Report: Direct Meet-in-the-Middle Attack

Matheus FERNANDES MORENO (21400700)
Daniel MACHADO CARNEIRO FALLER (21400117)

January 2025

1 Introduction

The Direct Meet-in-the-Middle (MitM) Attack is a well-established method in cryptanalysis designed to find a "golden collision", where two cryptographic functions f and g intersect at a common output under specific constraints. This project aims to implement and optimize a distributed version of this algorithm. We leverage parallel programming paradigms, namely MPI, to address performance bottlenecks associated with the sequential implementation provided. Our primary goal was to efficiently find solutions for increasing block sizes n, prioritizing higher values of n over execution speed as stressed by the project's assignment.

2 Implementation

The original sequential code fills out a dictionary from the outputs of $f(x)$ and probes it using $g(y)$. It uses a hashing function to do so, which ideally *uniformly allocates* entries across the dictionary's memory space. For large values of n, this process becomes increasingly memory- and compute-intensive, creating significant bottlenecks in both storage and processing time. Parallelization using MPI addresses these challenges by distributing the workload across multiple processes, thus enhancing scalability and efficiency.

2.1 Distributed Dictionary

The main strategy of the parallel algorithm is to use a *distributed dictionary*. Each process independently manages a *shard* of the global dictionary by doing insertions and look-ups on it. The sharding is straightforward: if the global dictionary has size N and we have P processes, the process with rank p will be responsible for the shard $[p \times N/P : (p+1) \times N/P]$. With the assumption that the hashing function results in an uniform distribution of entries across the dictionary, we should expect a balanced workload between processes.

Both the fill and probe computations are distributed between cores, i.e. each process computes only N/P instances of $f(x)$ and $g(y)$. This strategy, however, imposes a challenge: what to do when a rank computes an entry that does not belong to its shard? To address this problem, cores exchange computed entries using *buffers*.

2.2 Communication Buffers

To improve data handling and inter-process communication overhead, each process is equipped with a buffer dedicated to storing entries/look-ups associated with other ranks. These buffers batch accesses, significantly reducing the overhead caused by frequent communication between processes due to larger messages being transmitted.

2.2.1 Buffer Structure

Each process maintains a buffer for every rank, even itself. The buffer size is relative to the size of a shard: if the local dictionary can store E entries, then a buffer can store $E \times$ `BUFFER_RELATIVE_SIZE` entries. In our experiments, we observed that buffers with 0.1% of shard size presented a good trade-off between time and memory.

The buffers are implemented as an array divided into evenly spaced sections. An entry on the buffer consists of a key/value pair—for instance, $(f(x), x)$ in the fill phase—and thus occupies two memory slots. Additionally, since buffers have a limited size, we must keep track of their current capacity using counters; when a buffer is full, is necessary to start a buffer exchange and flush its content.

2.2.2 Buffer Exchange

Whenever the function `add_to_buffer()` returns `1`, it indicates that a buffer for a specific rank has reached its full capacity. This event triggers the `exchange_buffers()` operation, which distributes all buffered keys and values to their respective owners.

The `MPI_Alltoall()` collective operation is used twice in `exchange_buffers()`, first to exchange the number of elements in each buffer and then to exchange entries. A shortcoming of this approach is that one buffer being full does not implicate in all buffers being full, which means that processes can (and will) exchange garbage. However, if we again consider the assumption that entries are evenly distributed by the hashing function, we can also assume buffer occupancy will be balanced. By this logic, we deemed an implementation using `MPI_Alltoallv()` to be too complex and unnecessary. A metric called `buffer_occupancy` was defined to confirm our hypothesis that all buffers are relatively full when they are exchanged.

2.2.3 Batch Fill/Probe

The buffer management system operates within the two main steps of the program: filling the shards and probing them, both of which leverage cyclic load balancing to once again preserve the assumption of uniform entry distribution.

For instance, in the filling phase, each rank calculates the number of entries it must evaluate, iterates through them with a stride that is proportional to the number of processes, and adds results to their associated buffers. If a buffer reaches full capacity, the `exchange_buffers()` function is triggered, followed by a `batch_insert()` operation.

Both steps contain an "epilogue" that guarantees that every process will continue calling `exchange_buffers()` until all cores have finished filling/probing, thus avoiding any deadlocks that `MPI_Alltoall()` may cause.

