SORBONNE UNIVERSITÉ

PROJECT “GOLDEN COLLISON” REPORT

Prepared By

Kharef Okba & Nicholas Bell

Nicholas.Bell@etu.sorbonne-universite.fr

Okba.Kharef@etu.sorbonne-universite.fr

Professor

Charles Bouillaguet

Introduction

The following is a short summary of the problem and the sequential version of code. Feel free to skip to the next section.

The purpose of this project was to implement a parallelized version of a meet-in-the-middle attack program to find a "golden collision" for the problem defined as follows:

Given two unary functions f, g mapping an n-bit integer to n-bit integers (assumed to be cryptographic hash functions, i.e. quick evaluation, internal analysis infeasible), and a binary relation π on pairs of n-bit integers, a golden collision is defined as a pair of n-bit integers (x, y) such that f(x) = g(y) and π(x, y) = 1. The sequential program provided (sequential.c), computes a brute force inverse of f in the form of a dictionary, with which g can be used, for a given y, to find values x such that f(x) = g(y), after which π(x,y) can be verified. This serves as the baseline for parallelisation. This complexity of this approach is exponential, requiring at a minimum to compute 2^n evaluations of f, and then up to 2^n evaluations of g. The memory complexity is again exponential, requiring the storage of 2^n inputs to f in the dictionary.

The primary objective of parallelisation was to decrease the computation time and memory required per core, to increase the maximum input size, while also reducing total execution time.

Overview of Implementation :

1. Parallelization Strategy:

We used MPI to implement parallelisation and started by building a distributed dictionary using sharding, an approach where each MPI process creates a local dictionary of size `ceil(global_dict_size / world_size)`, ensuring distributed memory load and also entailing distributed lookup during the search stage. The dictionary being implemented as a hash map, which allows flexible size as the integers f(x) for all n are stored as 64-bit integers, we allocate to each MPI process a block of the indices (h(f(x)) := murmur64(f(x)) % global_dict_size). Given the potential for collisions as f is not assumed to be injective, we shift entries to the next available slot in the hash table. This means we lose the direct correspondence between h(f(x)) and the index, and to avoid recomputation of f(x), we store a secondary smaller hash p(f(x)) := f(x) % PRIME within the entry consisting of 32 bits rather than the full 64 bits of f(x). As such, each entry in memory is a pair (p(f(x)), x), stored at ~h(f(x)) % global_dict_size, as it is unlikely that two different f(x) will share a close index in the dictionary and the same p(f(x)). While in the sequential version, shifting to the next available index in the case of hash collisions does not cause issues, our approach does not send entries to other processes in the event that this shift places the entry out of bounds of the local hash table, and instead wraps around locally. This causes the issue that a local dictionary may not have sufficient allocated space in the hash table despite the global table being large enough. For small values of n this is within the realm of possibility, and the memory factor (1.125x) must be increased to avoid space issues, while in the average case for large values of n, importantly all those tested in our report, problems do not arise.

Each process is allocated a block of the input space to compute and store. When writing a new pair to the dictionary, the responsible process first computes the h(f(x)) to determine which process's shard the pair belongs to. If the pair belongs to the local shard, it is stored immediately. Otherwise, the pair is temporarily placed in a buffer designated for the corresponding process to reduce the frequency of message passing and therefore the communication overhead. Each process maintains an array of buffers which holds BUFFER_SIZE (= 1000) requests for each other process. Whenever the buffer is full, and once at the end of computation, it is emptied and sent to the appropriate target process, which then integrates the received values into its local dictionary shard. This approach is a compromise between memory overhead and computation time overhead due to parallelisation. This is illustrated the diagram below:

