Advanced Algorithms in Computational Biology DM845

Martin Østergaard Villumsen
mvill11@student.sdu.dk

University of Southern Denmark

May 15, 2015

Contents

1 Introduction

The human genome is diploid which means that each cell has two homologous copies of each chromosome, one from the mother and one from the father. In order to understand the genetic basis for different diseases (e.g. cancer) it is not sufficient to detect the single nucleotide polymorphisms (SNPs), we also need to assign each SNP to the two copies of the chromosome and current methods for doing so suffers from the fact that the reads are too short [1].

In this project we will develop a simple read simulator, which generates reads that are much longer than those obtained from e.g. Next-Gen sequencing machines. We will simulate reads with different parameters such as read length and sequencing error probability and assign SNPs to each chromosome from these reads using WhatsHap, a novel dynamic programming approach to haplotype assembly described in [1]. We will then evaluate the results and compare them with those presented in [1].

1.1 Biological Problem

The task of assigning SNPs to a concrete chromosome is called phasing and the resulting groups of SNPs are called haplotypes. We need to discover and phase all these SNPs in order to gain a better understanding for e.g. some diseases by linking possibly disease-causing SNPs with one another. By doing so we can reconstruct haplotypes from a collection of sequenced reads which is known as haplotype assembly.

1.2 Computational Problem

There are two groups of single nucleotide variants: Those that are homozygous and those that are heterozygous. Individuals that are homozygous at every locus or heterozygous at just one locus can easily be phased, however, if we have m heterozygous SNPs, there are 2^m possible haplotypes which illustrates that it is a hard computational problem. Therefore we are only concerned with heterozygous SNPs when doing haplotype assembly.

What also makes this a computational hard problem is the fact that we want to phase an reconstruct the haplotypes directly from sequencing reads.

2 Haplotype Assembly with WhatsHap

There are two major approaches to phasing variants: One approach relies on genotypes as input along with the zygosity status of the SNPs, and the other approach phases directly from sequencing read data [1]. WhatsHap belongs to the latter.

WhatsHap has been developed with the parsimony principle in mind, i.e. computing two haplotypes to which we can assign all reads while minimizing the amount of sequencing errors to be corrected or removed [1]. This resembles the minimum error correction (MEC) problem which consists of finding the minimum number of corrections to the SNP values to be made to the input in order to be able to arrange the reads into two haplotypes without conflicts [1]. This can easily be adapted to a weighted version of the problem, namely the weighted minimum error correction problem (wMEC). The weight in this case reflects the relative confidence that a single nucleotide is correctly sequenced.

Even though the wMEC problem is NP-hard WhatsHap creates an exact solution to the problem in linear time. This is done by the use of dynamic programming and assuming a bounded coverage, i.e. the implementation can solve the problem in linear time on datasets of maximum coverage up to 20x [1]. The algorithm is a fixed parameter tractable approach to the wMEC where the running time is only depending on the coverage, i.e. the number of reads that cover a SNP position.

The input to the wMEC problem is a matrix with entries $\in \{0, 1, -\}$ where each row corresponds to a read and each column corresponds to a SNP position. Each entry is associated with a weight telling how likely it is that the entry is correctly sequenced. We want to find a minimum weight solution and when these weights are log-likelihoods, summing them up corresponds to multiplying probabilities, thus finding the minimum weight solution corresponds to finding a maximum likelihood bipartition of the reads [1]. And this is basically what the authors of WhatsHap is using a dynamic programming approach to do. By using this approach they find an optimal solution to the wMEC problem for each column of the matrix and then an optimal bipartition of the reads can be obtained by backtracking along the columns of the dynamic programming table. See [1] for further details.

The implementation of WhatsHap can be found at `https://bitbucket.org/whatshap/whatshap` and the documentation can be found at `https://whatshap.readthedocs.org/en/latest/`. The software requires a `VCF`-file and a `BAM`-file as input. `VCF`-files describe gene sequence variations, i.e. a list of variants and genotypes for the diploid genome, and `BAM`-files are the compressed binary equivalent of `SAM`-files, which are *Sequence Alignment/Map*-files that contain sequence data in a series of tab delimited ASCII columns, i.e. aligned reads for the individual diploid genome that are coordinate sorted (i.e. by chromosal position).

