Parahaplo: A Heterogeneous Haplotype Solver
Accelerating Graph Search with GPU Processing

Robert Clucas
School of Electrical and Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg,South Africa

***Abstract*—Diploid organisms, such as humans, have two copies of each chromosome. Haplotyping involves partitioning sequences of aligned DNA reads from each of the two chromosomes to determine a haplotype from each chromosome. Minimum error correction (MEC) is one formulation of the problem, which involves correcting the minimum number of SNPs to infer the haplotypes from the reads, and is an NP-hard problem This report presents the graph based GPU component of Parahaplo, a heterogeneous haplotype solver. Parahaplo uses CPUs to process the input data before offloading the data to GPUs which find the haplotypes. The GPU component partitions sub-sets of the reads into two disjoint sets by calculating the similarity between the reads. The partitions are then refined until convergence. Parahaplo is compared to the current best existing solution. Results show that Parahaplo produces high, but not optimal haplotypes, on average, but that the variance between quality of the haplotypes is greater than desired. The performance of Parahaplo is shown to be far superior to same existing solution, with a x50 run-time improvement in some cases. Additionally, Parahaplo scales significantly better as the coverage and number of SNPs in the input increases. Future work is required to verify the effectiveness of Parahaplo on the largest real-life input data sets.**

***Keywords*—Fragment; Graph; GPU; Haplotype; Reduce**

I. Introduction

Determining the haplotypes for a strand of DNA is a major problem in the field of bioinformatics, as haplotypes can significantly impact areas such as drug discovery and disease diagnosis [1], [2]. Modern DNA sequencing technologies can provide accurate *fragments*, each covering a number of *single nucleotide polymorphisms* (SNPs) from an individual's DNA. However, for species which have multiple chromosomes there are multiple possible *nucleotide base pairs* for each SNP site. The sequencing technology is unable to distinguish which of the chromosomes each of the nucleotides in the pair belongs to, and is only able to specify the type of each of the nucleotides in the base pair. For *diploid* organisms, which have two copies of each chromosome, a *haplotype* (h) is a sequence of nucleotides from a chromosome, while a *haplotype pair* (H) consists of the haplotypes from each chromosome.

Consider Figure 1 (left matrix), H_1 and H_2 are possible haplotype pairs. Within H_1 and H_2, the top sequence of nucleotides is from one chromosome, while the bottom is from another, and each sequence is a haplotype. The sequencing technology reports that the nucleotide bases for each SNP are {A,A}, {T,T}, {G,C}, {T,A}, {A,T}, {A,T}, but does not specify which chromosome each nucleotide in the pair comes from. The haplotyping problem involves determining the order of the base pairs, which produces the haplotypes. The problem is combinatorial, and has 2^n possible solutions, where n is the length of the haplotypes, since there are two combinations of the nucleotides at each position. The red and purple sequences show two possible orderings of the unknown positions for the input shown in Figure 1.

There are two main forms of computational methods for solving the haplotyping problem. The first is *haplotype inference* (HI), where the haplotypes are inferred from the genotype samples of a population. The work presented in [3], [4] uses this model. The second is *haplotype assembly* (HA), based on shotgun sequencing technology which generates a vast number of short fragments of an individual's genome. Each of the fragments are sequences of aligned SNP sites. Fragments are also called *reads*. For each SNP site, the site is *homozygous* if all nucleotides have the same value, and *heterozygous* if nucleotide values differ. The HA problem is suited to large scale haplotyping [5], and has received a lot of interest in recent years, resulting in numerous formulations of the problem. *Minimum fragment removal* (MFR) and *minimum SNP removal* (MSR) are presented in [6], while [7] present the *minimum error correction* (MEC) formulation, and prove it to be NP-hard. This paper presents Parahaplo, a heterogeneous haplotype solver for the MEC formulation. The focus of the paper is the GPU accelerated graph search component of the solver.

The remainder of the report is structured as follows. Section II reviews existing solutions, and details GPU architectures. Section III describes the problem. Section IV gives an overview of Parahaplo. Section V describes the GPU accelerated graph search algorithm, and its implementation. Section VI presents the results. Section VII provides an analysis of the results. Section VIII describes possible future work. Section IX concludes.

