ABSTRACT

Title of Document:	HIGH PERFORMANCE COMPUTING FOR DNA SEQUENCE ALIGNMENT AND ASSEMBLY.
	Michael Christopher Schatz, Doctor of Philosophy, 2010.
Directed By:	Professor Steven L. Salzberg Department of Computer Science

Recent advances in DNA sequencing technology have dramatically increased the scale and scope of DNA sequencing. These data are used for a wide variety of important biological analyzes, including genome sequencing, comparative genomics, transcriptome analysis, and personalized medicine but are complicated by the volume and complexity of the data involved. Given the massive size of these datasets, computational biology must draw on the advances of high performance computing.

Two fundamental computations in computational biology are read alignment and genome assembly. Read alignment maps short DNA sequences to a reference genome to discover conserved and polymorphic regions of the genome. Genome assembly computes the sequence of a genome from many short DNA sequences. Both computations benefit from recent advances in high performance computing to efficiently process the huge datasets involved, including using highly parallel graphics processing units (GPUs) as high performance desktop processors, and using the MapReduce framework coupled with cloud computing to parallelize computation to large compute grids. This dissertation demonstrates how these technologies can be used to accelerate these computations by orders of magnitude, and have the potential to make otherwise infeasible computations practical.

HIGH PERFORMANCE COMPUTING FOR DNA SEQUENCE ALIGNMENT AND ASSEMBLY.

By

Michael Christopher Schatz

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:
Professor Steven L. Salzberg, Chair
Professor Mihai Pop
Professor Jimmy Lin
Dr. Arthur L. Delcher
Professor Steven Mount (Dean’s Representative)

Preface

Portions of this thesis have either been published in peer-reviewed journals or are in preparation for submission. I am indebted to each of my co-authors on these studies – their dedication and expertise resulted in much stronger scientific papers. At the time of this writing, Chapters 2-5 & 7 have appeared in print and are reformatted here, and Chapter 6 has been submitted for publication. Chapter 8 has been written in preparation for submission, but is unpublished at this time. Permission for republication of this material has been granted and is available upon request.

Chapter 2

Schatz, M.C.[†], Trapnell, C.[†], Delcher, A.L., and A. Varshney. *High-throughput sequence alignment using Graphics Processing Units*. BMC Bioinformatics, 2007. 8: 474.

The authors would like to thank David Luebke from nVidia Research for providing an early release of CUDA, Julian Parkhill from the Sanger Institute for making the *S. suis* data available, Mihai Pop from CBCB for his assistance obtaining data, and Steven Salzberg from CBCB for editing the manuscript. This work was supported in part by National Institutes of Health grants R01-LM006845 and R01-LM007938, and National Science Foundation CISE RI grant CNS 04-03313.

[†] Equal Contribution

Chapter 3

Trapnell, C.† and M.C. Schatz.† ***Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment.*** **Parallel Computing, 2009. 35: 429-440.**

The authors would like to thank Steven Salzberg from the University of Maryland for helpful comments on the manuscript. This work was supported in part by National Institutes of Health Grants R01-LM006845 and R01-GM083873.

Chapter 4

Schatz, M.C. ***CloudBurst: highly sensitive read mapping with MapReduce.*** **Bioinformatics, 2009. 25: 1363-1369.**

I would like to thank Jimmy Lin for introducing me to Hadoop; Steven Salzberg for reviewing the manuscript; and Arthur Delcher, Cole Trapnell and Ben Langmead for their helpful discussions. I would also like to thank the generous hardware support of IBM and Google via the Academic Cloud Computing Initiative used in the development of CloudBurst, and the Amazon Web Services Hadoop Testing Program for providing access to the EC2. Funding: National Institutes of Health (grant R01 LM006845); Department of Homeland Security award NBCH207002.

Chapter 5

Langmead, B., Schatz, M.C., Lin, J., Pop, M., and S.L. Salzberg. *Searching for SNPs with cloud computing.* Genome Biology, 2009. 10:R134.

This work was funded in part by NSF grant IIS-0844494 (MP) and by NIH grants R01-LM006845 (SLS) and R01-HG004885 (MP). We thank the Amazon Web Services Hadoop Testing Program for providing credits, and Deepak Singh for his assistance. We also thank Miron Livny and his team for providing access to their compute cluster.

Chapter 6

Schatz, M.C., Delcher, A.L., and S.L. Salzberg. *Assembly of Large Genomes Using Second-Generation Sequencing.* Manuscript Under Review.

This work was supported in part by NIH grants R01-LM006845 and R01-GM083873 and by NSF grant IIS-0844494.

Chapter 7

Schatz, M.C., Phillippy, A.M., Shneiderman, B., and S.L. Salzberg. *Hawkeye: an interactive visual analytics tool for genome assemblies.* Genome Biology, 2007. 8: R34.

The authors would like to thank Arthur Delcher and Mihai Pop of the Center for Bioinformatics and Computational Biology at the University of Maryland for their extensive editing of the manuscript and continued guidance of this project; Jane Carlton and Jacques Ravel of The Institute for Genomic Research for providing us

with the T. vaginalis and B. megaterium data; and Lincoln Stein for his thoughtful review of the original manuscript. This work was supported in part by NIH award R01-LM06845, the National Institute of Allergy and Infectious Disease under contract NIH-NIAID-DMID-04-34, HHSN266200400038C, and DHS/HSARPA award W81XWH-05-2-0051 to SLS.

Chapter 8

Schatz, M.C., Sommer, D.D., Kelley, D.A., and M. Pop. *De novo Assembly of Large Genomes using Cloud Computing.* Manuscript in Preparation.

This work was funded in part by NSF grant IIS-0844494 and by NIH grants R01-LM006845 and R01-HG004885. We thank the Amazon Web Services Hadoop Testing Program for providing credits, and Deepak Singh for his assistance. We also thank Miron Livny and his team for providing access to their compute cluster.

Dedication

To my wife Emery, and all of my family, friends, and teachers. Thank you for all of your support and inspiration. You have all touched me deeply and made this work possible.

Acknowledgements

I have been truly blessed to know and work with many outstanding people throughout my career, and I am grateful to all of you.

I would first like to thank my advisor Steven Salzberg. It has been an absolute privilege to work with him for almost 8 years as a student at Maryland and before as an engineer at TIGR. I am grateful for the support, encouragement, and freedom you gave me to pursue my interests, and also for keeping me grounded and focused on the most challenging problems of the day. You have created a culture of open scientific endeavor of the highest caliber that I hope to create in my own lab one day.

Contributing deeply to this culture, I would like to thank all of my teachers, and especially CBCB faculty members Mihai Pop, Arthur Delcher, and Carl Kingsford, University of Maryland Computer Science faculty Jimmy Lin and Amitabh Varshney, University of Maryland School of Medicine faculty Jacques Ravel and Pawel Gajer, my TIGR mentor Martin Shumway, my STi mentor Peter Tran, and Carnegie Mellon faculty Mark Stehlik. You have challenged me to think harder and work harder than I thought possible, but have also been my greatest advocates.

I would like to thank all of my fellow students that have contributed to my life in a profound way, especially Adam Phillippy, Ben Langmead, Cole Trapnell, David Kelley, Elena Zheleva, James White, Saket Navalkha, and Sam Angiuoli. You have been a tremendous resource and inspiration for my research, and will forever be my academic brothers and sisters.

I would also like to thank and acknowledge all of the members of Lung Chuan Fa for supporting me as my extended family during my graduate career, and especially Shih-Fu Doug Moffett for his time and effort cultivating and challenging me to become a stronger person physically and tactically, but also of the highest character and morals. You have unlocked the indomitable spirit within me to never give up even in the most difficult challenges. I am also grateful and indebted to Bob Greene, Jennifer Fenton, Leah Baldo, Matt Fuhler, Nathan Toxopeus and all of the senior students for your support.

I would like to thank my parents, William and Joan Schatz, my brother Bill and his family Megan, Alex, and Kate, my grandfather William Groucaski, my parents-in-law, Russell Mikel and Alison Hurst, my brother-in-law Jory Mikel, and all of my aunts, uncles, cousins and relatives for a lifetime of love and support. I would also like to thank Rob Haynes, Beth Topf, Meghan Ruby, Dawn Taraszkiewicz, Alena D'Aniello, Scott and Tammy Brement, and William and Carol Sanzone, for being my most supportive critics, and my greatest fans.

Finally, and above all others, I would like to thank my wife Emery for her infinite patience and love.

Thank you to everyone.

Table of Contents

List of Tables

List of Figures

Chapter 1: Background and Significance

Motivation

Recent advances in DNA sequencing technology from Illumina (http://www.illumina.com/), 454 Life Sciences (http://www.454.com/), Applied Biosystems (http://www.appliedbiosystems.com), and other vendors have enabled DNA sequencing instruments to sequence the equivalent of the human genome in few days and at low cost. In contrast, the sequencing for the human genome project of the late 90's and early '00s required years of work on hundreds of machines with sequencing costs measured in hundreds of millions of dollars [1]. This dramatic increase in efficiency has spurred tremendous growth in applications for DNA sequencing. For example, whereas the human genome project sought to sequence the genome of a small group of individuals, the 1000 genomes project (http://www.1000genomes.org/) aims to catalog the genomes of 1000 individuals from all regions of the globe. Recent related projects aim to catalog all of the biologically active transcribed regions of the genome over a wide variety of environmental and disease conditions. Similar studies are also underway for model organisms such as mouse, rat, chicken, rice, and yeast, and other organisms of interest.

There is high demand for analyzing DNA sequences, but the raw outputs for these studies often exceed 1 terabyte of data and are pushing the limits of feasibility for the computations involved. Furthermore, biological datasets are only increasing in size, as data for more individuals and more environments are collected, so if we have not yet reached the breaking point for traditional models of computation for computational biology, it is just over the horizon. It is clear that the only long-term solution is to combine research in computational biology with advances from high performance computing (HPC), especially to parallelize computations to multiple processors, and to utilize high performance distributed file systems.

DNA Sequencing

The genome of an organism encodes genetic information within a long sequence of 4 different DNA nucleotides: adenine (A), cytosine (C), guanine (G) and thymine (T). The nucleotides are configured along two strands of a double helix, called the forward and reverse strands, with the nucleotides of one strand bonding with complementary nucleotides on the other. Under normal conditions adenine nucleotides only bond with thymine, and cytosine nucleotides only bond with guanine, so the sequence of one strand determines the sequence of the other, with each bonded nucleotide called a basepair (bp). The length and complexity of a genome sequence varies considerably depending on the complexity of the organism. For example, the genomes of small single-cell bacteria are typically a few million nucleotides long, while the genomes of higher organisms, such as humans, are billions of nucleotides long organized in several chromosomes. See Brown's classic textbook for a more complete introduction to DNA and genomics [2].

Traditional Sanger sequencing [3] uses chain termination with radioactively or fluorescently tagged nucleotides to sequence DNA. The technique is effective and widely used, but is limited to sequencing at most ~1000 consecutive nucleotides, each called a sequencing read. Newer methods use a variety of low cost and high throughput sequencing technologies, but are currently limited to ~25-500 consecutive nucleotides [4]. Reads from all technologies have sequencing errors in the form of miscalled, extra, or missing nucleotides, at a rate of 1% to 5% of bases. For example, the 454 Life Sciences sequencing technology is limited in its ability to correctly sequence homo-polymer sequences (sequences consisting of a single repeated nucleotide), and most sequencing technologies tend to have more errors at the ends of the reads as the biochemical sequencing reactions become less efficient [4].

Sequencing reads are much shorter than the full genome sequence, so the complete genome sequence cannot be directly sequenced. Instead genome sequencing projects commonly use whole genome shotgun sequencing (WGSS) [5] approach to sequence an entire genome. In WGSS, the genome is first randomly sheared into small fragments, and then those small fragments are individually sequenced with a DNA sequencer (Figure 1). Some sequencing technologies also allow for sequencing pairs of reads from both ends of a fragment, creating what is a called a mate-pair. Consequently, mate-pairs are separated in the genome by an approximately known distance (Figure 1), and provide long range linking information crucial for analyzing complex genomes. The short reads and mate-pairs are then computationally analyzed, and reads with consistent sequences are assembled into larger sequences, similar to how small puzzle pieces can be connected to form larger and larger blocks. The number of reads necessary to sequence an entire genome depends on the size of the genome, including the 8-fold to 30-fold oversampling necessary to ensure each position in the genome has been sequenced with high probability [6]. For large genomes, such as the human genome, literally billions of reads are used to analyze the full genome with raw outputs exceeding 1 TB of data and sequence data exceeding 100 GB.

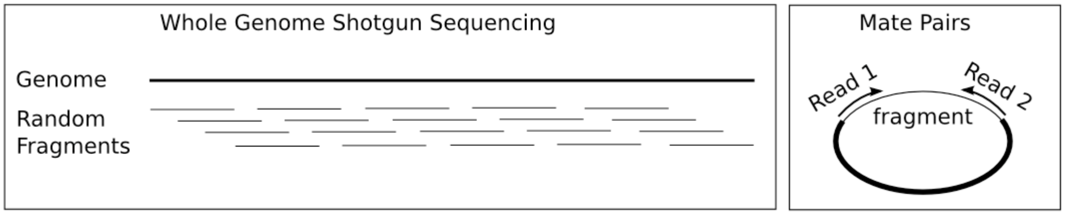


Figure 1. Whole Genome Shotgun Sequencing and Mate Pairs.
In WGSS, the genome is randomly sheared into many short pieces of DNA each called a fragment, which are later sequenced and analyzed (left). Each DNA fragment can be embedded within a cloning vector, with 2 mated reads sequenced from the same fragment (right).

Read Alignment

The genomes of two individuals of the same species or two individuals from closely related species are often very similar. In these cases it is possible to align or map a large fraction of the reads from one individual to a reference genome to find the most likely position each read occurs [7]. This information reports the regions of the genome that are conserved in the two genomes, and the regions with polymorphisms, including regions of the reference that are not present in the query reads at all (deletions). Regions of the query that are not present in the reference (insertions) are noted as unmappable reads, which consequently require de novo assembly for complete analysis (explained below).

