Semestral Project NI-PDP 2023/2024:

Parallel algorithm for solving the problem of exchanging knights on a chessboard

Shchapaniak Andrei

FIT CVUT

April 17, 2024

1 Definition of the problem

Input data

- **m,n** - natural numbers representing the size of the chessboard $S[1..m, 1..n]$, $m, n \geq 3$, $m, n \leq 20$
- **k** - a positive number representing the number of white/black knights, $1 \leq k < m \times n/2$
- **W** - a rectangular area on the chessboard, given by the coordinates of the ends of the diagonal, $k = |W|$
- **B** - a rectangular area on the chessboard, given by the coordinates of the ends of the diagonal, $k = |B|$
- the B and W areas are mutually disjoint

Rules and objective of the game

- **The initial state** is given by the positions of the white knights (located in area W) and black knights (located in area B).
- Each knight moves on the chessboard according to chess rules (move in the shape of an L to a free square).
- **The goal of the game** is to alternate moves of white and black knights to transition from the initial state to the target state, in which white knights occupy area B and black knights occupy area W.
- Since $k = |B| = |W|$, the areas B and W are always fully occupied at the beginning.
- The alternation of the knights' move colors stops when all knights of one color are already in their target area, and only knights of the opposite color make moves.

Task

Find the shortest sequence of moves leading from the initial state to the final state.

Output of the algorithm

- A number indicating the minimum number of moves after completing the exchange and a printout of the sequence of moves leading from the initial state to the target state.
- Print each move as the state of the entire chessboard after it is made.

2 Sequential algorithm

The sequential algorithm is implemented using the Branch and Bound Depth-First Search (**B&B DFS**) approach. The implementation can be divided into three primary components:

1. **Termination Check:** The recursion terminates if it reaches the maximum depth or if, based on heuristic estimates, the algorithm determines that it cannot find a solution better than the current best. This heuristic cutoff is mathematically expressed as follows:

$$\text{if current\_moves}(S) + d(S') + 1 \geq \text{current\_upper\_bound}, \tag{1}$$

   where `current_moves(`S`)` is the count of moves taken to reach state S, $d(S')$ is the heuristic distance to the goal from the new state S', and `current_upper_bound` represents the move count of the best solution found till now.

2. **Move Generation and Prioritization:** The algorithm computes all legal moves for the knight currently under consideration. These moves are then prioritized according to a heuristic that estimates the distance to the goal state, helping to guide the search toward promising areas of the search space.

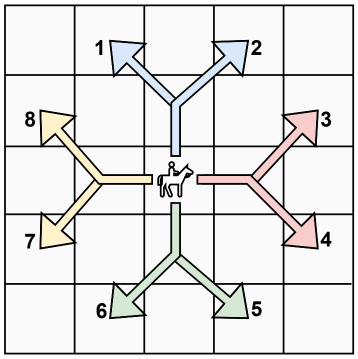


Figure 1: The 8 possible moves of a knight on a chessboard.

3. **Main Evaluation Loop:** Each candidate move is evaluated, and the algorithm recursively invokes itself with an incremented depth, thereby expanding the search tree.

The efficiency of the algorithm is improved through the use of bounding techniques:

- **Upper Bound:** The initial upper bound is established based on the maximum depth limit at the start of the program. As the search progresses and better solutions are found, this bound is dynamically reduced to reflect the depth at which these solutions are discovered.

- **Lower Bound:** Prior to each move, the algorithm calculates a lower bound to improve its efficiency. This bound is the aggregate of the minimum distances from the positions of the knights not yet in the final state to their nearest target squares in the goal configuration. This is formulated as:

$$d(X) = \sum_{\text{knight } i \notin \text{final state}} \text{min\_distance}(\text{position}_i, \text{final state}) \tag{2}$$

3 Parallel algorithm

3.1 Task parallelism using OpenMP

The task parallelism algorithm implemented with OpenMP has notable similarities to the sequential algorithm, but it is designed to leverage the power of parallel processing. It partitions the workload by dispersing discrete tasks across multiple threads. This partitioning is executed when the depth of the recursive function is at or below 35% of the maximum allowed recursion depth. Beyond this threshold, the algorithm changes the approach to the sequential one, completing the problem-solving process without further parallelization. This threshold-based switch from parallel to sequential processing is pivotal. It ensures that the overhead associated with parallel task management is justified by the complexity of the computation.

A critical aspect of this algorithm is how it handles the comparison and updating of the globally best solution found. This comparison is managed within a critical section `#pragma omp critical` to guarantee that any updates to the best solution are safely conducted, despite the concurrent nature of the task execution.