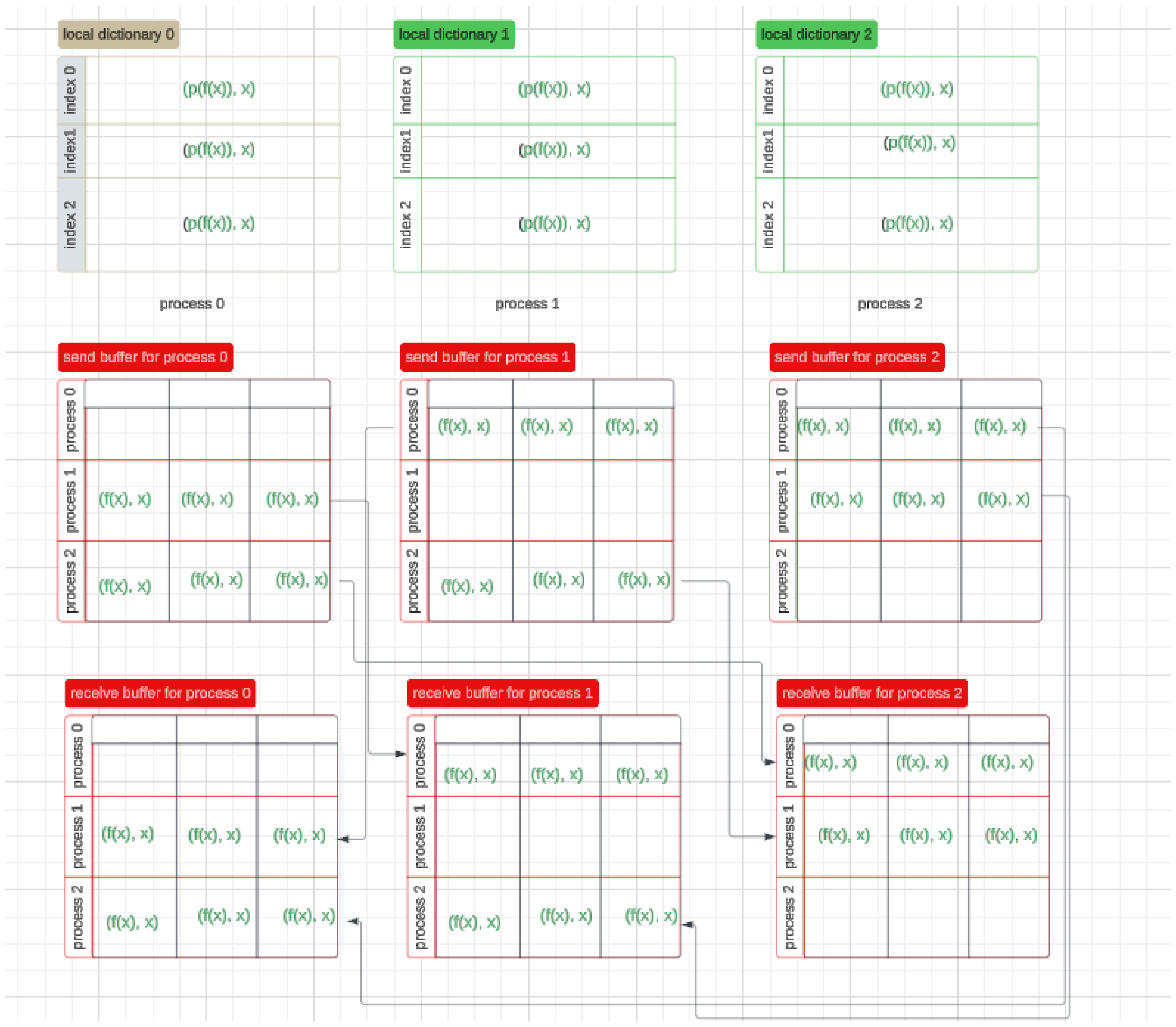

local dictionary 0
local dictionary 1
local dictionary 2
index 0
index1
index 2
(p(f(x)), x)
process 0
process 1
process 2
send buffer for process 0
send buffer for process 1
send buffer for process 2
(f(x), x)
receive buffer for process 0
receive buffer for process 1
receive buffer for process 2

Diagram 01: showing the structure of the dictionary

During the probing stage, each process computes the function g(y) for its assigned subset of inputs and probes the distributed dictionary to identify potential collisions where f(x)=g(y). We use an identical set of buffers to those in the writing stage. While in a general sharded dictionary, a read would require a response, here we can cut corners. A process receiving a read request for g(y) does not return the corresponding values of x. We instead send y along with g(y), and the receiving process checks the candidates itself and if needed adds them to its own solution buffer. This cuts out an extra stage of communication, which both simplifies the code and reduces overhead.

For both writing and reading, we need periodic buffer checks to see if there are any incoming requests. If these are not properly received, the sending process cannot clear its buffer and blocks on the next entry to that

Finally, verified solutions are gathered and consolidated using `gather_solutions`, allowing the results to be aggregated on a single process (root) to be printed out. This design leverages MPI's distributed computing capabilities to achieve efficient parallel processing and synchronization across all processes.

