Optimal Throughput Bioinformatics

Abstract

The amount of sequenced DNA is increasing quickly. To analyse this increasing amount of data, faster computers help, but also, faster algorithms and data structures are needed. These new methods may be fast in theory, but more importantly, they should be fast in practice.

In this thesis, our goal is to write algorithms that achieve *close-to-optimal throughput*. Ideally, we can prove that no other methods can be more than, say, twice as fast as our method. This has a few implications. Our code should use an algorithm that not only has a good *complexity*, but is also *efficient*. Further, and the focus of this thesis, this means algorithms should be *implemented* optimally. This requires a deep understanding of how modern CPUs execute our code, so that we can help it doing so efficiently.

We strive for optimality of a few different problems. First, we look at the problem of *pairwise alignment*: the task of finding the number of mutations between two DNA sequences. For example, one can count the number of mutations between two strains of the Sars-CoV-2 (COVID) virus. Research on this problem has a long and rich history, spanning over 50 years. We survey this history and introduce a new pairwise aligner, A*PA2, that combines a good, near-linear, complexity with a highly efficient implementation. We then extend these results to *text searching* or *mapping*, where shorter pieces of DNA must be found in longer strings.

Secondly, we consider *minimizers*. On a high level, this is a technique that takes a long DNA sequence and downsamples it to a small fraction of substrings. This is done in such a way that every sufficiently long piece of sequence contains at least one sample. We can then ask the question: what is the smallest number of samples we must select (or: the highest compression ratio we can achieve), while still satisfying this condition. We explore the theory of this problem, and give a new, near-tight, bound on the maximal compression ratio that can be achieved. We also develop minimizer schemes that reach a compression level (*density*) close to this bound.

Lastly, we look more into writing and optimizing high performance code in two categories: *compute-bound* and *memory-bound* code. We first implement a method to quickly compute minimizers, `simd-minimizers`, that is much faster than previous methods. Since this method is used to compress the input, it is the only part of a longer pipeline that works on the full data, rather than the compressed version, so that it can easily become a bottleneck.

In PtrHash, we implement a fast *minimal perfect hash function*, which is a data structure used in many applications, and specifically also in datastructures to index genomic sequences. We design a data structure that minimizes the number of memory accesses and that interleaves these as much as possible, to fully saturate the memory bandwidth of the hardware. Again, this achieves significant speedup over other methods.

In conclusion, we achieve a speedup on the order of 10× on three different problems, by using carefully designed algorithms and implementations. Thus, my thesis is that optimal software can only be achieved by designing algorithms in parallel with their implementation, and that these can not be considered independently.