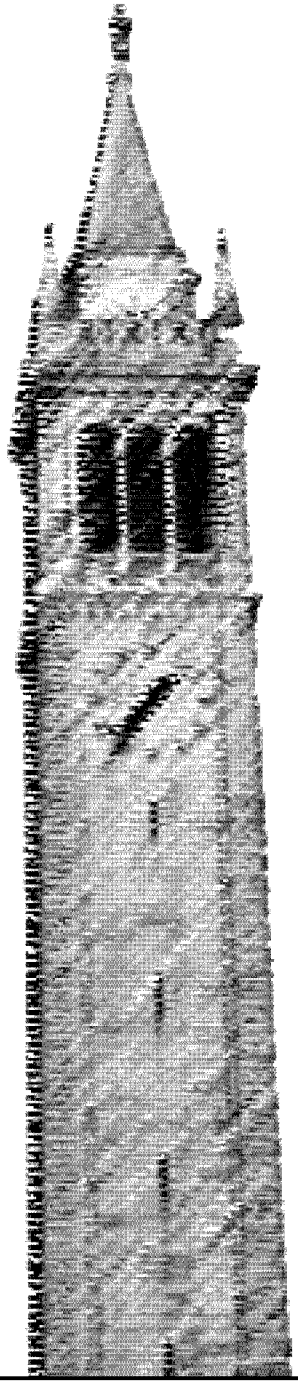
Technical Report: Benchmarking and Evaluating Whole Genome Assembly



Faraz Tavakoli Farahani

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-196
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-196.html

December 6, 2017

Technical Report: Benchmarking and Evaluating Whole Genome Assembly

by Faraz Tavakoli

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor *Katherine A. Yelick*
Research Advisor

(Date)

* * * * * * *

Professor *Thomas Courtade*
Second Reader

(Date)

Technical Report: Benchmarking and Evaluating Whole Genome Assembly

ft@berkeley.edu

October 13, 2017

Progress in long-read next-generation genome sequencing technology continues to sustain the trend of decrease in cost while increasing the read length. With a price that is now only a few thousands of dollars for the human DNA, *de novo* whole genome assembly is gaining clinical adoption in diagnosis and prevention of diseases. Meanwhile, the research community is employing the long-read technology to sequence an ever increasing number of new species and organisms for which there is no reference genome. Despite of these developments, a typical assembly pipeline lacks rigorous benchmarks and reproducible evaluations. Particularly, as the read length increases and a greater number of repetitive structures can be accurately resolved, the possibility of fully automated assembly becomes a reality for which there needs to be standardized qualitative and quantitative metrics. In this paper we purpose a deterministic and declarative framework for assessing long-read genome assemblers and provide a reference implementation applicable to a wide range of system infrastructures and operating environments. Moreover, we have constructed a set of synthesized reads which could be used both by developers to improve their assembly applications or the end users to identify the most appropriate software for their needs.

Introduction

The two dominant trends of reduction in cost and accretion of read length in next-generation sequencing (NGS), has made automated whole genome *de novo* assembly an attainable prospect with far reaching implications for both clinical and scientific applications. Table 1 provides a comparative overview and their basic capabilities (Goodwin et al., 2016). Despite these exciting improvements, assembling raw data to a complete genome remains a challenging endeavor. In particular, the complexity of genomes beyond that of simple organisms, such as long repetitive sequence combined with variations in structural elements and copy number alterations, has greatly limited applicability of short-read platforms, curtailing their extremely high-throughput advantages (Treangen and Salzberg, 2012). Meanwhile, long-read technology still suffers from a lower throughput, substantially different error profiles, and a greater cost. From a computational standpoint, assembly is an intensive task with high CPU, memory, and IO requirements for even moderately sized projects. Algorithmically, there are very few theoretical and empirical guarantees provided by the mainstream techniques, such as those based on greedy algorithms (Nagarajan and Pop, 2013). Furthermore, the quality of the end result and the uncertainty about the constructed genome is still mostly un-quantified even for widely used reference sequences. This is greatly exacerbated by opaque software implementations employing numerous heuristics while interfacing a complicated web of tune-able parameters. Additionally, with an involved pipeline of data processing routines consisting of disparate software packages, sometimes suffering from incompatible file formats or inconsistent API specifications, reproducibility still poses as a major challenge.

