PPAR – Final Project

Federico Crivellaro: 21502450
Gabriele Argentieri: 21502063

Academic Year 2025–2026
Date: 4 January 2026

Submitted as part of the PPAR Course requirements at Sorbonne University

1 Introduction

The baseline implementation solves the golden-collision search using a two-phase meet-in-the-middle structure. While correct, it becomes impractical at large instances mainly because of **memory**, with performance as a secondary concern.

For moderate sizes ($n \leq 32$–33), in fact, the main limitation is runtime, and our focus is on accelerating the two dominant phases (*Fill* and *Probe*). This is achieved implementing per-rank enumeration and local processing with *OpenMP*.

As n increases, memory becomes the limiting factor: the dictionary built during Fill grows exponentially and quickly exceeds the RAM available on a single node, making it unrealistic to scale by simply multiplying resources. To address this, **MPI** is used to handle a **distributed sharded dictionary**, where each rank stores only a fraction of the entries, using a deterministic ownership rule (hash and modulo) to route both insertions and lookups to the correct rank.

For the largest instances ($n \geq 39$), even distributed storage can be insufficient. In this regime we additionally apply a **time–memory tradeoff** by partitioning the search into multiple passes, so that each pass handles only a slice of the dictionary, reducing peak memory while preserving correctness.

2 Parallel Architecture

Distributed Dictionary (Sharding): To distribute the storage of the dictionary across P MPI ranks, we implement deterministic sharding. Every generated dictionary key z is assigned to a unique owner rank:

$$\text{Owner}(z) = H(z) \bmod P, \tag{1}$$

where $H(\cdot)$ is a 64-bit Murmur-style hash. Hashing provides a near-uniform key distribution, balancing: memory usage (each rank stores only its shard) and work (insertions and lookups avoid hot-spots).

A critical correctness property is that both phases use the same ownership rule. Therefore, if a match exists, the corresponding Fill insertion and Probe lookup are guaranteed to reach the same owner rank.

Data Layout Optimization: Instead of using an array of packed structures (AoS), we implemented a **Structure of Arrays (SoA)** layout, splitting the hash table into two separate arrays: one for the keys (`uint32_t *A_k`) and one for the values (`uint64_t *A_v`). This approach leads to two benefits:

- **Memory Density without Misalignment:** We maintain the optimal memory footprint of 12 bytes per entry without packed structures. Since each array is allocated separately, access to both keys and values remains aligned, avoiding the architectural penalties of unaligned memory access.
- **Cache Efficiency:** During the probe phase, the CPU iterates through the table checking keys. In an AoS layout, a single cache line fetch loads only 4 keys, as the memory bandwidth is wasted on value bytes that are not needed for the comparison. With our SoA layout, the array consists solely of 4-byte keys, allowing a single cache line to load 16.

Adaptive Load Factor: We implement an adaptive memory allocation strategy based on the problem size n. For large instances ($n \geq 32$) where runtime is the primary bottleneck, we allocate the dictionary with a 2.0x padding (targeting a load factor of 0.5) to minimize linear probing collisions. For intermediate sizes where memory pressure is higher, we automatically switch to a tighter padding (approx. 1.125x), prioritizing memory density over raw access speed.

Intra-rank Parallelism and Thread-Local Batching: Within each MPI rank, we utilize OpenMP to parallelize the enumeration of the search space (x in Fill, y in Probe). A naive coupling of multithreading and MPI would lead to severe contention, as MPI calls often require serialization (e.g., via `OMP critical`). To mitigate this, we implement a **thread-local batching strategy**: each thread accumulates generated entries in a private buffer (size 256) and enters the critical section to flush data to the main MPI buffers only when full. Additionally, concurrent lock-free insertions are managed via GCC atomics with strict memory barriers to ensure that values are fully initialized before their corresponding keys become visible to other threads.

3 Asynchronous Communication Protocol

A naive distributed implementation would send one MPI message per generated key, resulting in an extreme number of tiny messages and dominating runtime via network latency. We therefore implement batching and non-blocking progress.

Batching and Non-Blocking Sends: For both Fill and Probe, each rank maintains one outgoing buffer per destination rank. Entries are appended to the buffer and flushed when reaching `BUFFER_SIZE`, using `MPI_Isend`. This amortizes latency and increases effective throughput.