3.2 Data parallelism using OpenMP

The data parallelism algorithm utilizing OpenMP is based on the pre-generation of initial states using the Breadth-First Search (**BFS**) algorithm. It generates unique states that are gradually divided for processing among multiple threads through `#pragma omp parallel for schedule(dynamic)` approach.

Additionally, for this task, a `State` structure has been introduced to facilitate the passing of necessary data into the recursive function conveniently. This structure encapsulates the chessboard configuration, a record of the best moves discovered, and the current depth of the recursion.

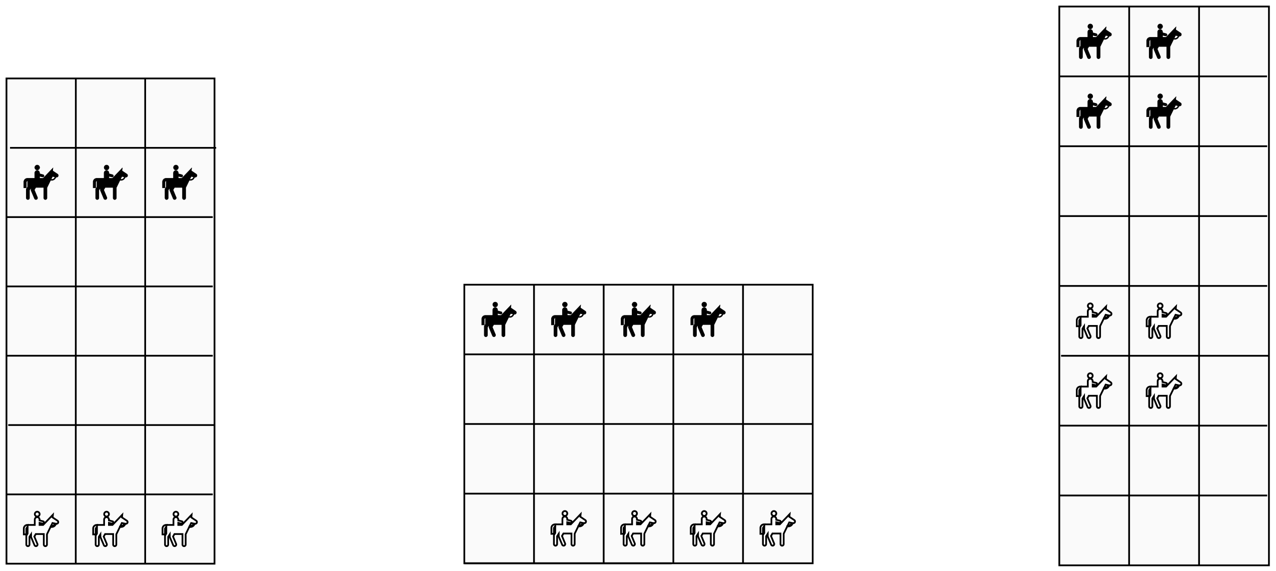
4 Task parallelism using MPI

The parallel algorithm using Message Passing Interface (**MPI**) extends the parallel solution approach to work effectively on clusters with distributed memory. The algorithm employs several processes, including one master process and several slave processes. The master process pre-generates states using the BFS algorithm which are then distributed to the slave processes. Upon receiving a state, a slave process solves it using OpenMP's task parallelism, and after completing sends the best cost and corresponding moves back to the master process. The master process incrementally updates the best value and eventually outputs the optimal solution.

In addition to the primary communication where the master distributes tasks to slaves and receives solutions, to accelerate the algorithm, when a slave finds a solution, it broadcasts the new maximum depth to the other slaves. This is facilitated using asynchronous message passing functions `MPI_Isend()`, `MPI_Irecv()` and `MPI_Test()` allowing slaves to dynamically adjust their search depth based on the latest information. This approach can significantly reduce unnecessary computations and lead to faster convergence on the optimal solution.

5 Measured results and evaluation

5.1 Instances



(a) 7_3_3.txt (b) 4_5_4.txt (c) 8_3_4.txt

Figure 2: Three instances for testing.