The mapping processes is computationally challenging because the amount of sequence data is very large and the mapping algorithm must allow for differences between a given read and the reference genome, for both biological and technological reasons. For example, the recently published analysis of the genomes of an African [8] and an Asian [9] individual from the 1000 genomes project required 4.0 and 3.3 billion 35bp reads, respectively, and hundreds of hours of computation.

Many read mapping algorithms use a technique called seed-and-extend to accelerate the mapping computation [10-15]. The key insight for this technique is if a read maps to the genome at a particular location with a relatively small number of differences, then a significant fraction of the read must map without any error at all. A difference can be a change of character (mismatch), additional characters in the query (insertion), or missing characters from the reference (deletion) (Figure 2).

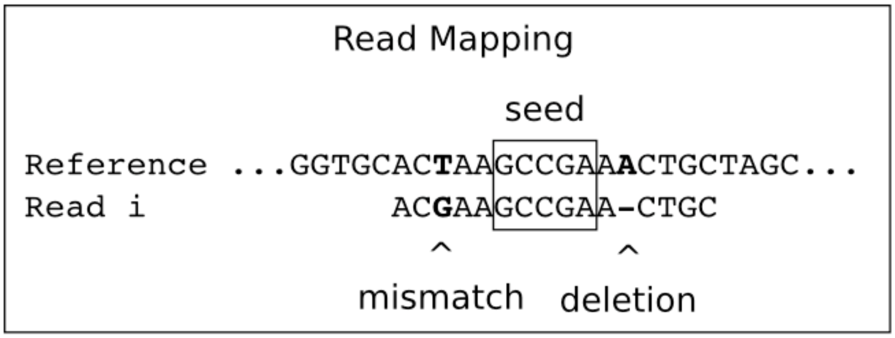


Figure 2. Seed-and-extend read mapping.
The 5bp sequence GCCGA is shared by the read and the reference sequence and acts as an alignment seed. The left flanking sequence has a mismatch, and the right flanking sequence has a deletion for a total of 2 differences for this alignment.

For example, if a 40bp read aligns to a reference with at most one difference, and the first base differs, the remaining 39 must match exactly. If only the 2nd base differs, the remaining 38 must match exactly, and so forth, until if the 20th is the only difference, then the remaining 20 bases must match exactly (the length of the exactly matching sequence is symmetric for the remainder of the sequence). More generally if a read of length R maps to a sequence with at most k errors, then it must contain a substring of length $\lfloor R/(k+1) \rfloor$ that exactly matches. This fact is due to a simple application of the pigeon-hole principle: if the read is divided into k+1 chunks, then the k errors must leave 1 chunk unmodified which will exactly match [16]. This insight leads to a general strategy for quickly computing end-to-end alignments: first short exactly matching regions called seeds in the read and the reference sequence are found, and then those seeds are extended into end-to-end matches using a more sensitive algorithm that allows for errors. The seeds are chosen so all alignments of sufficient quality are detected, but only regions with potential for high quality alignments are investigated.

The read mapping algorithms use several different approaches for finding seeds. The method used by RMAP [12] and others is to construct a hash table of the non-overlapping substrings of length $\lfloor R/(k+1) \rfloor$ in the reads. Then the genome sequence is scanned to consider each $\lfloor R/(k+1) \rfloor$ length substring of the genome. At each position in the reference, the set of reads with the current substring is retrieved from the hash table, and the end-to-end sequence of each read is compared to the genome allowing for a number of mismatches. The main limitation of this approach is the space requirements for the hash table may be very large, and it must be recomputed for different values of R or k. In addition, if the seed length becomes very small, then many chance occurrences of the seed may be present in the genome that will not lead to a high quality alignment.

Given the limitations of a hash table approach, other methods have been developed using a more sophisticated index to quickly evaluate candidate seeds, such as a suffix tree [14]. A suffix tree encodes all suffixes of string along a path from the root to a leaf, so the presence of a query sequence in a suffix tree of a reference sequence is determined by walking from the root of the tree along the edges according to the sequence of the query (Figure 3). The suffix tree can be quickly constructed in time proportional to the length of the sequence, and can be reused for queries of any length. Once the exact matches are found in the suffix tree, a more sensitive alignment algorithm is used for to align the flanking sequence into end-to-end alignments.

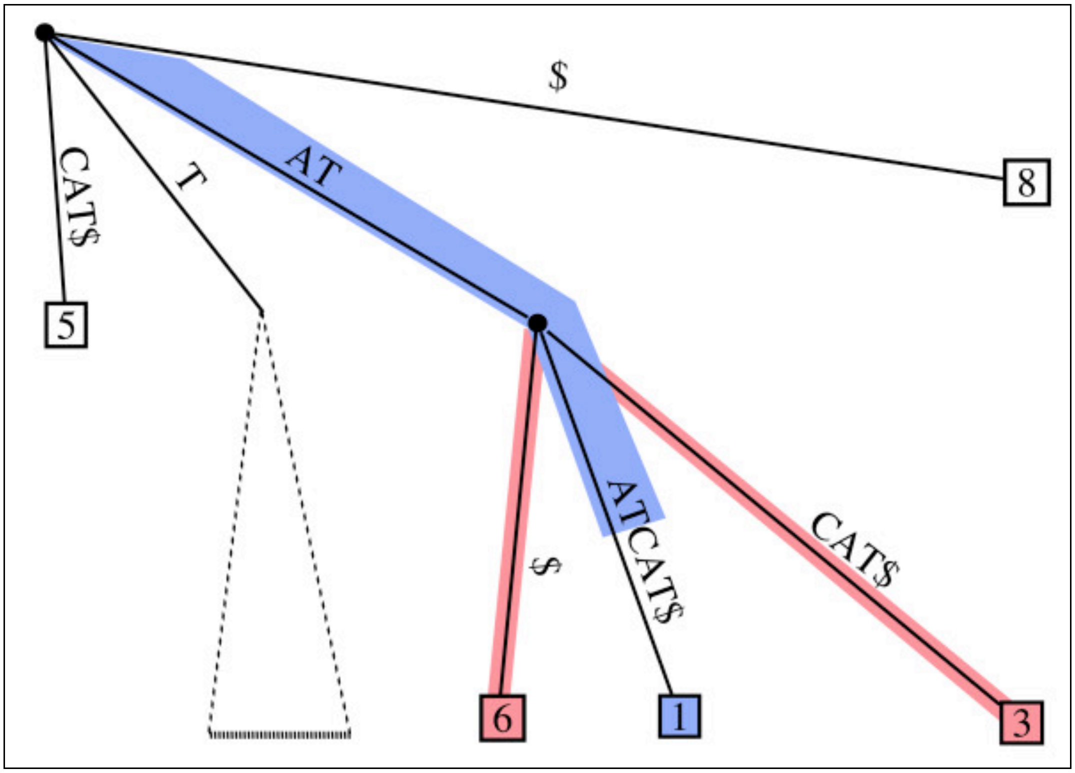


Figure 3. Aligning a query against a suffix tree.
Aligning the query ATAT against the suffix tree for ATATCAT$. The path from the root to each leaf encodes a sequence that occurs in the reference at the label of that leaf. The blue path shows the extent of the alignment in the tree. The query occurs at position 1 in the reference, and partially match with length 2 (AT) at positions 6 and 3 as shown by the red paths. (Figure from [17])

Genome Assembly

Early genome assembler including phrap [18] and the TIGR Assembler [19] used a greedy algorithm to assemble the genome from a collection of reads. In these assemblers, reads with the longest overlap, meaning the suffix of one read matches the prefix of another, are iteratively merged into longer and longer sequences. This approach is sufficient for the simplest genomes, but in general fails to produce a correct genome sequence because genomes contain repeated sequences and reads from these repeated sequences will have “false overlaps”, meaning these reads should not be assembled based on their original placement in the genome even though they are sufficiently similar. When a greedy assembler incorporates false overlaps, distant regions of the genome become falsely connected and incorrectly reconstructed [20].

The limitations of the greedy assembly algorithms lead to the development of a graph theoretic approach for genome assembly called overlap-layout-consensus implemented in several modern assemblers including the Celera Assembler [21], which was used for the private effort to sequence and assemble the human genome in 2001 and dozens of organisms since, and Arachne [22, 23], another widely used assembler for large genomes. In the overlap stage, all pairs of reads are compared for overlaps, allowing for a small amount of difference in the overlapping region from sequencing error. The result of these comparisons is an overlap graph, where reads in the graph have an edge if they have an overlap with sufficient quality. In the layout stage, consistent paths of overlaps are chosen from the overlap graph. A correct path of overlaps should visit each node exactly one time, a Hamiltonian path. The scale of the problem prevents a search for an exact Hamiltonian path, so heuristics are employed to simplify the problem. Regions of the genome between the boundaries of repeats create non-branching chains of overlaps that can be found and collapsed into linear paths, called unitigs in the terminology of the Celera Assembler. At repeat boundaries, the overlap graph forks where reads in the repeat overlap reads from different regions of the genome, but the reads from the different regions do not overlap (Figure 4). Finally in the consensus stage, each layout is refined to produce a consensus sequence called a contig, correcting for sequencing error in the underlying reads (Figure 5).

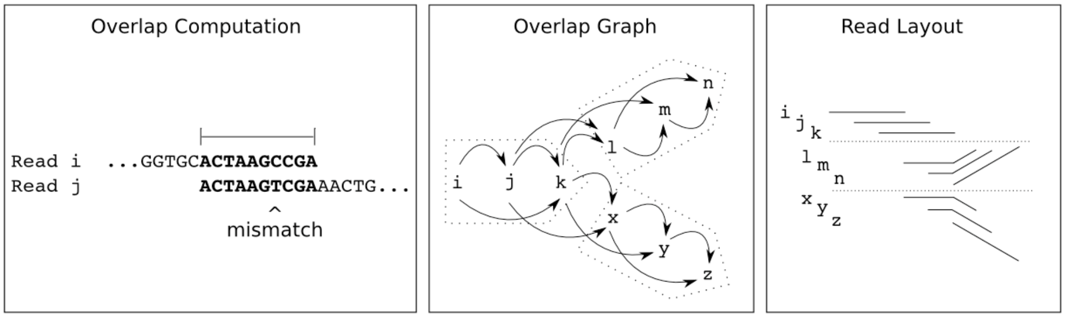


Figure 4. Overlaps and Layouts.
In the overlap stage, all pairs of reads are compared for overlaps. Here the last 10bp of Read i overlaps the first 10bp of Read j with 1 mismatch (left). The collection of overlaps form an overlap graph where reads are connected by a directed edge from a to b if the suffix of the read a overlaps the prefix of read b with sufficient quality (middle). The overlap graph is then analyzed to compute the layout of reads with an approximate offset of each read (right). The overlap graph forks at read k, so three separate unitig layouts are generated.

If mate-pairs are available, the linking information is used to better resolve true overlaps from false overlaps, and improve the layouts. For example, if the mate-pairs indicate reads l,m,n should immediately follow i,j,k and reads x,y,z are from an unrelated region of the genome, the layouts will be arranged and expanded appropriately. The contigs can also be ordered and oriented into larger scaffolds, with gaps between contigs representing missing sequence or ambiguous repeats. Ideally a single scaffold will represent each chromosome of the organism, but if the organism has repeats larger than the span of the mate pairs, then the assembler will create multiple scaffolds for the different unambiguous regions.

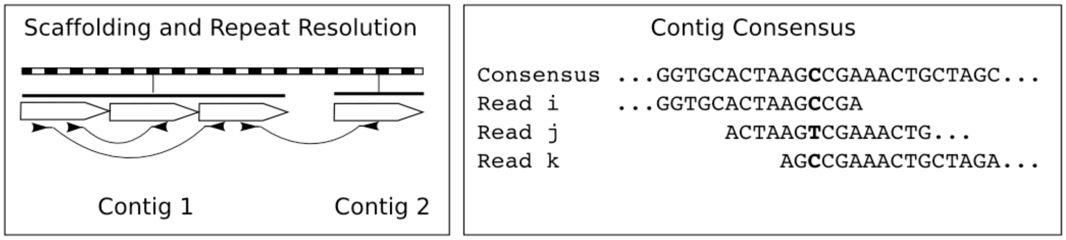

Scaffolding and Repeat Resolution
Contig 1
Contig 2
Contig Consensus
Consensus ...GGTGCACTAAGCCGAAACTGCTAGC...
Read i ...GGTGCACTAAGCCGA
Read j ACTAAGTCGAAACTG...
Read k AGCCGAAACTGCTAGA...

Figure 5. Scaffolding and Consensus.
In the scaffolding phase (left), mate pairs, shown as curved lines, indicate the correct order of the unitigs. If the resolved unitigs overlap, the unitigs are merged into a longer contiguous sequence as in Contig 1. If the unitigs do not overlap, the contigs are placed into a scaffold with a gap between the contigs of known size but unknown sequence. In the consensus computation (right), the unitig or contig layouts are refined into a true multiple alignment of reads correcting for sequencing error in the individual reads.

The primary complicating factor in genome assembly is the presence of repeats, which create false overlaps that confound the assembly process [24]. If the false overlaps are incorrectly incorporated, they can cause mis-assemblies that corrupt the genome sequence by rearranging the sequence of the genome, mis-representing repeat instances, fragmenting contigs, or otherwise mis-representing the true genome sequences. The fraction of a genome that is repetitive depends on the sequence composition of the genome and the read length, but also a tension in the assembly problem between allowing for sequencing error and mis-assembling repeats: the assembler must allow for some amount of sequencing error between overlapping reads, but allowing for differences in overlaps causes more of the genome to appear repetitive. As such, modern assemblers attempt to statistically detect and correct sequencing error, but are limited in their ability to correct all errors.

