Π-cyc: A Reference-free SNP Discovery Application using Parallel Graph Search
Reda Younsi¹*, Jing Tang² and Liisa Holm¹
1* The Institute of Biotechnology (BI)
2 The Institute of Molecular Medicine Finland (FIMM)
September 2018

Keywords: Parallel graph search, SNP calling, Population genomics.

ABSTRACT

Motivation: Working with a large number of genomes simultaneously is of great interest in genetic population and comparative genomics research. Bubbles discovery in multi-genomes coloured de Bruijn graph for de novo genome assembly is a problem that can be translated to cycles enumeration in graph theory. Cycle enumerations algorithms in big and complex de Bruijn graphs are time consuming. Specialised fast algorithms for efficient bubble search are needed for coloured de Bruijn graph variant calling applications. In coloured de Bruijn graphs, bubble paths coverages are used in downstream variants calling analysis.

Results: In this paper, we introduce a fast parallel graph search for different K-mer cycle sizes. Coloured path coverages are used for SNP prediction. The graph search method uses a combined multi-node and multi-core design to speed up cycles enumeration. The search algorithm uses an index extracted from the raw assembly of a coloured de Bruijn graph stored in a hash table. The index is distributed across different CPU-cores, in a shared memory HPC compute node, to build undirected subgraphs then search independently and simultaneously specific cycle sizes. This same index can also be split between several HPC compute nodes to take advantage of as many CPU-cores available to the user. The local neighbourhood parallel search approach reduces the graph’s complexity and facilitate cycles search of a multi-colour de Bruijn graph. The search algorithm is incorporated into Π-cyc application and tested on a number of Schizosaccharomyces Pombe genomes.

Availability: Π-cyc is an open-source software available at www.github.com/2kplus2P

Contact: * reda.younsi@helsinki.fi

INTRODUCTION

De novo de Bruijn graph assemblers for short reads are mainly divided into Eulerian and Hamiltonian approaches. In both approaches, cycles, called bubbles when searching directly a de Bruijn graph, represent errors, heterozygous variants or repeats and are mainly graph artifacts that require genomic corrections for accurate assembly. Taking a different approach, showed, by using the concept of colours to represent multiple genomes, can generate multicolor graph paths, in form of bubbles, which may contain genomic variants. This coloured bubble search was first used successfully in Cortex-var application to call genomic variants, mainly SNPs and short indels, using de Bruijn graphs. Several methods using coloured bubble concept have since been developed for variant calling using de bruijn graphs.

In this paper, we propose a fully integrated approach to SNP calling using a parallel cycle enumeration algorithm in a transformed colored de Bruijn graph (cdbg) which we call Π-cyc. The proposed method can use a single, or several compute, nodes with multi-cores CPUs to build undirected subgraphs and search for cycles in parallel. A similar algorithm was first used sequentially to enumerate specific $2k + 2$ bubbles, $k$ being the k-mer size used for the cdbg. These cycles where shown to harbour a large number of real SNPs but missed a substantial number of other SNPs found in different cycle sizes, those that are normally found within a k-mer path. Since we don’t know the exact number of nucleotide disagreements that may form multiple SNPs on each path, enumerating all the different sizes of cycles starting from a branching vertex can potentially return all the variants on the paths but at the expense of extended the path beyond $k + 1$ steps. This enumeration of cycles requires $k + n$, $k$ is fixed, however $n$ takes values from 1 to $k$, since the paths of these bubbles expand by the number of disagreeing nucleotides that forms $k + n$ steps in paths. This cycle enumeration process could be an expensive task as two loops are required. The first loop for the branching vertices and the second loop for $n$, which could take up to $O(n^2)$ in complexity. We extend the $2k + 2$ bubbles search method to take advantage of the multi-cores parallel computing found in modern CPUs. The aim is to search for all cycle sizes which could improve the SNP detection and classification down-stream analysis task. We show that the multi-core and multi-node design for our graph decomposition search method improves significantly cycle enumeration running time for colored de Bruijn graph based assemblers.

In the first part of the paper, we introduce Π-cyc application and its different modules. Sections 2, describes related work. In section three, we describe the algorithms and methods used for building Π-cyc application. Section four is the experimental evaluation of our cycles search method and the comparative results. Section five is a discussion and we conclude in section six.