5.2 Future formulas

- $SU^K(n)$ defines the upper bound on the time complexity for solving a given problem K. It represents the worst-case time complexity of the most efficient known sequential algorithm for problem K.
- $T(n, p)$ denotes the actual time elapsed from the start of the parallel computation to the moment when the last (and thus the slowest) processor has completed its computation:
- $S(n, p)$ is the speedup achieved by using p processors to solve the problem of size n compared to the sequential execution. It is computed as the ratio of the upper bound on the sequential execution time to the parallel execution time:

$$S(n,p) = \frac{SU^K(n)}{T(n,p)} \tag{3}$$

- $C(n, p)$ is the total computational cost of a parallel algorithm when executed with p processors. This cost accounts for the cumulative effort of all processors and can be expressed as the product of the number of processors and the time taken for the computation to complete:

$$C(n,p) = p \times T(n,p) \tag{4}$$

- $E(n,p)$ is the ratio of the upper bound on time complexity for solving a problem using the best-known sequential algorithm to the parallel cost, and its computational cost:

$$E(n,p) = \frac{SU(n)}{C(n,p)} \tag{5}$$

It reflects how well the parallel algorithm uses the computational resources, ideally approaching 1, which would indicate perfect scaling with the addition of more processors.

5.3 Results

Sequential algorithm

Instance	Time [s]
7_3_3.txt	177.890
4_5_4.txt	151.024
8_3_4.txt	108.919

Task parallelism - OpenMP

Instance	# Threads	Time [s]	Speed-up	Computational Cost	Parallel Efficiency
7_3_3.txt	2	32.540	5.467	65.080	2.733
	4	37.815	4.704	151.260	1.176
	8	10.967	16.220	87.736	2.021
	16	10.119	17.579	161.904	1.098
	32	10.369	17.156	331.808	0.536
4_5_4.txt	2	12.885	11.711	25.770	5.862
	4	7.141	21.149	28.564	5.287
	8	4.270	35.360	34.160	4.421
	16	4.016	37.606	64.256	2.351
	32	3.963	38.109	126.816	1.191
8_3_4.txt	2	40.477	2.690	80.954	1.345
	4	14.367	7.581	57.468	1.895
	8	8.618	12.639	68.944	1.579
	16	8.088	13.466	129.408	0.841
	32	8.099	13.448	259.168	0.421