2. Attempted OpenMP Optimisation:

We attempted to implement some OpenMP optimisations in the following ways.

Initialisation of the dictionary: placing EMPTY in each slot of the local dictionary is easily locally parallelisable. We placed a simple #pragma omp parallel for before this loop which we found decreased the execution time.

Writing to the dictionary: our sections of modified shared memory are the local dictionary and each pair of send/receive buffers and counts. We wanted to implement a critical section for writing to the local dictionary, and a critical section for dealing with each process's set of buffers. Given the buffer count could be read by the section checking the count, while another process was already modifying the buffer, we also needed to protect this. However, because the buffer used was determined at runtime, we couldn't use a standard OpenMP critical section to encapsulate the code modifying the buffer, and we tried using OpenMP locks instead. Unfortunately, there was an issue with this approach, and we got segmentation faults with large values of n, perhaps due to a lack of understanding on our part on the behaviour of OpenMP. We have left this extra code commented out in our submission.

Probing the dictionary: we tried reasoning about the reads in a similar way. In this case we have the same buffer modification concerns, and so we wanted a lock for each other process. We also needed to be careful with two variables: num_solutions and local_candidates. Because num_solutions modifies the next insertion point for a solution, we wrapped it in a critical block, and local_candidates needed to be atomic to avoid data races. Unfortunately, this caused an increase in execution time, so we removed this and left it as commented code.

3. Perfomance Analysis:

a. Resource Reservation:

We reserved enough nodes for 128 cores in the gros cluster in Nancy, so we could test the speedup for a large number of MPI processes. We had trouble running this in non-interactive mode, so we instead reserved resources in interactive mode.

b. Compiling and Running the program:

Basic commands:

All keys used for the analysis were from https://ppar.tme-crypto.fr/nbell/<n>

Our makefile has the flags `-Wall -Wextra -O2 -lm -fopenmp` for our parallel code and the same without -fopenmp for the sequential code.

An example of our execution is the following

```
    mpiexec -hostfile $OAR_NODEFILE -np 64 ./mitm -n 30 –C0 6be38c825fae1108  –C1
be76d5e30c21f609  > data/mim-par-30-64-out.txt && ./data/extract1.sh data/mim-par-30-64-out.txt
```

The raw output of the program contained a lot of warnings at the end, which we believe was quadratic with the number of processes, perhaps some non-blocking sends or receives we overlooked. We weren't able to identify the problem but were able to pipe the output into a file which we stored in our git. We cleaned this data using a script provided by Vibhu Kapadia, which we then modified, who experienced the same issue. Since we wanted to run many variations of the program over many processes and different key sizes, we decided to run a list of commands at once and store the outputs in a collection of separate files.

c. Comparing sequential with parallel:

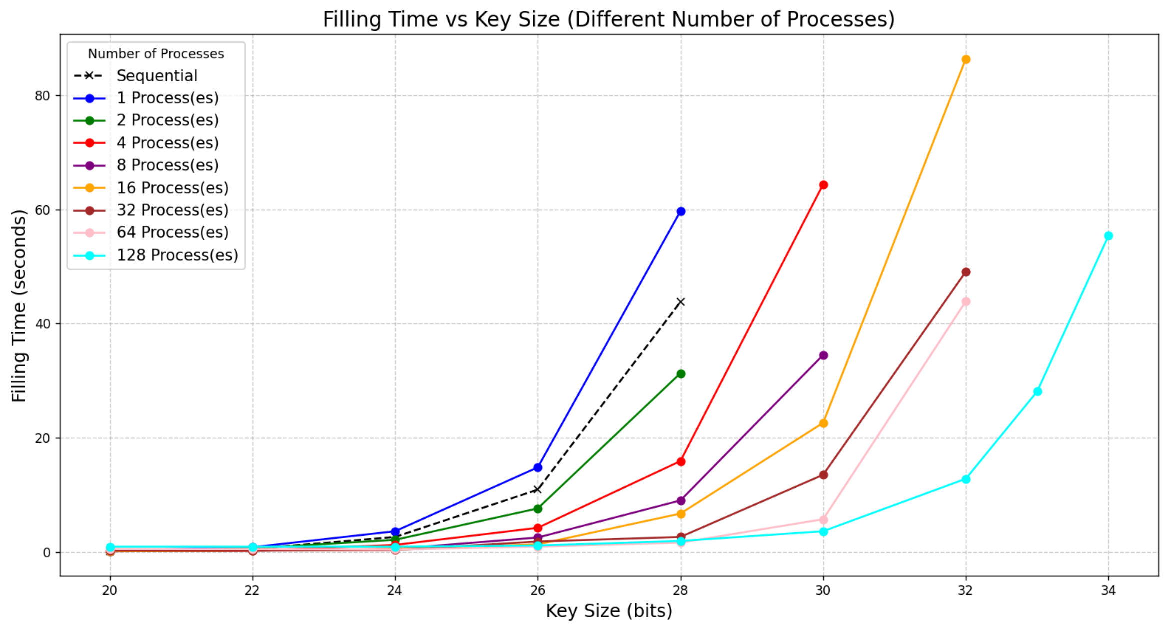
i. PARALLEL: varying the keysize and the number of processes:

This plot demonstrates the trade-off between computational speedup and the overhead of parallel execution. Parallelization is highly effective for large key sizes (≥28 bits), where the filling time is significantly reduced as the number of processes increases. However, beyond 64 processes, the marginal performance gains diminish, and for 128 processes, the performance even degrades for key sizes ≥32 bits.

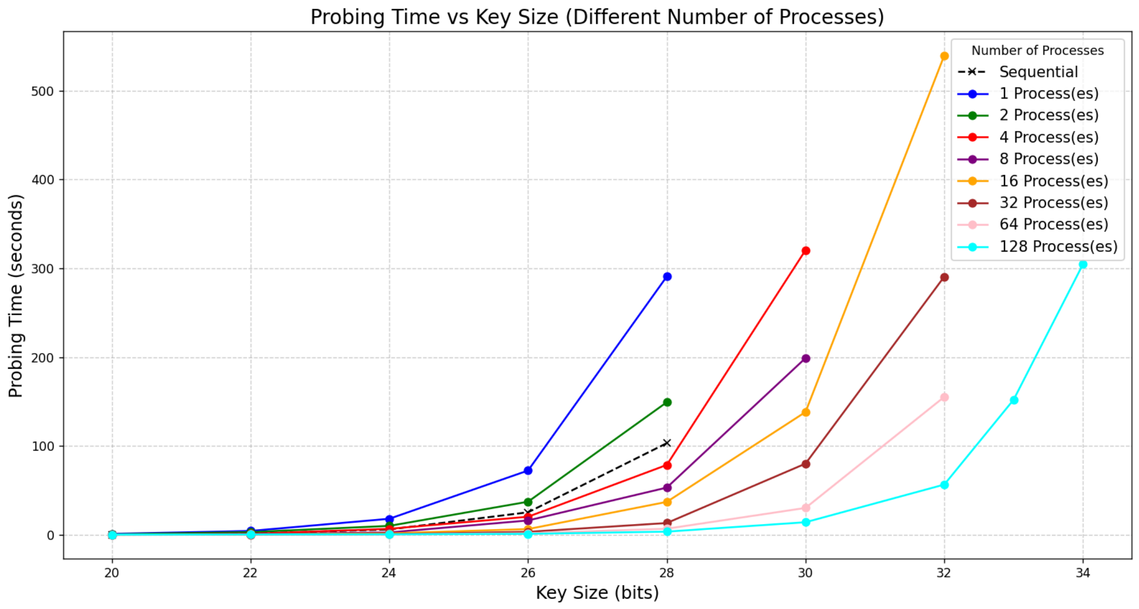
To maximize efficiency:

- For small key sizes (e.g., ≤26 bits), sequential or minimally parallel configurations (1-8 processes) are sufficient, as the parallel overhead outweighs the computational savings.
- For large key sizes (≥28 bits), increasing the number of processes up to 64 or 128 yields substantial performance gains. However, extra processes increase the parallelisation overhead, so the key size needs to be sufficiently large.

Further optimisation could focus on minimizing communication overhead and improving load balancing, especially for configurations with a high number of processes. Additionally, adaptive strategies to dynamically select the optimal number of processes based on the key size may improve overall performance. We could also reduce the memory factor (1.125x) in code as the key size increases, and implement multi-threaded optimisations to locally parallelise the code.