Given the tension from sequencing error and the presence of long identical repeats, genome assemblers act cautiously to detect and avoid mis-assembling repeats whenever possible. Even so, modern assemblers are still prone to make errors, such as collapsing multiple copies of a repeat into a single copy [24]. This is especially problematic if the repeat copies occur in tandem at adjacent positions in the true genome sequence (Figure 6). Since the reads from sufficiently large exact repeats overlap without error, it can be impossible to determine the correct placement from overlaps alone. Fortunately, collapsed repeats do have detectable mis-assembly signatures [20, 25], such invalid mate-pair relationships or as the depth of coverage (number of reads spanning a given position) suddenly increasing, so assemblies can be verified by analyzing the assembly and reviewing any suspicious region.

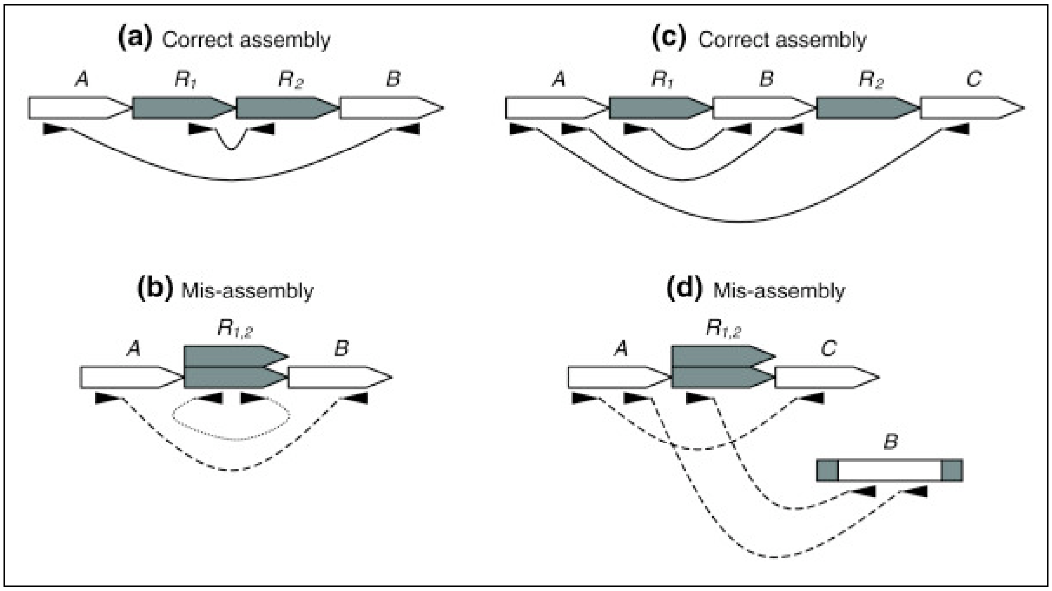


Figure 6. Mate pair signatures of collapsed repeats.
(a) Two copy tandem repeat R shown with properly sized and oriented mate-pairs. (b) Collapsed tandem repeat shown with compressed and mis-oriented mate-pairs. (c) Two copy repeat R, bounding unique sequence B, shown with properly sized and oriented mate-pairs. (d) Collapsed repeat shown with compressed and mis-linked mate-pairs. (Figure from [20])

Once the genome has been assembled, the genome sequence is analyzed for a variety of biological reasons. For example, the genome sequence is compared to other genomes to catalog the conserved (similar) and unique regions of the new genome. Protein encoding regions of the genome called genes are predicted using computational and laboratory techniques. Regulatory elements, transposable elements, insertion sequences, and a host of other biologically significant sequences are analyzed. In short, a complete and correct genome sequence is a fundamental requirement to a wide variety of analyses and is a key to unlocking the health and biology of the organism.

High Performance Computing

Research in high performance computing (HPC) has lead to many advances, including techniques for coordinating computation across multiple processors within one computer, and across multiple processors within multiple computers. The high level goals of this research are to use more processors to accelerate the end-to-end computation time for a given problem (strong scaling), or to increase the size of the computation possible in a fixed amount of time (weak scaling).

One of the simplest and most widely used techniques of HPC is called batch computing, in which, a (large) set of independent computations is partitioned into multiple batches, which are then simultaneously and separately evaluated on multiple processors [26]. For example, within computational biology this is a well known technique for accelerating protein alignment using the tool BLAST [11] using P processors, such as is performed using the tool mpiBLAST [27]. The input is a set of Q query sequences, and the goal is to align each query sequence to a database of known protein sequences. Since the alignment of one sequence is independent of the other sequences, the input can be trivially split into P batches each containing Q/P sequences, and then simultaneously execute P instances of BLAST. The total amount of computation performed remains the same, so a system like this should in principle achieve perfect linear speedup, meaning the wallclock running time should be decreased by a factor of P when running on P processors as compared to the serial runtime.

In practice, a parallel system will not always reach perfect linear speedup, but will often have less than 100% parallel efficiency, measured as the ratio of the serial runtime over the parallel runtime on P processors times P. In this example, the total number of alignments computed will be exactly the same while running in parallel, but extra computation and time is needed to distribute and monitor the work between the processors which reduces the parallel efficiency. Furthermore, if the system is not entirely parallel but contains a serial component, such as a sequential scan of the inputs before distributing the alignment computations, then the overall speedup will be limited relative to the fraction of serial work. This relationship, known as Amdahl's law [28], states that the overall speedup for a parallel system is limited to 1/(1-%P), where %P is the proportion of the program that is executed in parallel. For example, if 5% of the runtime is needed for serial computation, then the overall speedup is at most 20 times faster, assuming the remaining 95% of the runtime is reduce to zero.

Finally, some query sequences may take much longer than others to align than others, such as repetitive sequences with many more alignments or query sequences that are much longer than average. The batches with these sequences may take considerably longer than batches without those sequences, but the wallclock runtime will be dominated by the longest running batch. This load imbalance will also negatively impact the overall speedup of the system. In the extreme, if a single batch takes twice as long as the average, then the speedup will be cut in half by the single "straggler". Therefore it is generally beneficial to divide the input set into more than P batches so that any load imbalance can be hidden by scheduling fewer batches on machines analyzing the stragglers. However, if the size of the batch size is too small, then the extra communication overhead could negatively decrease performance. Clearly a careful balance between batch size and performance must be made depending on the characteristics of the parallel system.

In contrast to batch systems are parallel computations that cannot be partitioned into independent computations, but require communication between the different processors. A basic example is evaluating a function on a regular matrix, in which case each processor evaluates the function on a submatrix and uses interprocessor communication to coordinate the results at the boundaries of the submatrices. The most extreme versions are parallel computations on irregular data structures, such as computing the minimum spanning tree of a graph [29]. For very simple graphs it may be possible to partition the graph into non-overlapping components of approximately the same size that are examined independently, but this will not be possible for general graphs or will be difficult to find such a partitioning in a reasonable amount of time.

For these types of problems, a sophisticated parallel algorithm is necessary to coordinate the computation across processors [30]. These parallel algorithms are usually described using a variant of the parallel random access model (PRAM), which is an abstract model for synchronous shared memory computation. Under this model many processors execute an algorithm under control of a single clock with full access to a shared memory resource (See Jaja's seminal textbook for a full discussion of the model [31]). These requirements are clearly not realizable for very large systems with many processors, but inventing abstract PRAM algorithms frees the algorithm designer from low level system details to focus on the abstract computation. Furthermore many abstract PRAM algorithms can be efficiently simulated on non-synchronized, non-shared memory system using interprocessor and intermachine communication techniques such as the message passing MPI [32]. However, it may be very complicated or inefficient to simulate the PRAM algorithm because the communication costs are not included in the PRAM analysis.

The main benefit of the abstract PRAM model is that is useful for describing an algorithm and analyzing its efficiency, much like how the random access machine model (RAM) [33] is useful for analyzing serial computation. Central to this analysis are the concepts of parallel work W and parallel time T (sometimes also called depth). Parallel work is the total number of operations performed in a parallel system, and parallel time is total number of parallel steps. For example, consider the problem of computing the sum of *n* integers using up to *n/2* processors. A serial algorithm computes the sum in O(n) time using a sequential scan of the values. The well known parallel pairwise summation algorithm also performs a total of O(n) additions, but performs those additions in 3 steps, by summing pairs of items organized into a binary tree (Figure 7).

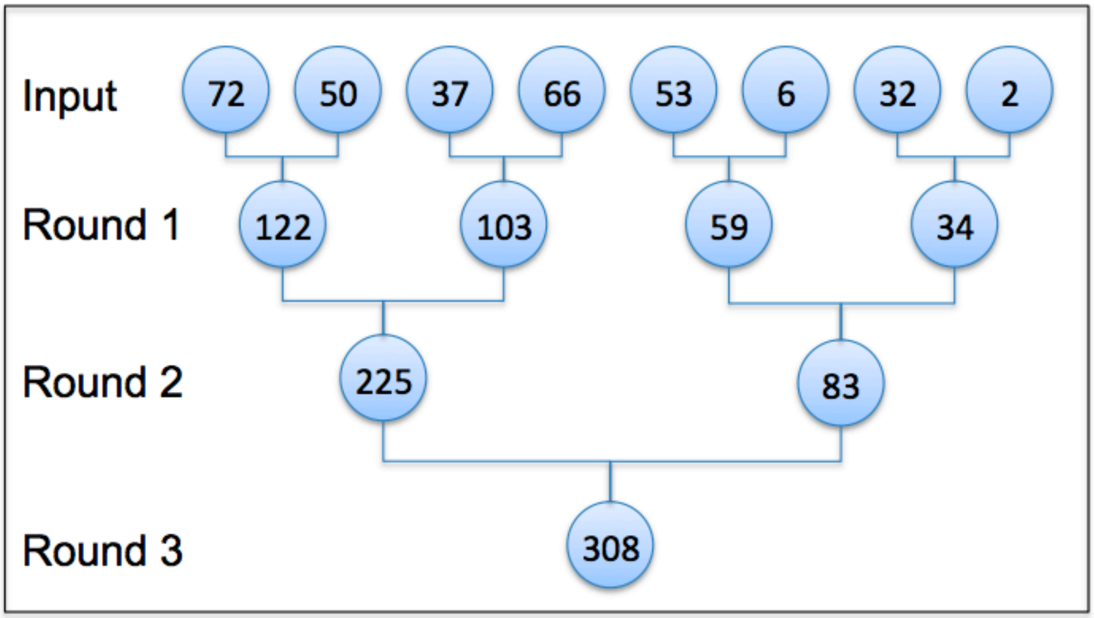


Figure 7. Illustration of Parallel Pairwise Summation Algorithm.
In each round, several additions are simultaneously computed by adding pairs of items from the previous round organized in a binary tree.

In every step, the number of items to sum is cut in half so this algorithm requires T=O(log n) total parallel steps. Since the total number of additions, the parallel work, is the same as the serial algorithm, this algorithm is considered work optimal. Furthermore since any other work optimal PRAM algorithm also requires T=O(log n) [34], this algorithm is both work and time optimal. In this way, many parallel algorithms that process n elements can be optimally implemented in W=O(n) and T=O(log n) [34].

GPGPU Computing

One recent development in HPC has been the growth of general purpose graphics processing unit (GPGPU) computation, where computation is executed on commodity but highly parallel graphics processing units (GPUs). While GPUs were original limited in capability and used exclusively for graphics processing, modern GPUs from nVidia and ATI contain dozens or hundreds of stream processors and can be programmed for non-graphics computation. The stream processors simultaneously execute the same instructions on different data items (SIMD computation), allowing for parallel, and nearly arbitrary computations, on large datasets (Figure 8).

The GPUs are programmed in a modified version of C, but have restricted programming capabilities such as a relatively small numbers of registers, no call stack, and no direct IO access. These limitations make implementing algorithms on the GPU challenging, but certain scientific and numeric GPGPU applications have demonstrated 10- to 100-fold improvements in running times on one GPU versus a traditional CPU. Consequently, there is great interest for developing GPGPU versions for computationally intensive applications. The most successful GPGPU applications typically have had high arithmetic intensity, meaning the computation is dominated by numerical and arithmetic operations as opposed to data access and comparisons. However, high performance data intensive algorithms are possible using techniques such as data reordering and register reduction to maximize cache performance and processor occupancy [35].

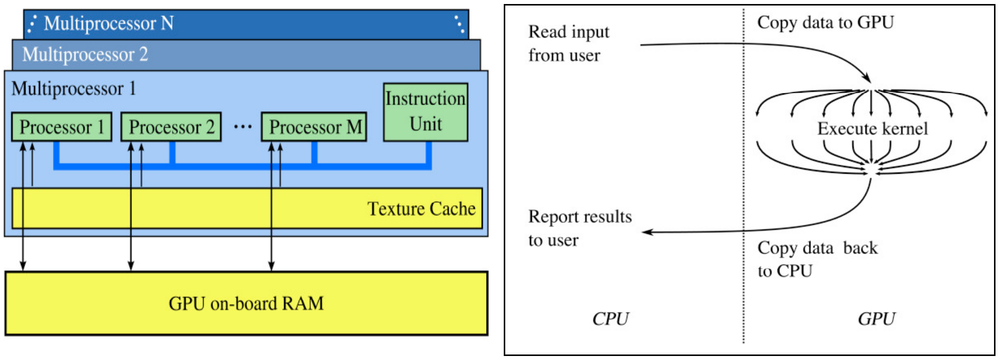


Figure 8. Overview of G80 Architecture and Workflow.
This figure shows how the GPU is organized into several (N) multiprocessors, each containing multiple (M) stream processors that simultaneously execute the same instruction (left). Each processor can access the texture cache very quickly, but reads and writes to the onboard RAM have high latency. Input data for a GPGPU application must be copied to the GPU's memory along with a pre-allocated output buffer prior to invoking the GPU-based kernel (right). Output from the kernel is read back into main memory and reported to the user. (Figures from [17])

MapReduce and Cloud Computing

Another advance in HPC is the MapReduce framework [36] developed at Google for their computations on extremely large data sets, including their index of more than 1 trillion web pages. Computation in MapReduce is structured into three main phases: the map phase, which emits key-value pairs from the input data, the sort/shuffle phase, which groups key-value pairs with the same key, and the reduce phase, which evaluates a function using all values with the same key (Figure 9). These operations conceptually construct a large distributed hash table (map and sort/shuffle functions), from which each bucket is independently evaluated (reduce function). MapReduce is primarily used with datasets much larger than can be stored in RAM, so MapReduce relies on the Google File System (GFS) to efficiently transfer data within the large clusters. The GFS is specially designed using redundancy, intelligent scheduling, and a lightweight directory master to provide performance and reliability even on commodity disks with high failure rates.