3 Building a DNA Simulator

There already exists different DNA read simulators but for this project we decided to design and implement our own simple simulator to generate data for WhatsHap. The lecturer of the course has provided a reference chromosome (`chr1.fa`) which we have used as a basis for the simulation of DNA reads. The chromosome is in `FASTA`-format (`.fa`) which is a text-based format for representing nucleotide sequences.

For this implementation we have used `Python 2.7.9`. Also, for this implementation we assume that there are no `N`s in the chromosome, i.e. the `N`s present in `chr1.fa` are removed and the only letters present are: `A, T, C, G`.

3.1 Generating a Chromosome with Mutations

Before building the actual simulator we needed a script to generate a `VCF`-file describing the chromosome with mutations. Generating the `VCF`-file is done with a simple `Python` script that reads the chromosome, removes all `N`s (if there are any) and copy the chromosome one nucleotide at a time. Every 400 + 'random integer between 0-100' we simulate a mutation (i.e. a single nucleotide variant) by randomly choosing one of the remaining nucleotides. E.g. if the current nucleotide is `A` we randomly choose a nucleotide from {`T, C, G`}. For every simulated mutation a line is written to the `VCF`-file with the following values:

- CHROM: `'chr1'`
- POS: The position of the nucleotide in the chromosome
- ID: The position of the nucleotide in the chromosome
- REF: The nucleotide from the reference chromosome
- ALT: The "new" nucleotide (i.e. the variant)
- QUAL: `'.'`
- FILTER: `'PASS'`
- INFO: `'.'`
- FORMAT: `'GT'`
- sample: `'0/1'`

These values were primarily chosen based on miscellaneous testdata for WhatsHap.

This `VCF`-file can then be used to generate a copy of the chromosome with mutations which we will need for simulating the DNA reads.

See `buildVCF.py` for details on the implementation.

3.2 Simulating DNA Reads

There are two types of variants which can occur in DNA reads: Sequencing errors and mutations. We need to take both into account when designing a read simulator. Also, we need to be able to choose how many reads the simulator should generate, how long the reads should be and the percentage of sequencing errors that should occur in the simulated reads. This is all given as parameters to the program. We have designed the read simulator as follows:

1. Read the `FASTA`-file (reference chromosome) and the `VCF`-file (both are given as input) and remove line breaks, comments etc.

2. Generate a copy of the reference chromosome with mutation from the `VCF`-file.

3. Generate reads one at a time as follows:

   a. Choose one of the two chromosomes at random.

   b. Choose a random start position for the read.

   c. Iterate over the next x nucleotides, where x is the read length.

   d. For each nucleotide in the read, alter the nucleotide with probability e, where e is the probability that a sequencing error occurs.

4. Repeat a. - d. r times where r is the number of reads we want to generate.

When all reads have been generated the program generates a `SAM`-file describing the reads, which is written to a file. We have chosen to leave out many of the values of the `SAM`-file to keep things simple. We have chosen to use the following header:

```
@HD VN:1.4 GO:none SO:coordinate
@SQ SN:chr1 LN:'length of chromosome'
@RG ID:1 PL:Platform LB:library SM:sample
```

along with the following values for each line of the file:

- QNAME: `'r00%d'` where '%d' is the start position of the read in the reference chromosome
- FLAG: `'0'`
- RNAME: `'chr1'`
- POS: 'Start position of the read in the reference chromosome'
- MAPQ: '255' (this indicates that the value is not available)
- CIGAR: `'%dM'` where '%d' is the length of the read
- RNEXT: `'='`
- PNEXT: `'0'`
- TLEN: 'Length of the chromosome'
- SEQ: 'The actual read'
- QUAL: 'A string of length r consisting of "J"s, where r is the length of the read'

These values were primarily chosen based on miscellaneous testdata for WhatsHap.

For details about the individual field see the SAM Format Specification at `https://samtools.github.io/hts-specs/SAMv1.pdf`. For details about the implementation of the simulator see `sim.py`.

