Direct Meet-in-the-Middle Attack Project

Kareem ABI KAEDBEY Victor ZHOU

1 Implementation strategy

Our primary objective was to find the golden collision for the highest possible value of N. We developed a distributed hybrid architecture combining MPI and OpenMP. Our implementation strategy focuses on two critical performance factors: minimizing the memory footprint per node and controlling network congestion.

1.1 Algorithmic Decomposition and Sharding

The Meet-in-the-Middle (MITM) algorithm splits the search space into two independent parts of size 2^n:

- Phase 1 Distributed Storage (Fill): Storing the entire dictionary on a single machine is impossible for high values of N due to RAM limitations. We implemented a sharding strategy where the global dictionary is distributed across all available MPI nodes. Each generated pair $(value, key)$ is assigned to a specific owner rank using the formula:

$$\text{Owner\_Rank} = \text{Hash}(value) \pmod{\text{Number\_Procs}}$$

- Phase 2 (Probe) : The second list is never stored globally. Instead, we adopted a streaming approach. Elements are generated on-the-fly and sent to their respective owner nodes (using the same hash formula as Phase 1). Upon reception, the owner node checks the incoming element against its local hash table. If a match is found, the collision is reconstructed; otherwise, the data is immediately discarded to conserve memory.

1.2 Hybrid Parallelism Model

To maximize the utilization of the Grid'5000 hardware, we leveraged two levels of parallelism:

- **Inter-node (MPI):** MPI handles the distribution of the search space and the synchronization between nodes. Each MPI process acts as an independent worker managing a fraction of the global hash table.
- **Intra-node (OpenMP):** Within each node, we used OpenMP to parallelize the computational workload. The filling of send-buffers and the insertion/lookup in the local hash table are multi-threaded. This allows us to fully exploit the 18 physical cores of the *Gros* nodes.

1.3 Network Traffic Management (Chunking Strategy)

During our tests, we identified network saturation as the primary bottleneck. The massive global data exchange via `MPI_Alltoall` exceeded the capacity of the network buffers, leading to job failures (error: *Transport retry count exceeded*).

To overcome this physical limitation, we implemented a **Chunking Mechanism**:

1. The total generation space is divided into `NUM_CHUNKS` subsets.
2. The program processes these chunks sequentially.
3. In each iteration, only a fraction of the total data (1/`NUM_CHUNKS`) is exchanged.

This approach transforms a massive traffic spike into a continuous, stable flow. We dynamically adjusted `NUM_CHUNKS` (e.g., 512 for $N = 38$ on 40 nodes, 256 for $N = 35$ on 20 nodes, and maybe more for larger sets but $N = 38$ was the highest value tested) to keep packet sizes within the network's optimal range (< 200 MB per process pair).

2 Presentation and Interpretation of Results

2.1 Comparison 1 – Strong Scalability ($n = 30$, chunks=32)

Experimental setup. All experiments use a key size $n = 30$ with `chunks=32`. The hybrid parallel implementation relies on one MPI rank per node and 18 OpenMP threads per rank. Execution times are reported separately for the *Fill* phase (dictionary construction) and the *Probe* phase (candidate testing). The reference execution corresponds to the sequential program `mitm_sequential` run on a single node.

Table 1: Execution times (s) for $n = 30$ and `chunks=32`.

Nodes p	Fill	Probe	Total	Speedup $S = T_{seq}/T_p$
Sequential (ref.)	177.6	429.5	607.1	1.00
1	136.2	279.9	416.1	1.46
2	69.3	141.3	210.6	2.88
4	70.8	30.7	101.5	5.98
8	73.4	33.0	106.4	5.71

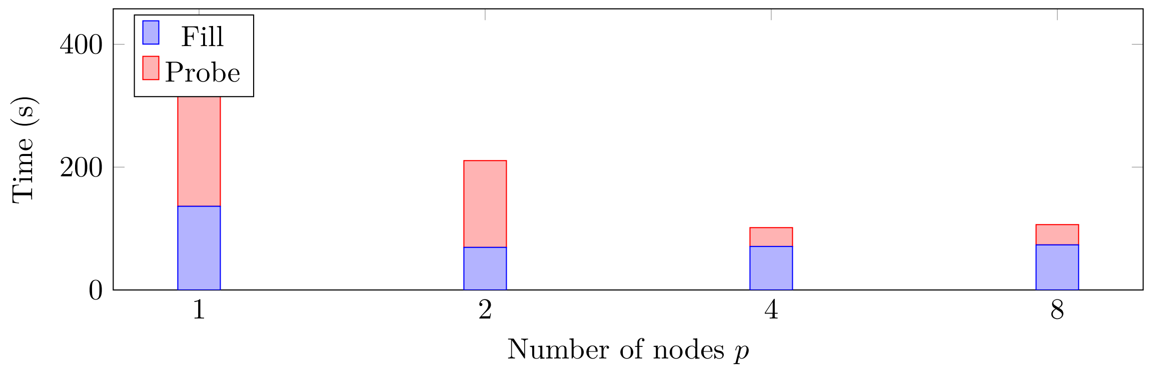

Figure 1: Stacked execution times for the Fill and Probe phases (MPI+OpenMP).