MapReduce aims to simplify large scale parallel programming so application developers need only implement custom map and reduce functions, and the MapReduce framework provides the scheduling, monitoring, fault tolerance, and other common parallel services automatically. The power of MapReduce is many instances of the map and reduce functions can execute in parallel, potentially on large compute grids with hundreds or thousands of compute nodes. The main challenges using MapReduce are casting the algorithm into a format compatible with the execution model, and then tuning the application to be as efficient as possible, especially to minimize overhead and maximize load balance between compute nodes. One exciting recent result showed than a large class of PRAM algorithms can be efficiently implemented in a MapReduce framework [37].

Open-source versions of MapReduce and the GFS, called Hadoop (http://hadoop.apache.org) and the Hadoop Distributed File System (HDFS), are actively developed and used by Google, Yahoo, Amazon, and other major vendors. Yahoo uses Hadoop clusters to support, in part, every web search result, and recently set a performance record for general purpose computing by sorting 1 terabyte of data (10 billion 100 byte records) in 209 seconds using 910 nodes with Hadoop (http://sortbenchmark.org/).

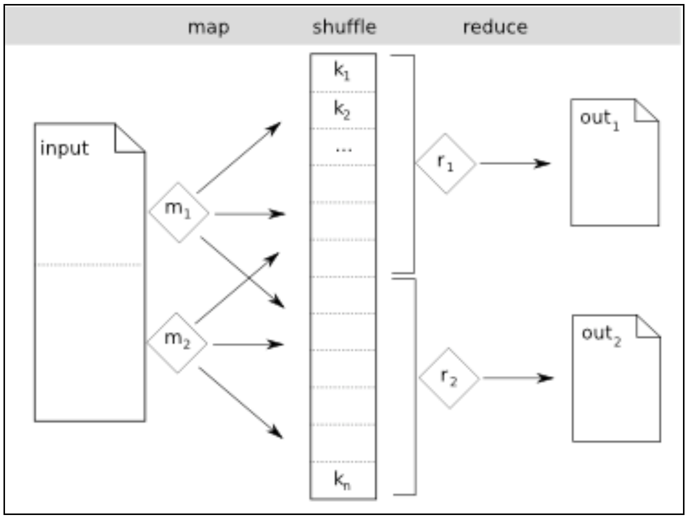


Figure 9. Schematic Overview of MapReduce.
The input file(s) are automatically partitioned into chunks depending on their size and the desired number of mappers. Each mapper (shown here as m_1 and m_2) executes a user-defined function on a chunk of the input and emits key-value pairs. The shuffle phase creates a list of values associated with each key (shown here as k_1, k_2, and k_n). The reducers (shown here as r_1 and r_2) evaluate a user-defined function for their subset of the keys and associated list of values, to create the set of output files. (Figure from [38])

Hadoop's capabilities for efficient computations on large data sets are starting to also draw attention for scientific computing, and some early applications have demonstrated orders of magnitude improvements in running time using Hadoop and MapReduce. Furthermore, Hadoop is becoming a de facto standard for cloud computing, where remote computing resources are accessed generically, without regard for location or specific configuration. Several companies, including Amazon (http://aws.amazon.com) and their Elastic Compute Cloud (EC2), lease compute time on their large clusters, and Hadoop is a recommended solution for executing large scale cloud computing with minimal effort.

Thesis Statement

GPGPU and MapReduce technologies can be successfully applied to accelerate and enable the problems of short read alignment and genome assembly from large volumes of short DNA sequences.

Towards this thesis, Chapters 2 and 3 describe MUMmerGPU, which uses GPGPU technology to accelerate the alignment of many query sequences against a suffix tree representation of the reference. Chapter 4 describes CloudBurst, which also accelerates the alignment of many query sequences against a genome, but uses a distributed inverted index of the reference within MapReduce for the alignment. Chapter 5 builds on this result, and describes the Crossbow pipeline for rapidly aligning and genotyping entire human genomes using MapReduce. Chapter 6 provides an in depth review of genome assembly, including the recent advances for assembling large genomes from short reads. Chapter 7 describes common mis-assembly problems and the visual analytics program Hawkeye for inspecting genome assemblies for mis-assemblies. Chapter 8 describes the genome assembler Contrail, which uses the MapReduce paradigm to make feasible the assembly of large genomes from short reads without requiring large amounts of main memory. Chapter 9 summarizes my contributions and describes possible avenues for future work.

Chapter 2: High-throughput Sequence Alignment Using Graphics Processing Units

Summary of Contribution

This chapter describes version 1.0 of the high-throughput sequence alignment program MUMmerGPU published in BMC Bioinformatics [17], and was developed in collaboration with Cole Trapnell, Art Delcher, and Amitabh Varshney at the University of Maryland. MUMmerGPU is a GPU accelerated implementation of the exact string matching component of the widely used sequence alignment program MUMmer [10, 14, 39], specifically designed to align large batches of short query sequences to a reference genome.

GPUs were originally designed to accelerate graphics computation for on-screen display, but can now be used to accelerate general purpose computation using their dozens or hundreds of lightweight stream processors composed of an ALU (arithmetic logic unit) and processor specific memory. MUMmerGPU uses the GPU to align many query sequences to the reference sequence in parallel, by aligning each query sequence on a different GPU stream processor. Each stream processor then executes the alignment kernel to match the given query string to a suffix tree representation of the reference stored on the GPU.

With dozens or hundreds of stream processors, GPUs are especially well suited to regular, numerically intensive computation, such as matrix computations or image processing. Accelerating data intensive programs require careful consideration of memory usage to maximize cache coherency and minimize cache misses. MUMmerGPU uses a novel space filling curve memory layout to reorder the suffix tree memory to improve locality for the characteristics of the GPU, and achieves a 10 fold speedup in the alignment kernel, and a 3.5 fold overall speedup in end-to-end application processing time.

Michael Schatz implemented the initial suffix tree construction, alignment kernel, and suffix tree reordering algorithms. Cole Trapnell and Michael Schatz worked together to complete the alignment kernel and CPU driver program, performed the performance experiments, and drafted the manuscript together. Arthur Delcher and Amitabh Varshney edited the manuscript and provided guidance for the project.

Abstract

The recent availability of new, less expensive high-throughput DNA sequencing technologies has yielded a dramatic increase in the volume of sequence data that must be analyzed. These data are being generated for several purposes, including genotyping, genome resequencing, metagenomics, and de novo genome assembly projects. Sequence alignment programs such as MUMmer have proven essential for analysis of these data, but researchers will need ever faster, high-throughput alignment tools running on inexpensive hardware to keep up with new sequence technologies.

This chapter describes MUMmerGPU, an open-source high-throughput parallel pairwise local sequence alignment program that runs on commodity Graphics Processing Units (GPUs) in common workstations. MUMmerGPU uses the new Compute Unified Device Architecture (CUDA) from nVidia to align multiple query

sequences against a single reference sequence stored as a suffix tree. By processing the queries in parallel on the highly parallel graphics card, MUMmerGPU achieves more than a 10-fold speedup over a serial CPU version of the sequence alignment kernel, and outperforms the exact alignment component of MUMmer on a high end CPU by 3.5-fold in total application time when aligning reads from recent sequencing projects using Solexa/Illumina, 454, and Sanger sequencing technologies.

MUMmerGPU is a low cost, ultra-fast sequence alignment program designed to handle the increasing volume of data produced by new, high-throughput sequencing technologies. MUMmerGPU demonstrates that even memory-intensive applications can run significantly faster on the relatively low-cost GPU than on the CPU.

Background

Sequence alignment has a long history in genomics research and continues to be a key component in the analysis of genes and genomes. Simply stated, sequence alignment algorithms find regions in one sequence, called here the query sequence, that are similar or identical to regions in another sequence, called the reference sequence. Such regions may represent genes, conserved regulatory regions, or any of a host of other sequence features. Alignment also plays a central role in de novo and comparative genome assembly [21, 40], where thousands or millions of sequencing reads are aligned to each other or to a previously sequenced reference genome. New, inexpensive large-scale sequencing technologies [41] can now generate enormous amounts of sequence data in a very short time, enabling researchers to attempt genome sequencing projects on a much larger scale than previously. Aligning these sequence data using current algorithms will require very high-performance computers, of the type currently available only at the largest sequencing and bioinformatics centers. Furthermore, realizing the dream of widespread personal genomics at hospitals and other clinical settings requires sequence alignment to be low cost in addition to high-throughput.

Most personal computer workstations today contain hardware for 3D graphics acceleration called Graphics Processing Units (GPUs). Recently, GPUs have been harnessed for non-graphical, general purpose (GPGPU) applications. GPUs feature hardware optimized for simultaneously performing many independent floating-point arithmetic operations for displaying 3D models and other graphics tasks. Thus, GPGPU programming has been successful primarily in the scientific computing disciplines which involve a high level of numeric computation. However, other applications could be successful, provided those applications feature significant parallelism.

In this chapter, we describe a GPGPU program called MUMmerGPU that performs exact sequence alignment using suffix trees on graphics hardware. Our implementation runs on recent hardware available from nVidia using a new software development kit (SDK) for GPGPU progamming called Compute Unified Device Architecture (CUDA). MUMmerGPU is targeted to tasks in which many small queries, such as reads from a sequencing project, are aligned to a large reference sequence. To assess the performance of MUMmerGPU we compare it to the exact alignment component of MUMmer called mummer. MUMmer is a very fast and widely used application for this type of task [14], and is also used as the alignment engine for the comparative assembler AMOScmp [40]. Overall MUMmerGPU is more than three times faster than mummer on typical sequence alignment tasks involving data from three recent sequencing projects. As implemented, MUMmerGPU is a direct replacement for mummer and can be used with any other programs that process mummer output, including the other components of MUMmer that post-process the exact alignments computed by mummer into larger inexact alignments.

Sequence alignment

One of the most successful algorithms for computing alignments between sequences is MUMmer. The first stage of MUMmer is performed by a component called mummer, which computes exact alignments between the pair of sequences. These alignments can be used directly to infer large-scale sequence structure, or they can be used to seed extensions to longer inexact alignments using the post-processing tools bundled with MUMmer [10, 14, 39]. Unlike other popular sequence alignment programs such as BLAST [11], FASTA [42], and LAGAN [43], which use fixed length seeds for constructing their alignments, mummer alignments are variable-length maximal exact matches, where maximal means that they cannot be extended on either end without introducing a mismatch. First, mummer pre-processes the reference sequence to create a data structure, called a suffix tree. This data structure allows mummer to then compute all maximal exact substring alignments of a query sequence in time proportional to the length of the query. The time to pre-process the reference sequence is proportional to its length (which may be considerable for very long sequences), but this time becomes insignificant when amortized across many query searches. Consequently, suffix trees are used in several alignment algorithms, including MGA [44] and REPuter [45]. The suffix tree [46] for string S is a tree that encodes every suffix of S with a unique path from the root to a leaf. For a string of length n, there are n leaf nodes for each of the n suffixes in S. Each edge in T is labeled with a substring of variable length of S called an edge-label. Concatenating edge-labels along a path from the root to a node i forms a substring, called i's path-label in S. Leaves in the tree are labeled with the position where the path-label begins in S. Internal nodes have at least 2 children, representing positions where repeated substrings diverge. The edge-labels of the children of a node each begin with a different character from the alphabet, so there is at most one child for each letter of the reference string's alphabet. Consequently, the depth of any leaf is at most n, and there are O(n) nodes in the tree.

A suffix tree can be constructed in O(n) time and O(n) space for a string over a fixed alphabet, such as for DNA or amino acids, by using additional pointers in the tree called suffix links. The suffix link of node v with path-label xα points to node v' with path-label α where x is a single character and α is a substring [47, 48]. Suffix links are used to navigate between equivalent nodes of consecutive suffixes without returning to the root of the tree.

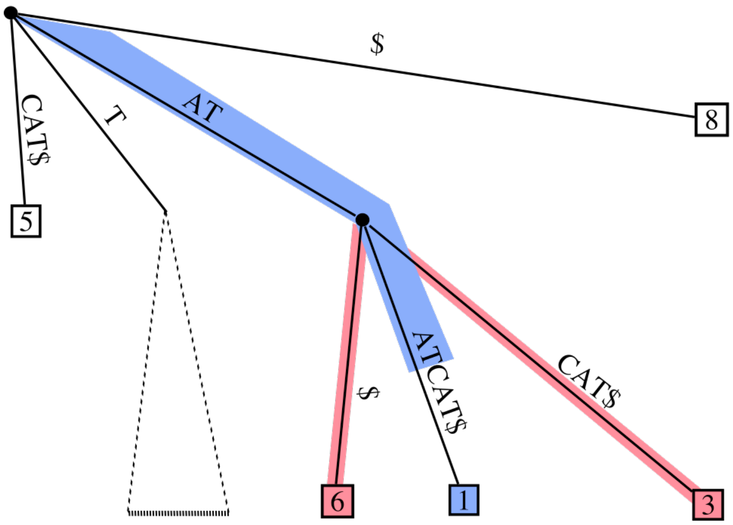


Figure 10. Aligning a query against a suffix tree.
Aligning the query ATAT against the suffix tree for ATATCAT$. The path from the root to each leaf encodes a sequence that occurs in the reference at the label of that leaf. The blue path shows the extent of the alignment in the tree. The query occurs at position 1 with an alignment length of 4. For l ≥ 2, MUMmerGPU will print the red nodes as alignments with an alignment length equal to 2, the sequence depth of the lowest common ancestor between the red nodes and the blue node.