II. Background

A. Haplotype Assembly

HapCut [8] is commonly used for comparison in related literature due the accuracy of the haplotypes it finds. The problem is formulated as a graph for which the max-cut must be found, which an NP-complete problem [9]. HapCut, however, does not scale well [10]. More recently, [11] present an integer linear programming solution. The MEC scores produced are the best amongst current literature. The high quality haplotypes are achieved by making the assumption that SNP sites can be both homozygous and heterozygous and that fragments may contain gaps, where other solutions assume only heterozygous columns and no gaps. The high quality haplotypes, however, come at the cost of long compute times for some inputs, which can be up to hundreds of times longer than HapCut for example (on average they are faster than HapCut). Another graph based solution is FastHap, proposed in [12], which requires $O(m^2)$ iterations, but does significant work before the iterative process. The produced MEC scores better than other most other literature, and the run-times are, on average, 8.1 times faster than HapCut. None of the existing implementations, however, make use the the wide range of available parallel hardware. As the computational requirements increase linearly with the coverage and quadratically with the total length of the reads [10], the existing solutions can require days of computation for large inputs. Parahaplo is a parallel implementation of [12] through a hybrid CPU-GPU system which offloads expensive computations to the system GPUs. Additionally, the approach of [12] is extended with Parahaplo by making the assumptions of [11].

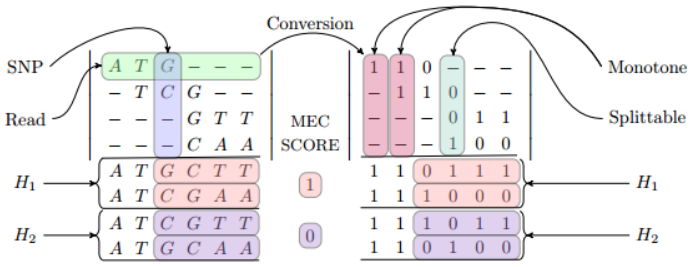

Fig. 1. Illustration of the input data and possible haplotypes.

B. Heterogeneous Systems & Parallel Processing

Heterogeneous systems use multiple types of processors to improve computational performance. Parahaplo uses CPUs (with Intel TBB [13]) for data processing and GPUs (with CUDA [14]) to accelerate the computation of haplotypes, in parallel. Parahaplo targets NVIDIA GPUs based on the sophistication of the CUDA software development kit, used to program the GPUs. NVIDIA GPUs are highly parallel, many-core architectures, which comprise of multiple streaming multiprocessors (SMs). SMs are special purpose computational units with dedicated memory and registers, and groups of threads called warps, which execute in a Single Instruction, Multiple Data (SIMD) manner. The multiple SMs allow the GPU to provide Single Instruction, Multiple Thread (SIMT) processing. SIMT executes an instruction across many threads on the GPU, in parallel. The highly parallel nature of GPUs makes them appropriate when the problem is data-parallel (the same operation can be used for all the data) rather than task-parallel (different tasks are required for different parts of the data), however, increased performance can be achieved in both instances. Threads are organised in blocks, and each block has a small shared memory space which can be accessed by only the threads associated with the block. Each thread also has access to a larger global memory space (up to 24GB for high end GPUs). Global memory is undesirable as the bandwidth is significantly lower than the shared memory regions. Due to the complex architectures of GPUs, achieving increased performance through the additional parallelism is typically difficult. Multiple parallel programming paradigms exist which intend to simplify parallelism, such as MapReduce [15], from which Parahaplo bases much of its operations – specifically the reduction component of MapReduce.

III. Problem Description

For each read, the 3-tuple {0,1,-} represents the nucleotides, where a '1' corresponds to a match with a reference sequence, a '0' represents a divergence from it, and a '-' represents a gap. This choice, however, does not matter so long as consistency is maintained. The aligned reads then make up a *block*, B, which is an mxn matrix whose elements come from the 3-tuple. For the remainder of the paper m is used for the number of reads, and n for the number of SNPs. The j^{th} SNP site from the i^{th} read of a block is given by x_{ij}. From B, the problem requires computing the haplotype pair $H = \{h, h'\}$, and two partitions of the reads from which each haplotypes are generated. The right matrix of Figure 1 shows an example of the converted input data and two possible haplotype pairs. The MEC formulation of the problem defines the MEC score for evaluating a solution as given by Equation 1.

$$MEC = \sum_{i=1}^{m} \min\{d(r_i, h), d(r_i, h')\} \tag{1}$$

Where r_i is the i^{th} read and d is the *Hamming Distance* between two binary strings and determines the number of positions for which the elements of two binary strings have different values. The definitions of an *intrinsically heterozygous* (IH) column and a *non-intrinsically heterozygous* (NIH) column from [11] are used to potentially produce better MEC scores. Essentially, an IH column is a column which is known to be heterozygous. NIH columns, however, could be either homozygous or heterozygous, but which specifically is not known with certainty due to errors in the input data. NIH columns require an additional step in the computation. Parahaplo assumes reads may contain gaps, since gaps are presents in real-life data sets. The following properties are defined for the input matrix. Figure 1 illustrates these properties.