2.3 Compression Factor

In the sequential code, the memory space required to store the dictionary increases exponentially with n. Our implementation contains a parameter called *compression factor* C to tackle this problem. Concretely, it reduces the size of the global dictionary by slicing it into 2^C parts and filling and probing each part "sequentially" (i.e. processes work on a single slice at once). Thus, given the total number of values to assess X, the effective number of entries for each *round* is calculated as $X_{\text{per round}} = X/2^C$. Round slices are also computed using a cyclic load-balancing strategy. The algorithm can automatically calculate the C if the user provides the total available memory with the `--mem` flag.

Note that the compression factor comes with a time-space trade-off: while we aim to divide the X keys across multiple rounds, each round still requires comparisons against the entire set of Y keys. Consequently, the Y keys are processed multiple times, which adversely affects the speed efficiency of the algorithm.

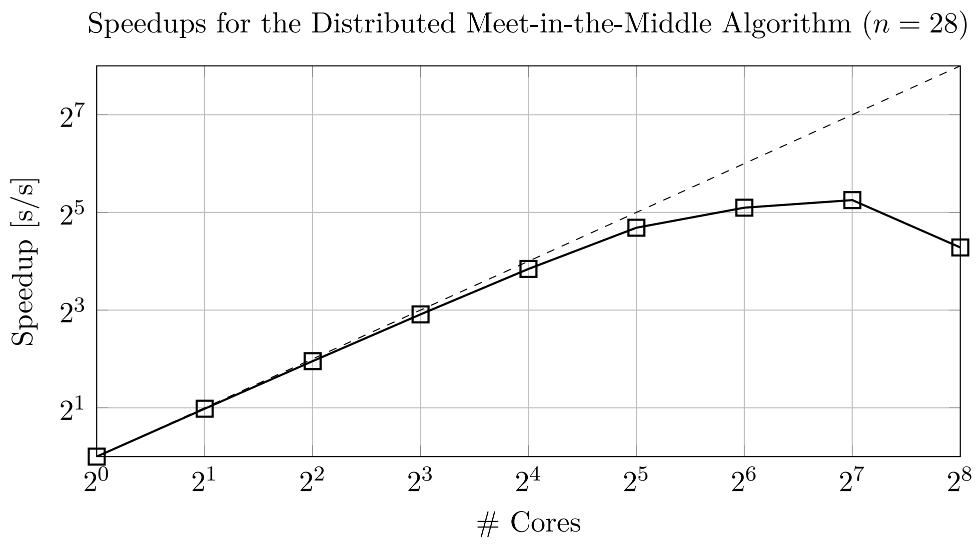
2.4 Early Exit

An early exit mechanism was implemented in the code to reduce runtime, particularly for larger values of n. By terminating the probing phase as soon as a solution was found, the algorithm avoids unnecessary computations in later iterations. This optimization must be configured at compile time by setting the `EARLY_EXIT` macro.

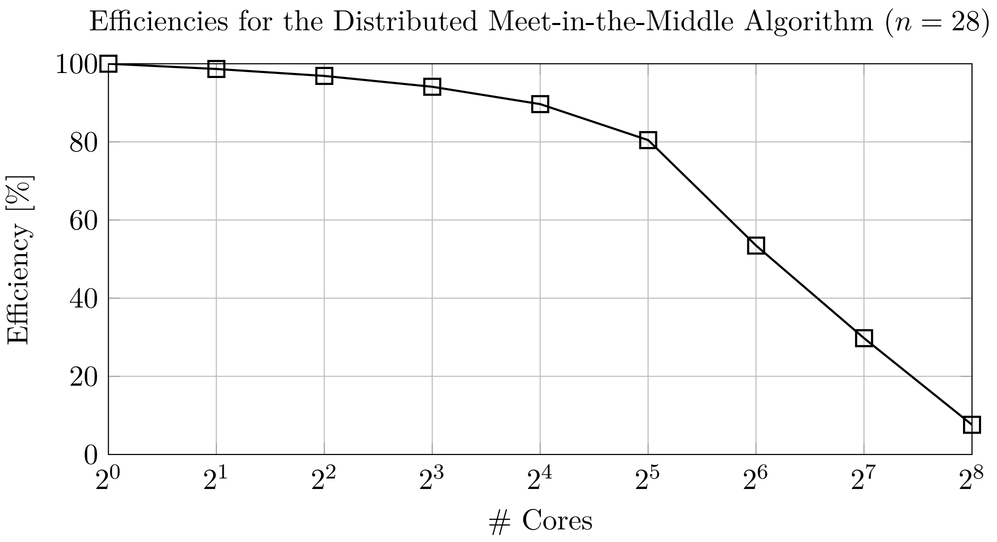
3 Performance Analysis

The following experiments were conducted at the Rennes site of Grid'5000 by running the script `performance_evaluation.sh` on non-interactive mode. More specifically, nodes `paradoxe-{4, 5, 6, 8, 9}` (of 52 cores each) were provisioned. Processes were *not* equally distributed between nodes and early exiting was disabled.

