Parallelizing the Individual Haplotying Assembly Problem

Robert J. clucas

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

Abstract:

Key words: Brach, Bound, GPU, Haplotyping, Simplex

1. INTRODUCTION

It is commonly accepted that all humans share ∼99% of the same DNA, however, small variations cause human beings to have different physical traits. Single nucleotide polymorphisms (SNPs), which are variations of a single DNA base from one individual to another, are believed to be able to address genetic differences. For diploid organisms, which have pairs of chromosomes, a *haplotype* is a sequence of SNPs in each copy of a pair of chromosomes. A *genotype* describes the conflated data of the haplotypes on a pair of chromosomes. Haplotypes are believed to contain more generic information than genotypes [1], however, obtaining haplotypes correctly is a difficult problem, which is broken into two subdomains: haplotype assembly and haplotype inference.

Haplotype inference uses the genotype of a set of individuals. The genotype data tells the status of each allele at a position, but does not distinguish which copy of the chromosome the allele came from. This negative aspects of this approach are that it cannot distinguish rare and novel SNPs [2], and there is no way of knowing if the inferred haplotype is completely correct.

Individual haplotype assembly uses fragments of sequences generated by sequencing technology to determine haplotypes. The fragments of a sequence come from the two copies of an individual's chromosome, the goal of the individual haplotyping problem is to correctly determine two haplotypes, where each haplotype corresponds to one of the two copies of the chromosome.

The haplotype assembly problem was proven to be NP-Hard [3]. The algorithms used to solve the problem are thus computationally complex and until recently, there was no practical exact algorithm to solve the problem using minimum error correction (MEC) [4], However, recently an exact solution was proposed by [5] which is capable of solving the MEC problem exactly, and can thus correctly infer all haplotypes from the fragment sequences. Due the NP-Hardness of the problem, the algorithm results in long run times - in the range of days for chromosomes with high errors rates. Using a parallel implementation of any of the proposed solutions could reduce the long run times, allowing useful haplotype information to be quickly inferred from the available datasets, having positive effects in fields such as drug discovery, prediction of diseases, and variations in gene expressions, to name a few.

Come back to introduction ...

Parallel programming makes use of devices which have many simple cores, but which can execute the same instructions on each of the cores at the same time. The effectiveness of parallel programming is dependant on the nature of the problem, as per Amdahl's law. The first attempts at parallel programming came from Graphics Processing Units (GPUs) which were used to render many pixels simultaneously. More recently, General-Purpose GPU (GPGPU) programming has become prominent with API's like CUDA and OpenCL, allowing access to GPUs from C and C++ programs.

2. HAPLOTYPE ASSEMBLY PROBLEM

This section will provide a brief overview of the haplotype assembly (HA) problem, and define the notation used through the rest of the paper. The input to the problem is a set of reads from a given genome sequence, where each read contains fragments from each of the two chromosomes which make up the genome sequence. These characters of a read consist of elements from a *ternary string*, where a ternary string has characters from the set {0, 1, -}. A value of 0 refers to the major allele at a site, a value of 1 to the minor allele, and a value of - to the lack of a read at the site, and is referred to as a *gap*. These reads are then combined to form a matrix, M, where each row of the matrix corresponds to a read.

Each column of the matrix is known as an SNP site. At each site, the data could be accurate, missing, or have error. The goal of the haplotype assembly problem is to determine a haplotype, H = {*h,h'*} from the matrix. The following terminology will be used to refer to properties of the matrix and the fragments.