Monotone Columns: A column is monotone if all values in the column which are not gaps have the same value. Monotone columns are equivalent to homozygous SNPs when the DNA data is not abstracted to binary data. The solution for a monotone column is trivial since the haplotype takes the value of the elements in the monotone column, thus monotone columns can be removed from B and replaced in the final solution.

Singular Rows: A row is singular if it has only one element. Singular rows do not affect the solution as they can be aligned to the opposite partition to ensure that they do not increase the MEC score. Singular rows are removed from B.

Splittable Columns: Informally, a column is splittable if for all the reads in B, the column does not lie between start and end column (non-inclusive) of any read, and is NIH. Splittable columns allow B to be decomposed into *unsplittable sub-blocks* (referred to as sub-blocks from here onward) which can be solved in parallel, since the solution of a sub-block does not increase the MEC score of another sub-block.

Read Similarity: The cross operator which acts on two variables x and y, from [12], is defined by Equation 2.

$$x \otimes y = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \textbf{ and } x, y \in \{0, 1\} \\ 0.5 & \text{otherwise} \end{cases} \tag{2}$$

The cross operator is used to determine the similarity, Δ_{ij}, between two reads i and j in a sub-block, as given by Equation 3.

$$\Delta_{ij} = \frac{1}{X_{ij}} \sum_{k=1}^{n} x_{ik} \otimes x_{jk} \tag{3}$$

Where n is the number of SNPs in the sub-block, and X_{ij} is the number of SNPs for which either read i or read j has a value which is not a gap. The similarity between all the reads make up the similarity matrix Δ.

IV. Parahaplo Overview

Parahaplo comprises of a splitting and reconstruction system, and a solving system. The two systems operate concurrently to split blocks, solve the haplotype for the resulting sub-blocks, and reconstruct the final haplotypes for a block.

A. Data Conversion

The input DNA data is first converted to its binary representation, as shown in Figure 1. Each of the elements in a read is converted to a 2-bit value and stored in a contiguous array. Gaps before the start and after the end of the read are not stored, thus the space complexity is $O(L)$, where L is the total length of the reads.

B. Splitting & Reconstruction System

The splitting and reconstruction (S&R) system uses the available CPUs to find all splittable columns, from which the sub-blocks are determined. Determining if a column is splittable requires searching through all reads for which the column lies between the start and end index, which requires $(mn)/p$ operations, where p is the number of CPU cores, as the evaluation of each column can be done in parallel. The worst case complexity, when there is only a single CPU core in the system, is $O(mn)$. Additionally, the S&R system removes monotone columns and singleton rows from a sub-block. Sub-blocks are then offloaded to the solving sub-system, which determines the haplotypes of the sub-blocks using the avaliable GPUs.

When the haplotypes of a sub-block are found, they are reconstructed using the CPUs to find the haplotypes of a block. Reconstruction involves replacing the monotone columns in the sub-block solutions and joining the solutions of each sub-block. This is done serially since each sub-block's solution may need to be inverted, based on the solution of the previous sub-block, since the start and end column of adjacent blocks are common. Inverting the sub-block solution does not increase the MEC score.

C. Solving System

Parahaplo's solving system uses the FastHap algorithm of [12], but performs all steps with parallel operations, and considers NIH columns. First, a fuzzy conflict graph, $G(V, E, \Delta)$ is created, where each vertex, V_i, is a read, E is a set of edges for which edge e_{ij} connects vertex i and j, and the similarity matrix Δ defines the weights of the edges, whose elements are determined by Equation 3. For here onward, vertex is used interchangeably with read. The initialization phase of the algorithm determines Δ from the vertexes, and sorts the vertexes in descending order of magnitude. The partitioning phase uses the similarity matrix is to create a partition set, $P = \{P_1, P_2\}$, of the vertexes. Each partition, P_1 or P_2, consists of vertexes which are most similar. Once the partitions are created, the refinement phase iteratively modifies the partitions by moving vertexes between partitions until the MEC score does not improve. Algorithm 1 shows a high-level overview of the solving process. Figure 2 shows an example of a graph, the corresponding similarity matrix, and the result of the initialization phase.