Interpretation. Execution time decreases significantly up to 4 nodes, reaching 101.5s (speedup 5.98), but slightly increases at 8 nodes. As shown in Figure 1, this limitation is caused by the *Fill* phase, whose execution time plateaus around 70–73s from 2 nodes onward. In contrast, the *Probe* phase scales very efficiently up to 4 nodes (429.5s $\rightarrow$ 30.7s), which is consistent with the reduction of the local dictionary size (from 14.5GB down to 3.6GB per node) and improved

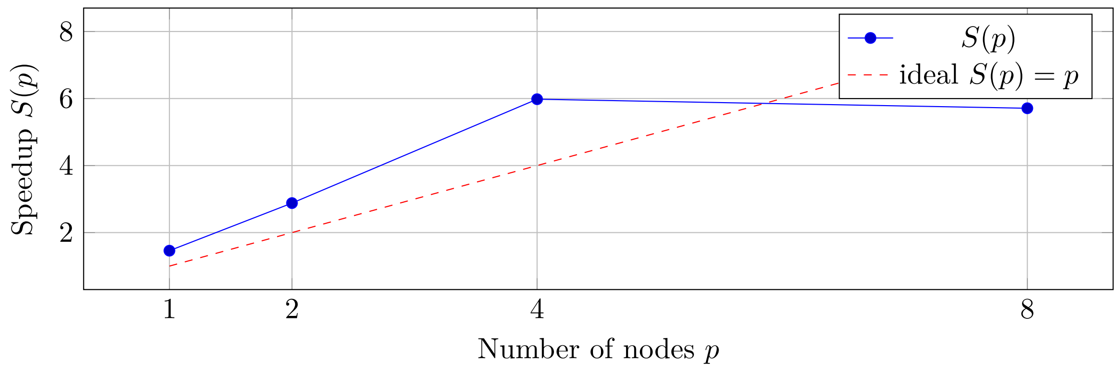

Figure 2: Overall speedup relative to the sequential execution ($T_{seq} = 607.1$s).

memory locality. Beyond 4 nodes, the additional speedup in *Probe* no longer compensates for the stagnation of *Fill*, explaining the loss of scalability at 8 nodes. The overall efficiency remains close to (or slightly above) 1 up to 4 nodes, then drops to 0.71 at 8 nodes, confirming the scalability limit for this configuration.

2.2 Comparison 2 – Impact of Chunk Size

Experimental setup. This experiment evaluates the impact of task granularity on performance at fixed scale. All runs use a key size $n = 30$ on 8 nodes, with one MPI rank per node and 18 OpenMP threads per rank. Only the chunk size is varied, while all other parameters are kept constant.

Table 2: Impact of chunk size on performance ($n = 30$, 8 nodes).

Chunks	Fill (s)	Probe (s)	Total (s)
8	73.1	30.5	103.6
32	73.4	33.0	106.4
128	77.7	42.0	119.7
512	94.5	100.4	194.9

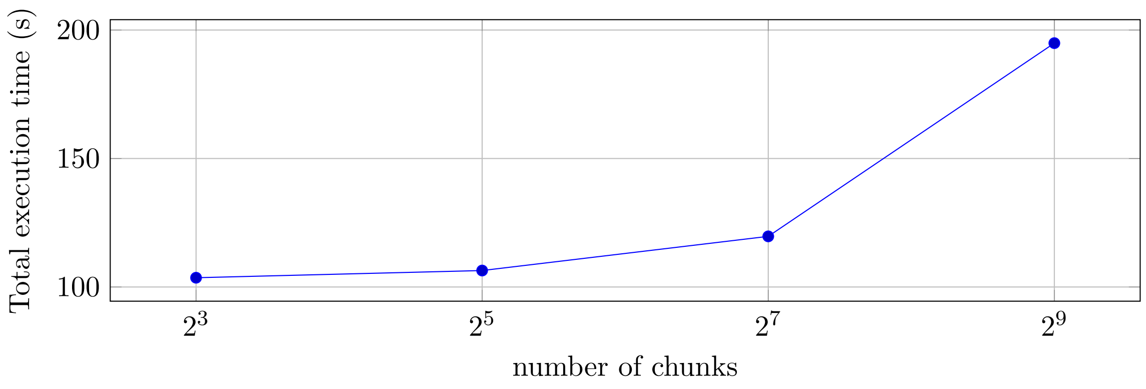

Figure 3: Total execution time as a function of number of chunks ($n = 30$, 8 nodes).

Interpretation. Number of chunks directly impacts the efficiency of the communication phase, which relies on an `MPI_Alltoall` exchange of dictionary entries. In principle, performing this exchange in a single step would minimize synchronization and communication overhead. However, as the problem size increases, the volume of data involved in the all-to-all communication becomes very large, leading to high memory pressure and network contention. In practice, a single large `MPI_Alltoall` call may become inefficient or even impractical due to buffer allocation constraints and limited communication scalability.

