Report for the final project of SPM course

Parallelizing discrete logarithm computation over groups of smooth order

Giorgio Dell'Immagine, July 2023

1 Introduction

1.1 Discrete logarithm problem algorithms

The discrete logarithm problem[3] is the problem of finding in group G with order n an element x such that $g^x = b$ given g and b. No efficient algorithm is known to compute the discrete logarithm in the general case and it is thus considered an intractable problem. This form the security basis for modern cryptographic primitives, like the Diffie-Hellman key exchange (which uses the multiplicative group $\mathbb{Z}_p^*$), and elliptic curve cryptography (which uses elliptic curves groups).
The naive algorithm just tries all possible values of x and has a time complexity of $\mathcal{O}(n)$, thus exponential in the dimension of n. The baby-step giant-step algorithm[1] improves on this complexity by using a *meet in the middle* approach, lowering the time complexity to $\mathcal{O}(\sqrt{n})$, but using $\mathcal{O}(\sqrt{n})$ space.
In some special cases, however, the discrete logarithm can be computed faster. For example, if the order of the group G is a smooth number (i.e., its largest prime factor is relatively small) the Pohlig-Hellman algorithm[5] can be used, which applies a *divide and conquer* approach. The idea behind this algorithm is to reduce the computation of the discrete logarithm to the sub-groups of G, and then recombine the results exploiting the solution given by the Chinese Remainder Theorem. To compute the discrete logarithms in the sub-groups, the baby-set giant-step approach can be used. The time complexity of this algorithm can be in practice approximated to $\mathcal{O}(\sqrt{p_m})$ where p_m is the largest prime factor of the order n. The space occupancy improves as well to $\mathcal{O}(\sqrt{p_m})$.
When the order n is not smooth, the Pohlig-Hellman algorithm does not improve over other approaches, so for secure cryptographic purposes a group with at least one big prime factor of the order is chosen, in order to have $p_m = \Theta(n)$.

1.2 Project objectives

Parallel implementations of other discrete logarithm algorithms have been proposed (e.g., in [2] in the context of elliptic curves) to try pushing the group order bit length of feasible computation.
The objective of the project is providing sequential and parallel implementations of the Pohlig-Hellman and baby-step giant-step algorithms to compute the discrete logarithm efficiently over $\mathbb{Z}_p^*$, assuming that the order $p-1$ is smooth. The program takes in input the base g, the result b, the prime p and the factorization of $p-1$, and shall produce in output the integer x such that $g^x = b \mod p$.

1.3 Methodology

The problems and algorithms have been first analyzed theoretically, then a straight-forward sequential implementation has been created, to locate and analyse the major bottlenecks and performance hot spots. Therefore a parallel implementation has been devised, both using native `c++` threads and `FastFlow`. Finally the implementations have been evaluated empirically by performing various tests.
The typical use case of this application would be to compute one instance of the discrete logarithm, therefore we are interested mainly in end-to-end completion time. For this reason all the time measurements also include the time it takes to create and manage the workers.

2 Problem analysis

2.1 Analysis of baby-step giant-step algorithm

The analysis and optimization of baby-step giant-step is crucial for the whole Pohlig-Hellman algorithm, since it is used as a subroutine multiple times, and it is by far the most expensive computation step in the whole algorithm.
Taking a look at the algorithm (of which a pseudocode is given in appendix A) we can see that it is composed of two loops of equal number of iterations: the first loop builds a table, and the second loop looks for values in that table. One first simple observation is that we cannot start computing the second loop before having built the whole table, because there is a data dependency between the two sections.
Since all the operations we make in the loops are relatively fast (multiplications and modulo reductions) we should expect that the majority of computation time is spent in managing the elements of the table, mostly in allocating and inserting them in the table. This is confirmed empirically: profiling a straight forward sequential c++ implementation exploiting `std::unordered_map` reveals that over 25% of the total time is spent allocating the elements of the table, and another 20% is spent in deleting the elements from the heap. This is also worsen by the fact that we are allocating arbitrary precision integers, which can be quite large.
We can make further observations on the structure of the algorithm:

- the second loop is trivially parallelizable, since the table is read-only, so we can split the space $(0 \dots m-1)$ into nw chunks and computing the loop body in parallel,
- the random memory access pattern, both in item insertion and membership check, is not really avoidable, since the function $i \mapsto g^i \mod p$ yields unpredictable results, and it is well-known that they are evenly spread on the $(1 \dots p-1)$ space.[1] Therefore, we can never achieve good cache locality, since both insertions and membership checks are made using the results of this function as the index.