All substrings of a query string Q of length m that occur in a string S can be determined in time proportional to m by navigating the suffix tree T of S to follow the characters in Q. The algorithm begins by finding the longest prefix of Q that occurs in T, descending from the root of T and following exactly aligning characters in Q for as long as possible. Assume that substring Q[1, i] is found in T along the path-label to node v, but there is no edge from v labeled with the next character in Q because Q[1, i + 1] is not present in S. The algorithm can then report the occurrences of Q[1, i] at the positions represented by all leaves in the subtree rooted at v after checking the alignments are maximal by comparing the left flanking base of the query and reference. The algorithm then continues by finding the longest substrings for each of the m - 1 remaining start positions in Q. However, instead of navigating the tree from the root each time, the algorithm resumes aligning with Q[i + 1] after following the suffix link from v to v' and without reprocessing previously aligned characters.

Given a user-specifed minimum length l and a query Q, suppose there is an exact alignment of length M ≥ l for the substring starting at position i in the query and ending at or along the edge to node N. The length of the alignment (M) is equal to the length of the path-label of the parent of node N plus the length along the edge to N. Starting from N, the algorithm follows successive parent links up the tree, subtracting the edge length of each link from the alignment length, until the alignment length is less than l as shown in Figure 10. Let R be the node with the smallest string depth greater than l on this path. For each leaf L in the subtree rooted by R, the path-label to the lowest common ancestor of N and L defines a substring starting at i in Q which occurs in both Q and S at the reference position defined by the leaf label of L. For a thorough discussion of suffix trees and their applications, see Gusfield's classic work on sequence analysis [48].

GPGPU programming

As the GPU has become increasingly more powerful and ubiquitous, researchers have begun exploring ways to tap its power for non-graphics, or general-purpose (GPGPU) applications [49]. This has proven challenging for a variety of reasons. Traditionally, GPUs have been highly specialized with two distinct classes of graphics stream processors: vertex processors, which compute geometric transformations on meshes, and fragment processors, which shade and illuminate the rasterized products of the vertex processors. The GPUs are organized in a streaming, data-parallel model in which the processors execute the same instructions on multiple data streams simultaneously. Modern GPUs include several (tens to hundreds) of each type of stream processor, so both graphical and GPGPU applications are faced with parallelization challenges [50]. Furthermore, on-chip caches for the processing units on GPUs are very small (often limited to what is needed for texture filtering operations) compared to general purpose processors, which feature caches measured in megabytes. Thus, read and write operations can have very high latency relative to the same operations when performed by a CPU in main memory.

Most GPGPU successes stem from scientific computing or other areas with a homogeneous numerical computational component [51, 52]. These applications are well suited for running on graphics hardware because they have high arithmetic intensity – the ratio of time spent performing arithmetic to the time spent transferring data to and from memory [53]. In general, the applications that have performed well as a GPGPU application are those that can decompose their problems into highly independent components each having high arithmetic intensity [54]. Some bioinformatics applications with these properties have been successfully ported to graphics hardware. Liu et al. implemented the Smith-Waterman local sequence alignment algorithm to run on the nVidia GeForce 6800 GTO and GeForce 7800 GTX, and reported an approximate 16× speedup by computing the alignment score of multiple cells simultaneously [55]. Charalambous et al. ported an expensive loop from RAxML, an application for phylogenetic tree construction, and achieved a 1.2× speedup on the nVidia GeForce 5700 LE [56].

nVidia's new G80 architecture radically departs from the traditional vertex+fragment processor pipeline. It features a set of multiprocessors that each contain a number of stream processors (Figure 11). Graphics applications can use these as either vertex or fragment processors, and GPGPU applications can program them for general computation. All processors on a single multiprocessor simultaneously execute the same instruction, but different multiprocessors can execute different instructions. nVidia anticipated the benefits of such a unified architecture for GPGPU computing, and released the Compute Unified Device Architecture (CUDA) SDK to assist developers in creating non-graphics applications that run on the G80 and future GPUs. CUDA offers improved flexibility over previous GPGPU programming tools, and does not require application writers to recast operations in terms of geometric primitives, as was required by earlier GPGPU environments [57].

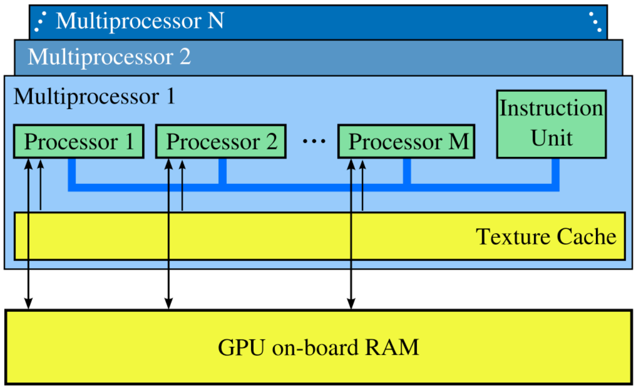


Figure 11. Simplified view of the nVidia G80 Architecture.
This figure, inspired by a similar figure in [57] shows how the GPU is organized into several (N) multiprocessors, each containing multiple (M) stream processors that simultaneously execute the same instruction. Each processor can access the texture cache very quickly, but reads and writes to the onboard RAM have high latency.

CUDA enables programmers to write programs that run on the GPU in a restricted form of the C programming language, and compiled into G80 bytecode. CUDA programs typically consist of a component that runs on the CPU, or host, and a smaller but computationally intensive component called the kernel that runs in parallel on the GPU (Figure 12). The kernel cannot access the CPU's main memory directly – input data for the kernel must be copied to the GPU's on-board memory prior to invoking the kernel, and output data also must first be written to the GPU's memory. All memory used by the kernel must be preallocated, and the kernel cannot use recursion or other features requiring a stack, but loops and conditionals are allowed. Furthermore, the number of registers per multiprocessor is limited and the multiprocessor schedules fewer processors to compute simultaneously if the number of registers used per kernel is too high. Consequently, high-performance kernel code requires careful tuning to reduce the number of registers used and limit the amount of branching.

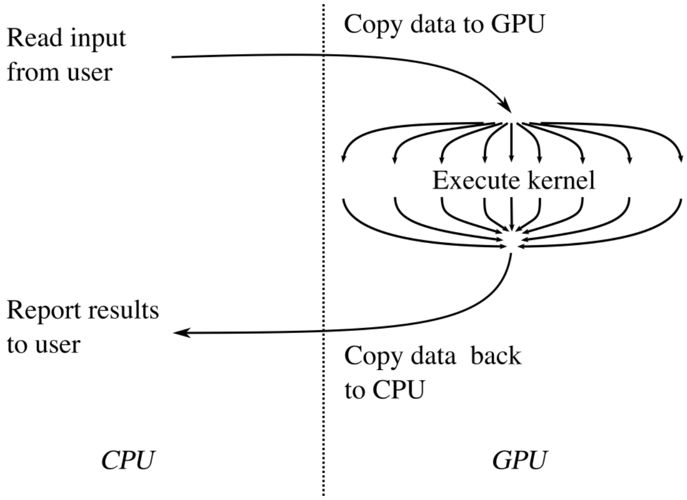


Figure 12. Typical GPGPU application flow.
Input data for a GPGPU application must be copied to the GPU's memory along with a pre-allocated output buffer prior to invoking the GPU-based kernel. Output from the kernel is read back into main memory and reported to the user.

The improved flexibility of CUDA does not solve the more fundamental problems caused by the G80's stream-computing organization: the relatively small cache and associated high memory latency for memory intensive programs. However, the G80's texture memory is cached to speed up memory intensive texture mapping operations, and can be used by GPGPU programs. GPGPU programs can pack their data structures into one-, two-, or three-dimensional arrays stored in texture memory, and thus use the cache for read-only memory accesses to these data structures [57]. Performance is further improved by utilizing one of several software techniques for maximizing the benefit offered by even a small cache. One such class of techniques involves reordering either the data in memory or the operations on those data to maximize data and temporal locality. Mellor-Crummey et al. reported significant speedup in particle interaction simulations, which feature highly irregular access patterns, by reordering both the locations of particles in memory and the order in which interactions were processed. They tested a reordering strategy based on space-filling curves, such as the Hilbert and Morton curves [58].

Implementation

The MUMmerGPU algorithm performs parallelized exact string alignment on the GPU (Figure 13). First a suffix tree of the reference sequence is constructed on the CPU using Ukkonen's algorithm [47] and transfered to the GPU. Then the query sequences are transfered to the GPU, and are aligned to the tree on the GPU using the alignment algorithm described above. Alignment results are temporarily written to the GPU's memory, and then transfered in bulk to host RAM once the alignment kernel is complete for all queries. Finally, all maximal alignments longer than a user-supplied value (l) are reported by post-processing the raw alignment results on the CPU. The output format and many parameters of MUMmerGPU are identical to those of mummer (with the -maxmatch option), up to the order in which alignments appear in the output for each query, and thus MUMmerGPU can be used as a drop-in replacement for mummer. In particular, all programs in the NUCmer suite of programs that use the output of mummer, including those that extend the exact alignment seeds to larger inexact alignments, can take advantage of the GPU parallelization [10, 14, 39].

Algorithm 1: MUMmerGPU

```
Build $k$ overlapping suffix trees from reference
foreach $QueryBlock$ do
    $Output \leftarrow \emptyset$
    Load $QueryBlock$ onto GPU
    foreach $Tree$ do
        Load $Tree$ onto GPU
        Run alignment kernel on $(QueryBlock, Tree)$
        Add kernel output to $Output$
        Unload $Tree$
    foreach $query \in QueryBlock$ do
        Print output for $query$ in $Output$
    Unload $QueryBlock$
```

Figure 13. MUMmerGPU Algorithm.
MUMmerGPU builds multiple suffix trees of the reference and partitions the query sequences into sets, called QueryBlocks, depending on the memory available on the GPU. Sequences within a given QueryBlock are aligned in parallel on the GPU.

The G80 has a relatively small amount of on-board memory, so the data are partitioned into large blocks so that the reference suffix tree, query sequences, and output buffers will fit on the GPU. As of this writing, the amount of on-board memory for a G80 ranges from 256 MB to 768 MB. A suffix tree built from a large reference sequence, such as a human chromosome, will exceed this size, so MUMmerGPU builds k smaller suffix trees from overlapping segments of the reference. MUMmerGPU computes k at runtime to fill approximately one third of the total GPU device memory with tree data. The trees overlap in the reference sequence by the maximum query length m supported by MUMmerGPU (currently 8192 bp) to guarantee all alignments in the reference are found, but alignments in the overlapping regions are reported only once.

After building the trees, MUMmerGPU computes the amount of GPU memory available for storing query data and alignment results. The queries are read from disk in blocks that will fill the remaining memory, concatenated into a single large buffer (separated by null characters), and transferred to the GPU. An auxiliary 1D array, also transfered to the GPU, stores the offset of each query in the query buffer. Each multiprocessor on the GPU is assigned a subset of queries to process in parallel, depending on the number of multiprocessors and processors available. The executable code running on each processor, the kernel, aligns a single query sequence from the multiprocessor's subset to the reference. The kernel aligns the query to the reference by navigating the tree using the suffix-links to avoid reprocessing the same character of the query, as described above. Reverse complement alignments are computed using a second version of the kernel which reverse complements the query sequences on-the-fly while aligning, allowing for computing both forward and reverse alignments without any additional data transfer overhead. The output buffer contains a slot to record the alignment result for each of the m - l + 1 substrings for a query of length m. The fixed size alignment result consists of the node id of the last visited node in the tree and length of the substring that exactly aligns. This information is sufficient to print all positions in the reference that exactly align the substring on the CPU.

After the kernel is complete for all the queries, the output buffer on the GPU is transfered to host RAM and the alignments are printed by the CPU. Each slot in the output buffer corresponds to a specific substring of a query. If multiple trees were built from the reference (k > 1), then the output slots for each tree are preserved until the queries in a block have been aligned against each tree. This way all of the alignments for a given query can be printed in a single block, following the syntax used by mummer.

GPU Memory Layout

The suffix tree is "flattened" into two 2D textures, the node texture and the child texture. Each tree node is stored in a pair of 16-byte texels (texture elements) in these two textures. The node texture stores half the information for a node, including the start and end coordinates of the edge sequence in the reference, and the suffix link for the node. The remaining information for a node – the pointers to its A, C, G & T children – is stored in the child texture, addressed in parallel to the node texture. An auxiliary table containing each node's edge length, sequence depth, parent pointer, and suffix number for leaf nodes, is stored in RAM and is used during the output phase.

In the CUDA architecture, a program can store read-only data as cached textures. The G80's proprietary caching scheme takes advantage of 2D locality common in texturing operations. Therefore, the algorithm attempts to optimize the 2D locality of the tree structure in these textures by organizing the nodes in 32 × 32 texel blocks as shown in Figure 14. Near the root of the tree (node depth <16), nodes are assigned using a level-order (breadth-first) traversal of the tree creating "wide" blocks of the tree. This ensures that all nodes near the root of the tree are placed in the first 32 × 32 texel blocks, and guarantees the children of a given node will be at (nearly) adjacent cells in the texture. This is useful because at this depth, loading a single 32 × 32 block for one kernel is likely to be reused for the other kernels running in parallel. Further from the root (depth ≥ 16), nodes are arranged in "tall" blocks so that a node, its children, grandchildren, and great-grandchildren are adjacently placed in the same (or adjacent) 32 × 32 block. As multiple queries are aligned against lower parts of the tree, it becomes less likely that their kernels will access many of the same nodes. Thus, the data reordering scheme attempts to increase the cache hit rate for a single thread. The exact specification of the G80's caching scheme is proprietary information, but empirically, this hybrid layout seems to maximize the cache hit rate near the root of the tree, and towards the leaves where the kernel access patterns are radically different.