Algorithm 1 High-level solving algorithm

```
Define: e_ijn = next edge to partition
Inputs: V, E, Δ, P
Outputs: P, H

Δ ← DetermineSimilarity(reads)                (1) Initialization
Δ ← Sort(Δ)
Δ ← RemoveInvalid(Δ)

while not all V ∈ P do                        (2) Partitioning
    e_ijn ← Max(Δ)                              if i or j ∈ P
    P     ← MoveVertex(V_i, V_j)
    e_ijn ← Min(Δ)                              if i or j ∈ P
    P     ← MoveVertex(V_i, V_j)
    P     ← MoveAllVertexes(V)                  if e_ijn not found
end while

while MEC improves do                         (3) Refinement
    H ← FindHaplotypes(P, V)
    H ← EvaluateNihColumns(V,H)
    P ← SwapWorstVertex(P,V)
end while
```

V. Graph Solving System

A. Initialization Phase

The initialization phase first creates the similarity matrix. Each element in the matrix requires n operations (for each SNP) for the sum in Equation 3. To improve the computational performance, m thread blocks, each with n threads, are used to perform the sum in parallel. Each block determines the similarity for a pair of vertexes i, j (i.e computes Equation 3). Each of the n threads loads the result of the cross operation at SNP site k ($x_{ik} \otimes x_{jk}$) into a shared memory array. Each array is then reduced [16] by the n threads to find the similarity for a vertex pair. The overall complexity is $O(\log(n))$, since each reduction requires $\log(n)$ operations, and each of the m reductions is done in parallel. Serially this has $O(mn)$ complexity.

Following the creation of the matrix, the elements are sorted in descending value. The similarity matrix is flattened in memory by storing the elements in a row-major format, allowing contiguous storage, which is beneficial for GPU performance [17]. Only the upper or lower half of the similarity matrix is stored since it is symmetric ($\Delta_{ij} = \Delta_{ji}$), which requires $m(m-1)/2$ space. The matrix is then sorted using a bitonic sort, which is shown in [18] to achieve better performance than other parallel sorting implementations when optimized. The bitonic sort requires a power of two number of threads, thus while only storing half the similarity matrix is beneficial in terms of memory, computationally the benefits are less significant. However, since time complexity of the bitonic sort, $O(\log^2(m^2))$, is logarithmic, the difference is negligible.

Lastly, the initialization removes all elements from the sorted similarity matrix which have a value of 0.5. A value of 0.5 means that there is not sufficient information from the fragments to accurately select a partition during the partition phase. The sort implementation is designed to move all elements with a value of 0.5 to the end of the array, allowing them to be easily removed. The process of determining the similarity matrix, and the operations on it, is illustrated in Figure 2.

B. Partitioning Phase

The goal of the partitioning phase is to determine the partitions, P_1 and P_2, such that all vertexes in P_1 are as similar as possible, all vertexes in P_2 are as similar as possible, and that vertexes in P_1 are as dissimilar as possible to vertexes in P_2. The intuition is that if all vertexes in a partition are similar, a haplotype can be produced from the partition with a minimum MEC score (if the reads in the partition are all the same, no bits