For the input matrix M, the number of fragments is denoted by m, which is the number of rows in M. The number of SNP sites is denoted by n, which is the number of columns in M, while the j^{th} site of the i^{th} fragment is given by f_{ij}. Furthermore, two fragments are said to conflict if the following conditions are true:

- $f_{ik} \neq f_{jk}$ **and** $f_{ik} \neq$ '-' **and** $f_{jk} \neq$ '-'

Essentially this means that for two fragments i and j, if at an SNP site k, the reads do not have error and are not gaps, the reads have different values (fragment i has a value 0 at site k, while fragment j has a value 1 at site k, or vice versa).
Following the notion of a conflict, the *distance* between two fragments (or ternary strings) is denoted by d(f_i, f_j) is the total number of positions for which the two fragments f_i and f_j conflict. Furthermore, to understand some of the problem formulations from the fragment data, it is useful to define a *conflict graph* [6] G = { V, E }, where V corresponds to a fragment, and E corresponds to and edge between two fragments if they conflict. If the matrix M contains no errors, then none of the fragments from the same chromosome will conflict and G will be bipartate. Fragments from the same chromosome may conflict. However, if there are errors (as is usually the case) in M, G will not be bipartate. The haplotype assembly problem then requires the correction of G from a non-bipartate graph to a bipartate graph. There are numerous methods for solving the problem:

- **Minimum Fragment Removal (MFR) :** This invloves removing the least number of fragments from the input data such that the resultant graph G is bipartate. It is shown in [6] that this can be solved in polynomial time.
- **Weighted Minimum Edge Removal (WMER) [7] :** This method is a recent method, which requires determining the minimum number of edges to remove such that removal of the edges results in G being bipartate.
- **Longest Haplotype Reconstruction (LHR) :** This requires finding a set of fragments which, when they are removed from M, result in G being bipartate and the length of the resultant haplotypes being maximized [8].
- **Minimum Error Correction (MEC) :** This method involves correcting the minimum number of elements (sites for all fragments) in the matrix M which allow the graph G to be bipartate. Although being the most complex method, it is the most widely used method as it provides the highest accuracy rates. Only recently has an exact algorith for the MEC problem formulation been proposed which can provide an exact solution for all cases.

Due to the accuracy and more extensive previous work, the MEC method for solving the problem was decided upon as the method for which a parallel implementation will be proposed.

2.1 Minimum Error Correction Implementations

Much research has been done on solving the MEC formulation of the HA problem. The first exact algorithm for solving the problem was a branch and bound algorithm proposed by [9]. The algorithm creates a tree which covers the search space of all possible corrections. The tree is then traversed to find the best solution. They use an upper bound for when branching that allows branches to be abandoned when a better solution than the current best cannot result from further exploration of the branch, thus improving performance. However, this exact method has time complexity of $O(2^m)$, where m is the number of fragments in the input data, and hence cannot produce solutions for large problem sizes. They also provide a heuristic *genetic algorithm* (GA) to improve the computational time, which only requires run times up to three orders of magnitude faster than the branch and bound implementation. While the GA implementation gives very similar results to the branch and bound implementation, it is slightly worse.
A dynamic programming solution was proposed by [10] which addresses the run time problem for large input sizes. The algorithm has time complexity of $O(mk3^k + mlogm + mk)$, where k is the maximum number of SNP sites a fragment covers. In practice k is usually small, and results were shown small k (less than 100). The proposed dynamic programming solution had significant run time improvements over the solution proposed by [9]. However, for larger k values, the algorithm cannot solve the MEC case of the HA problem in a feasible amount of time.
More recently, [5] proposed an exact algorithm for solving the MEC problem. The proposed algorithm is the currently the only algorithm which can solve the HA problem for both the homozygous (the alleles at a site are identical) and hetrozygous (the alleles at an SNP site are different) case. Most other work assumes that the input fragment data is hetrozygous, which, while true for most of the SNP sites in the input matrix, is normally false for a small number of the SNP sites. The exact solution removes unnecessary data from the matrix, and partitions the matrix into smaller sub-matrices. The problem is then formulated as an *integer linear programming* (ILP) problem and solved as an optimization problem. The implementation was run using an Intel i7-3960X CPU, and the HuRef dataset required 15 days to solve for the general case, and 24h for the all-hetrozygous case. Heuristics methods are also proposed which speed up computation time by up to 15 times. However, the results are only shown for smaller input sizes, and the heuristic methods do not always determine the optimal solution. Furthermore, the solutions for the general case show lower MEC scores, meaning that the all-hetrozygous assumption is not always valid.