3.1 Speedup and Efficiency



(a) Observed speedup values. The dashed line represents a linear speedup.



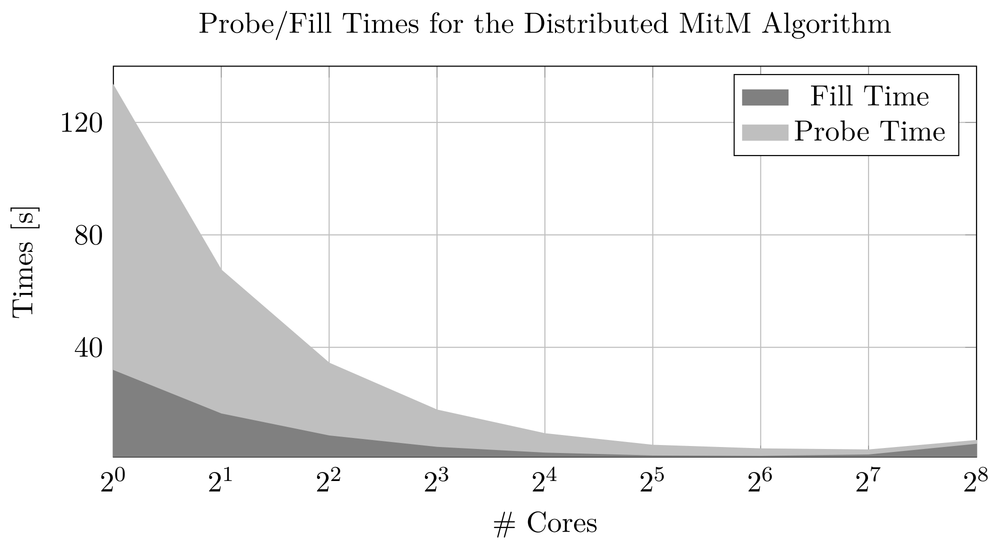
(b) Observed efficiency values.

Figure 1: Speedups and efficiencies for the distributed MitM algorithm for $n = 28$.

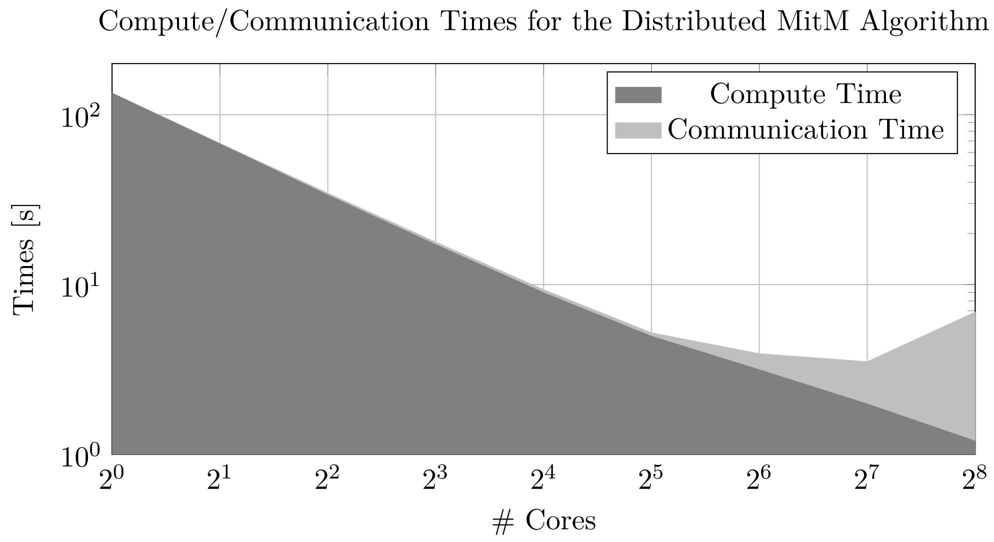
The performance of the algorithm as a distributed program can be analyzed by evaluating its *speedup* and *efficiency*, as shown in Figure 1.