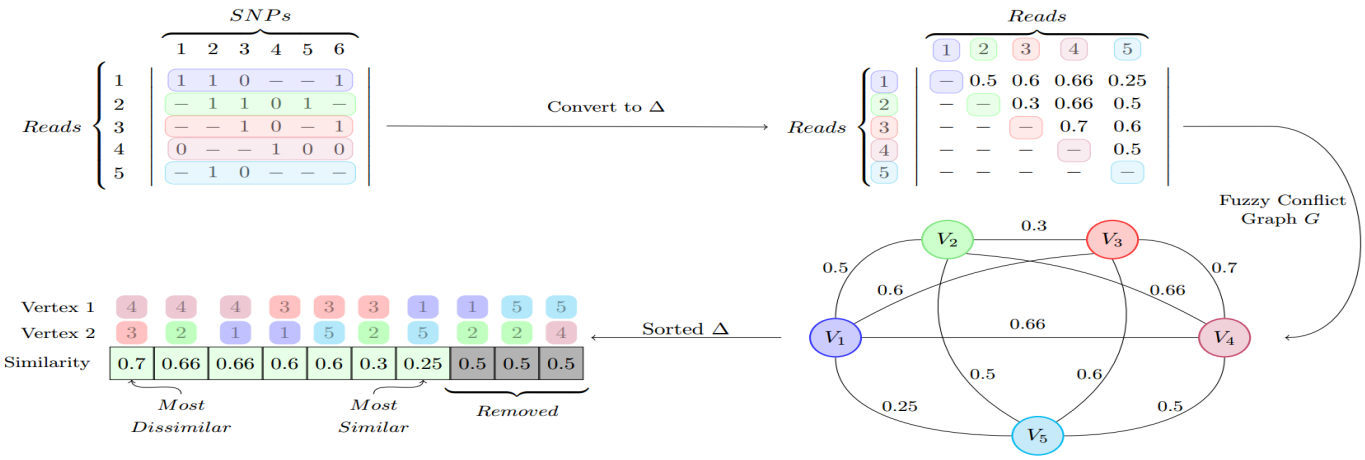

Fig. 2. Illustration of the relationship between the input data, similarity matrix, graph, and the sorted similarity matrix.

in the haplotype will need to be flipped, resulting in a perfect MEC score).

The similarity matrix allows an informed choice when moving vertexes into one of the partitions. For two reads, Equation 3 produces a value of zero for identical reads, and a value of one for exactly opposite reads, thus a similarity close to one represents large dissimilarity, and a similarity close to zero represents large similarity. The bitonic sort of the similarity matrix thus arranges the similarities such that the most dissimilar edge is at the start of the array, and the most similar edge is at the end of the array, which is illustrated by Figure 2. The partitioning begins by assigning the vertexes corresponding to the most dissimilar edge to opposite partitions (3 and 4 in Figure 2). The next most dissimilar edge (**Max**(Δ)), such that one of the vertexes which the edge joins is in either P_1 or P_2, is then found (edge 2-4 in Figure 2, since 4 would be partitioned in the first step). It then assigns the partitioned vertex to partition opposite to which the connected (and partitioned) vertex belongs (2 will be assigned to the opposite partition from 4). The same process is applied for the most similar edge (**Min**(Δ)), but the vertex is assigned to the same partition, since the vertexes are similar. This process continues alternating between most and least similar until there are two complete partitions.

A binary array of of length m is used to represent each partition, which requires $2m$ space for both partitions. Each binary element in the array represents whether the vertex at the array index exists in the partition. Only m space is required in the optimal case, since a vertex can only be in either partition. However, the partitions need to be searched constantly since for an edge e_{ij}, it needs to be known if either V_i or V_j is in the partition. Using only m space would make the look up and insertion complex, thus the decision to use the sub-optimal space in order to get $O(1)$ look up and insertion performance. The justification is that the additional space of size m, is tolerable for practical systems. Each index requires one byte, resulting in m extra bytes, thus even for large sub-blocks with over a million reads the wasted space is in the range of MB, which is acceptable for even low end GPUs.

If no edges are found for either of the stages above such that one of the vertexes exists in a partition, then the partitioning process ends, and all vertexes which do not belong to a partition are added to the partition which they are most similar. For an unpartitioned vertex, the similarity between itself and the partitioned vertexes to which it is connected, are loaded into two shared memory arrays, one for each partition. Each array is reduced to determine the similarity between the vertex and the partition. The vertex is then added to the partition to which it is most similar.

The partitioning loop, in the worst case, requires partitioning all m vertexes in the loop, which has time complexity of $O(m)$ since lookup and insertion into the partitions are constant time. In the case that no elements are partitioned initially, adding vertexes to the partitions to which they are most similar requires $O(m \log(m))$, since the similarity for each vertex requires a single reduction, and the partitions are updated each time a fragment is added to a partition so all fragments can be added simultaneously.

C. Refinement Phase

The refinement phase aims to move fragments which were incorrectly partitioned. The refinement phase extends the FastHap algorithm by evaluating NIH columns. First the haplotypes are found from the existing partitions, by determining, for each SNP site in a partition, if there are more ones or zeros from the reads which make up the partition. Determining the number of ones and zeros is essentially finding sum of two arrays whose elements represent if the element at a SNP position is a one or a zero. Which requires $O(\log(n))$ time for each SNP with a parallel reduction. The haplotype then takes the value of the element with the greatest number of occurrences, since the count of the other element is the contribution to the MEC score.

For NIH columns, the optimal value for each haplotype can be set, since the haplotype can be monotone or non-monotone, however, for IH columns the haplotype values must be opposite, which is considered when setting the haplotype, and is the additional constraint Parahaplo enforces.