2.2 Parallelization

Of the MEC implementations discussed in Section 2.1, the branch and bound solution proposed by [9], and the ILP formulation of the problem proposed by [5] have the most potential for a parallel implementation. While there are other methods for solving ILP prob-

lems [11] the most common methods for solving ILP problems are:

- **Branch and bound :** The search space is divided into sub spaces, each of which is explored for the optimal solution. Bounds are enforced to ensure that sections of the sub spaces for which an optimal solution will not be found, will not be searched.
- **Branch and cut :** The problem is first solved without the integer constraints to find the optimal solution. Cutting planes are then used to divide the search space and then branch and bound is applied.

Thus both the MEC methods mentioned above can be solved using a branch and bound algorithm. The branch and cut algorithm involves using a branch and bound method as well as the simplex method, and is thus more complex, hence the choice was made to use the branch and bound algorithm.

3. BRANCH AND BOUND

The branch and bound algorithm can be parallelized in multiple ways [12]. Either specific, computationally difficult functions can be accelerated by a parallel implementation, for example matrix inversion, or the entire tree can be search in parallel, or a combination of both can be employed. Using tree based approaches will require communication between each of the processes which searches a branch, as the branch solution will need to be compared against the solutions of the other branches. This becomes a bottleneck for both CPU and GPU implementations as groups of threads are assigned local memory, for which access is much faster than the global memory space. There have been numerous parallel branch and bound methods proposed for both the CPU and GPU which deal with these problems in different ways.

3.1 CPU Implementations

The ALPS framework [13] is written in C++ and provides a parallel CPU implementation of the branch and bound algorithm. The framework is tested on the knapsack problem [14], which is an ILP problem and is known to be NP-hard. The speedup achieved is near linear in the number of nodes when the number of nodes is small, but diminishes as more nodes are added due to the amount of communication between the nodes. A speedup of 8 times is achieved for 8 nodes, and 26 times for 32 nodes. It is likely that for the HA problem, the number of nodes could be extremely large in which case this methods will be insufficient.
The MALLBA framework [15] is also written in C++ and provides a parallel branch and bound algorithm. It labels nodes as *master* or *slave* nodes, which defines the type of work that the nodes do. The framework also provides heuristic methods to solve the problems more efficiently, however, the accuracy of the results is less, which while acceptable for many applications, is not acceptable for the HA problems. The performance results are similar to the ALPS framework, where near linear speed up is achieved for a small number of additional nodes, but levels off for larger numbers of nodes.

3.2 GPU Implementations

A hetrogeneous CPU-GPU implementation was proposed by [16] and was applied to the knapsack problem. Due to the overhead of transferring data from the CPU to the GPU before computation, the model only uses GPUs when the tree has a large number of nodes (> 5000), otherwise the CPUs are used Furthermore, the tree is built using a *breadth first* strategy to favour the parallel nature of the GPUs. Both the branching and bounding steps can be performed on the CPU or GPU. If the number of nodes is sufficiently large such that full occupancy of the GPU is ensured, then for each iteration of the algorithm the GPU does the branching and bounding on a list of nodes, eliminates nodes for which an optimal solution cannot be found, and return the list back to the CPU. This process continues until convergence. This regular communication between the CPU and GPU is expensive, which lessens the speedup achieved by the implementation. However, a speedup of up to 9.27 times was achieved for larger problem sizes, validating the feasibility of the branch and bound algorithm for parallel implementation. It must be noted that the algorithm was developed for Nvidia's Fermi architecture, which does not support dynamic parallelism [17], hence results in the vast CPU-GPU communication. The newer Kepler architecture, however, does support dynamic parallelism.
In [18, 19]. algorithms are proposed which target the bounding operation for parallelism, as well as dealing with the problem of thread divergence when using GPUs for branch and bound, which comes from the position of the nodes in the tree, and the need to for branches to communicate with other branches to compare solutions. Depending on the problem to which branch and bound is applied, the tree structure can be irregular which makes the entire tree search unsuited for GPUs not supporting recursion and dynamic parallelism, as was the case when the algorithm was proposed - hence the focus on only he bounding operation. The algorithm was applied to the Flow-Shop scheduling problem, for which it is shown that $\sim$98% of the computation is spent on the bound operation. Speed ups of up to 100 times were achieved by the GPU implementation over a multi-threaded CPU implementation. However, this kind of speedup will only be seen in problems where the calculation of the bounds is the bottleneck. Nevertheless, even if significantly less time is spent calculating the bounds, the speed up should still be considerable.
The algorithms of [18, 19] are extended by [20] to in-