A few parallel approaches were considered: first, the simplest one is to construct the table sequentially and then perform the search in parallel. For the sake of this argument, let's assume that both loops have equal run time. This approach has a theoretical speedup limit of 2, since the sequential part is one half of the time of total computation. In reality this is even worse, because the first loop has greater run time of the second, so the theoretical speedup is actually lower.
Since the total run time is dominated by table insertions and not by other computations, having a single shared table protected by a mutex would effectively render the insertions sequential, making this solution equivalent to the one above.
Another solution that has been considered is to build in parallel nw tables, splitting the input space $(0 \dots m-1)$ into nw chunks, then merging the tables and searching in the resulting union table. This also is not ideal, since the merging of the tables has to be done sequentially (we cannot employ a tree parallel reduction pattern, since we are not reducing to a single value, but we are constructing the union table; in any case this has a lower bound of the sequential processing of order of m items).
We could avoid constructing the union table by modifying the search algorithm and search in the nw tables. Since, however, the search loop run time is dominated by the membership lookup, this has the effect of multiplying by nw the run time of one search iteration. The run time of the whole parallel search is therefore $\mathcal{O}(\frac{m}{nw} \cdot nw) = \mathcal{O}(m)$, again giving us a theoretical speedup of 2 for reasons analogous of those above.

2.2 Proposed parallel approach for baby-step giant-step

We can make some observations on the structure of the problem: the key $g^i \mod p$ is not really needed to be stored, since it can be reconstructed very quickly from i and g, using binary exponentiation. Also, the keys are unique, i.e., there are no $g^i = g^j$ for $i \neq j$ (because g is a generator), so one very efficient approach would be to use a direct access table, and since there are no collisions, it could be parallelized easily. The only problem is that this table would have to be huge, and even for relatively small examples it would even exceed the addressable space of x86. As a concrete example, let's take $\mathbb{Z}_p^*$ where p has 64 bits. The discrete logarithm problem is theoretically feasible in manageable times,[2] as it requires in the best case 2^{32} 32-bit words of memory (around 17Gb), and order of 2^{32} operations. But the direct access table would have to be 2^{64} 32-bit words long.
A better approach, that still exploits some of the benefits of a direct address table, is open addressing. This was also partly inspired by Chapter 5 of [7] and by [4]. We construct the table using an array of $(1/\alpha) \cdot m$ items (where α is the load factor) and resolve collisions using either linear probing, quadratic probing or double hashing. This approach has a few advantages compared to conventional hash tables and direct access tables:

[1] This is one of the main reasons why this function is very useful in cryptography.
[2] Since we are in the domain of cryptography, as "manageable times" we can consider even a few hours of computation.

- insertions can be fully parallelized without any locks, exploiting atomic variables and the compare-and-swap atomic operation.
- It uses memory proportional to m and not p.
- It avoids multiple allocations on the heap, only one allocation for the array.
- The initialization, the insertion and the lookup in the table can all be performed in parallel, and the only synchronization needed is in using atomic variables and barriers between stages.

In experimental result it has been found that even the sequential implementation of this approach (using `uint64_t` instead of atomics) is almost twice as fast as the original approach using `std::unordered_map`, so this has been selected as the fastest sequential implementation for speedup calculation.

2.3 Analysis of Pohlig-Hellman

The Pohlig-Hellman algorithm is composed of two sub-algorithms (algorithm 2 and algorithm 3, both described in appendix A). The analysis in this case is much simpler than baby-step giant-step: algorithm 2 cannot be parallelized, since there result is calculated incrementally, and there is data dependency between the loop iterations. On the other hand, algorithm 3 can be easily parallelized, since at each iteration the computation depends only on the index. The final algorithm to solve the simultaneous congruences (using the constructive method of the Chinese remainder Theorem) cannot be parallelized, but that's not a big problem since the run time is very small compared to the other procedures. In general, we can state that the operations made by Pohlig-Hellman, excluding the calls to baby-step giant-step are very cheap, and are done only to generate parameters for baby-step giant-step computation and recombine their outputs.
Since the loop of algorithm 2 has to be computed sequentially, the minimum run time of the whole algorithm is determined by the maximum time of the calls to such algorithm. Calling T_i the time it takes to compute the i-th iteration of Algorithm 3, the maximum theoretical speedup achievable is