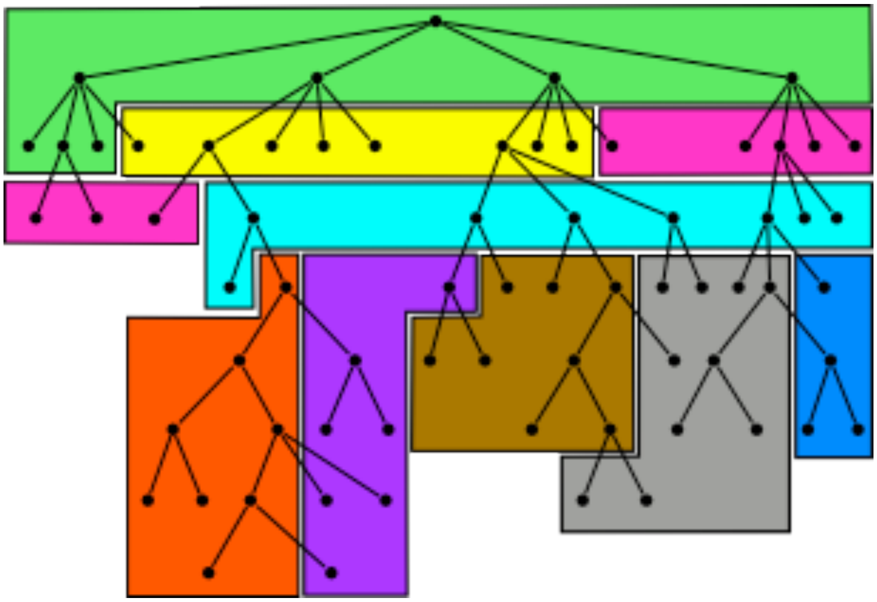


Figure 14. Suffix Tree Layout in MUMmerGPU.
The nodes of the suffix tree are rearranged into cache blocks to optimize 2D locality. Near the root of the tree, nodes of the same depth are placed into the same "wide" block. Futher down the tree, nodes from the same subtree are placed into the same "tall" block. MUMmerGPU uses blocks of 32 × 32 nodes, but for clarity, 8 nodes cache blocks are displayed here.

The reference sequence for the tree is transferred to the GPU as a third 2D texture, and is reordered along a simple 2D space-filling curve to maximize the cache hit rate for subsequent accesses along a node's edge. The sequence is reordered so that beginning with the first character, every four characters in the reference become the topmost four characters in the columns of the 2D array. Once the array contains 4 × 65,536 characters, successive four-character chunks become the next four characters in the columns, left-to-right, and so on. We experimented with a variety of other data reordering schemes, including along a Morton curve and other space filling curves, and found this to have the best performance on several reference sequences. Altogether, using cache memory organized with the spacing-filing curves for the suffix tree and reference sequence improved the kernel execution speed by several fold.

Complexity of MUMmerGPU

MUMmerGPU constructs its suffix trees in O(n) time with Ukkonen's algorithm, where n is the length of the reference. The alignment kernel running on the card computes all exact substring alignments for each query in time linear in the length of the query. The kernel is an implementation of existing alignment methods [48], but with many independent instances running simultaneously on the GPU.

MUMmerGPU uses both GPU memory and main system memory. Suffix trees use an amount of memory linear in the length of the reference from which they are constructed [48]. The suffix trees in MUMmerGPU thus each occupy O(n/k + m) space, where k is the number of overlapping trees specified by the user, and m is the maximum query length supported by MUMmerGPU. Note that for most expected uses of MUMmerGPU n >> m. Only a fraction of that total space is actually transferred to the GPU. In the current implementation, 32 out of every 48 bytes per node are transferred. The remaining bytes are stored in the host-only auxiliary table used only for printing results by the CPU. For each query, MUMmerGPU transfers the null terminated query sequence prepended with a special mismatch character, along with two 4-byte entries in auxiliary tables used by the kernel. For a query of length m, and a minimum substring length l, m - l + 1 output slots are reserved to record the query's substring alignments, and each output slot occupies 8 bytes. The total space required on both the CPU and the GPU for each query is 8(m - l + 1) + (m + 10) bytes. On a G80 with 768 MB of on-board RAM, there is sufficient RAM to store a tree for a 5 Mbp reference sequence, and 5 million 25 bp or 500,000 100 bp query sequences.

Results and Discussion

We measured the relative performance of MUMmerGPU by comparing the execution time of the GPU and CPU version of the alignment code, and the total application runtime of MUMmerGPU versus the serial application mummer. The test machine has a 3.0 GHz dual-core Intel Xeon 5160 with 2 GB of RAM, and an nVidia GeForce 8800 GTX. The 8800 GTX has 768 MB of on-board RAM and a G80 with 16 multiprocessors, each of which has 8 stream processors. At the time of this writing, the retail price of the 8800 GTX card is $529, and a retail-boxed Intel Xeon 5160 CPU is $882 (http://www.newegg.com). Input and output was to a local 15,000 RPM SATA disk. The machine was running Red Hat Enterprise Linux release 4 update 5 (32 bit), CUDA 1.0, and mummer 3.19.

We ported the MUMmerGPU alignment kernel to use the CPU instead of the GPU to isolate the benefit of using graphics hardware over running the same algorithm on the CPU. CUDA allows programmers to write in a variant of C, so porting MUMmerGPU to the CPU required only straightforward syntactic changes, and involved no algorithmic changes. Where the CUDA runtime invokes many instances of the kernel on the GPU simultaneously, the CPU executes each query in the block sequentially.

The first test scenario was to align synthetically constructed reads to a bacterial genome. We used synthetic reads in order to explore MUMmerGPU's performance in the absence of errors and over a wider variety of query lengths than are available with genuine reads. The synthetic test reads consisted of 50-, 100-, 200-, 400-, and 800-character substrings (uniformly randomly) sampled from the Bacillus anthracis genome (GenBank ID: NC_003997.3). Thus, each read exactly aligns to the genome end-to-end at least once, and possibly more depending on the repeat content of the genome. When aligning each of the five sets of reads, we used l equal to the read size for the set. Each set contained exactly 250,000,000 base pairs of query sequence divided evenly among all the reads in the set.

The time for building the suffix tree, reading queries from disk, and printing alignment output is the same regardless of whether MUMmerGPU ran on the CPU or the GPU, since those parts of MUMmerGPU always run on the CPU. The actual sequence alignment portion of MUMmerGPU ran dramatically faster, over 10× faster, on the GPU, despite the added cost of transferring the tree and query data to the GPU. The speedup of MUMmerGPU (not including the costs mentioned above shared by both variants) running on the GPU over MUMmerGPU on the CPU is shown in Figure 15.

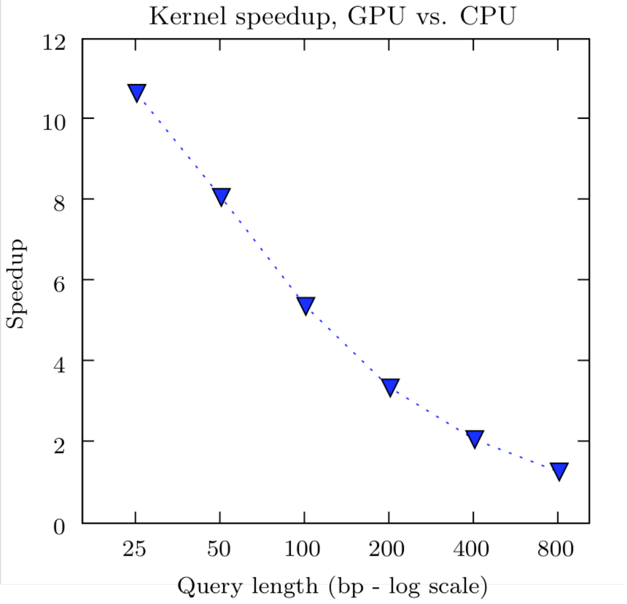


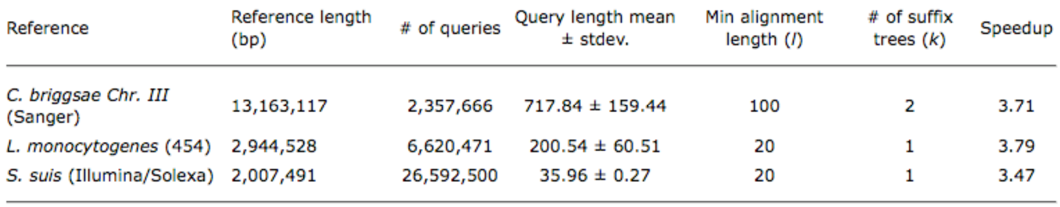
Figure 15. MUMmerGPU Speedup on the GPU over the CPU.
The decrease in speedup when processing error-free synthetic reads as read length increases is due to a combination of thread divergence and poor cache hit rate.

For longer reads, the speedup of using the GPU is diminished, because of poor cache performance and thread divergence, both of which are acknowledged as potential performance problems on the G80 [57]. All queries begin at the root of the tree, and many queries will share common nodes on their paths in the tree. However, as the kernel travels deeper into the tree for longer reads, the texture elements stored in the cache are reused less often, thus reducing the cache hit rate, and increasing the overall average access time. In addition, even though queries are the same length, the alignment kernel may not visit the same number of nodes, nor spend the same amount of time comparing to edges, because edges in suffix trees have variable length. This creates divergence among the threads processing queries, and the multiprocessor will be forced to serialize their instruction streams. It is difficult to quantify the relative contribution of these effects, but it is likely that both are significant sources of performance loss.

In addition to the test with synthetic data, we also aligned reads from several recent sequencing projects against the genomes from which the reads were generated. The projects included *Streptococcus suis* sequenced with the Solexa/Illumina sequencer (http://www.sanger.ac.uk/Projects/S_suis/), multiple strains of *Listeria monocytogenes* sequenced using 454 pyrosequencing (Genome GenBank ID: NC_003210.1, read TI numbers 1405533909 – 1405634798, 1406562010 – 1406781638, 1407073020 – 1411183505, 1413490052 – 1415592095, 1415816363 – 1415903784) and *Caenorhabditis briggsae* sequenced with standard ABI 3730xl Sanger-type sequencing [59]. We aligned the reads against both strands of the chromosomal DNA for *L. monocytogenes* and *S. suis*, and against both strands of

Chromosome III of *C. briggsae*. Little data from Solexa/Illumina has been made public at the time of this writing, and the public data set available had only a single lane's worth of data. To represent the full set of reads from a full Solexa/Illumina run, we concatenated 10 copies of the publicly available file containing 2,659,250 36 bp reads to form the *S. suis* query set. The reference sequence and queries in all three tests did not include ambiguous bases. For these three tasks, Table 1 shows the runtime parameters used and the overall speedup of MUMmerGPU over mummer. Figure 16 shows the wall-clock time spent by MUMmerGPU in the various phases of the algorithm, including kernel execution and I/O between CPU and GPU.

Table 1. MUMmerGPU Runtime Parameters and Speedup.



Reference	Reference length (bp)	# of queries	Query length mean ± stdev.	Min alignment length (*l*)	# of suffix trees (*k*)	Speedup
C. briggsae Chr. III (Sanger)	13,163,117	2,357,666	717.84 ± 159.44	100	2	3.71
L. monocytogenes (454)	2,944,528	6,620,471	200.54 ± 60.51	20	1	3.79
S. suis (Illumina/Solexa)	2,007,491	26,592,500	35.96 ± 0.27	20	1	3.47

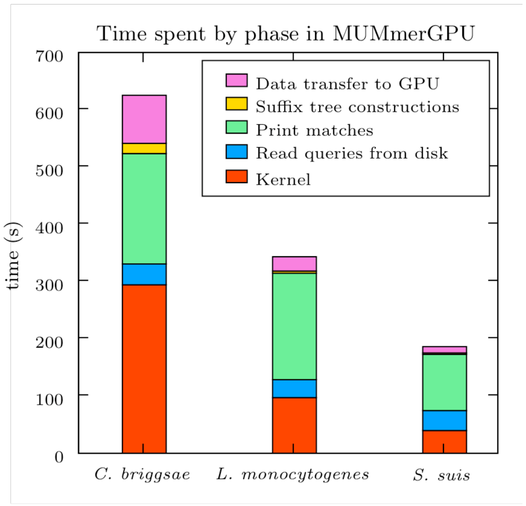


Figure 16. MUMmerGPU Runtime by Algorithm Phase
The stacked bar charts indicate the amount of time spent in each phase of the MUMmerGPU for the three test sets. Given a sufficiently large number of sequencing reads, the time spent building the suffix tree is small compared to time spent aligning queries.

For each of the alignment tasks, MUMmerGPU was between 3.47 and 3.79 times faster than mummer. For *C. briggsae*, MUMmerGPU spent most of its time aligning queries on the GPU. Because we aligned all of the reads from the sequence project against chromosome III of the *C. briggsae*, many of the reads did not align anywhere in the reference. As a result, a relatively short amount of time was spent in writing alignment output to disk. For other alignments, such as for the *L. monocytogenes* and *S. suis* test sets, the output phase dominates the running time of MUMmerGPU. For these tasks, printing the output in parallel with aligning a block of queries would provide substantial speedup, as it would hide much of the time spent aligning queries on the card. We plan to adopt this strategy in a future release of MUMmerGPU.

Despite the performance hazards experienced for longer simulated reads, MUMmerGPU on the GPU consistently outperforms mummer on real sequencing data by more than a factor of three in wall-clock application running time. Unlike the idealized simulated reads, these reads are variable length and have sequencing error, which will cause further divergence in the kernel executions. Furthermore, the *C. briggsae* alignment required the use of a segmented suffix tree and associated data transfer overhead. In general, MUMmerGPU confers significant speedup over mummer on tasks in which many short queries are aligned to a single long reference.