The observed speedup closely follows the ideal behavior up to 32 cores. Beyond this point, the speedup begins to deviate from the linear trend due to the overhead of inter-node communication. Despite this, the algorithm maintains a significant performance gain with increasing core counts, achieving a maximum speedup of approximately 34 with 128 cores. This indicates that the algorithm effectively utilizes additional cores, although diminishing returns become apparent at higher core counts.

Efficiency remains near 100% for up to 8 cores, signifying that the algorithm is highly scalable with minimal overhead in this range. However, as the number of processes increases, efficiency starts to decline, dropping below 50% at 128 cores. As we will see in the following subsection, this degradation is primarily attributed to the increasing cost of communication.



(a) Observed fill and probe times.



(b) Observed compute and communication times. A log scale for the y-axis better illustrates the exponential decrease of compute time and minimal impact of communication time for $n < 64$.

Figure 2: Times for the distributed MitM algorithm for various number of cores, $n = 28$ and no compression. We can see that the local probing is the bottleneck of the algorithm when a small number of cores are concentrated on the minimum amount of machines.

3.2 Bottlenecks

Now, we want to identify the bottlenecks of the algorithm. Figure 2 shows times for various numbers of cores and $n = 28$.

In Figure 2a, we can see that as the number of cores increases, both fill and probe times decrease proportionally, demonstrating effective parallelization and workload distribution. Moreover, while probe time generally dominates the fill time in most configurations, this relationship begins to shift after 128 cores; beyond this point, the fill time surpasses the probe time, highlighting a potential bottleneck due to increased communication overhead or load imbalance during dictionary construction.

Figure 2b highlights the breakdown between compute and communication times. A log scale is used for the y-axis to emphasize the impact of communication costs. We can observe that communication time remains minimal until 64 cores, which is consistent with the fact that beyond this point inter-node communication must take place. After this point, communication overhead becomes the algorithm's bottleneck. Conversely, for a small/moderate amount of cores concentrated on the minimum amount of nodes, the compute time (more specifically, the probing phase) is the bottleneck.

3.3 Compression Level

Figure 3 presents the algorithm's execution times under varying compression levels C. We can observe a clear exponential growth, a direct result from the division of the search space into smaller segments. In other words, the cumulative effect of processing the same Y-keys repeatedly leads to a substantial rise in computational overhead, overshadowing the initial efficiency gains achieved by reducing the search space size per process.

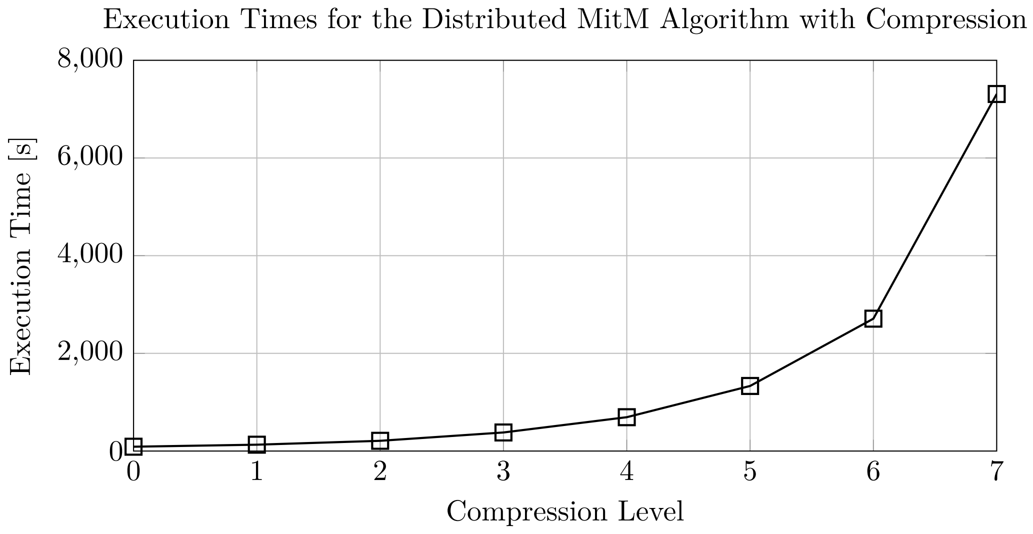


Figure 3: Observed execution times for the distributed MitM algorithm for $n = 33$, 128 cores, and different levels of compression.