$$\frac{T_{seq}}{\max_i\{T_i\}}.$$

Now, calling $T_{bsgs}(o)$ the time it takes to compute the baby-step giant-step algorithm on groups of order o, we can say that the run time of algorithm 2 is order of $e_i \cdot T_{bsgs}(p_i)$ for every term $p_i^{e_i}$ in the factorization of the order of the input group of Pohlig-Hellman. We can write that

$$T_{seq} = \sum_i e_i \cdot T_{bsgs}(p_i)$$

so the theoretical speedup limit is

$$\frac{\sum_i e_i \cdot T_{bsgs}(p_i)}{\max_i e_i \cdot T_{bsgs}(p_i)}.$$

Taking into consideration also the speedup of the baby-step giant-step sub-routines, the maximum speedup achievable for Pohlig-Hellman with parallel baby-step giant-step computation is the product of the two speedups, since each iteration of Pohlig-Hellman will have speedup approximately equal to the one of baby-step giant-step.

3 Implementation details

The two algorithms have been implemented both using native `c++` threads and using `FastFlow`. The two implementations make use of the publicly available BigNum library `libgmp`[3] to handle arbitrary precision integers. Those are needed because interesting and "real-world" sizes of the groups order have a big bit length (order of magnitude of 10^3 [6]), and thus this library is used throughout the implementations.
The rest of this section discusses some implementation details. Appendix C discusses how to compile and run the implementation and how to reproduce the experiments.

3.1 Lock-free hash table

The main data structure used by the implementations is the lock-free hash table. This is implemented as a pre-allocated `std::vector` of `atomic_uint64_t`. Storing 64 bit unsigned integers is more than enough to compute all reasonable-sized inputs, since we need to store in the table the numbers $(0 \dots m-1)$. With this implementation we can handle the maximum case of holding $2^{64} - 1$ 64 bits items, and that is way over our memory capacity.[4]

[3] The library's home page can be found at `https://gmplib.org/`
[4] A table of that size for example would not even be addressable in `x86`, since that architecture has "only" 48-bits addressing.

The table size, and in particular the load factor of the table, are one of the major parameters that affect the performance. On one hand smaller load factor keeps the data structure more compact, resulting is fewer cache misses, but on the other hand it increases the probability of collisions, resulting in slower insertions and look-ups. Appendix B discusses empirically the impact of the load factor on performance.
Another observation is that we do not really need a general hash function, because at each step of insertion we are evaluating the function $i \mapsto g^i$, that produces an "evenly spread" output on the $(1 \dots p-1)$ space. We can then use the very simple and fast hash function

$$h(x) = x \mod (\text{size of the table})$$

which is effective in practice.

3.2 Baby-step giant-step parallel implementation

The baby-step giant-step algorithm have been implemented in three data-parallel stages, each time splitting the work into chunks among the threads. The stages are: table initialization, in which every element of the table is initialized to a sentinel value, table construction and table search.
The threads are synchronized using two memory barriers between the stages, to ensure that the computation of one stage finishes fully before starting computing the next one.
During the last phase, a termination flag of type `atomic_bool` has been used to terminate as soon as the first thread finds a match in the table. Basically it is initialized to false, and at each iteration each thread checks if the flag is set to true, in which case it terminates. When a thread finds a match, it sets the flag, causing the termination of all the threads. Also, the result variable is protected by a `mutex`, since in some corner cases it may happen that multiple threads find concurrently a match in the table; however it is locked only when a match is found, i.e., when a result is calculated, so it does not add significant overhead to the whole computation.

3.3 Thread pool for Pohlig-Hellman

Parallel Pohlig-Hellman has been implemented using a thread pool. Tasks are taken from a shared queue protected by a mutex, and are in the form of $\langle i, p_i, e_i \rangle$, where $p_i^{e_i}$ is the i-th term of the factorization of the order. Each worker takes a task from the queue, executes the Pohlig-Hellman on groups of prime power order, and inserts into a shared array at position i the result. This output array is not protected, since the insertion positions are all different for each task.
Ultimately, these workers will end up calling the parallel implementation of baby-step giant-step, spawning other workers to perform the computation as described above. In this case the master is not considered as adding one to the parallelism degree, because it simply spawns the workers and immediately wait for them to return.