2 RELATED WORK

The bubble caller developed for cortex_var uses a single thread to search complex de Bruijn graphs, a serious limitation for very large and multiple assembled genomes analysis. In contrast, Π-cyc concurrently searches very large multi color graphs for $2(k+n)+2$ cycles using graph decomposition then uses a heuristic method for SNP predictions based on path coverages. Thus, reducing the overhead of a single thread cycles
enumeration and improves significantly the CPU time. cortex\_var bubble search was not designed for multiple genomes. And hence, we can not use it in the comparison section of the paper. In contrast, DiscoSNP++ application is a population based approach for de novo variant calling where samples or genomes, also using de Bruijn graph assembly based on bloom filter, searches for bubbles.\(^2\) However, the algorithm used for bubble search misses a large number of SNPs as shown in\(^3\). DiscoSNP++ uses a breadth-first search approach whereas cortex\_var bubbles search uses Dijkstra’s algorithm. II-cyc multi-node and multi-core approach effectively enumerates \(2(k + n) + 2\) graph cycles, also known as nested bubbles, leading to the discovery of potentially important variants in reference-free sequenced samples.

3 THE METHOD

II-cyc finds cycles by graph decomposition. The method is similar in principle to the method shown in\(^1\). First, an edge list representing the linked de Bruijn graph is extracted from a binary colored de Bruijn graph. Second, undirected subgraph are created using the edge list for cycles enumeration. An important difference is the extraction of the hash table index and the ad-hoc subgraph creation. A full multicolour de Bruijn graph is extracted from a binary colored de Bruijn graph. An important difference is the extraction of the hash table index and the ad-hoc subgraph creation. A full multicolour de Bruijn graph is extracted from a binary colored de Bruijn graph.

3.1 Software main features

![Flowchart](image)

Algorithm 1: Parallel Subgraph creation and cycle search.

---

1. Read Input files: Illumina reads from input files\(^1\).
2. Build a hash table: using K-mers \(k-mer\) windowing and Hashing\(^1\).
3. Combined de Bruijn graphs is stored in a single Hash table.
4. An index is calculated from the Hash table.
5. The index is divided among CPU-cores (see Algorithm\(^1\)).
6. Subgraphs are dynamically created using a fast neighborhood building block method in parallel (see Algorithm\(^2\)).
7. Searching for cycles is executed in parallel (see Algorithm\(^3\)).

In general, the application uses several steps to build colored de Bruijn graph which is stored in a hash table, undirected subgraphs are created in parallel for cycles search. A full description of Cortex\_con can be found in\(^1\).

---

Algorithm 1 describes the main loops used for subgraph creation and cycles enumeration. The first loop takes the computed index, or part of it in an HPC distributed system, and divide it among the CPUs. Each CPU-core works with a part of the index to create subgraphs and search for cycles.

The two main Algorithms\(^2\) used in subgraph creation, and\(^2\) used for cycle search, are:

- Build a subgraph using a fast neighborhood building block method.
- Parallel \(2(k + n) + 2\) cycles enumeration.

The subgraph creation process described in Algorithm\(^2\) uses a neighborhood building block method. Initially, a branching vertex is selected from the subindex. Every other adjacent vertex is selected and including in the subgraph creation and the process incrementally adds vertices to the size of the subgraph up to a user defined threshold. Genome assemblies from multi-colored de Bruijn graphs can be very complex. The objective of a neighborhood method is to build subgraphs.

---

1. We used cortex\_con C implementation [http://cortexassembler.sourceforge.net/](http://cortexassembler.sourceforge.net/)
2. See footnote 2
small enough to reduce graph complexity but large enough to search for \(2(k + n) + 2\) cycles.

### Algorithm 2: Graph Neighbourhood.

**Data:** Branching vertex \(j\) from index

**Result:** Subgraph \(g\)

1. Set max subgraph size \(v = x, x = \min G, \ldots, n; n \in \mathbb{N};\)
2. Initialise subgraph vector \(e = 0;\)
3. Initialise edge list map \(m = 0\) and \(i = 0;\)
4. \(c_0 = j;\)
5. while \(e < v\) do
   6. \(m = \text{GraphEdgeList}(e);\) // get adjacent vertex
      \(\{a_i\}\) from the hash table;
   7. if \(a_i \notin m\) then
      8. \(e \leftarrow a_i;\)
   9. \(i++\)
10. end
11. \(g \leftarrow m;\)

The cycle enumeration Algorithm 3 takes a branching vertex from the subgraph and walks \(k + n + 1\) steps and records whether another branching vertex is found. A full cycle is found when the walk returns to the starting vertex in \(2(k + n) + 2\) steps. Only cycles that have equidistant branching vertices are kept for further analysis. Cycles are decomposed into paths and labels are read from the hash table with their potential SNPs. The SNP prediction algorithm uses the extracted k-mer path coverages for each colour.