4 Golden Collisions

Table 1 presents the golden collisions discovered for the username `matheus.daniel` alongside execution statistics, including the number of cores, compression levels, and execution times. The experiments were conducted for $n \geq 25$, as we noticed that the sequential implementation began to exhibit a substantial execution time starting from this value.

All collisions (except for $n = 42$) were found on the same job execution at the `paradoxe` cluster, with 16 nodes of 52 cores each, using the `collision_finder.sh` script on non-interactive mode. One major difference between this script and the one used for the performance analysis is that, in this case, cores were equally distributed between nodes to optimize memory distribution. Moreover, the early exit optimization was enabled to find collisions as quickly as possible.

Table 1: Golden collisions found for the `matheus.daniel` username.

n	**Solution**	**# Cores**	**Compression**	**Time**
25	`(ea2686, 1ee6b15)`	128	-	00:00:30
26	`(27faf5e, 18cc538)`	128	-	00:00:25
27	`(5bfe71b, 2245cd7)`	128	-	00:00:29
28	`(f3662fd, d62c4dc)`	128	-	00:00:30
29	`(18dd3985, 1b8de076)`	128	-	00:00:32
30	`(2b486c95, 2a16ec38)`	128	-	00:00:35
31	`(5c6edcc0, 4a96788f)`	128	-	00:00:33
32	`(f6b6aa8f, 62c35b6)`	128	-	00:00:42
33	`(199887836, 1739ea3c9)`	128	-	00:01:08
34	`(394c28f82, e24dc73d)`	128	-	00:01:21
35	`(10526d7d6, 668757cda)`	128	-	00:03:31
36	`(3b643d6cd, b98ff8488)`	128	-	00:06:33
37	`(1d6cc2b682, 22e0a9a2f)`	128	-	00:05:56
38	`(272abfbb80, 1f5ac1df01)`	128	-	00:21:10
39	`(475395a5a1, 3c84e20c9c)`	128	1	01:08:08
40	`(94f05a1c13, 59ddbb310f)`	128	2	03:58:00
41	`(101ffa5a298, 22770e2f75)`	128	3	00:16:50
42	`(1086fc94d9f, 2ad2d125aee)`	256	4	33:10:49

The results demonstrate an increasing execution time as n grows, which is expected due to the exponential growth of the search space. For $n = 42$, even with 256 cores, the runtime extended to over 33 hours. However, in some cases, we can see that the early exit optimization strategy greatly improved performance; for instance, we found a collision for $n = 41$ in only 17 minutes.

5 Difficulties and Challenges

The development of the algorithm presented several challenges, primarily due to the fine-tuning of multiple parameters. For example, the `BUFFER_RELATIVE_SIZE` macro, which determines the size of the local buffers for each process, required careful balancing: setting it too low led to significant communication overhead, while setting it too high caused local memory to overflow.

Similarly, the compression factor C greatly reduced memory usage, enabling the algorithm to handle larger values of n. However, this came at the cost of increased computational overhead, as Y-keys had to be processed repeatedly in each round. After implementing the automatic computation of C, it was a matter of finding the optimal site on Grid'5000 to conduct our experiments. We quickly realized that the `paradoxe` cluster on Rennes resulted in the best memory per core ratio when several nodes were allotted.

Combining these parameters with different numbers of cores and input sizes considerably expanded the testing space, causing the evaluation of all potential configurations to be very time-consuming and resource-intensive. Nevertheless, the flexibility of the implementation showed that it could be adapted to various environments and requirements.

6 Conclusion

We have presented a detailed analysis of a distributed implementation of the Meet-in-the-Middle algorithm. The results demonstrated the effectiveness of parallelization and compression techniques in addressing the challenges posed by memory constraints. The algorithm presented excellent scalability up to a moderate number of cores, with near-linear speedup for ≤ 32 cores. However, the analysis also highlighted diminishing returns and efficiency drops at higher core counts due to communication overhead.

The compression method proved to be a valuable tool for handling memory limitations, allowing the algorithm to operate in resource-limited environments. However, the trade-off between memory usage and execution time ultimately became apparent, particularly at higher compression levels, causing an exponential growth in execution time to be observed.

Overall, the algorithm is suitable for distributed systems with various system specifications. However, achieving optimal performance requires careful consideration of hardware constraints, input size, and the tuning of various parameters.