3.4 FastFlow implementation

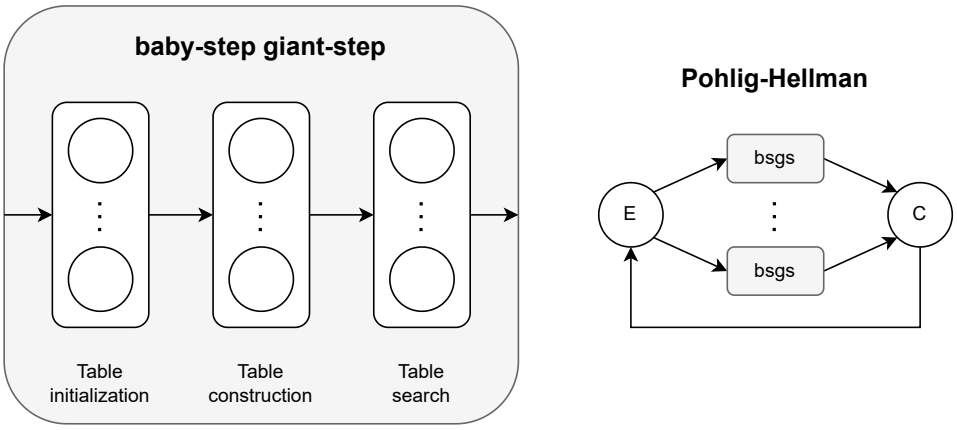

Figure 1: Structure of `FastFlow` blocks of the implementation. The baby-step giant-step algorithm is implemented as a pipeline of three `ff_Map`, while Pohlig-Hellman is implemented through a farm with a feedback-channel, the workers are exactly the blocks that compute baby-step giant-step.

The implementation of the baby-step giant-step algorithm maps quite naturally to three *parallel for* patterns, for table initialization, construction and search. This has been implemented in `FastFlow` using a pipeline of three

FF_Map stages, exploiting the provided parallel_for method to split the computation among workers. One thing to notice is that for the table construction and search stages, theoretically the computation is dependent only on the index, in practice it is far better to compute into large chunks, since for each i we need to compute some g^i. By using g and i this requires to run the binary exponentiation algorithm, which performs a few tens of operations, both additions, multiplications and modulo reduction. Having computed g^i we can compute efficiently the value at next iteration $g^{i+1} = g^i \cdot g \mod p$ by performing only one multiplication and one modulo reduction. Therefore, the parallel_for is used with a high chunk size.
To be able to perform the Pohlig-Hellman computation, these baby-step giant-step blocks have been arranged in a farm, with an emitter and a collector, and a feedback channel. The emitter first parses the task, creates some objects that keep the status of the computation of Pohlig-Hellman on prime power orders, created the task of baby-step giant-step representing the first iteration and sends them to the workers. The collector takes the results and

- if the competed baby-step giant-step task is the last one to be computed in the loop, then store it in the results array,
- if there are other iterations of baby-step giant-step to be computed, then calculate the next task and send it to the emitter via the feedback channel,
- if all the tasks have been completed, combine them into the overall result, and terminate the computation sending EOS over the feedback channel.

The collector when it receives a task from the feedback channel it simply forwards it to the farm. The termination have been addressed implementing the eosnotify method on the emitter, broadcasting EOS to all the farm workers. Since the overall computation is essentially dominated by data parallel patterns, the queues are only used to distribute and pass around tasks to be computed, and this is done very infrequently w.r.t. the overall computation. For this reason, all queues have been set to blocking mode at compile time, to reduce the overhead of active wait for tasks.

4 Experimental results

The experiments have been run on the physical NUMA machine provided, that has 32 cores and 64 hardware contexts. All input data has been generated using some scripts that can be found in the scripts folder. All data is taken on an average of 10 inputs, since the run-time of the algorithms is input dependent (mainly because the search phase terminates once a result is found).