Draining Incoming Traffic: To avoid deadlocks and prevent network buffers from filling while computation proceeds, each rank periodically calls a `drain_incoming()` routine. This routine progresses posted non-blocking receives, processes completed batches, and updates the local shard (in Fill) or answers lookup requests (in Probe). This design also enables overlap of computation and communication.

Persistent Receive Windows: To maximize network throughput, we maintain a pool of persistent receive requests posted *before* computation begins. As soon as a request completes and its data is consumed, the slot is **immediately reposted**. This ensures that the network layer always has available descriptors for incoming packets, minimizing flow control stalls.

4 Phase 1: Fill

Fill builds the distributed dictionary shard-by-shard. Each rank enumerates its own contiguous range of x values, computes the key z, and routes the insertion to `Owner`(z). If the owner is local, the entry is inserted directly, otherwise it is buffered and sent to the owner rank in batches.

5 Phase 2: Probe

Probe mirrors Fill. Each rank enumerates its range of y values, computes the lookup key k, and routes the request to `Owner`(k) using the same ownership rule. If the owner is local, the local shard is probed directly; otherwise, the request is buffered and sent.

Remote Candidate Verification: The probe phase follows a "push-compute" model. Instead of fetching dictionary entries over the network, the rank generating a query y sends the tuple (y, k_2) to `Owner`$(g(y))$. Upon receipt, the owner checks its local shard for $g(y)$. If matches are found, the owner locally performs the golden collision check $\pi(x, y)$ and stores any valid solutions found. This minimizes round-trip latency, as no data needs to be returned to the requester unless a solution is found (which is collected at the end).

Candidate Handling and Golden Check: Upon receiving a probe request, the owner rank looks up k in its local shard. If candidates exist, it tests each candidate (x, y) against the golden condition and records valid solutions. As in Fill, ranks periodically drain incoming probe traffic while continuing their local enumeration.

6 Time-Memory Tradeoff Optimization

For large instances, where the dictionary size exceeds available RAM even when distributed, a time-memory tradeoff is implemented by partitioning the search space into S slices (a power of two). Each iteration builds a dictionary containing only a fraction $1/S$ of the total entries. Then, the entire space of probe keys y is tested against this partial dictionary. This allows the computation to fit in limited RAM by repeating the probe workload S times.

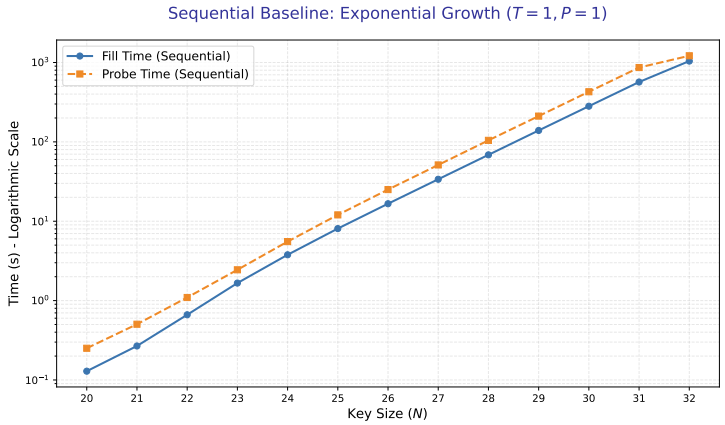
7 Experimental Results

All experiments were compiled with `mpicc -O3 -march=native -fopenmp -Wall -o mitm mitm.c -fopenmp` by running the *Makefile* provided, then executed on *Grid5000*, precisely on the *Paradoxe* cluster in the site Rennes. The command utilized to run was:

```
oarsub -l "{cluster='paradoxe'}/nodes=<NUM_NODES>,walltime=<WALLTIME>"
   -n "MITM" --stdout "mitm_%jobid%.out" --stderr "mitm_%jobid%.err"
   "mpirun -np <NUM_RANKS> --hostfile \$OAR_NODEFILE
   --map-by ppr:1:socket:pe=<NUM_THREADS> --bind-to core
   ./mitm --n <PROBLEM_SIZE> --C0 <C0> --C1 <C1> --verbose"
```

7.1 Baseline: Sequential Growth

The sequential implementation exhibits the expected exponential growth in n. Figure 7.1 reports the measured Fill and Probe times using a **logarithmic y-axis**. In this representation, both curves are approximately straight lines, which indicates that **Fill and Probe grow exponentially** with n, reaching $n = 32$ in about 2094.8 s (≈ 35 min).