3.3 Using the Simulator

In order use the simulator that is described above we first need to simulate a `VCF`-file describing a mutated version of the reference chromosome. As mentioned earlier we have used chromosome 1 (`chr1.fa`) for reference chromosome. The script to generate a `VCF`-file is run with the following command:

```
python buildVCF.py chr1.fa chr1.vcf
```

where `chr1.vcf` is the file that will be generated by the script. `chr1.vcf` can then be used for simulating the DNA read data. This is done with the following command:

```
python sim.py chr1.fa chr1.vcf numberReads readLength errorProb chr1.sam
```

where `numberReads` and `readLength` are integers stating the number of reads to simulate and the length of the reads respectively. `errorProb` is a float and states the sequencing error probability and `chr1.sam` is the sequence alignment/map file that will be generated which describes the simulated reads.

In order to run WhatsHap on the simulated data we need to convert the `SAM`-files to their binary equivalent `BAM`-files. This is done by the use of `samtools`[1] and a small shell script:

```
for f in *.sam; do samtools view -bS $f | samtools sort -
    ${f\%.sam}.sorted; done
```

which convert and sort every `SAM`-file in a folder and output their equivalent `BAM`-files as `filename.sorted.bam`, where `filename` is the filename of the input `SAM`-file. The output `BAM`-files can then be used as input to WhatsHap along with the generated `VCF`-file (`chr1.vcf`).

3.4 Phasing the Simulated Data

In order to phase the simulated data we ran WhatsHap with the following command:

```
python3 -m whatshap --ignore-read-groups chr1.vcf chr1.sorted.bam >
    phased.vcf
```

where `chr1.vcf` and `chr1.sorted.bam` are the input files which were generated with `buildVCF.py` and `sim.py` and `phased.vcf` is the output file describing the SNPs and what chromosome each variant belongs to.

4 Results and Discussion

WhatsHap has been developed with future generation of sequencing machines in mind which generates much longer reads than then currently used technologies such as NextGen. Current technologies generate reads of ≈ 150bp - 250bp while future generation sequencing machines generate reads of ≈ 1000bp, however, the sequencing error rate is also higher than the currently used technologies. We have therefore chosen to simulate data with the following parameters:

- Number of reads:

  100.000

  500.000

  1.000.000

- Read length:

  500

  1000

  1500

[1] See `http://www.htslib.org/` for more information about `samtools`

- Error rate:

  5%

  10%

  15%

Giving us a total of 27 simulations which we have phased using WhatsHap, however we have only been able to phase 15 of those simulations. When trying to phase the remaining 12 simulations with WhatsHap we received the following assertion error when trying to read the BAM-file given as input:

```
File "/home/martin_15/.local/lib/python3.4/site-packages/whatshap/
    scripts/whatshap.py", line 153, in read
   assert 0 < len(readlist) <= 2
AssertionError
```

and after digging into the source code of WhatsHap (more specifically whatshap.py line 153) it seems that the error was caused by the fact that none of the simulated reads cover at least one variant. Also, the 12 simulations that we were unable to phase included all simulations with 1.000.000 reads and 3 with 500.000 reads, so it seems that the more reads the more likely it is that the assertion error arise. We suspect that this is due to some error in the implementation of our read simulator but we have not been able to determine the concrete problem.

4.1 Results

In order to evaluate the results of the phasings we were able to do we made a small Python script to compare the 'sample'-value (i.e. the value stating which chromosome the SNPs belongs to, e.g. '0/1') of the chr1.vcf-file and the output VCF-files from WhatsHap and counting matches and mismatches in order to calculate the percentage of correctly phased SNPs. This script was called with the following command:

```
python evaluate.py chr1.vcf phased.vcf
```

where chr1.vcf is the file generated by buildVCF.py and phased.vcf is the output file from WhatsHap. For details of the implementation see evaluate.py.

The results have been plotted using an R-script[2] and the percentage of correctly phased SNPs is illustrated in Figure 1 for 100.000 reads and and Figure 2 for 500.000 reads. As we can see on the two plots it seems that the shorter the reads the more matches are obtained and also the percentage of matches is a bit higher if we use 100.000 reads instead of 500.000 reads. This seems a bit odd since WhatsHap have been developed for longer reads. However, the error rate does not seem to influence the number of matches by much.