4.1 Baby-step giant-step

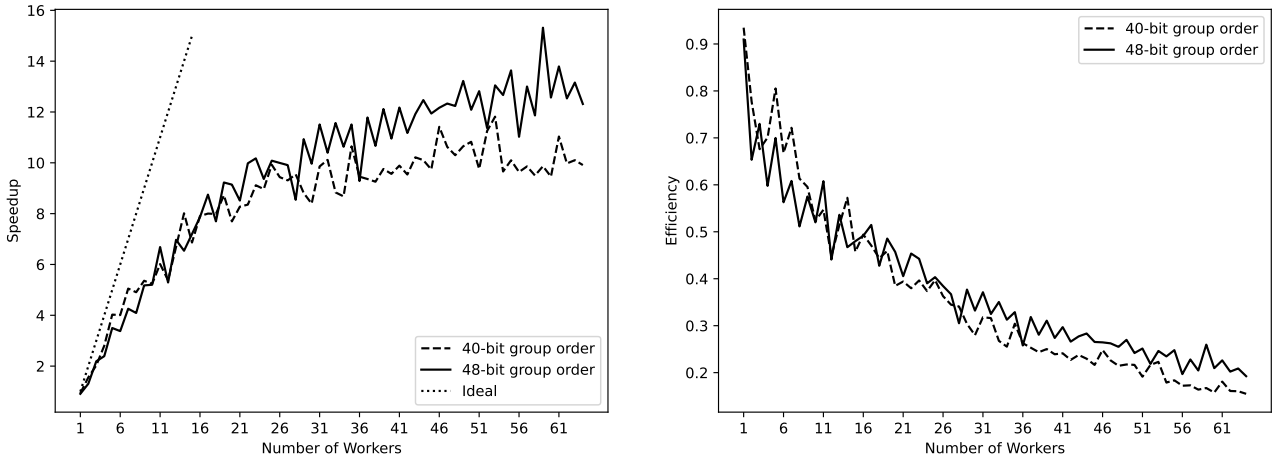

Figure 2: Speedup and efficiency for baby-step giant-step implementation.

Figure 2 shows the speedup and efficiency of the baby-step giant-step implementation using c++ threads. The experiments have been run on two sets of inputs, each one composed of 10 discrete log problems on groups of 40 and 48 bits. Run times have been averaged for each set. We can make some considerations on these results:

- the sequential version is very close in terms of run time to the version with degree of parallelism equal to one. This is because they both employ the same approach using open addressing, differing only in having elements of type `atomic_uint64_t` or `uint64_t`. As we should expect, the performance of using atomics without contention is comparable to not using atomics at all.
- The speedup curve is very far from the ideal one, and the efficiency rapidly drops, this is probably because all computation has to be done on a single shared data structure, and there are a lot of random accesses to the same memory area. However, putting it in perspective also the alternatives described above, in which the theoretical speedup was 2, this approach is far better.
- Even though the machine has only 32 physical cores, this algorithm is able to exploit, although not much, the two hardware contexts of each core, probably because the algorithm makes a lot of random accesses, which means a lot of cache misses that result in waits of the computations.