With further advancements in NGS, increase in data production will consequently result in a greater demand for automating the laborious task of whole genome assembly. In turn, this will lead to a more pronounced emphasis on computational limitations, reproducibility considerations, and validation without a reference. In this work, our end goal is to specify and implement a framework to address each of the aforementioned three issues.

Technology	Average read length	Main source of error	Throughput per run
Sanger	700-900 b	Polymerase slippage Short repeats	900 b
Illumina	150-250 b	Single base substitution	150-300 Gb 1.8 Tb
PacBio	10-15 Kb	Indel	0.5-1 Gb
Oxford Nanopore (MinION)	6 Kb	Deletion	500 Mb

Table 1: An overview of major sequencing platforms.

Overview

In order to systematically examine effects of different structural patterns present in complex genomes in the final assembly, we've developed an embedded domain specific language (EDSL) to declarativly construct reference sequences. With parametrization of features such as repeat types, length, and counts the interface facilitates quantifying the effect of structural variations in the reference versus the assembled sequence. Moreover, we designated a predefined set of sequences containing repeat structures which causes ambiguities in the final assembly to serve as simple validation benchmarks. A deterministic, reproducible, and declarative build environment for the assemblers and associated utility applications is achieved via the *nix* package manager. Most of the required software, in fact, was added on top of a preexisting *nixpkgs*[1] repository hosted on github, making them available as standalone applications and maintainable by the open-source community independently. Wrapper utilities implement support of a wide array of runtime environments in a composable fashion. For instance, we evaluated the performance of a few long-read assembler by exporting them as Linux containers [2] equipped with the real-time monitoring and data collection platform, Netdata[3], in order to collect application specific and isolated performance measures in a multi-user host system; which would otherwise be non-trivial due to a multitude of processes running simultaneously. Although the system is intended to serve as an end-to-end benchmarking and validation pipeline, each component is developed separately of the other so that it could be used in existing workflows without requiring commitment to the other parts of the system.

The EDSL is implemented in Haskell; therefore, leveraging its type system to statically verify type level constraints for specified sequences.

[1] `https://github.com/NixOS`

The *nixpkgs* contribution guide is available at `https://nixos.org/nixpkgs/manual/`

[2] Our implementation uses Docker containers, but the choice of container engine is immaterial for this functionality.

[3] `https://my-netdata.io/`

Our software architecture and design decisions were heavily influenced by the following themes:

- building upon already available software was prioritized over implementing the desired functionality in house, tolerating trade-offs between adding additional dependencies and maintaining our own code-base in favor of the former.
- intermediate results in a multi-stage workflow were subjected to the same level of

validation and consistency as the final result of the assembly, despite the additional overhead.

The goal of our project was to develop a framework to benchmark and assess some common assemblers, although performing those assessments is beyond the scope of this report. The framework will reduce the human time involved in such a comparative study, but substantial computational resources would still be needed. Similarly, Transcriptomic or Metagenomic assembly was not considered or evaluated.

Reference Data

The next frontier, repetitive structures in genome

Organism	Count	Total bp
Primates	563	664160
Rodents	466	487006
Other mammals	347	243730
Other vertebrates	52	53994
Drosophila	65	167423
Arabidopsis	98	275516
Grasses	27	67789

Table 2: The count of various repetitive patterns in different species. Data from *RepeatMasker* database.

Different qualitative measures and quantitative metrics have been developed over the years to evaluate the performance of assemblers; however, due to technical challenges and biological interests repeat resolution has been mostly overlooked. A phenomena shared across a wide array of species, from bacterial to mammalian cells, repeats in DNA sequences are a common occurrence in a large part of the genome 2. In fact, in one form or the other, repeats account for almost half of the human genome 3. Domination of short-read sequencing technology, despite their high throughput, was insufficient to faithfully reconstruct these regions. In particular, repeats created ambiguities that could not be resolved leaving many assemblies at best fragmented and frequently incorrect. From a computational perspective, these repetitive regions posed a theoretical barrier to accurately reconstructing the complete genome. Of three major categories of repeats, Terminal, Tandom, and Interspersed, the later cause the greatest difficulty in assembly of new genomes due to their size which could not be bridged by typical read lengths. Although the function of repeated sequences are not clearly understood, resolution of these repetitive segments is biologically significant. For instance, dispersed repeats have been recognized as potential source of variation and structural regulation(Treangen and Salzberg, 2012).

