received from other nodes, can be executed in parallel. To accelerate the program, we utilize OpenMP to distribute the workload across multiple threads.

Since the information for eight vertices is handled within a single variable, updating different vertices may involve accessing the same variable. OpenMP provides directives such as #pragma omp atomic to ensure exclusive access; however, using these directives can introduce execution overhead due to synchronization delays and contention between threads waiting for write access.

To mitigate this issue, we sort the received data such that updates affecting the same variable are grouped. This allows only a single designated thread to perform the writes, eliminating the need for atomic operations. For sorting, we use the parallelizable and high-performance Radix Sort, specifically the implementation by Akiba [4].

The proposed algorithm is shown in Algorithm 2.

4.4 Reduction of Memory Usage

The memory bottleneck in this program arises from the send and receive buffers used for exchanging vertex information during all-to-all communication. When constructing a large ZPDB, memory shortages may occur.

Algorithm 2. Hybrid MPI/OpenMP parallel level-synchronized BFS

Require:
p: Rank of MPI process,
P: Number of MPI processes,
V_p: p-th vertex set of the graph,
v_{root}: Source vertex
Ensure: $dist_p[v]$ = upper bit of minimum distance from v_{root} to v modulo 4 for all $v \in V_p$

```
 1: S_p ← {v_root} if v_root ∈ V_p else φ
 2: closed_p ← [false, ··· , false]
 3: dist_p ← [0, ··· , 0]
 4: d ← 0
 5: while Σ_p |S_p| ≠ 0 do
 6:     for all p' ∈ {0, 1, ··· P − 1} do
 7:         S_{p,p'} ← φ
 8:     end for
 9:     for all v ∈ S_p do                          ▷ in parallel with OpenMP
10:         dist_p[v] ← ⌊d/2⌋ mod 2
11:         closed_p[v] ← true
12:         for all v+ ∈ neighbors(v) do
13:             p' ← rank of MPI process s.t. v+ ∈ V_{p'}
14:             S_{p,p'} ← S_{p,p'} ∪ {v+}
15:         end for
16:     end for
17:     Exchange |S_{p,p'}| for |S_{p',p}| by MPI_AlltoAll()
18:     Exchange S_{p,p'} for S_{p',p} by MPI_Alltoallv()
19:     S_p ← ⋃_{p'} S_{p',p}
20:     parallel_radix_sort(S_p)                    ▷ to avoid race condition
21:     d ← d + 1
22:     Calculate Σ_p |S_p| by MPI_Allreduce()
23: end while
```

To mitigate this issue, we set an upper limit on the amount of data sent and received at one time. If this limit is exceeded, communication is divided into multiple rounds, thereby preventing memory exhaustion.

Döbbelin et al. encountered a similar issue in their method for constructing a large APDB [11]. They also addressed it by dividing communication into multiple rounds.

5 Evaluation

5.1 Parallel Performance

We executed the proposed program with eight tiles of the 24-puzzle shown in Fig. 4 as input and evaluated its parallelization performance. The program was executed on the Pegasus supercomputer [5], which is operated by the Center for Computational Sciences, University of Tsukuba. The detailed system specifications are shown in Table 1.

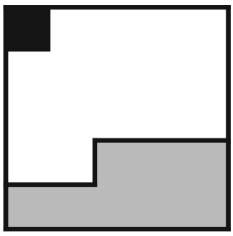

Fig. 4. Configuration of eight input pattern tiles

Table 1. Execution environment (Pegasus)

CPU	Intel®Xeon Platinum 8468 (2.1 GHz / 48 cores)
Memory	128 GiB + 2 TiB (DDR5-4800 16 GiB DIMM ×8, DDR5-4800 256 GiB Persistent Memory ×8)
Nodes	150
Network	InfiniBand NDR200 HCA
MPI library	OpenMPI 4.1.6
Compiler	GCC 11.4.0
Compile option	-std=gnu++17 -O2 -fopenmp

Number of MPI Processes. The program was executed with the number of MPI processes per node varied from 1 to 48. The number of nodes was fixed at 50. Each MPI process was run using a single thread. The results are shown in Fig. 5.