4.2 Pohlig-Hellman

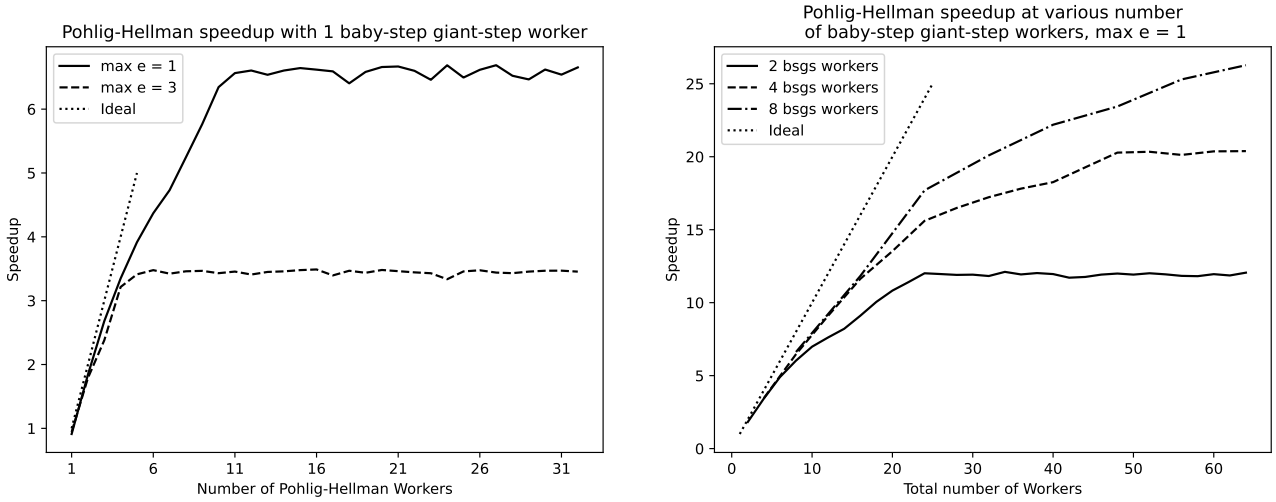

Figure 3: Speedup for Pohlig-Hellman implementation using native c++ threads.

Figure 3 shows two speedup graphs, displaying two notable cases. The graph on the left plots speedup of Pohlig-Hellman implementation having only one baby-step giant-step worker. It is plotted with two sets of inputs, in which the maximum exponent value in the group order factorization are 1 and 3 respectively. All factors have approximately 40 bit. The total number of workers has been calculated as the number of baby-step giant-step workers times the number of Pohlig-Hellman workers.
This graph empirically confirms the theoretical speedup of Pohlig-Hellman discussed above: a higher value of the maximum e found in the factorization will result in a lower speedup limit. Furthermore in this case, the speedup is pretty close to theoretical up until the theoretical limit, which is what we would expect since the tasks to be computed in parallel are completely independent.
The right plot shows the speedup at various number of baby-step giant-step workers. The interesting thing to notice is that having a speedup of x in the baby-step giant-step algorithm "highers" the limit of the speedup by a factor of x. In the graph this is well visible, since with two baby-step giant-step workers the speedup limit is approximately double the one with one worker (in the left graph), and the one with 4 workers is approximately four times that. This once again confirms the theoretical analysis that the overall speedup is the multiplications of the speedup of Pohlig-Hellman with the speedup of baby-step giant-step, taken individually.

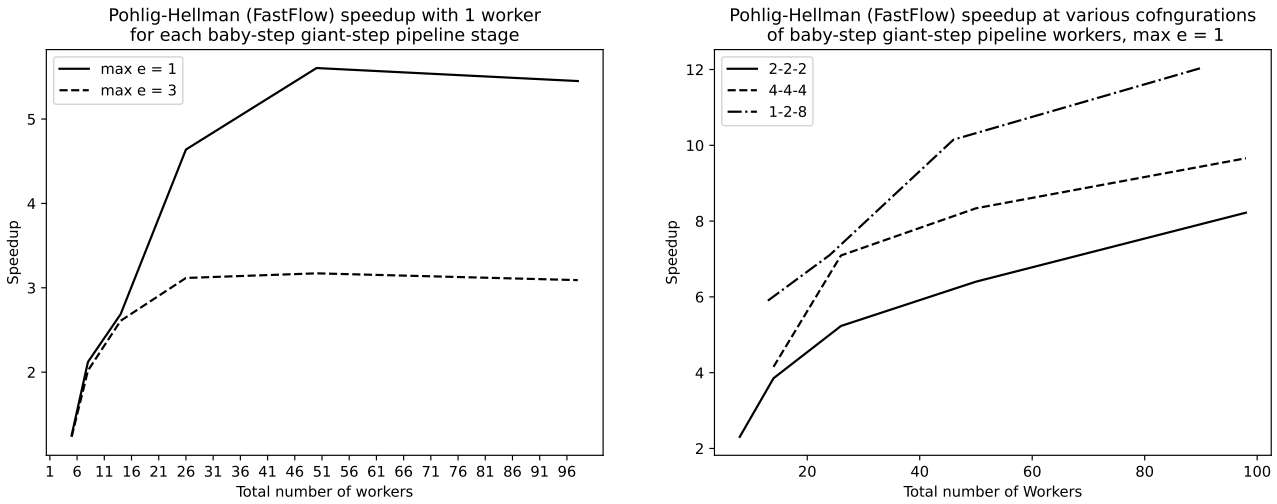

Figure 4: Speedup for Pohlig-Hellman implementation using `FastFlow`.