Reference Genome

In order to systematically devise synthetic sequences as well as providing an interface for declarative specification, we implemented a small EDSL in Haskell. The library

Repeat class	Repeat type	Count in hg19	Coverage	Length
Mini ǀ Micro satellite	Tandem	426,918	3	2 - 100
SINE	Interspersed	1,797,575	15	100 - 300
DNA transposon	Interspersed	463,776	3	200 - 2,000
LTR retrotransposon	Interspersed	718,125	9	200 - 5,000
LINE	Interspersed	1,506,845	21	500 - 8,000
rDNA (16S, 18S, 5.8S and 28S)	Tandem	698	0.01	2,000 - 43,000
Segmental duplications	Tandem or interspersed	2,270	0.20	1,000 - 100,000

Table 3: Repeated sequences in Human DNA (Treangen and Salzberg, 2012)

consists of sequence generators and associated monadic-combinators for constructing different repeat structures. The generators, construct sequences of a given length based a specified dynamic. For instance, `debruijn k` is a generator for a De-Bruijn sequence of order k on the four letter nucleotide alphabet A, C, G, T. Since each sub-sequence of length k appear exactly once, it can be used to construct a genome where each read of length greater than k can be uniquely aligned. Similarly, there are generators for a constant, first and second order Markov, as well as models for sampling from an input *FASTA-FASTQ* file. The core library provides combinators for basic composition of sequences in addition to utilities for constructing the three fundamental repeat types leading to ambiguities in contigs-graph (Kamath et al., 2017).

Reference Reads

Given a reference genome, we need to simulate the chemical and biological mechanism in order to generate synthetic reads used as the input to the assembler. Each read also needs to include alignment information for downstream quality analysis as aligning fragments and contigs to the reference genome could be error prone and adds an additional sources of variability to measuring assembler's performance. The recent survey by Escalona et al. (2016) provides an overview of the available techniques and accompanying software. However, read simulation still seems like a non-trivial task. For instance, they recommend *PBsim* for simulating long-reads generated by PacBio platform when there is a reference genome; regardless, the software has not been maintained since 2013 and does not accurately models the newest PacBio chemistry based on our comparison with actual data from the company [4]. There are few newly developed packages, not examined in the aforementioned review. For the same PacBio platform, *Simlord* is a more recent application which uses a different parametric model to sample reads. However, our benchmarks is still inconclusive about the quality of generated data and to our knowledge there is no single simulator that accurately models the latest chemistry of PacBio machines. Since read length distribution and error profile for *indel*s and substitutions are significantly different between different technologies, one of our goals was to study the effect of variations in read modeling with respect to the downstream analysis. Additionally, we evaluated the quality of simulated reads by comparing statistics from *FastQC* to that of our sampling procedure.

[4] We calculated basic sequence statistics (e.g. k-mer content, error rate, etc.) for each synthetic read dataset and calculated the empirical likelihood between true and generated data

Benchmarking framework and implementation

Deterministic build systems

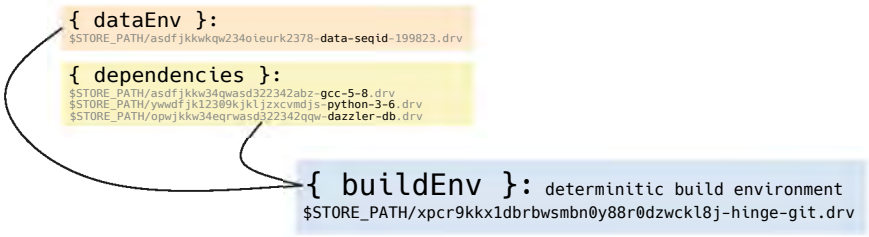


Figure 1: Each package is built in a fully deterministic environment. By computing the cryptographic hash for each derivation, we can guarantee that each invocation of the build leads to the same exact binary.