The plots show that probing time increases with key size, and parallelization significantly improves performance compared to sequential execution, especially for large key sizes (≥28 bits). For small key sizes (≤26 bits), parallelism provides limited benefits due to overhead. Moderate parallel configurations (16-64 processes) offer the best balance of reduced time and efficiency. High process counts (128 processes) yield diminishing returns and face overhead issues like communication bottlenecks for very large key sizes (≥32 bits). Sequential execution performs poorly as key size grows, underscoring the need for parallelism in larger problem spaces. Optimization efforts should focus on reducing synchronization overhead and improving load balancing.



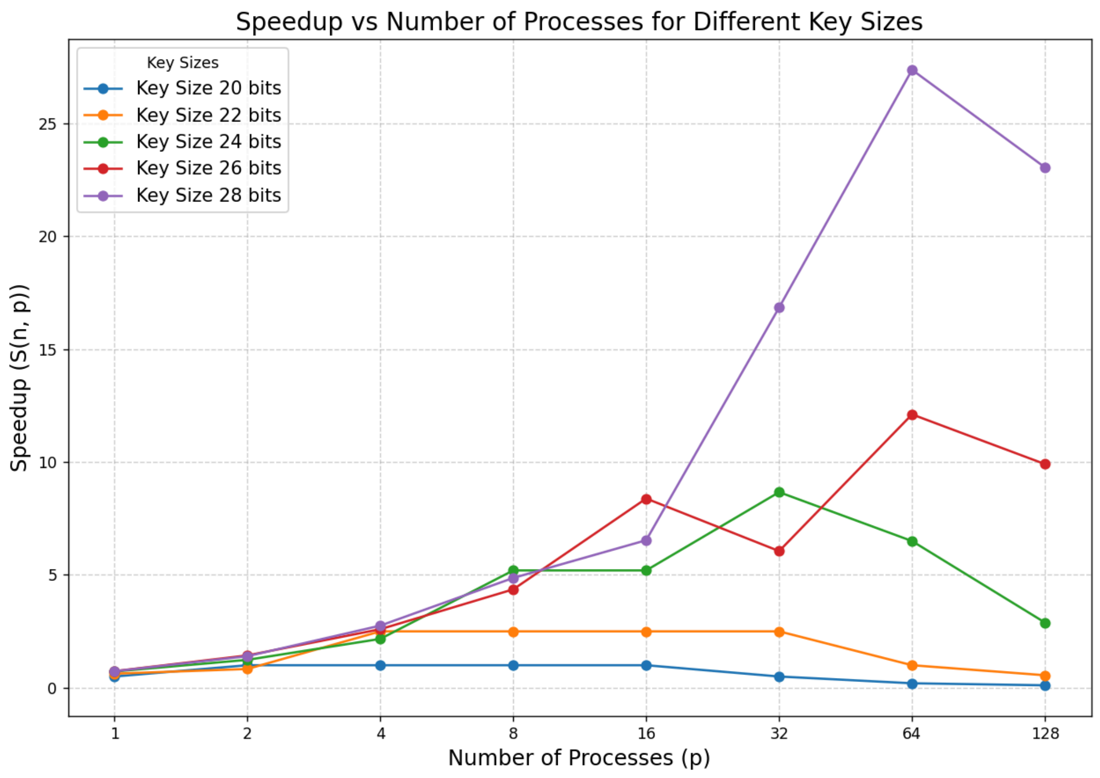
ii. Comparison:

Our implementation incurs a parallel overhead which makes it less efficient for one process, however for a sufficient key size parallelised version is faster for 2 processes and above. The maximum key size we were able to solve was n=34.