Figure 4 shows the same graph as above, but tested with the `FastFlow` implementation. The first thing to notice is that the number of workers is in general higher, for example the minimum is 5: one emitter, one collector and three stages for one baby-step giant-step pipeline. In general, it has been taken as the total number of worker the number of `FastFlow` entities, even though for instance the emitter or the collector will participate very little to the computation. This is the reason why it has been compiled with blocking mode set to true.
The results on the left are comparable with the native `c++` implementation, although the speedup limit is a bit lower, probably due to some overhead in the communication between threads.
On the right, it is plotted speedup at different configurations of number of workers in the pipeline. Through some manual search and measuring the times of computation of the various stages, it has been found that the configuration `1-2-8` (meaning 1 worker for initialization, 2 for construction and 8 for search) was the one that performed better, although it is still slower than the native `c++` threads version. This could be caused by overhead of communication between threads and overhead in the implicit barrier between multiple `ff_Map` (which could be a bit higher than using a plain `std::barrier`).

5 Conclusions

In this project it has been shown how it could be possible to parallelize the discrete logarithm computation over groups of smooth order, using parallel versions of baby-step giant-step and Pohlig-Hellman algorithms.
This project, however, has some shortcomings: possible improvements or future work could be to improve the efficiency and maximum speedup of the `FastFlow` version, also conducting a more thorough analysis on the best allocation of workers in the `FastFlow` implementation. This could be done by manual testing, or even better exploiting tools like `RplSh` to support the analysis.

A Pseudocode of algorithms

Algorithm 1 Baby-step giant-step, compute x such that $g^x = b \mod p$, with order of g being n

$m \leftarrow \lceil \sqrt{n} \rceil$
$T \leftarrow \{\}$ ▷ Initialize empty (hash) table
for i in $0 \dots m-1$ **do**
  $T[g^i \mod p] \leftarrow i$ ▷ In real implementation g^i is not recomputed every time, since $g^i = g \cdot g^{i-1} \mod p$
end for
$y \leftarrow b$
for i in $0 \dots m-1$ **do**
  if y in T **then** ▷ At each iteration it holds $y = b \cdot a^{-m \cdot i}$
    return $i \cdot m + T[y] \mod n$
  end if
  $y \leftarrow (y \cdot a^{-m}) \mod p$
end for

Algorithm 2 Pohlig-Hellman on prime power order, compute x such that $g^x = b \mod p$, with order of g being q^e

$x_0 \leftarrow 0$
$\gamma \leftarrow g^{q^{e-1}}$
for k in $0 \dots e-1$ **do**
  $h_k \leftarrow (g^{-x_k} \cdot h)^{p^{e-1-k}} \mod p$
  Using baby-step giant-step, compute d_k s.t. $\gamma^{d_k} = h_k \mod p$, the order of γ is q
  $x_{k+1} \leftarrow x_k + q^k d_k \mod p$
end for
return x_e

Algorithm 3 Pohlig-Hellman, compute x such that $g^x = b \mod p$, with n order of g having factorization $\Pi_{i=0}^{r} q_i^{e_i}$

for q_i, e_i in the factorization of the order **do**
  $o_i \leftarrow q_i^{e_i}$
  $g_i \leftarrow g^{n/o_i}$
  $h_i \leftarrow h^{n/o_i}$
  Using algorithm 2, compute x_i s.t. $g_i^{x_i} = h_i$, the order of g_i is o_i
end for
return x using Chinese remainder theorem, such that $x \equiv x_i \mod q_i^{e_i} \quad \forall i \in [0, r]$

B Impact of load factor on performance

The load factor is an important parameter for the efficiency of the implementation. There are indeed two conflicting forces:

- if the load factor is too high, then the insertions and searches will take a long time, since more collisions will be encountered; in the limit, a load factor of 1 will mean that the insertion of last element will have to traverse potentially the entire table to find a free spot.
- If, on the other hand, the load factor is too low, the table will be bigger and more sparse, and the initialization will take longer.

Figure 5 confirms empirically these statements: at all number of workers a high load factor degrades the performances, but a low load factor degrades the performances only at high number of workers. This is because the initialization phase has a "constant" impact on the run time (each element of the table must be initialized) and with high number of workers means that the computation time will be lower overall, thus the initialization will have a bigger relative impact on the total run time. Based on those results, a default load factor of 1/4.5 has been chosen, being it a trade-off value between space efficiency and performance.