As the number of MPI processes increased, the execution time decreased. When executed with 48 MPI processes per node, the program was approximately 24 times faster compared to running with 1 MPI process per node.

Number of Threads. The program was executed with various numbers of execution threads in the parts parallelized with OpenMP. It was run on 50 nodes, each of which was assigned one MPI process. The results are shown in Fig. 6. Increasing the number of threads reduced the execution time, but the speedup was more gradual compared to that obtained by increasing the number of MPI processes. Compared to running with a single thread, the execution with 48 threads resulted in a speedup of approximately 6.0 times.

MPI Process and Thread Allocation. The product of the number of MPI processes and the number of threads per node was fixed at 48, which is the number of cores per node. The program was executed while varying these values. The execution time is shown in Fig. 7. Since increasing the number of MPI processes led to a greater speedup than that obtained by increasing the number of threads, execution time decreased as the number of MPI processes increased. The

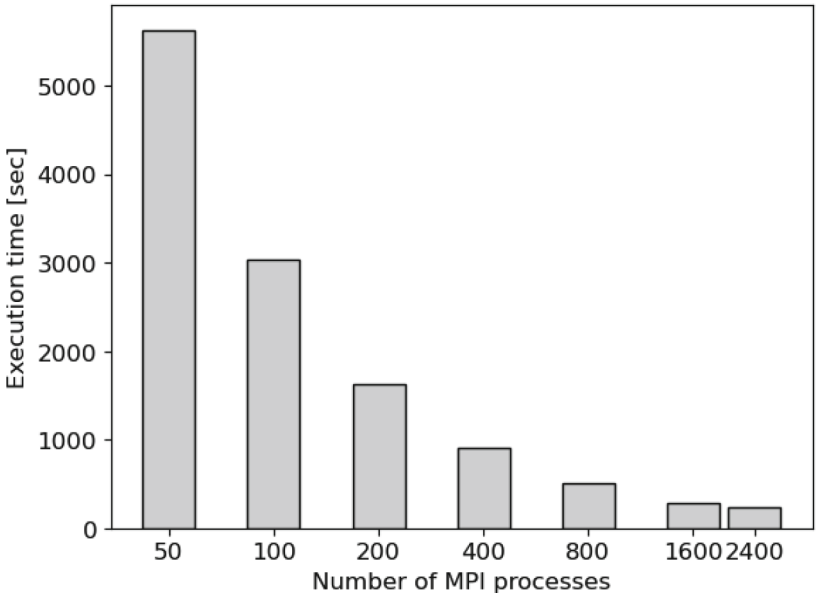

Fig. 5. Number of MPI processes vs. execution time (8 tiles, 50 nodes, 1 thread)

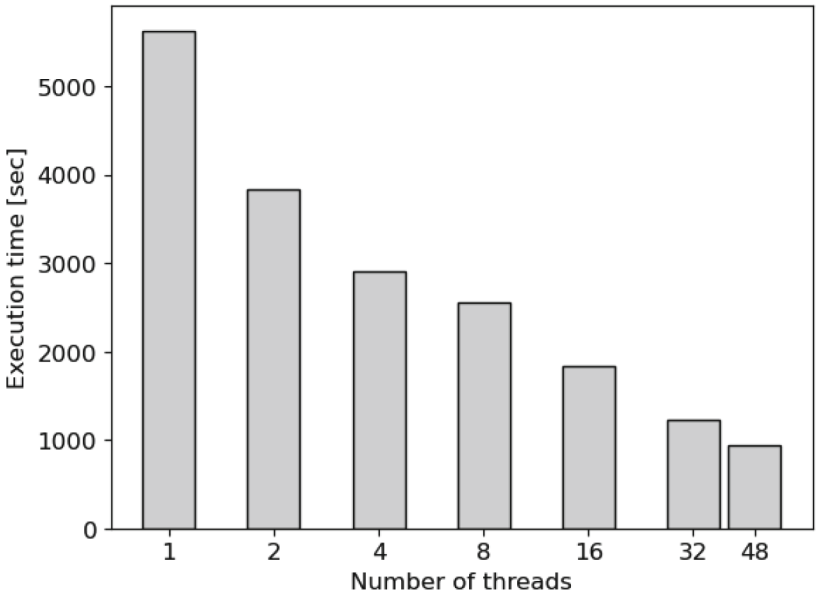

Fig. 6. Number of threads vs. execution time (8 tiles, 50 nodes)

fastest execution was achieved using Flat-MPI without OpenMP-based thread parallelization.