### Algorithm 3: Walk the graph.

**Data:** Subgraph \(g\)

**Result:** A set \(S = 2(k + n) + 2\) cycles

1. Set path Array \(p;\)
2. Initialise cycle vector \(v;\)
3. for \(b \leftarrow 1\) to \(n\) do // Select branching vertex \(b\) from \(g\)
   4. \(p_b \leftarrow b;\)
   5. while \(p < 2(k + n) + 2\) do
      6. if \(v \in v\) then
         7. \(v = \text{walk forward}(g);\)
      8. else
         9. \(v = \text{walk reverse}(g);\)
   10. end
11. end
12. \(S \leftarrow v;\)

## 4 RESULTS

The Pombe whole genome DNA sequencing datasets (DNA-seq)\(^4\) used for the CPU time evaluation and Cycle enumeration are described in Table 3.

### Table 1. \(2(k + n) + 2\) cycles found for different CPU-cores run for 60 minutes CPU time using the 10 Pombe genome strains in Table 3.

<table>
<thead>
<tr>
<th>cores</th>
<th>total cycles</th>
<th>used index</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>669</td>
<td>64</td>
<td>3607.50</td>
</tr>
<tr>
<td>5</td>
<td>2347</td>
<td>212</td>
<td>3648.70</td>
</tr>
<tr>
<td>10</td>
<td>3886</td>
<td>360</td>
<td>3652.03</td>
</tr>
<tr>
<td>20</td>
<td>7150</td>
<td>680</td>
<td>3604.22</td>
</tr>
<tr>
<td>30</td>
<td>10583</td>
<td>911</td>
<td>3617.55</td>
</tr>
<tr>
<td>40</td>
<td>13875</td>
<td>1344</td>
<td>4324.67</td>
</tr>
<tr>
<td>80</td>
<td>19255</td>
<td>1855</td>
<td>3803.10</td>
</tr>
</tbody>
</table>

The Pombe whole genome DNA sequencing datasets (DNA-seq)\(^2\) used for the CPU time evaluation and Cycle enumeration are described in Table 3.

4.1 CPU Time Analysis

We run the CPU time tests on a Dell EMC PowerEdge C6420 server\(^4\) with a twin Skylake 20-cores hyperthreaded CPUs for a total of 2x40 cores. The application was compiled with AVX2 option using intel C++ version 18.0.2 compiler (AVX-512 intel compiler option showed similar overall results).

The results shown in Table 1 are from a combination of 10 Pombe genomes, as shown in Table 2. The computed hash table index is 1314786 vertices and is divided among 5 compute nodes with 262957 vertices sub-index for each node. Each experiment is run with varying CPU-cores as shown in the table and with the same set of parameters. K-mer value \(K = 63\) and the subgraph size is set to 5000 vertices. The total number of cycles found for a single node is also shown in total cycles column (similar figures are found on each node). The used index on Table 1 shows the total number of vertices in the sub-index used to create subgraphs. The time column shows the total CPU time it took for the graph creation and cycle search after the allocated 60 minutes CPU time has ended. The extra time accounts for time spent in the loops before it exited.

The CPU time of the graph creation and cycles search on a single core is an average of 669 \(2(k + n) + 2\) cycles. A substantial increase in shown for the 80 cores with 19255 cycles found for a total of 3803.1 seconds. It took an extra 3.38 minutes for the search to finish for the 80 cores. The average CPU time of the 5, 10, 20, 30 and 40 cores parallel search is shown in Table 1 A parallel multi-core search that finds a similar number of cycles as found by 1 core will take far less time. The decrease in CPU time is mostly noticeable for the 80 cores (i.e. a complete single compute node in our HPC server) which finds 4358 cycles in 811.78 seconds,

---

\(^2\) https://wiki.helsinki.fi/display/it4sci/Kale+User+Guide
about 6.5 folds increase in cycles found, and 4 folds decrease in CPU time in comparison to one CPU-core.

The graph complexity is shown in figure 2. A simple measure is to calculate the number of connected vertices in a given subgraph (or total edges) then divide by the total number of vertices in the subgraph for normalisation. We notice that the increase in subgraph complexity increases with the number of genomes used in the experiments. The average complexity measure for 10 Pombe genomes is 0.25 while for the 50 Pombe genomes is 0.47. Similar samples that produces complex interconnected graphs will benefit from the parallel search method we developed.