7.2 OpenMP Strong Scaling (Single Node)

We evaluate intra-node **strong scaling** by fixing the problem size ($n \in \{28, 29, 30\}$) and increasing the number of OpenMP threads. The objective is to minimize runtime for a fixed workload.

Performance on $N = 28$ and $N = 29$. For these dataset sizes, the application demonstrates **robust scalability**. Both phases benefit significantly from increased parallelism up to the maximum available hardware threads. As shown in the Figures 1, the execution time decreases monotonically. This indicates that for memory footprints well within the node's capacity, the overhead of thread management is negligible compared to the computational gain.

Performance on $N = 30$ (Memory Saturation). At $N = 30$ (2^{30} entries), the memory pressure increases significantly. While the **Fill phase** continues to scale efficiently (reaching $\sim$ 15s at 26 threads), the **Probe phase** exhibits a memory-bound behavior. Table 2 shows that the total speedup peaks at **16 threads**. Beyond this point, the Probe phase suffers from memory bandwidth saturation, leading to a performance regression at 26 threads.

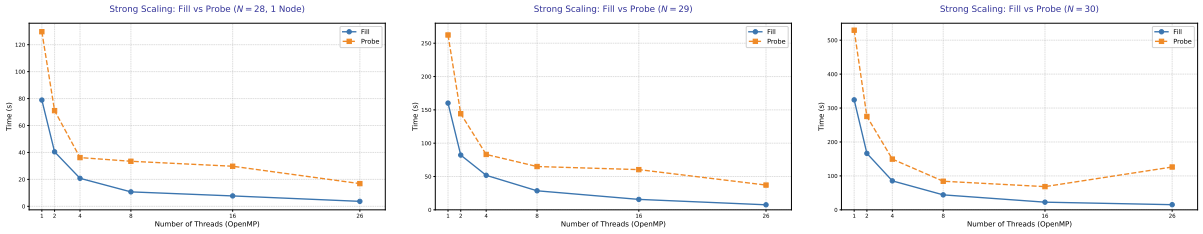


Figura 1: OpenMP strong scaling for different problem sizes.

8 Distributed Memory Analysis (MPI)

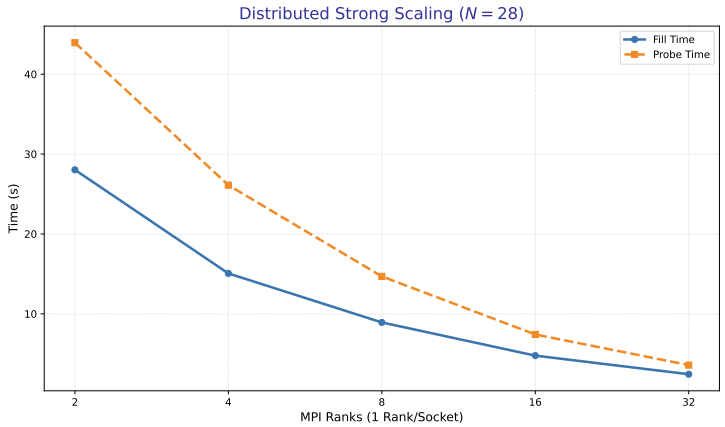
The results for $N = 30$ highlight the physical limits of a single node when handling massive random-access workloads. Since the memory bandwidth becomes the bottleneck before CPU capacity is fully exhausted, simply adding local threads is no longer sufficient. To address this limitation and further reduce execution time, the next phase of the analysis will involve increasing the number of compute nodes using **MPI**. This approach allows

us to distribute the memory pressure across multiple nodes, effectively increasing the aggregate memory bandwidth available to the application.

8.1 Distributed Strong Scaling ($N = 28$)

First, we validate the MPI implementation by performing a strong scaling test on a fixed workload of $N = 28$. We vary the number of MPI ranks from 2 to 32 (1 rank per socket, 2 sockets per node) to observe how the execution time decreases with added resources.