Chunking addresses this issue by splitting the communication into several smaller all-to-all exchanges, ensuring correct execution while keeping memory usage and communication volumes manageable. The results show that small number of chunks (8–32) achieve the best performance, indicating that the overhead of multiple collective calls remains negligible compared to the cost of large data transfers. Conversely, setting a **very high number of chunks (512)** significantly degrades performance (194.9s). Since the problem size ($N = 30$) is relatively small, dividing it into 512 parts results in tiny messages. The runtime is then dominated by the latency of hundreds of distinct MPI collective calls rather than actual data transfer.

These observations suggest that the number of chunks should be chosen as small as possible while still guaranteeing correct and efficient execution of the all-to-all communication, especially for larger problem sizes.

2.3 Large-scale run and practical limits ($n = 38$)

- **FOUND SOLUTION: k1=1ae969d8df, k2=9d50a6b5f with username : victor**

To characterize behavior at larger scale and highlight practical constraints, we report a large run performed at key size $n = 38$ on 40 nodes using 18 OpenMP threads per node and `chunks=512`. This configuration was chosen as a realistic upper bound within the project constraints: beyond this point, both memory footprint and runtime become prohibitive for repeated experimentation.

The measured timings were:

- Fill: 4466.1s
- Probe: 3835.1s
- Total: 8301.2s

During the Probe phase, the implementation tested 274,876,906,589 candidate pairs ($\approx 2.75 \times 10^{11}$), illustrating the rapid growth of the workload with n.

Discussion of limits for this test. This experiment highlights the main bottlenecks of the approach at large scale: (i) **memory capacity and memory bandwidth**: the dictionary grows quickly with n, leading to large memory footprints and heavy memory traffic; (ii) **communication and coordination overheads**: as the number of nodes increases, MPI-related overheads (collective operations, synchronization points, and coordination) represent a larger fraction of the runtime; (iii) **load balance and granularity**: uneven work distribution across chunks/ranks can introduce idle time, making the chunk size a critical tuning parameter (see Section 2.2). Overall, $n = 38$ constitutes a practical upper bound for demonstrating scalability on the target platform under realistic time and resource constraints.

Important Note on Stability and Reproducibility. It is crucial to highlight that the configuration used for this result ($N = 38$ on 40 nodes) represents a **critical stability threshold**. While the run reported above was successful, a subsequent attempt performed with identical parameters failed.

This instability suggests that 40 nodes are barely sufficient to handle the memory pressure, making the execution sensitive to cluster load and network variations (network timeouts).

Consequently, to ensure a guaranteed stable execution without failure (e.g., for grading or reproduction), we strongly recommend increasing the allocation to **50 nodes**.

3 Limits and Hardware Constraints

While our optimized implementation allowed us to solve high complexity instances up to $N = 38$, we eventually reached the physical limits of the available hardware. This section details the specifications of the cluster used and analyzes the memory bottleneck.

3.1 Cluster Specifications

All experiments were conducted on the *Gros* cluster located in Nancy. The specific hardware constraints of these nodes (particularly the RAM capacity) define the upper bound of what is computationally feasible.

Component	**Specification**
CPU	1 × Intel Xeon Gold 5220
Cores	18 cores per node
RAM Capacity	**96 GB** per node
Storage	447 GB + 894 GB SSD
Network	2 × 25 Gbps Ethernet
Total Nodes	124 nodes

Table 3: Hardware specifications of the cluster (Nancy).

3.2 The Memory Wall: Success at $N = 38$, Failure at $N = 39$

The primary limiting factor for the MITM attack is the RAM required to store the distributed dictionary (Phase 1).

1. The Saturation at $N = 38$: We successfully solved $N = 38$ using 40 nodes. The logs indicated that the local dictionary size consumed the vast majority of the available physical RAM. Considering the necessary memory reserved for the Operating System (kernel space) and the implicit overhead for network buffers (hidden MPI communication structures), the nodes were effectively operating at full capacity. This confirms that 40 nodes represented the strict minimum for this dataset size. Any reduction in node count would have mathematically forced the data size beyond the physical limit, resulting in immediate swapping or an Out-Of-Memory (OOM) crash.

2. The Crash at $N = 39$: When attempting to solve $N = 39$, the theoretical memory requirement grows by a factor of 2. Although we increased the resource allocation to 60 nodes to compensate : half of cluster's nodes, the job failed. This failure highlights a scale-out limitation: adding more nodes dilutes the dataset but simultaneously increases the **communication overhead**. With 60 nodes, the `MPI_Alltoall` primitive requires managing active connections and buffers for 59 peers simultaneously. This increased administrative memory cost, added to the expanded dataset, pushed the total usage beyond the 96 GB physical limit.

3.3 Conclusion on Scalability

Theoretically, solving $N = 39$ remains within the absolute boundaries of the cluster, but it presents significant logistical challenges. However, due to the increasing system overhead described above, a stable execution would likely require a significantly larger margin estimated

(60/70 nodes). Allocating this amount of resources represents **over 50% of the total capacity** of the cluster (124 nodes). In a shared academic environment (Grid'5000), obtaining such an exclusive reservation is operationally difficult and results in prohibitive queue wait times.