PPAR Project

Academic Year 2024/2025

Nhung Kieu Nguyen

Student ID: 21417723

Assia Mastor

Student ID: 28602563

Direct Meet-in-the-Middle Attack

1 Introduction

This project aims to solve a cryptographic problem where we need to find a pair (x, y) such that $f(x) = g(y)$ and $\pi(x, y) = 1$. We use a "meet-in-the-middle" attack, which is more efficient than brute-force search by reducing the time complexity. The provided C program performs this task, and the goal is to optimize the algorithm, particularly through parallelization techniques. The objective is to find a collision for the largest possible input size, while measuring the performance of the implementation.

2 Resource Selection

To execute our code we used this command to reserve our nodes:

```
oarsub -l nodes=10,walltime=04:00:00 -p ecotype -I
```

One of the main advantages of `ecotype` is its high availability. With its 48 nodes, it provides substantial capacity and is often less utilized than other more specialized or newer clusters (like gros) . This availability allowed for quick resource reservations without long waiting times, which was particularly beneficial for iterative tasks and experiments requiring frequent reservations.

But why ecotype and not another ?

- It uses `Intel Xeon E5-2630L v4` processors (10 cores per CPU, 2 CPUs per node), offering good performance for the parallelization.
- It has **128 GiB of RAM** per node, which is well-suited execution with a large n
- The **2 x 10 Gbps** network on most nodes ensures fast communication between processes

3 Optimizations and Parallelization

The main goal of this project was to reduce execution time and optimize memory usage by leveraging modern parallel computing architectures. To achieve this, two main approaches were combined:

- **MPI** to distribute the computations across multiple nodes.
- **OpenMP** to parallelize computations within each node.

Additional optimizations, such as SIMD vectorization, were also implemented.

3.1 Code Optimization with MPI and OpenMP Parallelization

3.1.1 Parallelization with MPI

Golden Search Parallelization

The objective is to distribute the workload evenly across all processes, ensuring that each process independently handles a portion of the problem while minimizing communication overhead.

To achieve this, we implemented a sharding approach where each process is responsible for only a specific subset of values. Specifically, each process handles keys that meet a criterion based on their least significant bits. The number of bits required to distinguish between processes is calculated using the base-2 logarithm of the total number of processes (`num_processes`). For instance, with 8 processes, 3 bits are used to distribute the keys. Consequently, the total number of processes must be a power of 2.

Each process is assigned a range of keys based on its rank. For example, a process with rank 3 will handle all keys whose three least significant bits are `011`. This method ensures a balanced workload across all processes.

To further optimize memory usage, we decided to split the search space by parity. We first process all even values, followed by all odd values. This approach significantly reduces the memory footprint.

Each process inserts only the keys that meet the criteria into its local dictionary. It then performs a local search by applying the criteria to the `g(y)` values and looking them up in its dictionary. This strategy avoids the overhead of searching through all values, thereby saving time.

The results are stored in local arrays (`k1` and `k2`), which are later aggregated by the root process in the main program. The function returns the number of solutions found locally by each process.

Main Program Execution

First, we allocate memory for the arrays $k1$ and $k2$, which will store the results found by each process. If the memory allocation fails, an error message is displayed, and the program stops using `MPI_Abort()`. Afterward, we call the `golden_claw_search_mpi()` function. This function runs the parallel computation on each process, and the results are stored in the $k1$ and $k2$ arrays for each process.

Once the computation is done, we need to gather the results. The first step is to collect the number of solutions found by each process. This is done using `MPI_Gather()`, where each process sends its local solution count (`local_nkey`) to process 0. Process 0 then calculates the total number of solutions and determines the displacement values to place the results in the final arrays correctly. Next, the actual solutions in $k1$ and $k2$ are gathered into two global arrays, `all_k1` and `all_k2`, using `MPI_Gatherv()`.

Finally, process 0 processes and prints the gathered results. For each solution, we check if it meets the required conditions using the `assert()` function. If the solution is valid, it is printed, confirming that it has been successfully verified.

3.1.2 Parallelization with OpenMP

Loop Parallelization We utilized OpenMP to parallelize loops in key parts of the program. Specifically, we applied `#pragma omp for` to distribute the tasks of dictionary insertion among the available threads. Similarly, the dictionary search steps were also parallelized, with each thread handling a portion of the data in parallel. This ensured optimal utilization of the available resources.