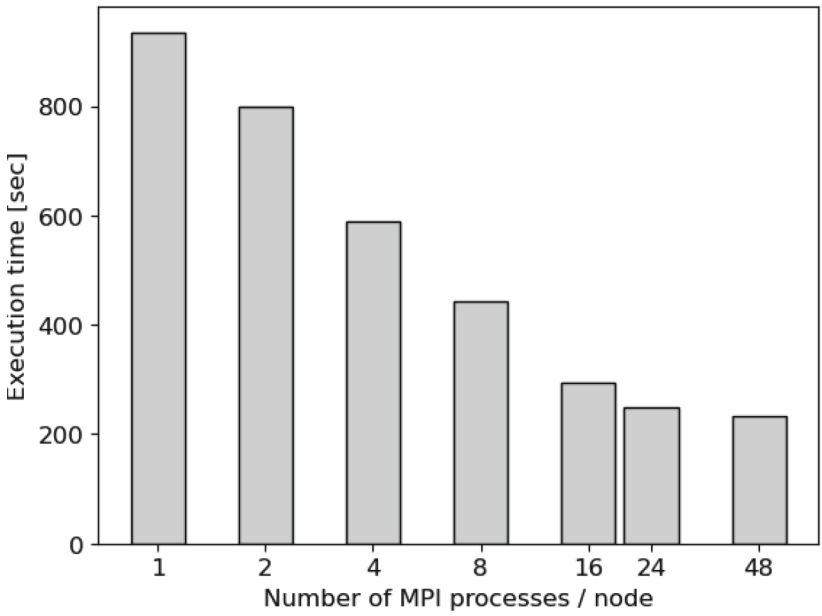

Fig. 7. Execution time with fixed product of MPI processes and threads per node set to 48 (50 nodes)

5.2 Construction of Large ZPDB

The proposed program was used to construct a ZPDB with the nine tiles shown in Fig. 8. It was executed on the Wisteria supercomputer [6], which is operated by the Information Technology Center, The University of Tokyo. The execution environment is shown in Table 2. The total number of patterns is $1\ 759\ 513\ 674\ 240 \approx 1.76 \times 10^{12}$.

The program used only MPI-based parallelization (i.e., OpenMP thread parallelization was not used). In addition, to reduce memory usage, communication was divided into multiple rounds. Specifically, during the BFS, the number of vertices whose edges are calculated at once was limited to $2^{18} = 262\ 144$ per MPI process. The node configuration was a three-dimensional mesh. The program was executed with two setups: 576 nodes with 27 600 MPI processes, and 2304 nodes with 110 400 MPI processes.

The execution time is shown in Table 3. Using 2304 nodes, the execution was completed in approximately 41 min. Since all-to-all communication was performed with more than 100 000 MPI processes, the communication cost was sig-

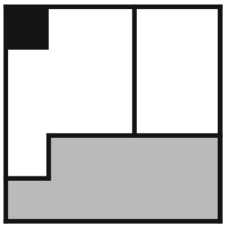

Fig. 8. Configuration of nine input pattern tiles, identical to one of Döbbelin's 9-9-6 partitions [11]

Table 2. Execution environment (Wisteria)

CPU	A64FX (2.2 GHz / 48 cores)
Memory	32 GiB
Nodes	7680
Network	Tofu Interconnect D
MPI library	Fujitsu MPI
Compiler	Fujitsu compiler
Compile option	-Kfast

nificant, accounting for more than one-third of the total execution time. Regarding communication, the maximum number of all-to-all communication rounds via MPI_Alltoall performed in one step of the level-synchronized BFS was 80 with 576 nodes and 26 with 2304 nodes.