Since *nixpkgs* is still heavily under development and lacks widespread adoption and community support for computational biology and bioinformatics software, the initial phase of our implementation focused on packaging some commonly used assemblers and other related tools. Although other package managers, such as Anaconda by Continuum and the sister community-maintained repository, Bioconda, enjoy a greater user-base in data analytics; they don't provide the granular level of control over the build environment required for rigorous benchmarking. As an example, our implementation can accurately test between two different versions of the same assembler since we can keep all other dependencies exactly identical. In contrast, binaries for even the same version obtained from different *Conda* repositories are not guaranteed to be identical, yet alone their dependencies. The following code snippet shows a sample *nix-derivation* for *HNGE*(Kamath et al., 2017) assembler.

```
# nix expression for building the HINGE assembler
{
  stdenv,
  gcc,
  zlib,
  cmake,
  boost
}:

stdenv.mkDerivation rec {

    version = "dev";
    name = "hinge-${version}";

    src = fetchgit ( builtins.fromJSON ''
      {
        "url": "https://github.com/HingeAssembler/HINGE",
        "rev": "ddcc8e2e34931d1895d15e08664e17c28ad61b22",
        "sha256": "1hqf5564q5cs9r6i2j9zh0z2v40sr60v9md7icwni6kbw29665z2",
        "fetchSubmodules": true
      }
    '' );


    buildInputs = [ gcc zlib cmake boost ];

    patchPhase = ''
      substituteInPlace src/layout/CMakeLists.txt \
        --replace 'set(Boost_USE_STATIC_LIBS   ON)'   \
                  'set(Boost_USE_STATIC_LIBS  OFF)'
    '';

};
```

User space

Currently, in house HPC clusters constitute the majority of deployment systems used for computational biology. The assembly pipelines for any realistic genome length require computational resources not readily available on cloud services or consumer PCs. One of the major challenges of developing software for multi-user environments is that it's rarely the case that user have privileged access (i.e. root) to the system. To address this issue, our current implementation uses the *UNSHARE* capability of the Linux kernel to manage the resources of the child process. In particular, by mounting the default root store path in the sub-process to a directory in user's home, we can initially install the compiler toolchain (e.g. GNU gcc) from a binary release source. Then, from within the *UNSHARE* process context, we bootstrap the dependencies with a $PREFIX path inside the user's directory.

Some of the major shortcomings we encountered pursuing this approach were

1. The growth in binary size
2. Computation time and massive rebuilds could also be of concern. When a package on which a lot of other packages are dependent changes, that triggers a rebuild for each one of the dependencies downstream.
3. Adding large files to the nix store is a memory intensive task. The current implementation requires loading the file into the memory in order to compute the cryptographic hash which could be problematic for large genomes.
4. Since outside of the build system, our paths are marked as read-only, access via multiple machines sharing a single file can be handled effortlessly. However, our implementation lacks support for concurrent writes.

Runtime isolation and resource sharing

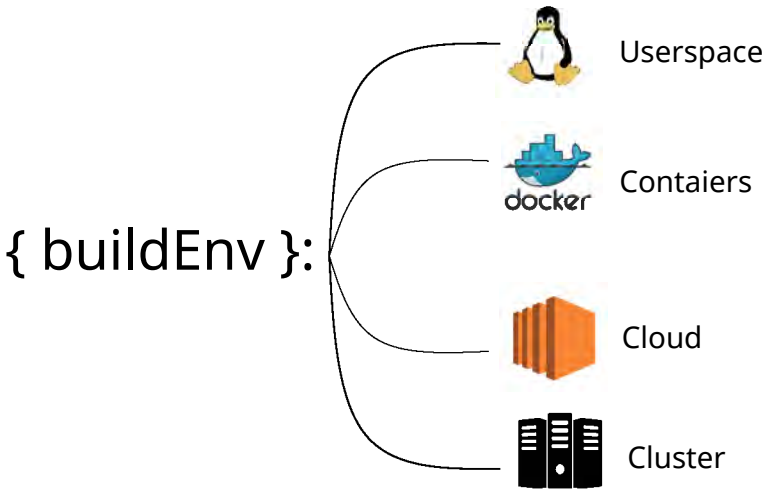


Figure 2: The supported runtime environment. The frame work can generate and deploy native application, lightweight containers, virtual machine and cloud services, as well as jobs for queue managers such as slurp. However, only docker containers and native applications have been tested so far.

The platform currently supports a few execution models with little to no boilerplate, including managed cloud orchestration via *nixops*, Linux containers via *Docker* images, and lightweight user isolated environments via kernel user namespaces.