In heavily interconnect graphs found in a multi-coloured setup, we noticed that as the search progress less cycles are discovered which eventually returns small number of cycles towards the end of the subgraphs creations as shown in Figure 3. Using the entire branching index in a heavily interconnect de Bruijn graph will result is searching for duplicate cycles and will require increased resources for searching the entire de Bruijn graph. An efficient way to reduce the search load is to select only a third of the entire index for subgraph creation.

### 4.2 SNP Calling Analysis

In order to assess the results of the contigs in the fasta file produced by the application, we use the blast program with default parameters to align each path to the reference genome *Schizosaccharomyces Pombe ASM294v2*. The SNP results are processed by an in-house program that search for nucleotide differences in the blasted file that match those in the produced contig fasta file.

We made certain to select randomly genomes from 57 strains included in SNP calling analysis. A summary of the Table 3 includes 3 of the 57 datasets used in the 10-Pombe experiments, 13 and 25 for the 30-pombe and 50-pombe experiments.

Table 3 shows the results of the Pombe datasets, using $k = 63$ for the de Bruijn graph construction, for a single compute node. The application generated approximately 200000 cycles for a combination of 10 genomes, 500000 for 30 genomes and a million for 50 genomes. Filtering these sets keeps only cycles that have relatively matched paths using a parameter allowing maximum mismatch value $f = 15$ and shown in *Pred* column. The number of predicted SNPs using path coverages are shown in *Matched* column. We used a SNP validation set as reported in Table 3 to confirm the SNP found in the filtered sets and reported as *Matched* in Table 3. The index column shows the branching vertices of the coloured de Bruijn graph. The subG column shows the total number of subgraphs processed using a maximum of 3000 vertices per subgraph. The index values was divided among 10 compute nodes for the 10 and 50 genomes experiments and 15 nodes for the 30 genomes experiment.

Many of the cycles in the Cyc row shown in Table 3 are indels that are not reported in this paper. The 10-genomes experiments found 84806 SNPs, 30-genomes experiment 110410 SNPs, and the 50-genomes experiment found 122631 SNPs that matched the SNP set. The increase in number of SNPs found with the increase in the number of genomes used, and the relatively small size of index used for subgraph creation indicates the design approach we used is useful to enumerate a large number of SNPs when combining many genomes. The graph complexity is indeed a major issue for very large number of colours.

Table 3 compares our method to DiscoSNP++ application, a similar de novo multi-colour de Bruijn graph bubble search that uses Bloom filter. We run the experiments without a reference genome to avoid any bias in bubble search. The parameters used for DiscoSNP++ are $k = 63$, $b=2$, $P=15$, and the remaining are kept as default values. According to the user document that comes with the application, $b$ and $P$ are the two parameters that are directly involved in bubble search. We choose $b = 2$ as we wanted to get as many bubbles as possible from DiscoSNP++ results in order to align the paths with the Pombe reference genome, and $P = 15$ to allow for large number of SNPs to be reported per bubble. The default values are $b = 1$ and $P = 1$, and as reported by the authors will ensure.

![Cycles number decrease with the index](image-url)
Table 4. DiscoSnp++ Results Analysis. #: number of genomes used. Matched: SNPs that matched the SNP set. bwa_coh: Found bubbles in the coherent set are run with bwa on the reference genome. bwa_uncoh: Found bubbles in the uncoherent set are run with bwa on the reference genome.

<table>
<thead>
<tr>
<th>#</th>
<th>bwa_coh</th>
<th>Matched</th>
<th>bwa_uncoh</th>
<th>Matched</th>
<th>tot</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>57006</td>
<td>53347</td>
<td>12710</td>
<td>10876</td>
<td>64223</td>
</tr>
<tr>
<td>30</td>
<td>90740</td>
<td>82926</td>
<td>15169</td>
<td>11118</td>
<td>94044</td>
</tr>
<tr>
<td>50</td>
<td>103424</td>
<td>99110</td>
<td>17732</td>
<td>12722</td>
<td>111832</td>
</tr>
</tbody>
</table>

5 DISCUSSION

The multi-genomes coloured graph produced by the raw assembly can be big, yet simple in structure and complexity if raw sequencing reads are cleaner in comparison to smaller graphs with very noisy reads. Cycles enumeration takes far longer to run in complex interconnected graphs built with large nucleotides sequencing error rates. A parallel graph decomposition, implemented as a neighbourhood subgraph as shown in this paper, lowers graph vertices interconnection complexity while the distributed search using multiple compute nodes and CPU-cores decrease dramatically the overall cycle’s search. This type of graph decomposition is also ideal for a hybrid CPU/GPU compute node. Running large complex genomes simultaneously will have a major speed boost in de novo coloured de Bruijn graph for variants calling.