Reduction of Thread Conflicts To avoid thread conflicts during simultaneous access to shared data, synchronization mechanisms were implemented. During the insertion of elements into the dictionary, this operation was encapsulated within a critical section (`#pragma omp critical`), ensuring that only one thread at a time could perform the operation. Additionally, the results of the searches, which are stored in global arrays, were inserted within a critical section to maintain data integrity. Finally, a reduction was used on the `ncandidates` variable, shared among threads, to ensure a consistent and conflict-free update of the accumulated values.

3.1.3 Advanced Optimizations

In addition to the basic parallelization with MPI and OpenMP, we applied several optimizations to further enhance performance.

SIMD Vectorization In the code, SIMD is applied in the loop that processes the candidate pairs in the `golden_claw_search_mpi` function. Specifically, the `#pragma omp simd` directive is used in the loop where we check if the key-value pairs are valid by calling the `is_good_pair` function. This allows the compiler to vectorize the loop and process multiple iterations simultaneously, thereby improving performance.

Data Prefetching We used the `__builtin_prefetch` function to prefetch data into memory before it is needed. This reduces data access latency and optimizes performance.

Profiling and Performance Analysis To further optimize the code, we utilized profiling data generation through the `-pg` compilation option. This option generates detailed information about the program's execution, which can then be analyzed using tools such as `gprof`.

After each execution, the resulting `gmon.out` file was analyzed to identify:

- The most time-consuming functions.
- Code sections exhibiting bottlenecks.

- Frequently called functions, enabling targeted optimizations.

Through this analysis, we were able to prioritize our efforts on the slowest sections of the code, leading to targeted improvements and significant performance gains.

3.1.4 Compilation Optimization

In addition to code optimizations, we aimed to maximize performance during the compilation phase. To do this, we used advanced compilation options with `mpicc`, the MPI compiler. Here are the main options used:

```
mpicc -O3 -ftree-vectorize -march=native -fopenmp -pg -o mitm mitm.c -lm
```

- `-O3` : enables high-level optimizations, such as function inlining and reducing unnecessary memory accesses.
- `-ftree-vectorize` : allows automatic vectorization of the code, improving performance on modern architectures.
- `-march=native` : generates code optimized for the processor architecture on which the compilation takes place, enabling the use of processor-specific instructions.
- `-pg` : generates profiling information to analyze performance after execution.

These compilation options allowed us to leverage the system's hardware and software capabilities to maximize the program's efficiency.

4 Experimental Results

4.1 Execution Commands

Below are the execution commands used for different values of n:

```
mpiexec -n 2 ./mitm --n 11 --C0 83649243f5f3b1ec --C1 d11ddbcb5fb40ee1
mpiexec -n 2 ./mitm --n 12 --C0 9752e6e08295f952 --C1 c1107f6e7b9e680e
mpiexec -n 2 ./mitm --n 13 --C0 91ad21220294f3cd --C1 6da5e588ca006480
mpiexec -n 8 ./mitm --n 20 --C0 ad50aadf0a066d23 --C1 224e747fa59d03b5
mpiexec -n 8 ./mitm --n 25 --C0 fe92bbe84aaeb1b3 --C1 45cdd6952dfa2cc8
mpiexec -n 8 ./mitm --n 28 --C0 4ef4f74e47903cd1 --C1 aff400ced6e0c1a2
mpiexec -n 8 ./mitm --n 30 --C0 b35de13b51802ec6 --C1 1fae4ee8767747ef
mpiexec -n 8 ./mitm --n 31 --C0 11f09a4c5a9e6279 --C1 66654d8de55b7162
mpiexec -n 8 ./mitm --n 32 --C0 9bd576fa6fe18df8 --C1 e3bc8b2f4ea88b96
mpiexec -n 16 ./mitm --n 33 --C0 41aa0c7205358a1a --C1 0272ea126e047507
mpiexec -machinefile $OAR_NODEFILE -n 64 ./mitm --n 34 --C0 3
    ac9c956adbec01a --C1 73b85c64dd305ea2
mpiexec -machinefile $OAR_NODEFILE -n 64 ./mitm --n 35 --C0 030632
    fbb21252b9 --C1 27c29cb58821cf3e
```

4.2 Results

The experimental results are summarized in the table below (with the username asmastor) :