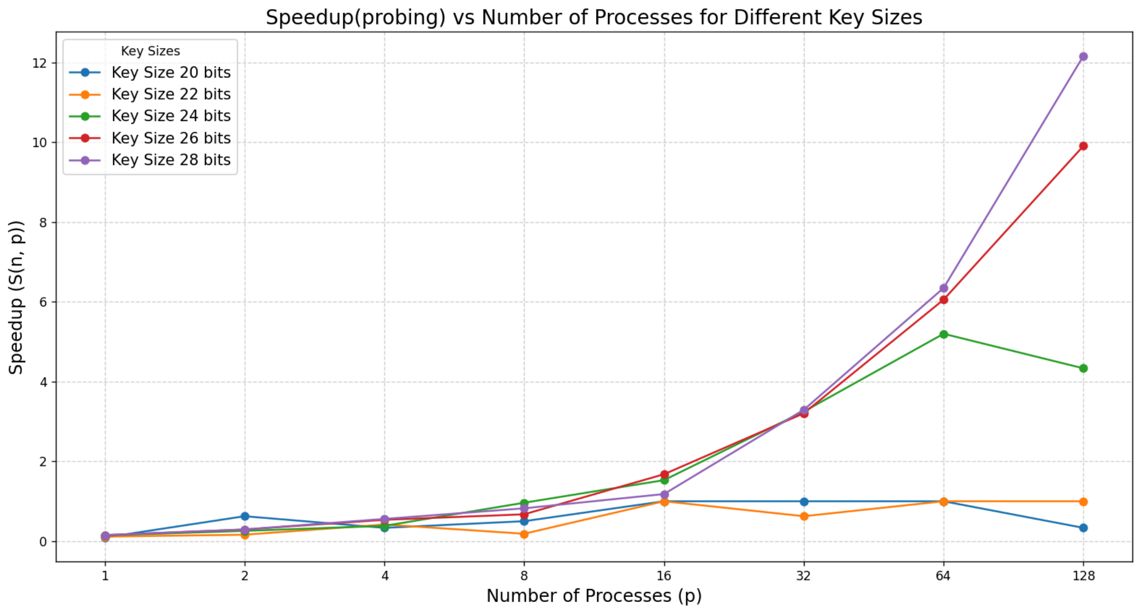
d. Plots:

i. Speedup plot:

The speedup graph illustrates the relationship between speedup S(n,p) and the number of processes p for different key sizes. Larger key sizes (e.g., 28 bits) achieve higher speedup, peaking at around 64 processes, demonstrating effective parallelization. Smaller key sizes (e.g., 20 bits) show limited speedup, plateauing or declining beyond a few processes due to the parallelisation overhead being the dominating factor. The diminishing returns for all key sizes as p increases past the optimal range highlight the impact of communication overhead. We cannot calculate speedup for a large enough number of bits because the execution of the sequential program becomes too slow.

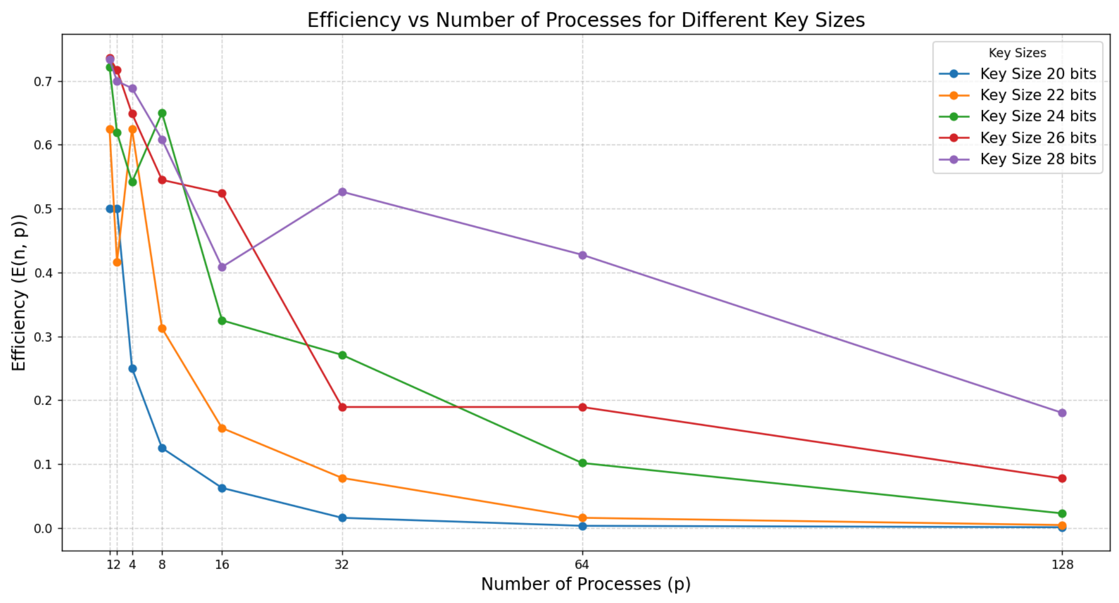


This plot highlights that parallelization achieves significant speedup only for larger key sizes (≥26 bits). Key size 28 bits benefits the most, achieving nearly 12× speedup with 128 processes. For smaller key sizes (≤24 bits), the overhead of managing parallel processes outweighs the performance gains. The optimal number of processes depends on the key size: for larger keys, 32-64 processes offer a good balance between efficiency and scalability, while higher process counts show diminishing returns. Optimization should focus on minimizing parallel overhead and ensuring load balancing, particularly for higher process counts.