Conclusion

Operations on the suffix tree have extremely low arithmetic intensity – they consist mostly of following a series of pointers. Thus, sequence alignment with a suffix tree might be expected to be a poor candidate for a parallel GPGPU application. However, our results show that a significant speedup, as much as a 10-fold speedup, can be achieved through the use of cached texture memory and data reordering to improve access locality. This speedup is realized only for large sets of short queries, but these read characteristics are beginning to dominate the marketplace for genome sequencing. For example Solexa/Illumina sequencing machines create on the order of 20 million 50 bp reads in a single run. For a single human genotyping application, reads from a few such runs need to be aligned against the entire human reference genome. Thus our application should perform extremely well on workloads commonly found in the near future. The success of our application is in large part the result of the first truly general purpose GPU programming environment, CUDA, which allowed us to directly formulate and implement our algorithm in terms of suffix tree navigation and not geometric or graphics operations. This environment made it possible to efficiently utilize the highly parallel and high speed 8800 GTX. An 8800 GTX is similar in price to a single 3.0 Ghz Xeon core, but offers up to 3.79× speedup in total application runtime. Furthermore, in the near future, a common commodity workstation is likely to contain a CUDA compliant GPU that could be used without any additional cost.

Even though MUMmerGPU is a low arithmetic memory intensive program, and the size of the stream processor cache on the G80 is limited, MUMmerGPU achieved a significant speedup, in part, by reordering the nodes to match the access patterns and fully use the cache. We therefore expect with careful analysis of the access pattern, essentially any highly parallel algorithm to perform extremely well on a relatively inexpensive GPU, and anticipate widespread use of GPGPU and other highly parallel multicore technologies in the near future. We hope by making MUMmerGPU available open source, it will act as a roadmap for a wide class of bioinformatics algorithms for multi-processor environments.

Chapter 3: Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment

Summary of Contribution

This chapter describes version 2.0 of the high-throughput sequence alignment program MUMmerGPU published in the journal Parallel Computing [35], and was developed in collaboration with Cole Trapnell at the University of Maryland. MUMmerGPU 2.0 improves on MUMmerGPU 1.0 [17] to increase performance of aligning large batches of next generation sequence reads to a reference genome.

MUMmerGPU 1.0 accelerates this computation with a single GPU kernel that computes maximal substring matches between the query sequence and a suffix tree representation of the reference genome stored on the GPU. The match kernel outputs the longest exact match between the query and the reference starting at each position of the query. MUMmerGPU 1.0 would then post-process the exact matches on the CPU to compute all maximal exact matches between the query and the reference. MUMmerGPU 2.0 accelerates this post-processing with a second GPU kernel using a novel stackless depth first search of the suffix tree. A stackless depth first search was necessary to overcome a major limitation of the GPU that memory cannot be dynamically allocated on the GPU. The new kernel was 10 times faster than the CPU version of the depth first search.

MUMmerGPU 2.0 also carefully examined the memory and kernel parameters to better tune the performance on a variety of common workloads. A total of 128 different configurations were evaluated, and we determine higher stream processor occupancy is the dominate factor towards achieving high GPU performance.

Michael Schatz implemented the second GPU kernel and improved performance of the match kernel. Cole Trapnell evaluated the memory and kernel parameters, implemented support for the second GPU kernel in the driver program, and made major improvements to the driver program for processing large genomes. Cole Trapnell and Michael Schatz wrote the paper together.

Abstract

MUMmerGPU uses highly-parallel commodity graphics processing units (GPU) to accelerate the data-intensive computation of aligning next generation DNA sequence data to a reference sequence for use in diverse applications such as disease genotyping and personal genomics. MUMmerGPU 2.0 features a new stackless depth-first-search print kernel and is 13x faster than the serial CPU version of the alignment code and nearly 4x faster in total computation time than MUMmerGPU 1.0. We exhaustively examined 128 GPU data layout configurations to improve register footprint and running time and conclude higher occupancy has greater impact than reduced latency. MUMmerGPU is available open-source at http://mummergpu.sourceforge.net.

Introduction

Graphics Processing Units (GPUs) were originally designed for efficient data-parallel graphics computations, such as in scene rasterization or lighting effects. However, as GPUs have become more powerful with dozens or hundreds of stream processors, researchers have begun using them for general-purpose (GPGPU) computations. Early attempts to exploit GPU's high level of parallelism for non-graphical tasks required application developers first re-cast their problem into graphics primitives, and re-interpret graphical results. However, recent toolkits from both nVidia [57] and ATI [60] have enabled developers to write functions called kernels in a restricted variant of C that execute in parallel on the stream processors. High-level toolkits coupled with powerful, low cost hardware have sparked huge interest in developing GPGPU versions of data-parallel applications.

Read mapping is a data-parallel computation essential to genome re-sequencing, a rapidly growing area of research. In this computation, millions of short DNA sequences, called reads, obtained from a donor are individually aligned to a reference genome to find all locations where each read occurs in the reference sequence, with allowance for slight mismatches for biological and technical reasons. Read mapping can be used, for example, to catalog differences in one person's genome relative to the reference human genome, or compare the genomes of model organisms such as Drosophila melanogaster (fruit fly) or Arabidopsis thaliana (thale cress). Researchers use this information for a wide variety of analyses, since even a single nucleotide difference can have a dramatic effect on health and disease. Next-generation DNA sequencing technologies from Illumina, 454 Life Sciences, and

Applied Biosystems have recently become extremely popular because they can create billions of bases of sequence data in a single sequencing run at relatively low cost [41]. The DNA of James Watson, a co-discoverer of the molecule's structure, was recently sequenced using technology from 454 Life Sciences in just two months. Biotechnology researchers believe that within the next several years, an individual will be able to have his or her DNA sequenced in only a few days and for as little as $1000 [61]. Despite their popularity, the most widely used sequence alignment programs are unable to handle the extreme workload required by the new technology. The MUMmerGPU system uses the highly parallel graphics cards from nVidia and their CUDA GPGPU toolkit to process next generation sequencing reads in a fraction of the time of other programs [17].

MUMmerGPU 2.0 uses the same suffix tree based match kernel as described in the original version of MUMmerGPU, but we have added several significant improvements to increase performance and capabilities for the overall application. First, we implemented a new query streaming model in which reads are streamed past overlapping segments of the reference, allowing us to compute alignments to Mammalian-sized reference genomes. Second, we implemented a new GPU-based print-kernel that post-processes the tree coordinates from the match kernel into exact alignment coordinates suitable for printing. This computation had previously been the limiting factor in end-to-end application time for commonly used parameters. The print kernel performs the computation via an iterative depth-first-search on the suffix tree using a constant amount of memory and no stack. This non-traditional implementation is required to meet the severe restrictions on kernel code, but is between 1.5- and 4-fold faster than the previous (CPU-based) version of the routine. Popov et al recently reported a different algorithm for traversing trees in a CUDA kernel [62] which requires additional pointers between the leaf nodes in a kd-tree, but our technique is applicable to any tree without additional pointers. Finally, we optimized performance for both kernels by identifying the best organization of the DNA sequencing reads and suffix tree in GPU memory. We explore and report on 128 variations of the data layout policy, and quantify the tradeoffs involved for kernel complexity, cache use, and data placement. We find that optimizing these choices can greatly accelerate performance, and mistuned choices have an equal but negative effect on performance compared to the naïve version. Processor occupancy dominated performance for our data-intensive application, but techniques that reduce GPU memory latency without compromising occupancy were also generally beneficial. We describe several techniques to reduce kernel register footprint and thus improve occupancy that are widely applicable to GPGPU programs. Overall, MUMmerGPU 2.0 is nearly 4x faster in total computation time than the originally published version of the code for the most commonly encountered workloads.

GPGPU Programming

Recent GPUs from nVidia have up to 256 stream processors running at a core frequency of up to 650 MHz. [63] Each stream processor has an individual arithmetic logic unit (ALU), but the stream processors are grouped into multiprocessors such that all of the stream processors in the same multiprocessor execute the same instruction at the same time (SIMD architecture). The functions that execute on the stream processors are called kernels, and a single instance of a running kernel is called a thread. Threads are launched in groups of 32 called warps that the multiprocessor uses for scheduling, and are further organized in larger groups called thread blocks of user specified size with the guarantee that all threads in the same thread block will execute concurrently. A GPU has up to 1.5 GB of on-board memory, but very small data caches compared to general purpose CPUs (only 8KB per multiprocessor). Cached memory is only available for read-only data and for a small number of word-aligned data types called textures. Non-cached memory has very high latency (400 to 600 clock cycles), but multiprocessors attempt to hide this latency by switching between warps as they stall.

Kernel code is written in a restricted variant of C and compiled to GPU specific machine code using the CUDA compiler, NVCC. Developing kernel code can be challenging because commonly used programming features, such as dynamic memory allocation and recursion, are not available. Loops and conditionals are allowed in kernel code, but if different threads in the same warp follow different branches, then the multiprocessor will automatically serialize or stall execution until the threads resynchronize, thus cutting effective parallelism and end-to-end application performance. Furthermore, each multiprocessor has a fixed number of registers available for its stream processors, so the number of threads that can execute concurrently is determined in part by how many registers each thread requires. The percent of stream processors in a multiprocessor that execute concurrently, processor occupancy, is available in discrete levels depending on the number of registers used by each thread, the thread block size, and the physical characteristics of the device including the number of registers present on each multiprocessor, the maximum number of concurrent warps, and the maximum number of concurrent thread blocks Threads are executed in discrete units of the thread block size such that the total number of registers used by all concurrent threads is at most the number available on the device. For example, an nVidia 8800 GTX has 8192 registers per multiprocessor, and can execute at most 8 concurrent thread blocks per multiprocessor and at most 24 concurrent warps of 32 threads per multiprocessor (a maximum of 768 concurrent threads total). If the thread block size is 256 a kernel will have 100% occupancy if it uses at most 10 registers (allowing 3 complete thread blocks), 66% occupancy for at most 16 registers (allowing 2 complete thread blocks), 33% occupancy for at most 32 registers (allowing 1 complete thread block) , and fail to launch if each thread requires more than 32 registers because one thread block would require more than 8192 registers Finally, kernel code cannot directly address main memory nor other devices, so inputs to the kernel must be copied to the GPU's on-board memory prior to execution and outputs must be copied to main memory from on-board memory after execution. The full details of the device capabilities and programming model are described in the CUDA documentation. [57]

GPU accelerated versions of data parallel-applications have been developed for numerous application domains, including molecular dynamics, numerical analysis, meteorology, astrophysics, cryptography, and computational biology. [55, 64-66] The most successful GPGPU applications have generally had high arithmetic intensity, meaning processing time is dominated by arithmetic operations with relatively few memory requests. These applications are well suited to the numerical capabilities of the stream processors. In contrast, data intensive applications requiring fast random access to large data sets have been generally less successful on the GPU, because of the GPUs small data caches and relatively high latency (400-600 clock cycles) for on-board memory accesses.

DNA Sequence Alignment

DNA is the molecule that encodes the genetic blueprint for the development and traits of an organism. It is composed of a long sequence of four possible nucelotides or ‘base pairs’ (bp): adenine (abbreviated A), cytosine (C), guanine (G) and thymine (T). The sequence of base pairs in biologically active regions called genes determines the amino acid sequence and function of biologically active molecules called proteins. Even a single nucleotide difference in a gene between two individuals can substantially change the function of its protein product and lead to disease. Larger insertion, deletion, or rearrangement events of several nucleotides can have profound effect on development, such as the chromosomal duplication responsible for Down syndrome. Numerous other human diseases and traits have been linked to both small-scale single nucleotide polymorphisms and larger genetic variations, and thus make DNA sequence analysis an extremely active and important field of research. [67]

Until recently, the most widely used protocol for determining the sequence of nucleotides in a genome used Sanger sequencing. Sanger sequencing can determine the order of ~1000 consecutive nucleotides by separating fluorescently tagged molecules based on their charge; each sequence fragment is called a “read.” Longer regions, including full genomes, are sequenced by sequencing random overlapping fragments, and then stitching the reads together computationally into the original full-length sequence. [21] New DNA sequencing protocols from Illumina, 454 Life Sciences, and Applied Biosystems sequence DNA at a much higher rate and dramatically lower cost, but the reads are significantly shorter (30-200bp). Nevertheless, there has been a dramatic shift towards using the cheaper sequencing protocols and placing the burden on computational resources to analyze the result with less information per read.

One of the most widely performed DNA analysis tasks is to align a pair of sequences to find regions that are similar. In the case of short sequencing reads, researchers will generally require that the entire read aligns end-to-end to a reference sequence except for a small number of differences, which may be real polymorphisms or sequencing errors. Modern sequence alignment algorithms use a technique called seed-and-extend to quickly perform the alignment by focusing the search to regions that are reasonably similar. In the first phase, the algorithms find substrings of sufficient length called seeds that are shared between the sequences. In the second phase, the algorithms extend the relatively short exact seeds into longer inexact alignments using a more sensitive dynamic programming algorithm. The widely used BLAST algorithm considers all possible fixed-length substrings called k-mers as seeds [11]. In contrast, the popular MUMmer algorithm and our high-performance variant MUMmerGPU compute variable length maximal exact matches (MEMs) as seeds for alignment. Both algorithms are much faster than using the original Smith-Waterman local sequence alignment algorithm, which requires time that is quadratic with respect to the input sequence sizes. By contrast, MUMmer and MUMmerGPU find MEMs using a suffix tree, which requires linear space and enables substring matching in linear time. [10, 17, 39] MUMmerGPU uses a very similar output format as MUMmer, and thus one can reuse MUMmer's components for extending the exact seeds into longer inexact alignments.