Performance monitoring and data collection

Given the rapid development in sequencing technology and unabated data production, the performance bottleneck would inevitably become one of the major considerations for developing assembly software. We've implemented support for few profiling utilities for both real-time monitoring and collecting application traces (e.g netdata and perftools) as demonstrated in the following figures. Given netdata is intended as a lightweight real-time monitoring system, by default it only retains 1 hour of events. However, recording of sampled data could be achieved via support for persistence database engines through plugins. We envision using a network data-store to collect usage data from a global community of volunteers.

In order to demonstrate the utility of performance measurement, we've included snapshots of resource usage for a typical execution as would be seen through the real-time monitoring dashboard. Combined with ability to implement custom alerts; the time-series data can greatly facilitate performance engineering and optimization. As an example, a cursory glance at the disk usage would reveal an abrupt increase in IO backlog caused by the disk driver. Therefore, it can be conjectured that amortizing IO operation could lead to a significant speed up; assuming such backlogs are frequent. Likewise, the graphs reveal heavy under utilization of CPU resources which could be explained by the aforementioned disk bottleneck. It should be noted; however, these measurements are done inside the container and the corresponding virtualization system, so great care must be taken to allocate host resources exclusively in order to obtain accurate results. This is not a major technical challenge as this functionalities are already implemented in most major virtualization systems as well as dedicated cloud based solutions.

Conclusions and Future Directions

The most immediate extension of this work, as touched upon previously, is to benchmark existing assemblers to both highlight the deficiencies of our approach and avenues to improve the existing software. This will also provide the scientific community with a report card of assemblers, which we think is greatly lacking at the moment.

Over the technical front, our system is capable of identifying duplication in files and therefore storing only one copy on the disk. Nonetheless, we still record the data as files, and multiple files with small variations lead to a great storage overhead. Even though, end to end determinism make intermediate snapshots avoidable, they might still be retained for downstream exploratory analysis. Therefore, a more database oriented approach to store the data might prove more economical in the long run.

On the other hand, *Nix* itself is a rather non-intuitive environment and might require an upfront learning investment. There are few opensource alternatives, mainly *Guix*, but each suffer from very small user-base and not routinely updated package-base.

References

Merly Escalona, Sara Rocha, and David Posada. A comparison of tools for the simulation of genomic next-generation sequencing data. *Nature Reviews Genetics*, 17 (8):459–469, Jun 2016. ISSN 1471-0064. doi: 10.1038/nrg.2016.57. URL `http://dx.doi.org/10.1038/nrg.2016.57`.

Sara Goodwin, John D. McPherson, and W. Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*, 17(6):333–351, May 2016. ISSN 1471-0064. doi: 10.1038/nrg.2016.49. URL `http://dx.doi.org/10.1038/nrg.2016.49`.

Govinda M. Kamath, Ilan Shomorony, Fei Xia, Thomas A. Courtade, and David N. Tse. Hinge: long-read assembly achieves optimal repeat resolution. *Genome Research*,