The last step of the refinement involves finding the vertex which contributes most negatively to the MEC score. The argument of sum in Equation 1 defines the contribution of a single vertex to MEC score, which can again be computed with a parallel reduction, since it is a sum whose elements are independent. A block of threads is used to reduce each vertex to find its MEC contribution. The most negative contibuting vertex must then be found from the results, which requires an additional reduction. The time complexity of the first reduction is $O(\log(n))$ as the reduction is over the SNPs. The second

TABLE I. RESULTS FOR THE PARAHAPLO AND CHEN ET AL. [11]

c	General Case			All Heterozygous			Parahaplo		
	MEC	RR	Time	MEC	RR	Time	MEC	RR	Time
l = 100									
3	53.3	96.2	0.03	66.6	92.7	0.02	90.2	77.2	0.004
5	95.4	99.0	0.16	132.9	92.0	0.11	142.6	86.8	0.007
8	157.3	99.7	0.62	249.4	90.1	0.52	196.4	91.9	0.011
10	195.8	99.9	0.94	327.2	89.2	1.02	212.4	95.7	0.033
l = 350									
3	186.2	96.5	0.45	233.5	93.0	0.31	337.3	80.3	0.081
5	338.4	98.8	3.05	477.2	91.4	2.68	607.3	82.4	0.138
8	547.1	99.7	16.5	909.8	88.6	41.4	815.6	91.3	0.351
10	682.0	99.9	30.5	-	-	-	991.4	89.0	0.602

reduction has time complexity of $O(\log(m))$ as it is over the reads. The total complexity of finding the worst read is $O(\log(m) + \log(n))$.

VI. RESULTS

To evaluate the quality of the solutions, the simulated datasets from [10] are used. Simulated datasets are chosen for their ease of use, as well as their popularity amongst related work. The quality and complexity of the data is defined by three parameters, namely, the number of SNPs in the input matrix, l, the coverage of the reads, c, and the error rate, e. Results are shown for lengths $l = \{100, 350\}$, converages $c = \{3, 5, 8, 10\}$ and an error rate of 10%. Each result is an average of 20 runs across 3 different simulated matrices with the same configuration. A further benefit of the simulated data is that the exact solution is known, allowing the *reconstruction rate* (RR) to define how closely a solution matches the optimal solution, which is given by Equation 4.

$$RR = 1 - \frac{\min\{d(h, \hat{h}) + d(h', \hat{h}'), d(h, \hat{h}') + d(h', \hat{h})\}}{2l} \quad (4)$$

Where $\hat{H} = \{\hat{h}, \hat{h}'\}$ is optimal haplotype pair. The RR is a measure of the average number of corrections which need to be made to the haplotype such that becomes optimal. A RR of one represents a match with the optimal solution.

The results, for both the performance and the quality of the haplotypes, are compared to the solutions of the general and all-heterozygous cases from Chen et al. [11] since they are the most optimal amongst existing literature. Table I shows the results, while Figure 3 shows the total time required by Parahaplo to reconstruct the haplotypes for the simulated data. The figure also provides a breakdown of the amount of time required by each component of Parahaplo.

The results shown for Chen et al. are taken from their paper. They use an Intel i7-3960X (6-core) CPU and 31.4GB of system ram. Their solution uses IBMs CPLEX solver, a highly optimised, commercial linear programming solver. The results for Parahaplo use an Intel Xeon E5-2670 v3 (12-core) CPU, and 128GB of system ram, as well as an Nvidia K40 (2880 core, 12GB ram) GPU.

VII. ANALYSIS

A. Haplotype Assembly

The results show that the haplotypes which are reconstructed are comparable in quality to the all-heterozygous case of the solutions of Chen et al., but cannot provide the same quality as their general case. Since they currently provide the only solutions for the general case, and their all-heterozygous case has performance comparable to other related work, Parahaplo is able to construct good, but not optimal, haplotypes.

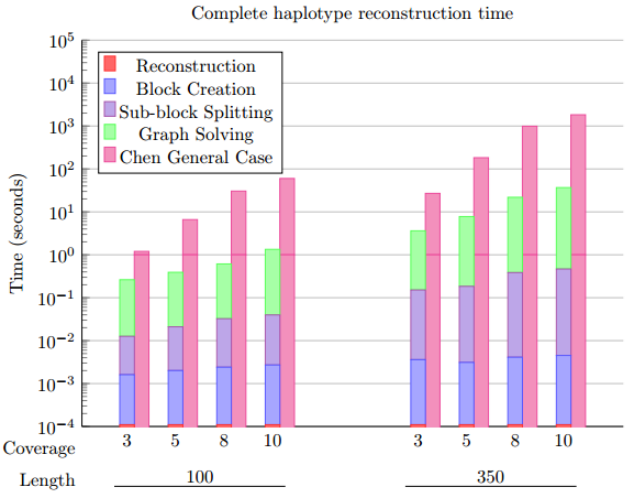

Fig. 3. Comparison of Parahaplo against the solution of Chen [11]. Higher blocks within columns show more time-critical sections. Note that the y-axis is logarithmic.

