Supplementary 2

Rationale

Initially, the de novo assembler Minia proposed a new approach avoiding Bloom filter false positives for de-Bruijn graph implementation. Thanks to its inner structure, analyzing complex genomes with very little memory has become feasible. From this seminal work, and following the Minia concept, new tools using de Bruijn graph implementation have been designed. However, from an engineering point of view, this "practical" reuse process was not optimal:

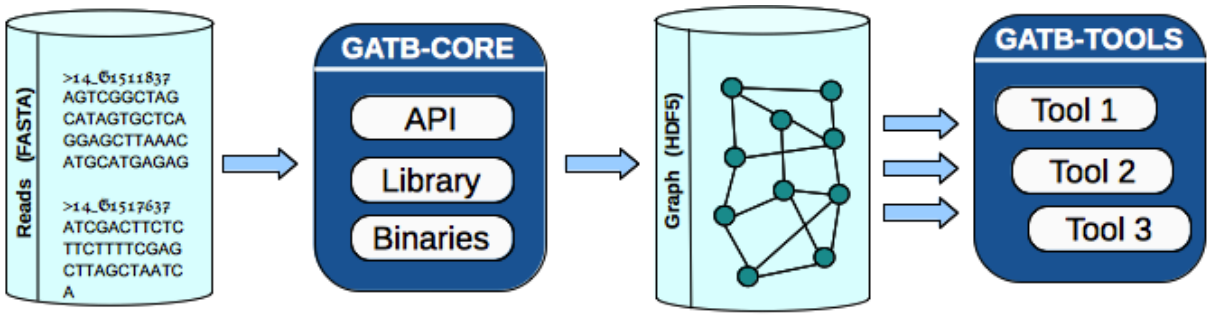
1. In most cases, new NGS tools just copied Minia's code in their own distribution, so any improvement in Minia's code had to be copied in each tool repository;

2. Minia's code was not first designed to be a library, so tools had to bother with some Minia's implementation details since there was no high level API for using de Bruijn graphs;

3. NGS tools first build de Bruijn graphs, then navigate graphs to extract some information specific to the application (contigs for Minia, SNP for DiscoSNP, etc.). This pattern had however an issue: it was not obvious to reuse a graph generated by a tool A in a tool B. As a matter of fact, de Bruin graph creation was not clearly separated from what was done on the graph just after.

These observations have been the foundation of the GATB-CORE project: designing a software library that 1) provides an implementation of the de Bruijn graph coming from Minia's software, 2) provides a tool that builds a de Bruijn graph and saves it on file system in a specific graph file format, 3) provides a public API for navigating a de Bruijn graph read from a graph file.

From this rationale, developing de Bruijn graph based tools becomes simple: just load a graph from the file generated by the graph builder provided by GATB-CORE, then navigate it with the GATB-CORE API.



Design initial choices

The primary design choice made in GATB-CORE was to follow the Object Orientation paradigm. In particular, it is of most importance to have a clear separation between interfaces and implementations. Once the library interfaces are defined and serve as the public API of the library, developers using this API won't have to change their code even if some part of the library implementation changes (a simple link with a new version of the library is needed).

Moreover, Object Orientation usage brings to the library the power of the Design Patterns. In our case, a central pattern is the Iterator because we always have to iterate elements such as reads, kmers, or nodes of graphs. Thus, the GATB-CORE library defines high level interfaces (iterable, iterator,

collection, etc.) and provides nice uniformity. For instance, iterating reads or nodes will be done with the same API, which makes the code clearer.

Programming Language

The GATB-CORE library aims 1) to provide fast NGS algorithms and 2) to be easy to use by developers. At first glance, these two concerns may be hard to achieve in the same library; for instance, the library may be over focused on performances but with a poor user API.

The C++ language allows meeting both worlds. As the successor of C language, it provides native binary (i.e. no slow interpreted code) and gives access to the low level resources of the computer (like fine memory management, SIMD development and so on); it also provides high-level structures that support Object Orientation paradigm. This is exactly what we need to achieve the GATB-CORE library objectives.