Value of n	**Sequential Time (s)**	**Parallel Time (s)**	**Number of Processes**	**Command**	**Solution**
11	0.0032359	0.000675	2	–C0 83649243f5f3b1ec –C1 d11ddbcb5fb40ee1	(38b, 38f)
12	0.0064299	0.001175	2	–C0 9752e6e08295f952 –C1 c1107f6e7b9e680e	(78b, 529)
13	0.0133359	0.001725	4	–C0 91ad21220294f3cd –C1 6da5e588ca006480	(e87, 1f9a)
20	1.8927109	0.05175	8	–C0 ad50aadf0a066d23 –C1 224e747fa59d03b5	(9089f, 1367f)
25	69.372874	2.53000	8	–C0 fe92bbe84aaeb1b3 –C1 45cdd6952dfa2cc8	(8bea65, 1d9c402)
28	588.339061	24.85000	8	–C0 4ef4f74e47903cd1 –C1 aff400ced6e0c1a2	(ef1dc64, a905251)
30	1643.4224951	102.10000	8	–C0 b35de13b51802ec6 –C1 1fae4ee8767747ef	(28e731f5, 2d77dba7)
31	3314.829761	206.35000	8	–C0 11f09a4c5a9e6279 –C1 66654d8de55b7162	(6a7d4e21, 887d84b)
32	Impossible	418.75000	8	–C0 9bd576fa6fe18df8 –C1 e3bc8b2f4ea88b96	(8423d250, b9f35490)
33	Impossible	1471.70172	16	–C0 41aa0c7205358a1a –C1 0272ea126e047507	(32f491e2, 145e8ff88)
34	Impossible	3872.88	64	–C0 3ac9c956adbec01a –C1 73b85c64dd305ea2	(2e09b8c19, 2e4dd6928)

Table 1: Performance comparison for different values of n with corresponding commands and solutions.

4.3 Analysis

4.3.1 Limits of Sequential Execution

The results show that it becomes impossible to execute the algorithm sequentially beyond a certain problem size ($n \geq 32$). This is due to the exponential growth of the hash table, which exceeds the available memory capacity. When memory is insufficient, the execution terminates prematurely with the message `killed`. This limitation highlights the necessity of parallelization to divide the computational load and reduce memory requirements per process.

4.3.2 Speedup through Parallelization

For values of n where sequential execution remains feasible, parallelization demonstrates remarkable performance improvements. For example, at $n = 25$, the sequential execution time of 69.37 s is reduced to just 2.53 s using 8 processes, achieving a speedup of approximately 27x. Similarly, for $n = 28$, parallel execution reduces the time from 588.34 s to 24.85 s, showcasing the efficiency of distributing the workload across multiple processes.

The results also highlight the critical role of parallelization in enabling the execution of larger problem sizes that are otherwise infeasible sequentially due to time or memory constraints. For instance, cases such as $n = 33$ and $n = 34$, which are impossible to handle sequentially, become manageable with 16 and 64 processes, respectively. This underscores the scalability of the approach for tackling increasingly complex computations.

However, the observed speedup is not perfectly linear as the number of processes increases. We can attributed it to:

- **The communication overhead**: As the number of processes grows, the cost of exchanging data between processes becomes more significant, particularly for larger problem sizes.
- **Diminished workload per process**: With more processes, the workload assigned to each process decreases, potentially leading to underutilization of computational resources.
- **Load balancing challenges**: Uneven distribution of the search space across processes may result in some processes completing their tasks earlier than others, leaving resources idle.

Overall, these results demonstrate the effectiveness of parallelization in reducing execution times and extending the solvable range of problem sizes. However, they also emphasize the importance of optimizing process communication and workload distribution to maximize performance as the scale of parallelization increases.

For n=35, the execution completed successfully without any warnings or errors. However, no solutions were identified across multiple nodes (tested on both gros and ecotype clusters). This outcome highlights the inherent complexity of the problem and the challenges associated with efficiently distributing and exploring the search space in a parallelized environment. The results underscore the importance of precise partitioning and load balancing when dealing with highly combinatorial problems of this nature.

For n=36, the computational demands increased significantly, as the size of the search space exceeded the combined memory capacity of the reserved nodes. While additional resources might have mitigated this limitation, cluster availability constraints at the time of testing restricted our ability to reserve more nodes.

ALso, despite various optimization attempts and modifications to the code, some bottlenecks remained difficult to improve. First and foremost, the search with the g variable represented a major performance bottleneck, closely followed by the insertion process. The search, in particular, was the most time-consuming part of the computation. Even after applying several optimization techniques, it continued to take up 43.35% of the total execution time.