An important conclusion from the results is that Parahaplo is not able to effectively use the additional information provided by the NIH columns for all inputs. While the average results are presented, the results had significant variation. For some inputs, such as for length 100 and coverage 10, and length 350 and coverage 8, Parahaplo found solutions with RRs of 97.5 and 94.5 respectively, providing evidence that Parahaplo has the potential to provide extremely accurate haplotypes. However, the variance of the haplotypes requires a solution. It is also noted that the quality of Parahaplo's solutions increases with the coverage, which is a positive result. This, however, is expected considering the algorithm. As the algorithm heavily utilises the similarity between the reads, when the coverage is low the algorithm cannot determine good partitions because the covariance of the similarity matrix is minimal, and many elements are removed during the sort due to having a similarity of 0.5. When the coverage increases, the reads become more diverse, and the covariance of the similarity matrix is increases, allowing the algorithm to create good partitions.

B. Performance

Comparison of the compute times shows Parahaplo to be far superior to Chen et al. While this expected given the additional available hardware, the intention of the results is to show that Parahaplo can benefit from the parallelism the additional hardware provides, as there is no existing parallel solution. While the test system has more memory than the system used by Chen et al., the solving is done on the GPU which has x2.5 less memory. During benchmarking, it was determined that all runs used less than 100MB of memory, illustrating that Parahaplo provides an extremely memory-efficient solution. Parahaplo scales better with increased coverage than Chen et al. Where Chen et al. has an x66 increase in run-time for a x3.3 increase in coverage ({350,3} to {350,10}), Parahaplo has an increase of x7.5. More significantly, the run-time increase of Parahaplo, for increased read length across the coverages, is approximately constant i.e the increase between {100,3} and {350,3} is the same as {100,10} and {350,10}. However, for Chen et al.'s solution this is not the case, and there is a linear increase. This is shown in Figure 3 by the increasing difference in bar height between the two solutions as the coverage increases. The results suggests that the performance

of Parahaplo will scale better for the larger, real-life data sets. The maxmimum run-time improvement of Parahaplo over Chen et al. is x50 (for {350,10}).

VIII. Future Work

The results show that effective use is not made of the information provided by NIH columns, which results in sub-optimal haplotypes, and potentially the variance. As the implementation of Chen et al. utilises the information to significantly improve their solutions, the information is useful. Possible approaches may be to determine a similarity matrix between the SNPs, in addition to the matrix for the reads, and use the information to modify haplotypes during the refinement stage.

While the system is able to achieve significantly increased performance for the small simulated data sets, the results need to be verified on real-life data sets, which is where the benefit of the increased performance is required as existing solutions require hours to days of computation on sequences of human chromosomes. Additionally, recent developments in sequencing technology have produced long reads with an average of 8849 base pairs and an average coverage of 54 [19]. Future work would involve verifying the results on such data sets.

Analysis of the individual GPU graph search code sections showed that the partitioning phase severely limits the computational time of the system. The reason is most likely the simplicity of the implementation. Analysing and re-implementing critical sections of the graph search code could result in severely reduced compute times.

IX. Conclusion

Results show that Parahaplo is able to achieve significantly better run-times than the current optimal solutions in terms of haplotype quality. Parahaplo scales better when both the number of SNPs in the input matrix, and the coverage, increases. For some inputs, Parahaplo is able to achieve up to x50 run-time reduction for the hardest block over the solution of Chen et al. The quality of the reconstructed haplotypes is approximately equivalent with Chen et al.'s all-heterozygous solution, and better for large coverages. For real-life data, however, the inputs are not all-heterozygous. The FastHap algorithm on which Parahaplo is based, was extended to consider SNP sites which may be either homo or heterozygous. Parahaplo does not fully utilise the information. The result of the lack of information usage produces sub-optimal haplotypes when compared to Chen et al.'s general solution, except for some inputs for which Parahaplo was able to find an almost optimal haplotype, to within 5%. The cases where Parahaplo found almost optimal haplotypes were when the coverage was large, which is encouraging considering recent developments in sequencing technology allowing large coverage sequencing with many base pairs. Parahaplo shows that performance can be improved through parallelism, but that further work is required to make Parahaplo an optimal haplotype solver.