Table 3. Execution time of nine-tile ZPDB construction program

nodes	execution time [s]	communication time [s]	communication ratio
576 ($6 \times 12 \times 8$, mesh)	5807	392	0.0675
2304 ($12 \times 12 \times 16$, mesh)	2477	973	0.393

5.3 Comparison with Existing Program

The program developed by Clausecker was executed on a single Pegasus node using the tiles shown in Figs. 4 and 8 as input. The number of threads was set to 48. When using nine tiles were used as input, the available DRAM was insufficient, so persistent memory was used to extend the memory capacity.

The execution time for the eight-tile case was 11 min and 10 s. The nine-tile case required 3 h and 43 min and approximately 1.7 TB of memory.

Compared to the program implemented in this study, the existing shared-memory program requires fewer computational resources. In distributed-memory programs, additional time is required for communication and related processing. Moreover, since each MPI process handles only a subset of vertices, the workload in each BFS step is not evenly distributed. These factors likely contributed to the difference in execution efficiency between the two approaches.

6 Conclusion

A ZPDB construction program for distributed memory environments was developed. MPI and OpenMP were used for parallelization. With the parallelization of the level-synchronized BFS, increasing the number of MPI processes and threads reduced the execution time of the program. Increasing the number of MPI processes tended to result in a greater reduction in execution time than that obtained by increasing the number of threads. With a few thousand MPI processes, all-to-all communication was sufficiently fast, making Flat-MPI the most efficient choice. However, in an example where a nine-tile ZPDB for the 24-puzzle was constructed with over 100 000 MPI processes, the communication time accounted for more than one-third of the total execution time. To further increase the number of MPI processes, techniques for reducing communication costs are necessary. The program developed in this study is expected to be useful for constructing large ZPDBs that would be difficult to compute on a single machine.

Acknowledgments. This research was conducted using the computational resources of Pegasus, provided by the Multidisciplinary Cooperative Research Program at the Center for Computational Sciences, University of Tsukuba, and Wisteria, provided by the Supercomputing Division, Information Technology Center, The University of Tokyo.

References

1. MPI. https://www.mpi-forum.org. Accessed 14 Mar 2025
2. OpenMP. https://www.openmp.org. Accessed 14 Mar 2025
3. clausecker/24puzzle: Solve 24 puzzles using zero-aware pattern databases https://github.com/clausecker/24puzzle. Accessed 14 Mar 2025
4. iwiwi/parallel-radix-sort: An implementation of optimized parallel radix sort https://github.com/iwiwi/parallel-radix-sort. Accessed 14 Mar 2025
5. Center for computational sciences, University of Tsukuba: Supercomputer Pegasus. https://www.ccs.tsukuba.ac.jp/supercomputer. Accessed 14 Mar 2025
6. Wisteria/BDEC-01 Supercomputer System | Supercomputing Division, Information Technology Center, The University of Tokyo. https://www.cc.u-tokyo.ac.jp/supercomputer/wisteria/service/. Accessed 14 Mar 2025
7. Bader, D.A., Madduri, K.: Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In: International Conference on Parallel Processing, pp. 523–530 (2006). https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1690657

8. Clausecker, R., Reinefeld, A.: Zero-Aware Pattern Databases with 1-Bit Compression for sliding tile puzzles. In: 12th International Symposium on Combinatorial Search, vol. 10, no. 1, pp. 35–43 (2019)
9. Cooperman, G., Finkelstein, L.: New methods for using Cayley graphs in interconnection networks. Discret. Appl. Math. **37**(38), 95–118 (1992)
10. Culberson, J.C., Schaeffer, J.,: Searching with pattern databases, In: Conference of the Canadian Society for Computational Studies of Intelligence, pp. 402–416 (1996)
11. Döbbelin, R., Schütt, T., Reinefeld, A.: Building large compressed PDBs for the sliding tile puzzle. In: Workshop on Computer Games 2013 Proceedings, pp. 16–27 (2014)
12. Felner, A., Korf, R.E., Hanan, S.: Additive pattern database heuristics. J. Artif. Intell. Res. **22**, 279–318 (2004)
13. Korf, R.E.: Depth-first iterative-deepening: an optimal admissible tree search. Artif. Intell. **27**, 97–109 (1985)
14. Ratner, D., Warmuth, M.: The $(n^2 - 1)$-puzzle and related relocation problems. J. Symb. Comput. **10**(2), 111–137 (1990)