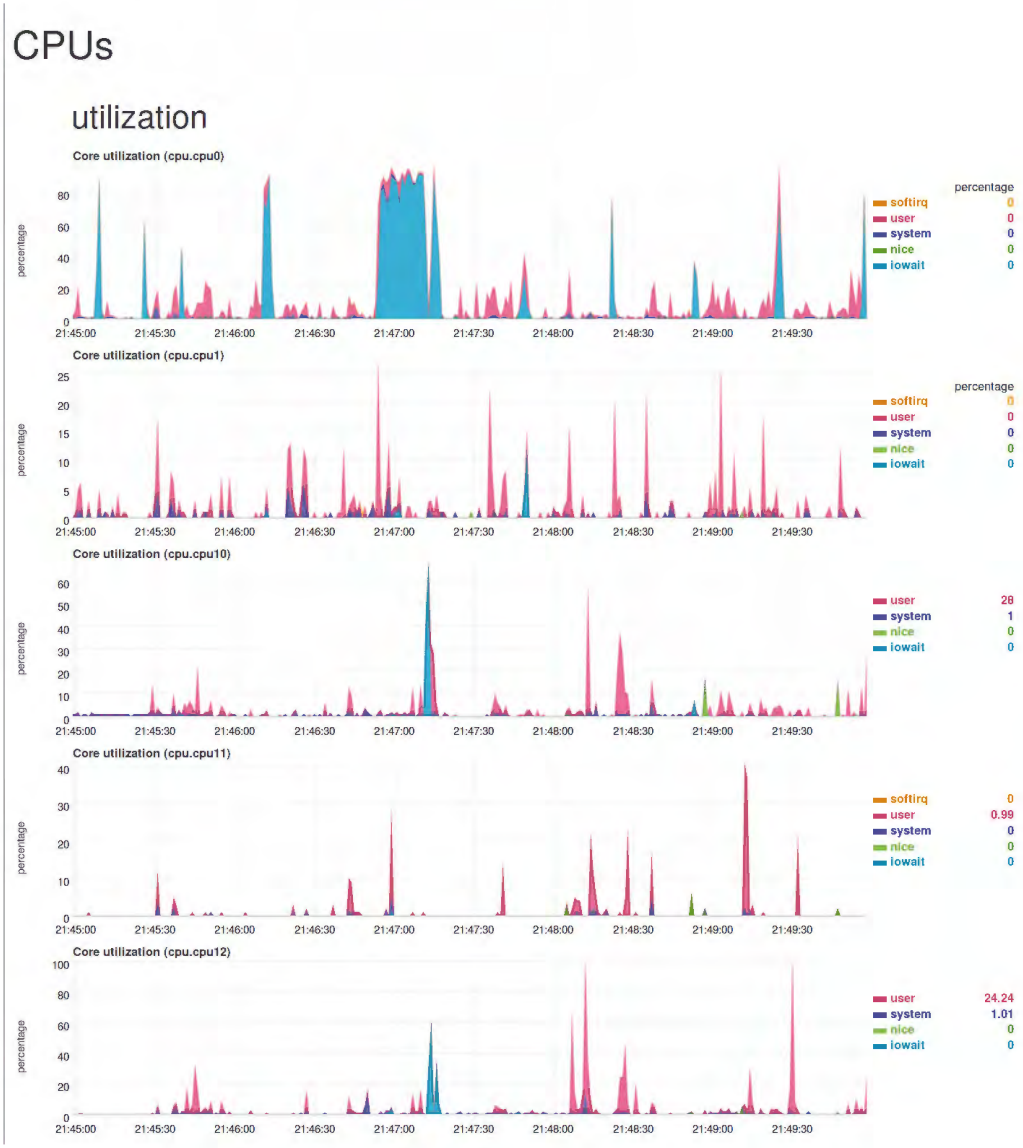


Figure 3: The CPU utilization time-series plot corresponding to an assembly process inside a container context. Each subplot corresponds to a single execution thread on a multi-core system.

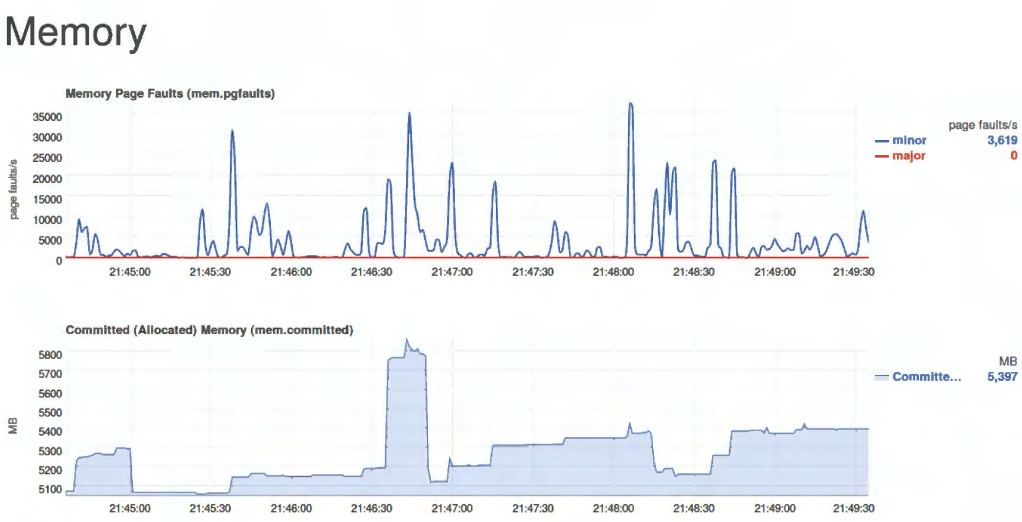


Figure 4: A view of memory management during the execution of the program.

Figure 5: These plots correspond to the process activity occurring within a container as part of an assembly pipeline.