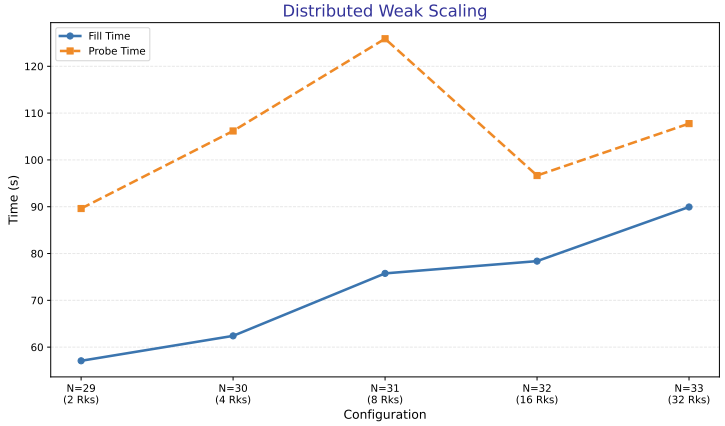
Figure 8.1 shows a clear trend: both Fill and Probe times decrease monotonically as the number of ranks increases. This confirms that the communication overhead introduced by MPI is well-managed and does not prevent the application from scaling down the execution time effectively for a fixed problem size.



8.2 Distributed Weak Scaling ($N = 28$ to $N = 33$)

To address larger datasets, we perform a weak scaling test, increasing the problem size from $N = 28$ to $N = 33$ while proportionally doubling the number of MPI ranks at each step (maintaining a constant workload per node).

As illustrated in Figure 8.2: the **Fill Phase** shows a controlled, linear increase in time, suggesting the distributed construction scales predictably, while the **Probe Phase** exhibits higher variance but stabilizes around 110 seconds for the largest configurations ($N = 33$).



8.3 Maximum Solved Instance and Memory Limits

With the largest non-partitioned configuration tested (**36 MPI ranks on 18 nodes**, **26 threads per rank**), we successfully solved up to $n = 38$ in $5518,82$ seconds (≈ 1.5 hour). Attempts to run instances with $N \geq 39$ using the standard implementation resulted in immediate **Out-Of-Memory (OOM)** errors during the dictionary allocation phase.

9 Time-Memory Tradeoff via Partitioning

To overcome the RAM limitation and solve the target instance of $N = 40$, we implemented a time-memory tradeoff by splitting the search space into S temporal partitions. This reduces the peak memory requirement per MPI rank by a factor of approximately $1/S$, at the cost of increased time.

Selection of the Number of Partitions To determine the optimal S, we compared the theoretical memory required to store the dictionary against the aggregate physical RAM available in the cluster. The memory requirement M_{req} for a given N is calculated as:

$$M_{\text{req}}(N) = 2^N \times \mathcal{E} \times \alpha \tag{2}$$

where $\mathcal{E} = 12$ bytes is the size of a packed entry, and $\alpha = 2.0$ is the safety load factor used in the partitioned implementation to minimize collisions in smaller sub-tables. The available memory M_{avail} on our cluster configuration (18 nodes) is:

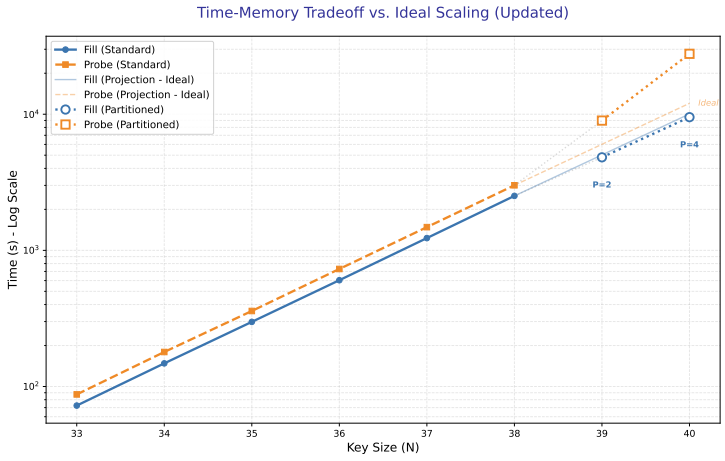
$$M_{\text{avail}} = 18 \times 384 \text{ GB} \approx 6,912 \text{ GB} \tag{3}$$

Since the implementation relies on efficient bitwise masking (`hash & (S-1)`) to route items to partitions, S must be a power of two.