We have also made two plots illustrating the performance of the phasing in terms of the percentage of unphasable positions versus the error rate. Figure 3 illustrates the performance of phasing for 100.000 reads and Figure 4 shows the same plot for 500.000 reads. These two plots clearly shows that the length of the reads have a great influence on the amount of unphasable positions and comparing Figure 3 and Figure 4 with Fig. 2 in [1] we achieve somewhat similar results

[2] Implemented in results.R

that show that the reads of length 1000 and 1500 have a lot lower percentage of unphasable positions than the reads of length 500. Also, the percentage of unphasable reads seems to increase slightly with the error rate but the increase is so small that it seems to be irrelevant.

4.2 Time and Memory Performance

One of the advantages of WhatsHap is the low memory and time usage, e.g. as mentioned in [1] the software can solve any problem instance with 15x coverage in less than 10 minutes on a single core CPU. Therefore we have also measured the memory and time usage of WhatsHap for the different parameters of our simulations. This was done using `/usr/bin/time` in the shell with the following command:

```
/usr/bin/time -v -o outputFile.txt python3 -m whatshap
    --ignore-read-groups chr1.vcf chr1.sorted.bam > chr1.out.vcf
```

which saves all memory and time usage in `outputFile.txt`. We have plotted all the results obtained from measuring the memory and time usage of the phasing for all our simulations. Looking at the measurements the error rate did not seem to have any significant influence on the memory and time usage so we have chosen to take the average of the measurements with the same number of reads and read length and used these average measurements for the plot.

The results are illustrated in Figure 5 which shows that even for the most time and memory consuming dataset (500.000 reads and reads of length 1500) WhatsHap only used approximately 3 minutes and 1.2GB of memory. The plot illustrates that the increase in time and memory usage seems to be growing linearly with the length of the reads and also that the number of reads has a great influence on the time and memory usage which seems quite obvious.

5 Conclusion

Haplotype assembly and phasing SNPs is important in terms of understanding, diagnosing and treating diseases such as cancer. WhatsHap is one tool among many for haplotype assembly, however, it is the first to solve the weighted minimum error correction problem with long reads to optimality in practice [1].

The purpose of this project was to develop a simple DNA read simulator to simulate long reads and use WhatsHap to phase the simulated data. We have developed and implemented a DNA read simulator which seems to be working and produce fairly acceptable results, however, when generating $\geq$ 500.000 reads the simulator produces reads where none of the reads seems to be covering any variants. At least that is why WhatsHap is failing and we have not been able to determine the concrete cause of this problem.

We simulated reads with a mixture of three different number of reads, length of reads and error rate, i.e. a total of 27 simulations, of which we were able to phase 15 simulations with WhatsHap.

The results of the haplotype assembly with the simulated data seems to agree somewhat with the results presented in [1] (See Figure 3 and 4) although the percentage of correctly phased SNPs in general seems to be a bit low with the simulated data compared to what is presented in [1]. So it seems that

WhatsHap is a very efficient tool for haplotype assembly for long reads and even though we were unable to run WhatsHap on the vast amount of simulated reads (i.e. $\geq$ 500.000) we still managed to obtain results that agree somewhat with those presented in [1].

References

[1] M. Patterson, T. Marschall, N. Pisanti, L. van Iersel, L. Stougie, G. W. Klau, and A. Schönhuth. Whatshap: Haplotype assembly for future-generation sequencing reads. In *Research in Computational Molecular Biology - 18th Annual International Conference, RECOMB 2014, Pittsburgh, PA, USA, April 2-5, 2014, Proceedings*, pages 237–249, 2014.

A Figures

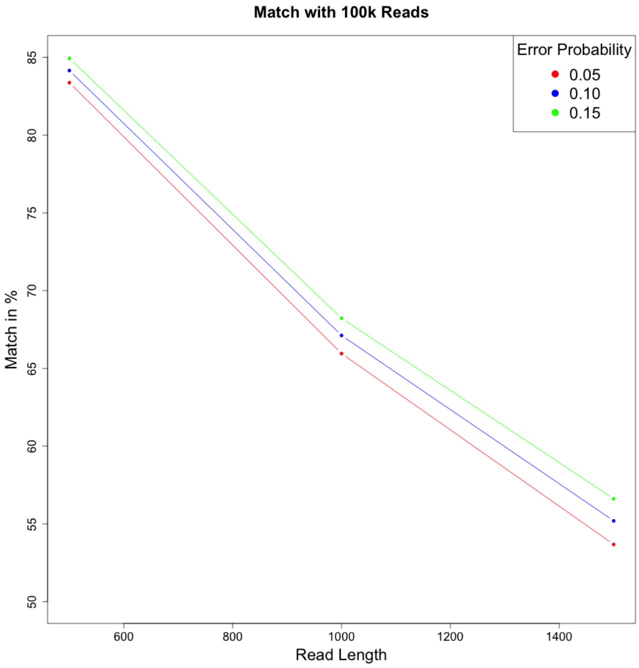

Figure 1: Results of haplotype assembly of 100.000 reads.