4.3.3 Comparison Across Clusters and Process Counts

Initially, we obtained precise results using the `écotype` cluster. However, we decided to extend our analysis by comparing the performance across different clusters and varying process counts.

To achieve this, we performed executions on Chiclet, Ecotype, Nova and Gros:

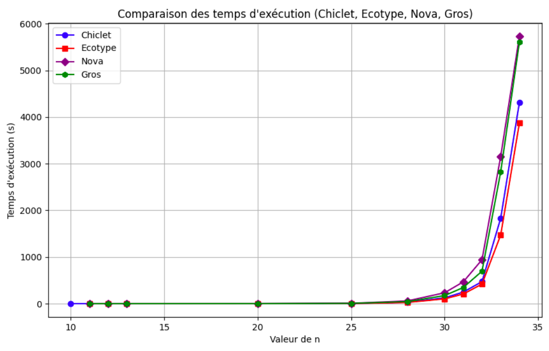

Figure 1: Execution on different cluster

Looking at the results, it's clear that the Ecotype and Gros clusters perform much better, especially for larger values of n like $n = 31$ and $n = 35$. This is mainly because their hardware is much better suited for intensive computations.

For example, Ecotype has Intel Xeon E5-2630L v4 processors with 10 cores per CPU, 128 GiB of memory, and a network bandwidth of up to 2x10 Gbps. Gros is even more powerful, with Intel Xeon Gold 5220 processors (18 cores per CPU), 96 GiB of memory, and 2x25 Gbps network bandwidth. These features make them very efficient at handling heavy workloads.

On the other hand, Chiclet and Nova don't perform as well. For instance, Nova only has 8 cores per CPU and 64 GiB of memory, which limits its ability to handle demanding tasks.

Interestingly, we noticed that Ecotype was consistently faster than Gros, despite Gros having better hardware on paper. To understand this better, we decided to investigate whether the number of processes used could explain why Ecotype outperformed Gros in our tests.

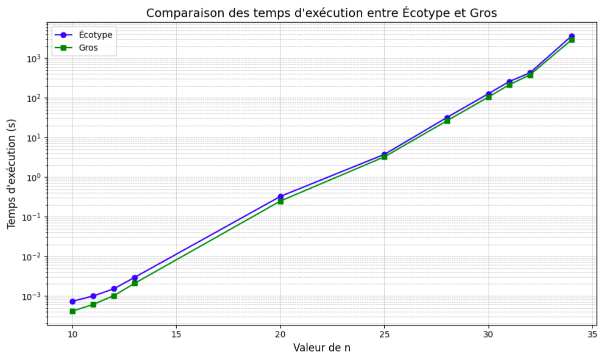

Figure 2: Execution for less processes than the first executions

We ran additional experiments by varying the number of processes. Specifically, we reduced the number of processes from 4 to 2 and from 8 to 4. The results clearly showed that Gros outperformed Ecotype in all cases, confirming that Gros's superior hardware provides a consistent advantage, especially when the number of processes is adjusted.

This suggests that the choice of the number of processes is crucial in avoiding overhead and ensuring optimal performance. While Ecotype showed some advantages with fewer processes, Gros's higher core count and faster network bandwidth allow it to handle larger workloads and parallelism more efficiently. Thus, selecting the right number of processes is key to fully leveraging the capabilities of the hardware and achieving the best performance.

5 Challenges Faced

During the course of the project, we encountered several significant challenges that impacted our progress and required careful consideration to address.

The first major issue was related to segmentation faults, which occurred unpredictably and were difficult to trace. This was primarily because numerous changes had been made to the code during the optimization process, making it challenging to pinpoint the exact source of the problem. Unfortunately, we realized the extent of this issue too late. After spending several days investigating the problem without success, we decided to restart the project from scratch. Subsequently, we attempted to reintroduce the optimizations incrementally. However, we were unable to fully reintegrate all optimizations without risking further segmentation faults. Consequently, we chose to rely on simpler and more stable optimizations.

Another challenge stemmed from resource sharing on the cluster. At times, we faced difficulties in securing clusters with a sufficient number of available slots, which delayed the execution and testing of our code. This limitation occasionally hindered our ability to progress efficiently and required us to adapt our scheduling to accommodate the availability of resources.

Despite these challenges, we managed to implement a functional and efficient solution while maintaining the stability of our codebase.

All the versions of the project, along with detailed updates and changes, can be found on our GitHub repository `https://github.com/sisaladiva/PPAR_Project`, which provides a transparent history of the work completed.