De Bruijn graph file format

In the rationale, we made an explicit distinction between the de-Bruijn graph creation and the way it can be used after. This distinction occurs at binary level, which means that a binary from GATB-CORE creates a graph and saves it in a specific format; then other binaries can load and navigate this graph file through the GATB-CORE API.

We therefore needed to define a format for the graph file. We had to consider the following:

- The de Bruijn graph implementation of GATB-CORE requires several information sources, each one being a typed list of items;

- Input set of reads can be huge, so does the information to be kept in the graph file.

The HDF5 technology is well suited for our purpose: it allows several collections of typed items (solid kmers, Bloom filters, etc...) to be to kept in a single file. Furthermore, it is designed to support massive amount of data. As a result, a file format gathers all the de Bruijn graph information. This format is also used as the output format of the 'dbgh5' tool (provided in GATB-CORE) that transforms the set of reads into a de Bruijn graph. The HDF5 is embedded in GATB-CORE as a third party.

Several example snippets are available here: http://gatb-core.gforge.inria.fr/snippets_graph.html

Operating system

GATB-CORE provides a light operating system abstraction (Linux, Macos and soon Windows) for file, memory and thread management. Making such abstraction allows client code to be independent from the OS, thus suppressing compilation directive inside the code or improving some OS accesses by hiding specific OS optimization.

Parallelism

The GATB-CORE library proposes a simple and powerful way to achieve parallelism and so get the power of multi cores architecture of modern computers. The idea is to provide a dispatching class that knows how to parallelize the way an iterator iterates its items. This dispatcher both relies on our iterator interface and our operating system thread abstraction. As a consequence, any implementation of our iterator interface can be easily parallelized through such a dispatcher.

Since the main loops of most algorithms are iterations over some items, the algorithms of the GATB-CORE library can easily be parallelized. Note that such a scheme supposes that each iterated item can be done without knowledge of the other iterated items, which is a common case in the GATB-CORE

library. Note also that synchronization mechanisms are available in the operating system abstraction when care must be taken while accessing a shared resource in different threads contexts.

Several example snippets are available here: http://gatb-core.gforge.inria.fr/snippets_multithread.html

Integer calculus and performance

The GATB-CORE library relies on the k-mer concept, i.e. a fixed sized word of nucleotides. A k-mer of k nucleotides is coded as an integer stored in 2k bits (a nucleotide can be stored on 2 bits). Today, 64 bits operating systems provide native and fast integer computation on 64 bits (even sometimes on 128 bits), so k-mers of maximal size 32 can be efficiently handled.

For k-mers larger than 32, computation on integers larger than 64 bits is required, leading to some arbitrary precision integers. There are many libraries providing such functionality but we choose to provide our own "large integer" class because 1) we need only a few integer functions needed for k-mers management, 2) we need to have an optimized integer class for a given k-mer size k.

The point 2 is important mainly for memory allocation concerns; since a lot of large k-mers/integers are created, we can't rely on the dynamic allocation storage because of performance issues (general-purpose dynamic allocators are well known to be inefficient for allocating many short buffers). The solution is to use the stack storage, which is fast. The issue is then, *for a choice of k that is known at runtime*, to choose which integer class to use: 64 bits native? 128 bits native, if available? Large integers up to K=96 ? Large integers up to K=128? And so on...

The solution is to use variants (in particular, we use Boost variants) that allow selection of types to be selected at runtime, which is exactly what we need for getting the best integer class for a k chosen by end user.

This design choice implies to use C++ templates; for instance, an algorithm A may uses k-mers whose type is a template T, with several specializations for different integer classes. Today, the GATB-CORE library supports 4 different integer classes for a k-mer size range from 1 to 128. Thanks to the variant usage, the algorithm uses the best integer class according to k chosen at runtime.

Compilation remarks

GATB-CORE uses CMake for artifacts generation. This is a natural choice since we target to support native libraries for several operating systems.

GATB-TOOLS also relies on CMake but it is not mandatory; developers could use other tools (make for instance) for building their binaries.

Several compilers have been tested so far; GCC for all OS and clang on MacOs are ok.