Thus we applied this formula to determine the configuration for the challenging instances:

$$R = \frac{M_{\text{req}}}{M_{\text{avail}}} = \frac{2^N \times 12 \times 2.0}{6,912 \times 10^9}$$

Thanks to this equation, we calculated that the required memory is approximately 13.2 TB for N=39 and around 26.4 TB for N=40.



Analysis of Overhead. The partitioned strategy introduces a necessary overhead due to the management of multiple passes. However, the impact varies significantly between phases:

- For the **Fill phase** (Blue), the performance adheres remarkably well to the ideal projection. The measured times for $N = 39$ and $N = 40$ lie almost exactly on the theoretical trend line, indicating that the overhead for the partitioned construction is negligible.
- For the **Probe phase** (Orange), the overhead is significant. At $N = 40$ with $P = 4$, the measured time is approximately **27,781 s** (~ 7.7 hours), a $\sim 2.31\times$ **slowdown** compared to the theoretical ideal scaling (i.e., the time projected if infinite RAM were available).

While the partitioned approach incurs a time penalty, it successfully achieves the primary goal of **feasibility**, allowing the cluster to solve the $N = 40$ problem which is otherwise impossible on this hardware configuration.

10 Discussion

Overall performance is shaped by three interacting bottlenecks:

- **Memory behavior.** Dictionary operations involve irregular accesses and limited locality, causing early saturation in OpenMP scaling and favoring multi-node scaling (more aggregate memory bandwidth).
- **Communication overhead.** In the Probe phase, sharded lookups generate significant message traffic; batching and periodic draining mitigate but do not eliminate this effect, and weak scaling highlights Probe sensitivity.
- **Memory capacity.** Without partitioning, the maximum n is limited by RAM; partitioning provides a practical mechanism to push beyond this boundary, at the cost of additional runtime.

A key qualitative result is that **partitioning enables larger instances** (up to $n = 40$ here), turning the problem from *"impossible due to RAM"* into *"solvable within acceptable time"*.

11 Conclusion

We implemented a hybrid MPI+OpenMP solution based on a distributed sharded dictionary. Non-blocking batched communication and periodic draining allow partial overlap of communication and computation, while thread-safe insertion enables intra-rank parallelism.

Experimentally, OpenMP strong scaling saturates early due to memory/contestion effects, whereas MPI strong scaling provides substantial speedup across nodes. The ultimate limitation is memory capacity: without partitioning we reached $n = 38$, while partitioning allowed us to solve up to $n = 40$, providing a practical time-memory tradeoff to remain within RAM bounds.

A Appendix: Detailed Experimental Results

All of the tests were carried out with the username `ppar`.

Tabella 1: Baseline Sequential Run

N	NP	Nodi	Threads	Partitions	Fill Time	Probe Time	Solutions
20	1	1	1	1	0.129s	0.251s	k1=c72ed, k2=1a6ce
21	1	1	1	1	0.268s	0.504s	k1=118da0, k2=d986f
22	1	1	1	1	0.663s	1.094s	k1=1700a3, k2=2e561d
23	1	1	1	1	1.663s	2.444s	k1=25c499, k2=304c59
24	1	1	1	1	3.779s	5.541s	k1=4e151c, k2=d09349
25	1	1	1	1	8.082s	12.021s	k1=c2bdbb, k2=1e2f58b
26	1	1	1	1	16.623s	25.054s	k1=1922aa7, k2=26c984c
27	1	1	1	1	33.800s	51.200s	k1=81056a, k2=2a8e250
28	1	1	1	1	68.636s	104.526s	k1=85acec8, k2=6c48b0c
29	1	1	1	1	139.253s	211.243s	k1=e980698, k2=7397b2a
30	1	1	1	1	281.511s	428.044s	k1=7a3f106, k2=2ce169ba
31	1	1	1	1	567.747s	864.387s	k1=7d240482, k2=47eccf23
32	1	1	1	1	1042.443s	1215.337s	k1=a7413f3d, k2=95900609

Tabella 2: OpenMP Strong Scaling (Single Node)