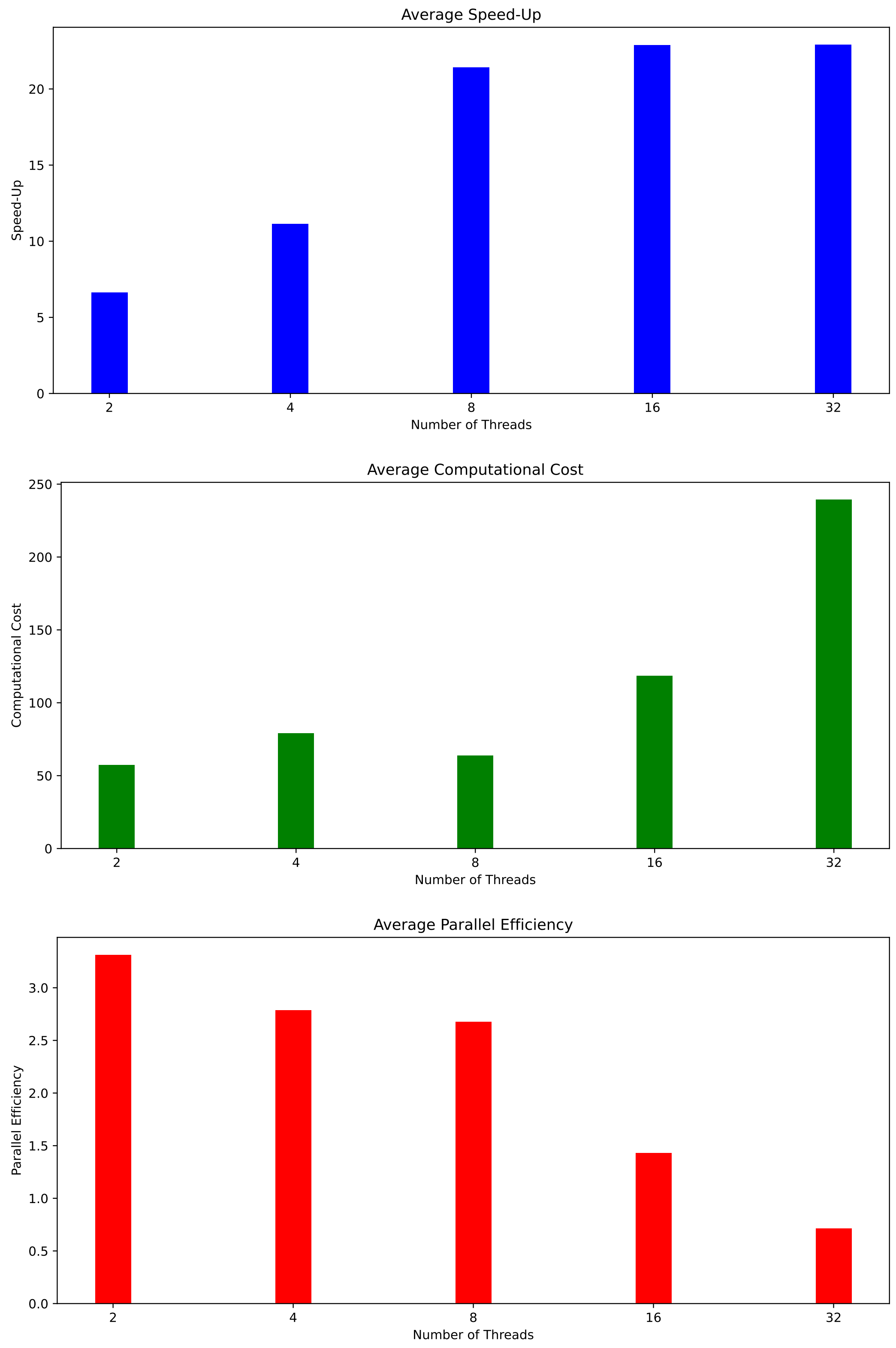


Figure 3: Average results for OpenMP solution.

- `Speed-Up`: the graph shows increased speed-up with the number of threads, peaking at 8 threads. However, there is a plateau from 8 to 32 threads, indicating no significant gains in speed-up beyond 8 threads.
- `Computational Cost`: the graph indicates that the computational cost is lower with 2, 8, and 16 threads compared to 4 and 32 threads. Notably, the cost at 32 threads spikes significantly, suggesting inefficiencies at higher thread counts.
- `Parallel Efficiency`: the graph shows a decline in parallel efficiency as the number of threads increases. Efficiency is highest at 2 threads and drops sharply after 8 threads, becoming very low at 32 threads.

The best number of threads for this OpenMP algorithm appears to be **8**, as it offers the highest speed-up without the efficiency losses seen at higher thread counts. Increasing the number of threads beyond 8 does not seem to benefit the speed-up significantly and can result in higher computational costs.

If the number of threads was to be increased further, the trend suggests that there would likely be a continued increase in computational cost and a decrease in parallel efficiency, without a substantial improvement in speed-up.

Parallel algorithm using MPI - 1 master + 3 slaves

Instance	# Threads	Time [s]	Speed-up	Computational Cost	Parallel Efficiency
7_3_3.txt	3×2	25.675	6.929	154.050	1.155
	3×4	24.791	7.175	297.492	0.598
	3×8	12.901	13.788	309.624	0.575
	3×16	9.671	18.394	464.208	0.383
	3×32	9.029	19.702	866.784	0.205
4_5_4.txt	3×2	12.630	11.957	75.780	1.993
	3×4	12.074	12.508	144.888	1.420
	3×8	6.901	21.884	165.624	0.912
	3×16	5.200	29.043	249.600	0.605
	3×32	5.292	28.538	508.032	0.297
8_3_4.txt	3×2	28.213	3.861	169.278	0.643
	3×4	20.121	5.413	241.452	0.451
	3×8	8.902	12.235	213.648	0.510
	3×16	6.298	17.294	302.304	0.360
	3×32	6.091	17.881	584.736	0.186