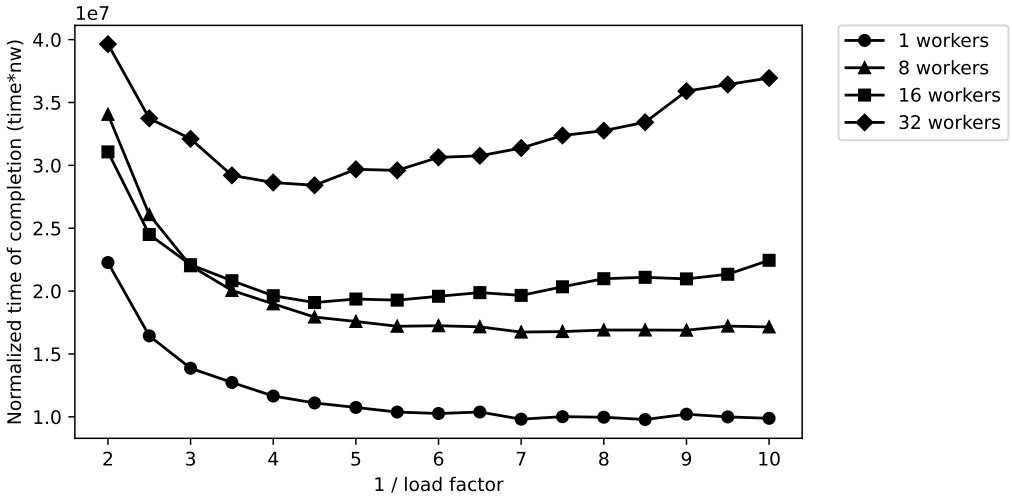

Figure 5: Experimental results of parallel baby-step giant-step implementation on an average of 5 discrete logarithm problems with group order of 48 bits. It is plotted normalized average time of completion relative to the table load factor, at different number of workers.

C Reproducing the experiments

The project is structured as follows. The folder `include` contains the include files, in particular `include/discrete_utils.h` contains some utilities for discrete math, and `include/oatable.hpp` contains the implementation of the open addressing hash table. The folders `src` and `tests` contain the parallel implementation of baby-step giant-step and Pohlig-Hellman using native `c++`, while the folder `ff` contains the implementation using `FastFlow`. The folder `lib` contains the libraries files of `libgmp` that handle arbitrary precision integers. The folder `plots` contains the Python notebook and all data that has been used to generate the plots. The folder `scripts` contains two Python scripts to generate the input files, which are stored in the `input` folder, and three Bash script that run the benchmark on the different implementations.

To compile the code run `make`, this will create three binaries under the folder `bin`. These are respectively the binary for standalone baby-step giant-step, for Pohlig-Hellman using `c++` threads and for Pohlig-Hellman using `FastFlow`. They all take as first command line parameter the filename of the input to be computed, and as subsequent parameters the number of workers. In both baby-step giant-step and Pohlig-Hellman, specifying as number of workers 0 will execute the sequential version, while in baby-step giant-step, specifying as number of workers -1 will run the "old" sequential implementation, exploiting `std::unordered_map`.
In all cases, `libgmp` should be in the loader library path, which can be accomplished by setting the environment variable `LD_LIBRARY_PATH=./lib/`.

References

[1] *Baby-step giant-step - Wikipedia*. URL: `https://en.wikipedia.org/wiki/Baby-step_giant-step`.

[2] Joppe W. Bos et al. "Pollard Rho on the PlayStation 3". In: 2009.

[3] *Discrete logarithm - Wikipedia*. URL: `https://en.wikipedia.org/wiki/Discrete_logarithm`.

[4] *Maximizing Performance with Massively Parallel Hash Maps on GPUs*. URL: `https://developer.nvidia.com/blog/maximizing-performance-with-massively-parallel-hash-maps-on-gpus/`.

[5] *Pohlig-Hellamn algorithm - Wikipedia*. URL: `https://en.wikipedia.org/wiki/Pohlig%E2%80%93Hellman_algorithm`.

[6] *RFC 3526 - More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*. URL: `https://datatracker.ietf.org/doc/html/rfc3526`.

[7] R. Robey and Y. Zamora. *Parallel and High Performance Computing*. Manning, 2021.