N	NP	Nodi	Threads	Partitions	Fill Time	Probe Time	Solutions
28	1	1	1	1	78.905s	129.633s	k1=85acec8, k2=6c48b0c
28	1	1	2	1	40.467s	71.042s	k1=85acec8, k2=6c48b0c
28	1	1	4	1	20.798s	36.188s	k1=85acec8, k2=6c48b0c
28	1	1	8	1	10.731s	33.447s	k1=85acec8, k2=6c48b0c
28	1	1	16	1	7.723s	29.762s	k1=85acec8, k2=6c48b0c
28	1	1	26	1	3.697s	16.866s	k1=85acec8, k2=6c48b0c
29	1	1	1	1	160.158s	262.358s	k1=e980698, k2=7397b2a
29	1	1	2	1	82.159s	144.185s	k1=e980698, k2=7397b2a
29	1	1	4	1	51.809s	83.096s	k1=e980698, k2=7397b2a
29	1	1	8	1	28.572s	64.983s	k1=e980698, k2=7397b2a
29	1	1	16	1	15.671s	60.346s	k1=e980698, k2=7397b2a
29	1	1	26	1	7.561s	37.160s	k1=e980698, k2=7397b2a
30	1	1	1	1	324.153s	529.463s	k1=7a3f106, k2=2ce169ba
30	1	1	2	1	166.352s	275.162s	k1=7a3f106, k2=2ce169ba
30	1	1	4	1	85.382s	149.501s	k1=7a3f106, k2=2ce169ba
30	1	1	8	1	44.115s	84.260s	k1=7a3f106, k2=2ce169ba
30	1	1	16	1	22.581s	68.420s	k1=7a3f106, k2=2ce169ba
30	1	1	26	1	15.186s	126.229s	k1=7a3f106, k2=2ce169ba

Tabella 3: Distributed Strong Scaling

N	NP	Nodi	Threads	Partitions	Fill Time	Probe Time	Solutions
28	2	1	26	1	28.031s	43.944s	k1=85acec8, k2=6c48b0c (rank 0)
28	4	2	26	1	15.059s	26.109s	k1=85acec8, k2=6c48b0c (rank 0)
28	8	4	26	1	8.925s	14.687s	k1=85acec8, k2=6c48b0c (rank 0)
28	16	8	26	1	4.769s	7.420s	k1=85acec8, k2=6c48b0c (rank 0)
28	32	16	26	1	2.442s	3.584s	k1=85acec8, k2=6c48b0c (rank 0)

Tabella 4: Distributed Weak Scaling

N	NP	Nodi	Threads	Partitions	Fill Time	Probe Time	Solutions
29	2	1	26	1	57.081s	89.614s	k1=e980698, k2=7397b2a (rank 0)
30	4	2	26	1	62.404s	106.180s	k1=7a3f106, k2=2ce169ba (rank 3)
31	8	4	26	1	75.755s	125.888s	k1=7d240482, k2=47eccf23 (rank 2)
32	16	8	26	1	78.374s	96.662s	k1=a7413f3d, k2=95900609 (rank 7)
33	32	16	26	1	89.943s	107.758s	k1=629869b, k2=1da9ad39a (rank 18)

Tabella 5: Running until OOM

N	NP	Nodi	Threads	Partitions	Fill Time	Probe Time	Solutions
33	36	18	26	1	72.427s	87.480s	k1=629869b, k2=1da9ad39a (rank 30)
34	36	18	26	1	147.908s	179.270s	k1=1c7f5a4e8, k2=34b6cceef (rank 2)
35	36	18	26	1	298.328s	358.674s	k1=70a23cb73, k2=87eea083 (rank 2)
36	36	18	26	1	603.656s	729.998s	k1=dc1edb9c0, k2=c95eb83b7 (rank 27)
37	36	18	26	1	1230.990s	1481.183s	k1=194843a4bb, k2=13ca02d470 (rank 31)
38	36	18	26	1	2512.619s	3006.199s	k1=3d4bdf4d77, k2=2f9bd24c41 (rank 9)
39	36	18	26	1	OOM	OOM	OOM
40	36	18	26	1	OOM	OOM	OOM

Tabella 6: Time-Memory Tradeoff via Partitioning

N	NP	Nodi	Threads	Partitions	Fill Time	Probe Time	Solutions
39	36	18	26	2	4824.49s	8980.895s	k1=478ad38aa0, k2=666f0b5d3f (rank 33, partition 1/2)
40	36	18	26	4	9515.384s	14447.991s	k1=cfabb5e696, k2=5c9cecd007 (rank 33, partition 3/4)