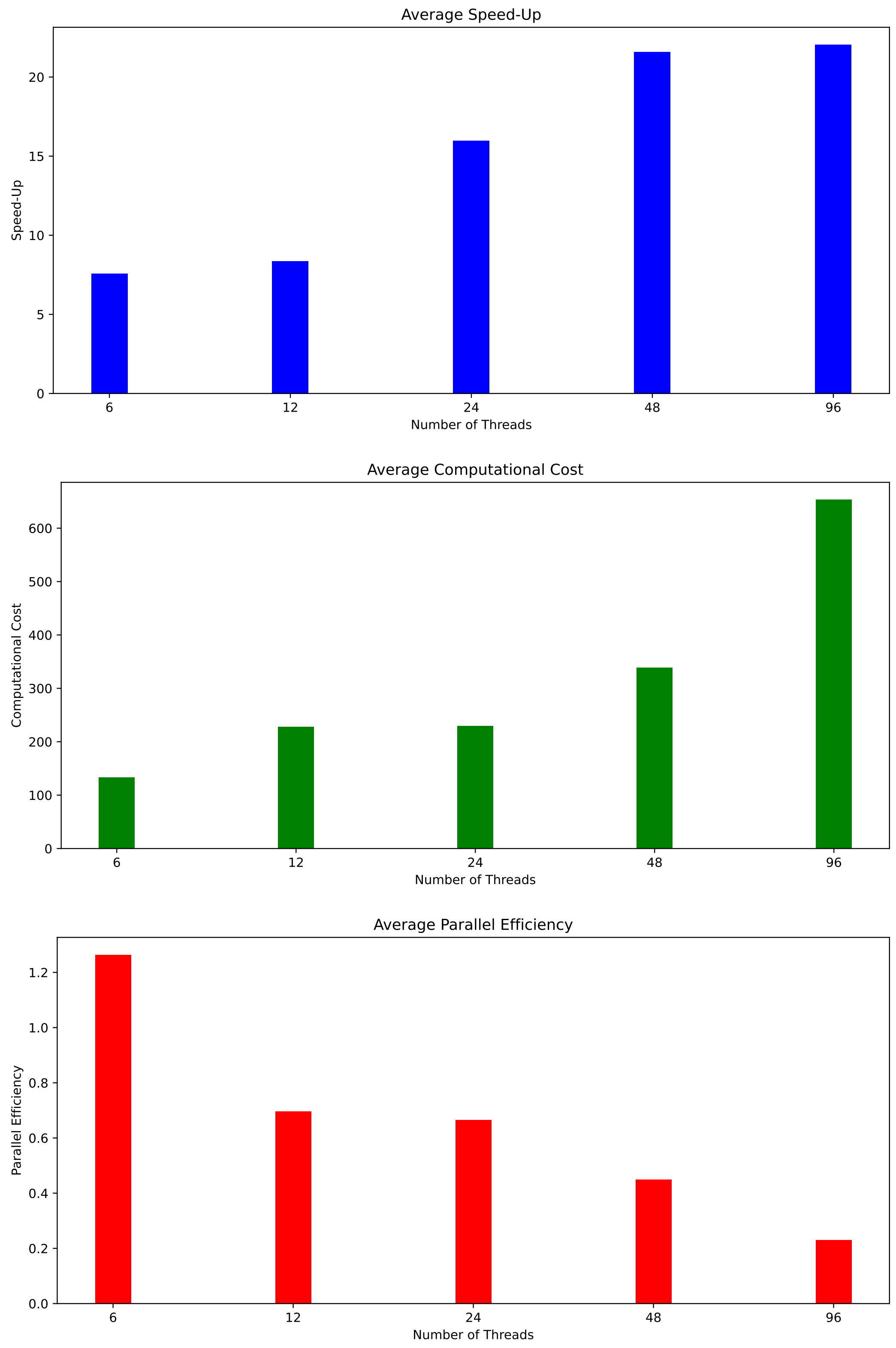


Figure 4: Average results for MPI solution.

- `Speed-Up`: There's a progressive increase in speed-up with more threads, peaking at 24 threads, but then it doesn't significantly increase.
- `Computational Cost`: It is lowest at 6 threads, increases at 12, decreases again at 24, but escalates sharply at 48 and 96 threads, suggesting inefficiency at higher thread counts.
- `Parallel Efficiency`: Parallel efficiency gradually decreases from 12 threads, indicating diminishing returns with more threads.

Considering speed-up, computational cost, and parallel efficiency, the optimal number of threads for this MPI algorithm is likely around **24** (**8** threads for each slave node). This provides a balance between the benefits of speed-up and the drawbacks of increased computational cost and decreased efficiency.

If the number of threads increases beyond 24, it's realistic to expect further declines in parallel efficiency and increases in computational cost without substantial improvements in speed-up.

6 Conclusion

In terms of performance, the observed trends in speed-up from increasing thread counts initially aligned with the expected benefits of parallel computing. However, the subsequent plateau in speed-up and the dip in parallel efficiency with further added threads highlighted the complexities and overhead involved in parallel processing. The empirical outcomes resonated with established theoretical concepts, notably Amdahl's Law.

To enhance efficiency, a more optimized approach to algorithm design could be beneficial. This could involve refining the way threads are created, synchronized, and how tasks are distributed among them. Application profiling might be key in pinpointing specific bottlenecks, such as significant thread contention or inefficient memory access patterns, and addressing these could lead to improved performance outcomes.

Working on this project presented a unique and interesting challenge in practical terms, especially since it was my first experience with parallel algorithms. The level of complexity matched my expectations, falling comfortably into the medium range. Through the engaging task of manipulating a chessboard, the core principles of the algorithm's operation were clearly demonstrated and easily visualized.