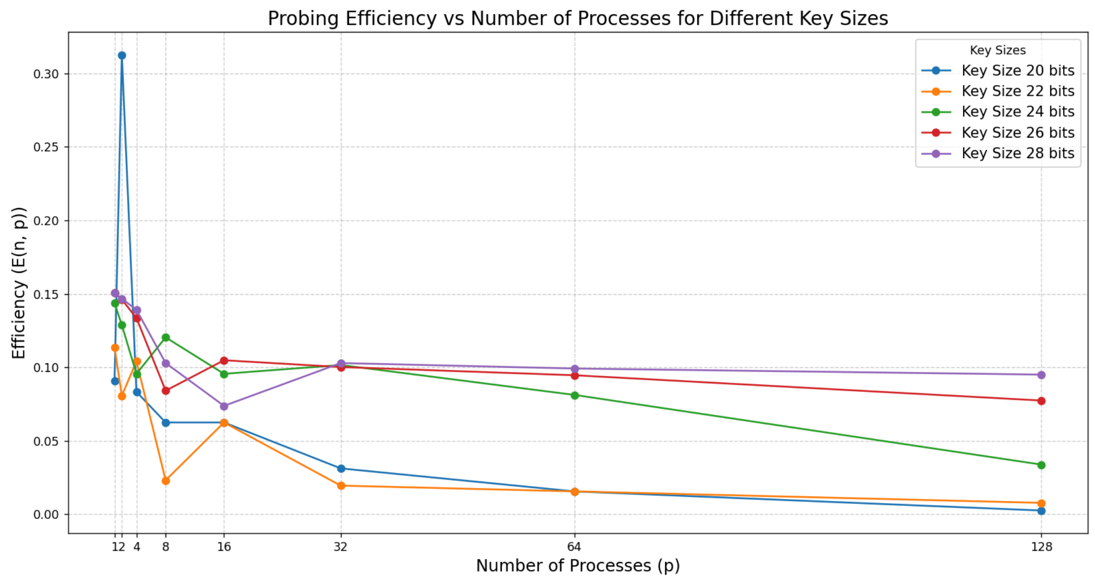


ii. Efficiency plot:

The graph shows the relationship between efficiency E(n,p) and the number of processes p for various key sizes. Efficiency decreases as the number of processes increases, with smaller key sizes (e.g., 20 bits) experiencing a faster decline due to parallelization overhead. Larger key sizes (e.g., 28 bits) maintain higher efficiency, making them more suitable for parallelization. Efficiency peaks when using fewer processes (1-8), beyond which diminishing returns are observed. The graph again highlights the effect of parallel overhead on small enough inputs.



The following diagram represents the probing efficiency



e. Memory bottleneck :

The memory required by our parallelised code expands exponentially with the bit size. The significant uses of memory are the following:

Dictionary array total: 32 + 64 = 96 bits per slot, 2^n search space, 1.125 * 2^n slots. So 108 * 2^n bits

Buffer per process: 64 + 64 = 128 bits per slot, BUFFER_SIZE (= 1000) slots, WORLD_SIZE buffers, 4 types of buffer (write_send, write_recv, read_send, read_recv). So 512000 * WORLD_RANK bits per process

We then have the following memory estimates for 128 processes:

n	Memory (total)	Memory (per process with 128 world size)
20	514MB	4.01MB
21	527MB	4.11MB
22	554MB	4.33MB
23	608MB	4.75MB
24	716MB	5.59MB
25	932MB	7.28MB
26	1.36MB	10.7MB
27	2.17GB	17.5MB
28	3.86GB	30.9MB
29	7.24GB	57.9MB
30	14.0GB	112MB
31	27.5GB	220MB
32	54.5GB	436MB
33	109GB	868MB
34	217GB	1.69GB
35	433GB	3.38GB
36	865GB	6.75GB

At a point the approach becomes feasible only with a large number of processors. Because each machine at Nancy (18 cores) had 96GB of memory, this starts becoming the limiting factor, however we did not reach this point.