References

[1] R. W. Morris and N. L. Kaplan, "On the advantage of haplotype analysis in the presence of multiple disease susceptibility alleles," *Genetic Epidemiology*, vol. 23, no. 3, pp. 221–233, 2002. [Online]. Available: http://dx.doi.org/10.1002/gepi.10200

[2] I. H. Project, "How will the hapmap benefit human health?" 2015, [Online]. Available: http://hapmap.ncbi.nlm.nih.gov/healthbenefit.html [Accessed: 19 October 2015].

[3] L. Wang and Y. Xu, "Haplotype inference by maximum parsimony," *Bioinformatics*, vol. 19, no. 14, pp. 1773–1780, 2003. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/19/14/1773.abstract

[4] E. Halperin and E. Eskin, "Haplotype reconstruction from genotype data using imperfect phylogeny," *Bioinformatics*, vol. 20, no. 12, pp. 1842–1849, 2004. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/20/12/1842.abstract

[5] R.-S. Wang, L.-Y. Wu, Z.-P. Li, and X.-S. Zhang, "Haplotype reconstruction from snp fragments by minimum error correction," *Bioinformatics*, vol. 21, no. 10, pp. 2456–2462, 2005. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/21/10/2456.abstract

[6] G. Lancia, V. Bafna, S. Istrail, R. Lippert, and R. Schwartz, "Snps problems, complexity, and algorithms," in *Algorithms ESA 2001*, ser. Lecture Notes in Computer Science, F. auf der Heide, Ed. Springer Berlin Heidelberg, 2001, vol. 2161, pp. 182–193. [Online]. Available: http://dx.doi.org/10.1007/3-540-44676-1 15

[7] R. Lippert, R. Schwartz, G. Lancia, and S. Istrail, "Algorithmic strategies for the single nucleotide polymorphism haplotype assembly problem," *Briefings in Bioinformatics*, vol. 3, no. 1, pp. 23–31, 2002. [Online]. Available: http://bib.oxfordjournals.org/content/3/1/23.abstract

[8] V. Bansal and V. Bafna, "Hapcut: an efficient and accurate algorithm for the haplotype assembly problem," *Bioinformatics*, vol. 24, no. 16, pp. i153–i159, 2008. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/24/16/i153.abstract

[9] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, ser. The IBM Research Symposia Series, R. Miller, J. Thatcher, and J. Bohlinger, Eds. Springer US, 1972, pp. 85–103. [Online]. Available: http://dx.doi.org/10.1007/978-1-4684-2001-29

[10] F. Geraci, "A comparison of several algorithms for the single individual snp haplotyping reconstruction problem," *Bioinformatics*, vol. 26, no. 18, pp. 2217–2225, 2010. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/26/18/2217.abstract

[11] Z.-Z. Chen, F. Deng, and L. Wang, "Exact algorithms for haplotype assembly from whole-genome sequence data," *Bioinformatics*, vol. 29, no. 16, pp. 1938–1945, 2013. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/29/16/1938.abstract

[12] S. Mazrouee and W. Wang, "Fasthap: fast and accurate single individual haplotype reconstruction using fuzzy conflict graphs," *Bioinformatics*, vol. 30, no. 17, pp. i371–i378, 2014. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/30/17/i371.abstract

[13] Intel, "Intel Thread Building Blocks," 2015. [Online]. Available: https://www.threadingbuildingblocks.org/docs/help/index.htm

[14] NVIDIA, "CUDA C Programming Guide," September 2015. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[15] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[16] J. Luitjens, "Faster parallel reductions on kepler," 2014. [Online]. Available: http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

[17] M. Harris, "How to access global memory efficiently in cuda c/c++ kernels," 2013. [Online]. Available: http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

[18] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, "Fast in-place, comparison-based sorting with cuda: A study with bitonic sort," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 7, pp. 681–693, May 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1686

[19] J. Huddleston, S. Ranade, M. Malig, F. Antonacci, M. Chaisson, L. Hon, P. H. Sudmant, T. A. Graves, C. Alkan, M. Y. Dennis, R. K. Wilson, S. W. Turner, J. Korlach, and E. E. Eichler, "Reconstructing complex regions of genomes using long-read sequencing technology," *Genome Research*, 2014. [Online]. Available: http://genome.cshlp.org/content/early/2014/01/13/gr.168450.113.abstract