clude not only parallelization of the bounding operator, but also the branching and pruning operators. This essentially uses the GPU for doing all the searching of the subspaces. However, the CPU is still used for the between each iteration as each GPU thread can only determine the solution of its first child nodes, rather than all child nodes until the leaves of the tree. Again this is due to the limitations of the Fermi architecture not supporting dynamic parallelism, despite this hardware limitation, the algorithm is still able to achieve a speed up of up to 166 times over a multi-threaded CPU implementation, for large problem sizes.

3.3 Potential Improvements

Sections 3.1 and 3.2 above provide evidence of the feasibility of taking a parallel approach to the branch and bound algorithms, and hence the feasibility of solving the HA problem with a parallel implementation. Some aspects of the related work have significant performance implications on the parallel implementation, such as:

- Using multiple CPUs does not scale effectively, and hence the relative performance increase for adding an additional CPU decreases for each CPU which is added
- The amount of communication between the CPU and the GPU is large, and is required on each iteration, which decreases the performance of the algorithm
- The structure of the tree depends on the problem - one problem may ensure a balanced tree while another may map to an unbalanced tree - which makes a GPU difficult due to its data-parallel nature

The above factors come from the limitations of the parallel hardware available at the time. Due to the lack of recursive ability for the Fermi architecture used, each time solutions need to be compared for each of the branches, and then to select the most relevant nodes from the list of searchable nodes, all results from the GPU kernels needed to be passed back to the CPU since the GPU cannot invoke kernels itself. Furthermore, the problem of an unbalanced tree was significant for the same reason. However, the newer Kepler architecture does allow kernels to call kernels, which should reduce the CPU-GPU communication and be able to handle unbalanced trees as the kernel can itself determine the best searchable nodes. The Intel Xeon Phi also supports recursion, and is thus also a good candidate for this approach.

REFERENCES

[1] J. Stephens. "Haplotype Variation and Linkage Disequilibrium in 313 Human Genes." *Science*, vol. 293, no. 5529, pp. 489–493, Jul. 2001. [Online]. Available at `http://dx.doi.org/10.1126/science.1059431` [Accessed 27 June 2015].

[2] D. He, A. Choi, K. Pipatsrisawat, A. Darwiche, and E. Eskin. "Optimal algorithms for haplotype assembly from whole-genome sequence data." vol. 26, no. 12, pp. i183–i190, 2010.

[3] R. Lippert, R. Schwartz, G. Lancia, and S. Istrail. "Algorithmic strategies for the single nucleotide polymorphism haplotype assembly problem." vol. 3, no. 1, pp. 23–31, 2002.

[4] P. Bonizzoni, G. Della Vedova, R. Dondi, and J. Li. "The Haplotyping problem: An overview of computational models and solutions." *Journal of Computer Science and Technology*, vol. 18, no. 6, pp. 675–688, 2003. [Online]. Available at `http://dx.doi.org/10.1007/BF02945456` [Accessed 27 June 2015].

[5] Z.-Z. Chen, F. Deng, and L. Wang. "Exact algorithms for haplotype assembly from whole-genome sequence data." vol. 29, no. 16, pp. 1938–1945, 2013.

[6] G. Lancia, V. Bafna, S. Istrail, R. Lippert, and R. Schwartz. "SNPs Problems, Complexity, and Algorithms." In F. auf der Heide, editor, *Algorithms ESA 2001*, vol. 2161 of *Lecture Notes in Computer Science*, pp. 182–193. Springer Berlin Heidelberg, 2001. [Online] Available at `http://dx.doi.org/10.1007/3-540-44676-1_15` [Accessed 16 July 2015].