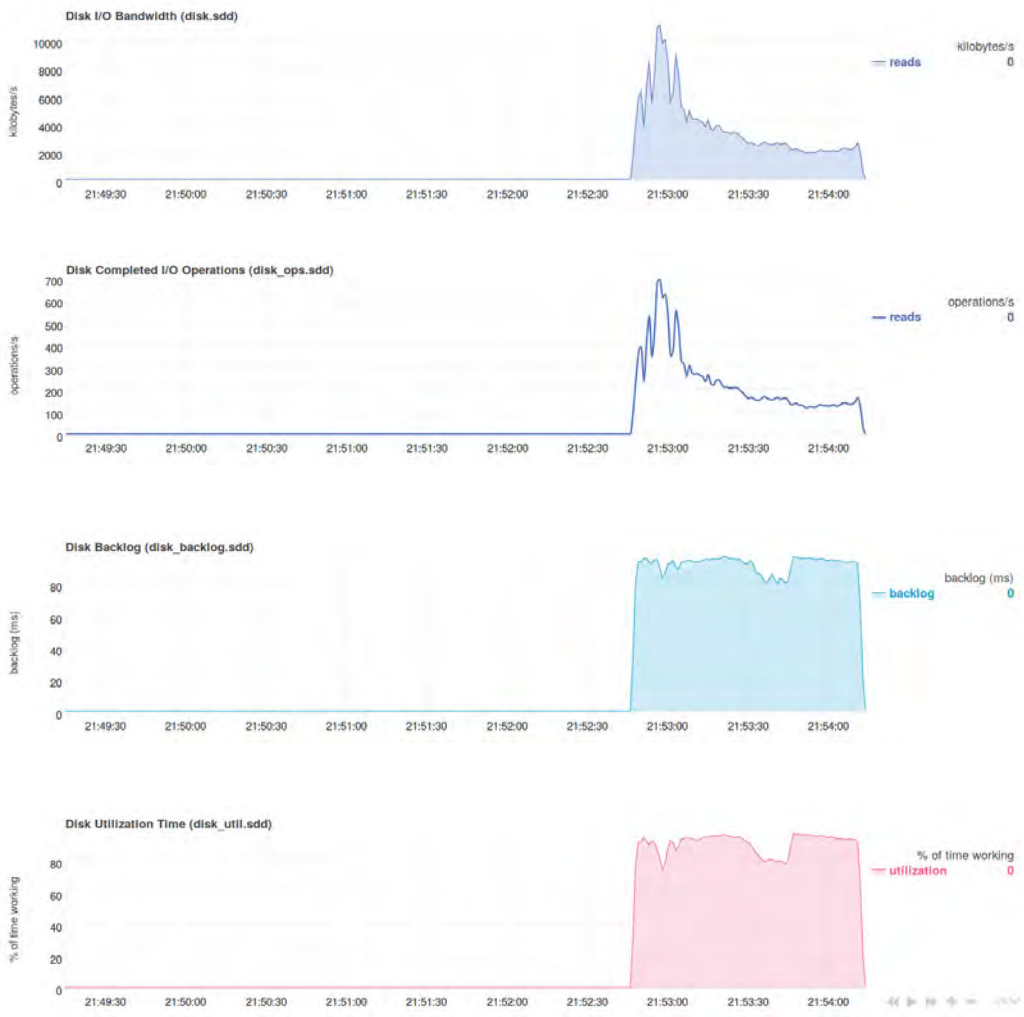


Figure 6: Disk usage statistics and activity monitoring could be of significant interest to developers as IO constitutes a major bottleneck in efficiency of assemblers implementation. The figure show a non-uniform heavy disk usage in a 5 minutes window of a long running process. Given heavy disk utilization (subplot 3 and 4) has created a backlog, a more uniform distribution of io subroutines could result in a greater performance.

27(5):747–756, Mar 2017. ISSN 1549-5469. doi: 10.1101/gr.216465.116. URL http://dx.doi.org/10.1101/gr.216465.116.

Niranjan Nagarajan and Mihai Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14(3):157–167, Jan 2013. ISSN 1471-0064. doi: 10.1038/nrg3367. URL http://dx.doi.org/10.1038/nrg3367.

Todd J. Treangen and Steven L. Salzberg. Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, Jan 2012. ISSN 1471-0064. doi: 10.1038/nrg3164. URL http://dx.doi.org/10.1038/nrg3164.