We used Cortex-con de Bruijn graph assembler which is outdated and it hasn’t been checked for code optimisation. We believe our parallel method can be used with any couloured de Bruijn graph assembler as part of a variant analysis pipeline. In general, algorithms manipulating data externally are well-known to be slower in comparison to full memory based representation, we believe that our graph decomposition method can be used successfully for very large number of colours using DBG. In addition, we believe that a hybrid GPU/CPU platform could be used efficiently to accelerate the search phase in very complex multi-colour de Bruijn graph. Graph error cleaning can improve drastically genome assembly and our method can be used in this context with no modification. The other main feature of the application is the down stream data analysis program for SNP predictions. The coverage pattern exhibited in cycles can be classified as either an allele in a heterozygous genome, a SNP or a repeat. However, genome sequences are not free from errors which is the reason path coverages can be misleading. Machine learning can deal with samples that contain noise and outliers which make it an ideal field to analyse coverage datasets. We developed a simple coverages method using a heuristic path analysis approach and we are looking at ways in extending the machine learning classification approach first proposed in.

The main issue encountered while designing II-cyc is the trade-off between CPU speed and memory overhead. That is, the additional time to access memory and the extra space needed for graph storage. An MPI architecture distributed among several compute nodes incur a higher time in memory access while in a shared memory architecture, using C++ openMP as an example, the issue is overcome. Especially that HPC compute node are increasing their CPU-cores and memory dramatically. However, the actual implementation uses stored hash table files that are uploaded on each compute node memory to increase CPUs availability for the parallel graph search. An alternative approach is using message passing interface on a single node to send requested edge list for subgraph creation. In this case, the speed and memory trade-off is also a design issue to be taken into consideration.

Furthermore, we believe that our parallel search method can be incorporated as a plugin for the several emerging de novo based colored de Bruijn graph assemblers using succinct data representation (SDBG). Succinct de Bruijn graphs showed promising results in compression using very large number of colours. SDBG have indeed
shown that coloured graphs can be used in de novo population based genomics assembly for variant calling especially that modern sequencing technologies are increasingly producing cleaner samples and in large quantities. The emphasis is on developing cycle enumeration algorithms for variant calling that are fast and efficient. Our “all versus all” method would be suitable for comparing a very large number of genomes simultaneously with or without a reference genome. We showed that a single HPC compute node with 80 CPU-cores can be used efficiently with Π-cyc application for SNP calling in transformed de Bruijn graph. We believe that our method is a step forward for using heterogeneous platforms, such as those found in the HPC compute nodes.

6 CONCLUSION

In this paper, we reported a novel design to parallelise the computational intensive task of finding bubbles in coloured de Bruijn graphs. The parallelism we introduced is incorporated in the graph creation and cycles search. The graph decomposition proposed take advantage of multi-core CPUs found in multi-node platforms. Our main approach is to allocate a portion of the graph index to all available multi-core CPUs in a shared memory compute node, and also designed to be use in a multi-nodes in a distributed memory platform. Cycle enumerations algorithms in big and complex graphs are time consuming. We showed that using our method, on an HPC parallel platform, can help reduce considerably the CPU time of a single/dual CPU-core machine. The time required to search complex graphs for specific sizes cycles depends on the number of machines available to the user. Found cycles can then be translated into SNP using path coverages. Our overall application can help the scientific community work with large and complex de novo genomes for SNP calling. We identified a number of possible parallel extensions that will improve graph building and memory access. In addition, we believe using heterogeneous GPU/CPU platforms can increase dramatically the acceleration of graph search which will allow the analysis of extremely large number of big genomes simultaneously, thus benefiting population and comparative genomic research.

ACKNOWLEDGEMENTS

The programming and testing phases were used on Taito HPC servers at the Finnish IT center for science in Espoo (CSC Finland: https://www.csc.fi/). Our experiments were conducted on Kale and Ukko2 HPC servers located at the University of Helsinki (https://wiki.helsinki.fi/display/itsci/)

FUNDING

Conception, implementation and testing by RY. This work was supported in part by JT with the Academy of Finland grant 317680.

REFERENCES