[7] D. Aguiar and S. Istrail. "HAPCOMPASS: A fast cycle basis algorithm for accurate haplotype assembly of sequence data." *Journal of Computational Biology*, vol. 19, no. 6, pp. 577–590, June 2012. [Online] Available at `http://www.brown.edu/Research/Istrail_Lab/papers/hapcompass_jcb_draft.pdf` [Accessed 16 July 2015].

[8] R. Schwartz. "Theory and Algorithms for the Haplotype Assembly Problem." *Commun. Inf. Syst.*, vol. 10, no. 1, pp. 23–38, 2010. [Online] Available at `http://projecteuclid.org/euclid.cis/1268143371` [Accessed 16 July 2015].

[9] R.-S. Wang, L.-Y. Wu, Z.-P. Li, and X.-S. Zhang. "Haplotype reconstruction from SNP fragments by minimum error correction." vol. 21, no. 10, pp. 2456–2462, 2005.

[10] M. Xie, J. Wang, and J. Chen. "A Practical Exact Algorithm for the Individual Haplotyping Problem MEC." In *BioMedical Engineering and Informatics, 2008. BMEI 2008. International Conference on*, vol. 1, pp. 72–76. May 2008.

[11] L. Galli. "Algorithms for Integer Programming.", December 2014. [Online] Available at `http://www.di.unipi.it/optimize/Courses/RO2IG/aa1415/IP_algorithms.pdf` [Accessed 27 June 2015].

[12] T. G. Crainic, B. Le Cun, and C. Roucairol. *Parallel Branch-and-Bound Algorithms*, pp. 1–

28. John Wiley & Sons, Inc., 2006. [Online] Available at http://dx.doi.org/10.1002/9780470053928.ch1 [Accessed 17 July 2015].

[13] Y. Xu, T. Ralphs, L. Ladnyi, and M. Saltzman. "Alps: A Framework for Implementing Parallel Tree Search Algorithms." In B. Golden, S. Raghavan, and E. Wasil, editors, *The Next Wave in Computing, Optimization, and Decision Technologies*, vol. 29 of *Operations Research/Computer Science Interfaces Series*, pp. 319–334. Springer US, 2005. URL http://dx.doi.org/10.1007/0-387-23529-9_21. [Online] Available at http://dx.doi.org/10.1007/0-387-23529-9_21 [Accessed 17 July 2015].

[14] M. Kedia. "Dynamic Programming.", October 2005. [Online] Available at http://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf [Accessed 27 June 2015].

[15] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Daz, I. Dorta, J. Gabarr, C. Len, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. In B. Monien and R. Feldmann, editors, *Euro-Par 2002 Parallel Processing*, vol. 2400 of *Lecture Notes in Computer Science*, pp. 927–932. Springer Berlin Heidelberg, 2002. [Online] Available at http://dx.doi.org/10.1007/3-540-45706-2_132 [Accessed 17 July 2015].

[16] A. Boukedjar, M. Lalami, and D. El-Baz. "Parallel Branch and Bound on a CPU-GPU System." In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pp. 392–398. Feb.

[17] Nvidia. "CUDA DYNAMIC PARALLELISM PROGRAMMING GUIDE.", August 2012. [Online] Available at https://www.clear.rice.edu/comp422/resources/cuda/html/cuda-dynamic-parallelism/index.html [Accessed 17 July 2015].

[18] N. Melab, I. Chakroun, M. Mezmaz, and D. Tuyttens. "A GPU-accelerated Branch-and-Bound Algorithm for the Flow-Shop Scheduling Problem." *CoRR*, vol. abs/1208.3933, 2012. [Online] Available at http://arxiv.org/abs/1208.3933 [Accessed 17 July 2015].

[19] I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi. "Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm." *Concurrency and Computation: Practice and Experience*, 2012. [Online] Available at https://hal.inria.fr/hal-00731859 [Accessed 17 July 2015].

[20] I. Chakroun and N. Melab. "Operator-level GPU-Accelerated Branch and Bound Algorithms." *Procedia Computer Science*, vol. 18, pp. 280 – 289, 2013. 2013 International Conference on Computational Science.