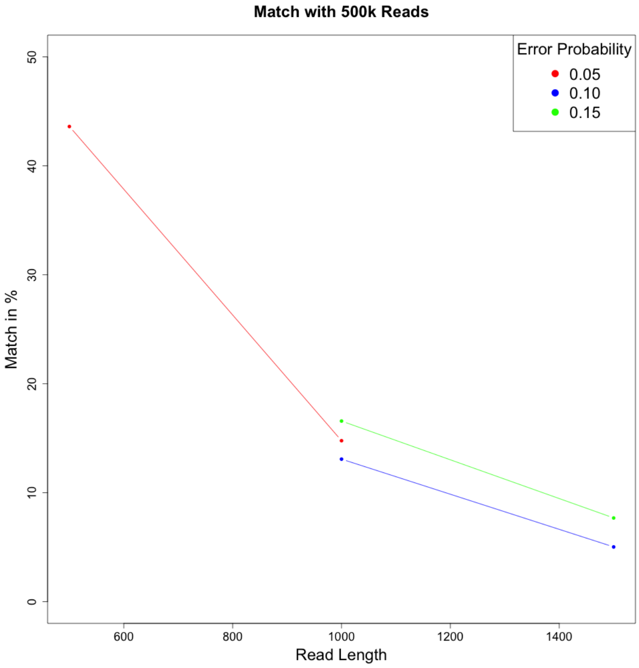

Figure 2: Results of haplotype assembly of 500.000 reads.

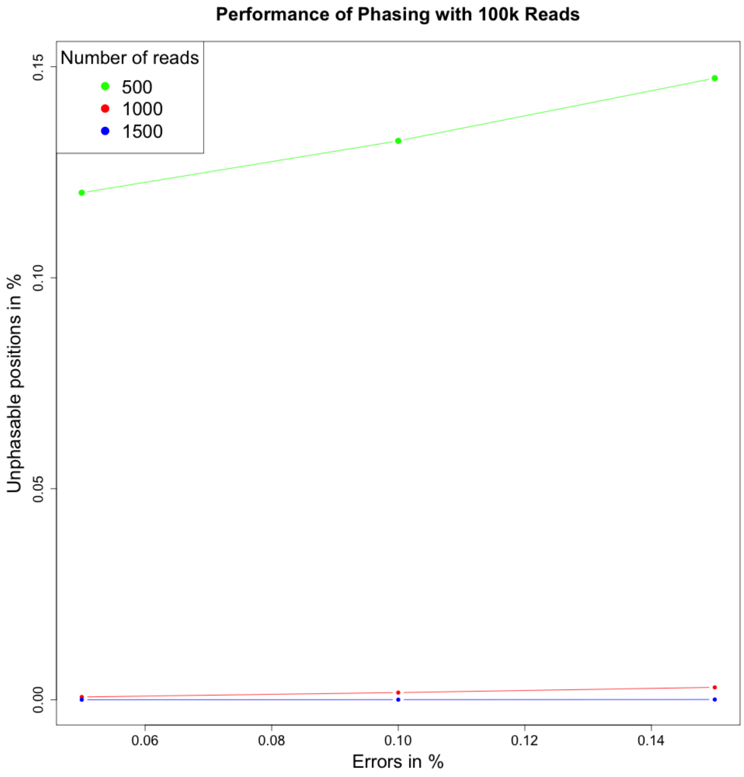

Figure 3: Performance of phasing with 100.000 reads.

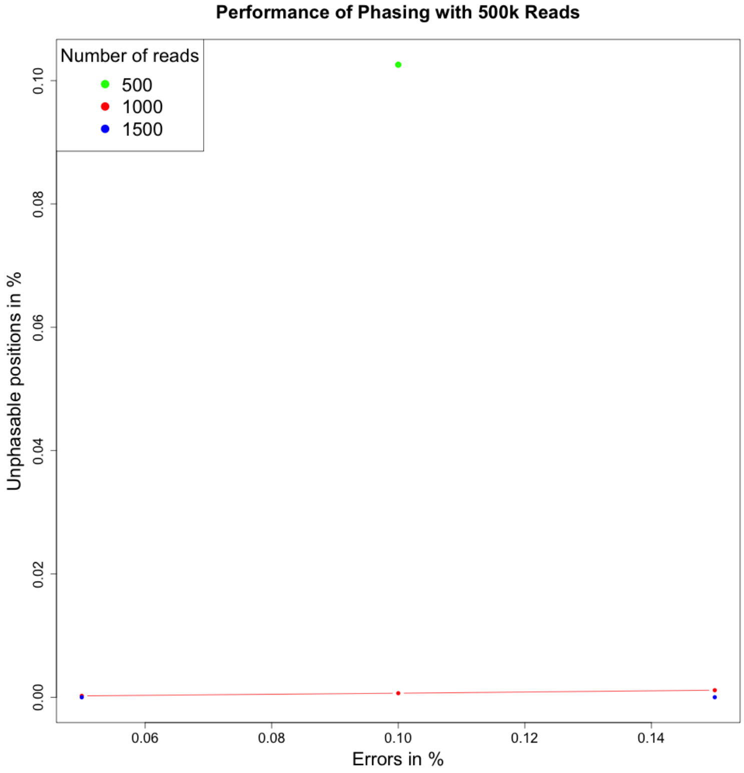

Figure 4: Performance of phasing with 500.000 reads.

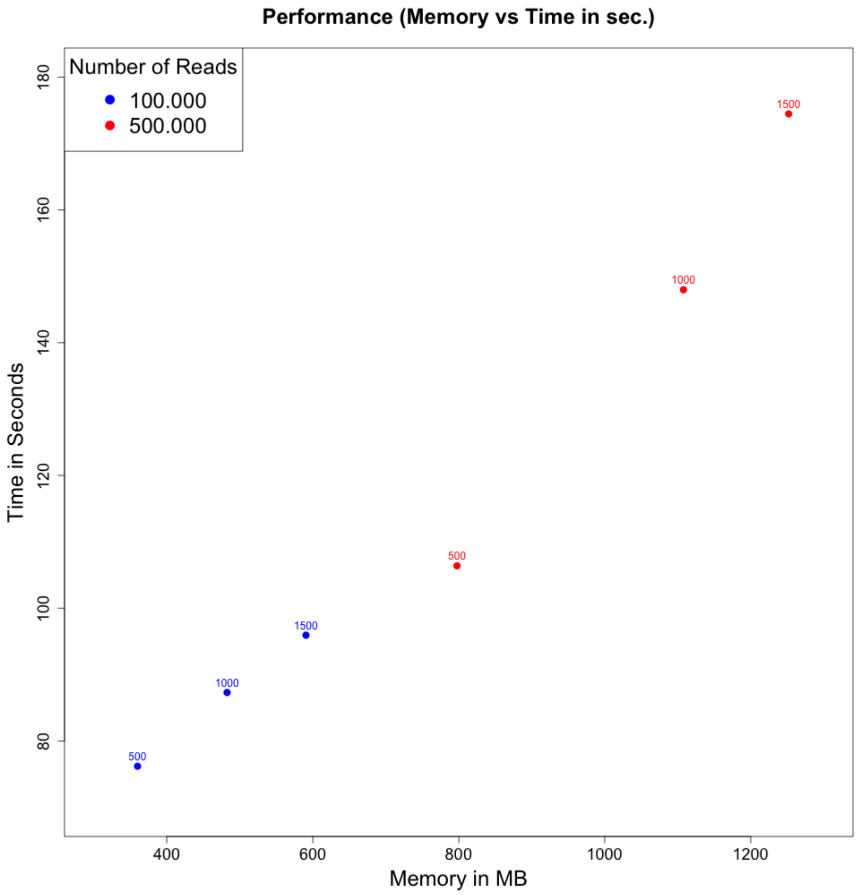

Figure 5: Performance measured in memory (in MB) and time (in sec.) and with two different number of reads. The numbers above each point indicates the length of the reads.