A suffix tree is a tree that encodes all suffixes of a string on a path from the root node to a leaf (Figure 17). A special character that does not occur in the original string ($) is appended to the reference string to ensure that each suffix ends at a unique leaf node, which is labeled by the starting position of the suffix called the leaf id. Edges of the tree are labeled with substrings of the reference, and internal nodes have at least 2 children representing positions where repeated suffix prefixes diverge. The path string of a node is the concatenation the edge labels along the path from the root to that node. The string depth of a node is the length of its path string. Suffix trees over fixed alphabets, such as for DNA nucleotides, can be constructed in linear time and space using additional pointers called suffix links, that point from node n with path string x∂ to node n' with path string ∂, where x is a single character and ∂ is a string. [47] Once built, a suffix tree allows one to find occurrences of a query string or substrings of a query string in the reference string in time proportional to the length of the query substring by matching characters of the query along the edges of the tree. Substring matches can be extended into MEMs by walking the suffix tree along the path of the substring matches as described below. For a complete description of suffix tree construction and search algorithms see the comprehensive reference by Gusfield. [48]

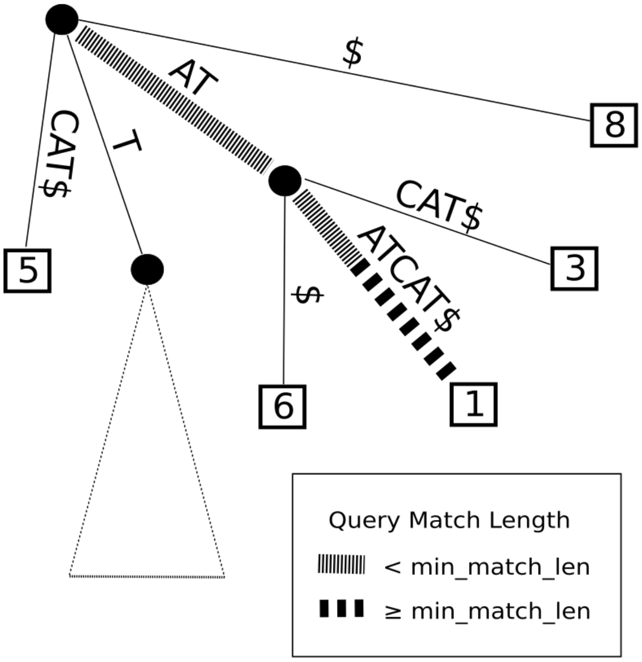


Figure 17. MUMmerGPU 2.0 Aligning a query to the suffix tree.
Aligning the query "ATAT" against the suffix tree for "ATATCAT". MUMmerGPU will report a match at position 1 in the reference, provided that the minimum match length l ≥ 4.

Alignment Algorithm

MUMmerGPU computes all MEMs that are at least the minimum match length (the parameter l) characters long between a reference sequence and a set of query sequences. The MEM computation is divided into four phases:

1. Reference Preprocessing – Load the reference from disk and construct a suffix tree of it.

2. Query Streaming – Load blocks of queries from disk and launch the alignment kernels.

3. Match Kernel – Match each suffix of each query to the suffix tree to find candidate MEMs.

4. Print Kernel – Post-processes the candidate MEMs to report all MEMs at least l characters long.

Both the match kernel and the print kernel are executed in parallel on the graphics card, as illustrated in Figure 18. A separate GPU thread running the match kernel processes each query. Then for each matching suffix of each query, a separate instance of the print kernel reports MEMs for that suffix. Suffix tree construction and I/O are executed serially on the CPU and require a small fraction of the overall runtime for large read sets.

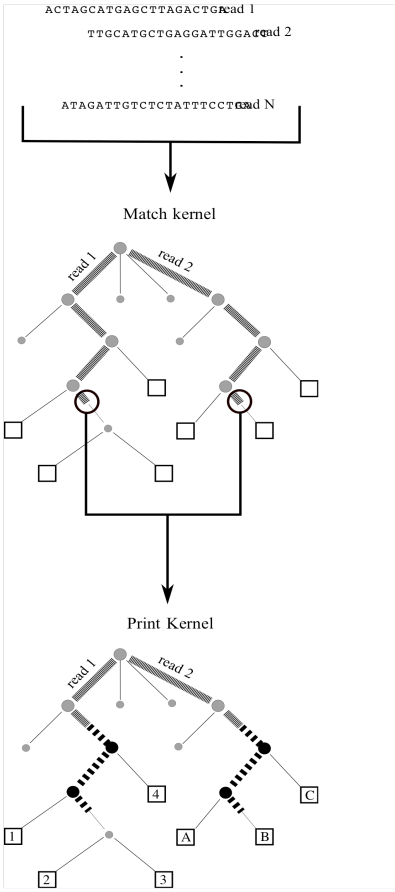

ACTAGCATGAGCTTAGACTGA read 1
TTGCATGCTGAGGATTGGACT read 2
ATAGATTGTCTCTATTTCCTGA read N
Match kernel
read 1
read 2
Print Kernel
read 1
read 2
4
C
1
A
B
2
3

Figure 18. MUMmerGPU 2.0 Overview
MUMmerGPU constructs suffix trees for overlapping sections of the reference string. Reads are first matched to the suffix tree in the match kernel. Tree coordinates are passed from the match kernel to the print kernel to convert tree coordinates to alignment coordinates in the reference string.

Reference Preprocessing

Since the reference sequence may be very large, the reference is divided into overlapping 8Mbp segments called pages. For each reference page, the algorithm constructs a suffix tree using Ukkonen's algorithm in linear time and flattens the tree into a large array suitable for processing on the GPU. Suffix tree construction time is generally a small fraction of the total runtime for typical datasets involving millions of reads [17]. Each suffix tree node requires 32 bytes of data, which is divided into two 16 byte structs called the node and children structs. The node struct contains the coordinates of the reference string for the edge label into that node, the string depth of the node, and the address of the parent and suffix nodes. The children struct contains the address of each of the five children (A,C,G,T,$) and a flag indicating if the node is a leaf. If the node is a leaf, then the leaf id and the character of the reference just prior to that suffix of the reference is stored instead of the children pointers. Node addresses are stored using 24 bit addresses to conserve space but limits the suffix tree to 16 million nodes, and the maximum page size to 8 million base pairs. The nodes of the tree are reordered using the previously described reordering scheme [17]. Briefly, nodes near the top of the tree are numbered according to a breath-first-traversal to maximize locality across threads, while nodes at depth ≥ 16 are assigned using a depth-first-traversal to maximize locality for a particular thread.

Query Streaming

Unlike previous versions of the code, which processed queries in memory across all reference pages, queries are streamed across the reference, meaning after a query is aligned to a reference page, the alignments are printed immediately and the query is flushed from memory. If the reference is larger than the page size, then it will be necessary to reload the queries multiple times from disk. This tradeoff was necessary to support aligning against very large reference sequence or aligning a large set of reads, either of which required a prohibitively large amount of host RAM in the previous version of the code.

Match Kernel

The match kernel is essentially the same as described in previous version of the code. Briefly, the kernel finds the longest matching substring of the query starting at each position of the query, i.e. each suffix of the query is considered. Starting with the first character of the query (i=1) and the root node of the reference suffix tree, the characters in the query are matched to the edges of the suffix tree one character at a time until a mismatch or the end of the query is reached. If the number of matching characters is at least l, the match is recorded in the output buffer for position i as the id of the lowest node visited and the length along the edge to that node. The next suffix of the query is then considered by following the suffix link from the parent of lowest node reached. This has the effect of removing the first base of the query from consideration, and allows the next suffix to be evaluated without returning to the root in the suffix tree.

Print Kernel

The print kernel post-processes the match kernel results into potentially many MEMs per match (see Figure 19). The match kernel reports the lowest node L in the tree that matches the ith suffix of the query. If the query match to L is longer than l, there are multiple substrings of ith suffix that match the reference and are at least l characters. Call node P the highest ancestor of L that has a string depth at least l characters long. The leaves of the subtree rooted at P determine where in the reference a substring starting at i occurs, and the string depth of the lowest common ancestor of those leaves and L determines the matching substring length. Because the match kernel reports the longest possible match for suffix i, all of the matches at the leaves are guaranteed to be right maximal. However, the print kernel must be careful to not report matches that are fully contained by matches to suffix i-1. That is, the raw matches for suffix i may not be left maximal so the left flanking base must be explicitly checked by comparing the i-1th character of the query to the corresponding character of the reference. The print kernel computes this check for all candidate MEMs via a depth-first-search of the suffix tree to all of the leaves in the subtree rooted at P.

The algorithm begins by following parent pointers from L to find node P by following the parent pointer stored in each node, and stopping when the string depth field is < l. It also finds the parent of P called node B. Starting at P, it attempts to traverse to the A child. If the A child is null, it tries the C child and so forth in lexicographical order. It proceeds down the tree in this way to the first (lexicographically smallest) leaf below P where the MEM criterion is evaluated by comparing the i-1th character of the query to the corresponding character in the reference string. This character is the character in the reference that is just before the suffix ending at that leaf, and is stored in the leaf node for efficiency. If the characters are different, the substring is a MEM and the coordinates of the substring are stored to the output buffer, as explained below. After processing the first leaf, the kernel traverses up to the parent of the leaf, and resumes processing with the lexicographically next child. Because leaves are always visited in lexicographic order and because the last child visited can be determined with a pointer comparison, the algorithm does not require a stack to determine where to search next. After processing the $ child, the kernel traverses up the tree and continues processing lexicographically. The algorithm ends when the current node is B.

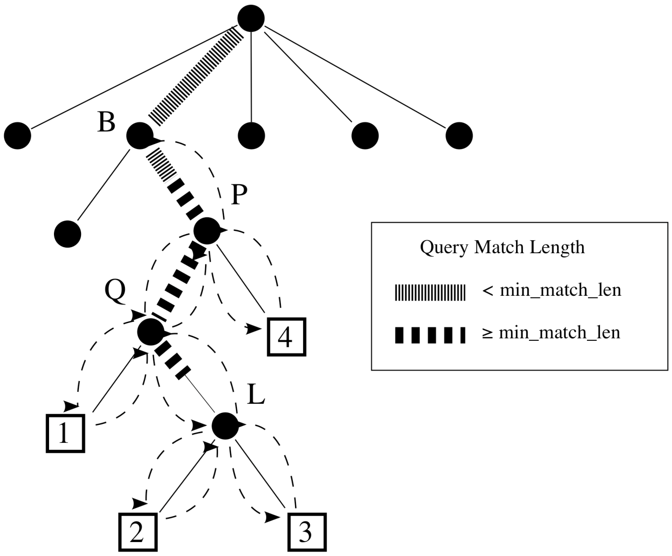


Figure 19. MUMmerGPU 2.0 Print Kernel Overview.
The query matches to the suffix tree along the bold path from the root to L. The print kernel post-processes this information to consider MEMs at the leaves by a stackless depth-first-search starting at P to leaves 1,2,3 & 4 shown by dashed arrows. The match length at leaf 1 is the string depth of Q, the match length at 2 and 3 is the string depth of Q plus the partial edge match to L, and the match length at leaf 4 is the string depth of P.

The coordinates of a MEM depend on the leaf id of the leaf for the start position, and the location of the leaf in the suffix tree relative to L, for the length. The length of the MEM is the string depth of the lowest common ancestor of the leaf node and L in all cases except for the leaves below L because the query may only have a partial match along the edge to L. Call the path of nodes between the parent of L and P the query path. Note the lowest common ancestor of a visited leaf must fall along the query path. When the traversal algorithm begins at P, the substring length is the string depth of P, and by definition P is along the query path. When traversing down the tree and the current node is along the query path, the algorithm checks if the child node is also on the query path. Call the character of the query at the string depth of the current node the query character. The query character determines which child of the current node is also in the query path. If the next node is along the query path, the matching substring length is the string depth of that node. If the next node is not on the query path, the matching substring length is not updated. Instead the kernel records and updates the distance to the query path throughout the traversal. When the distance returns to 0, the current node is once again in the query path and the algorithm resumes checking for the query child as before. Since there may be only a partial match to node L, a special condition checks when this node is visited, and the match length is set to the string depth of L's parent plus the partial edge match length reported by the match kernel.

Data Policies

MUMmerGPU 1.0 organized data on the GPU according to the few "best practices" that existed at the time. However, those best practices, such as whether or not to use texture memory, were developed for arithmetically intensive applications. In MUMmerGPU 2.0, we have revisited our decisions for seven possible boolean data organization policies and exhaustively tested all 128 possible combinations of choices. The policy choices are as follows.

1. Two-dimensional reference – store the reference string in a two-dimensional array instead of in linear memory.

2. Query texture – store the query strings in texture memory instead of global memory.

3. Reference texture – store the reference string in texture memory instead of global memory.

4. Tree texture – store the tree in a pair of textures instead of global memory.

5. Two-dimensional tree – store the tree in two-dimensional arrays instead of linear memory.

6. Tree reordering – reorder the nodes of the tree to improve locality instead of the node numbering determined by the construction algorithm.

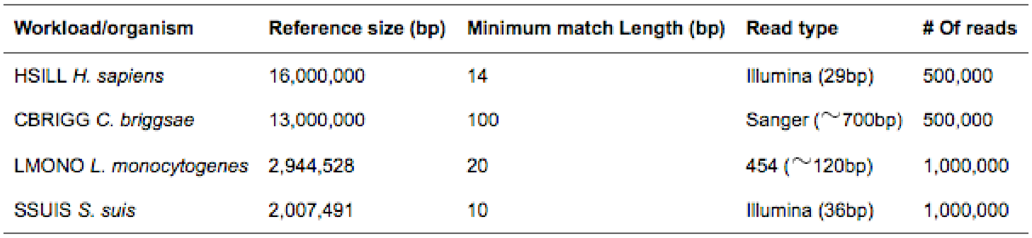
7. Merged tree – for a given node, store the node and children structs adjacent in memory, instead of two parallel arrays.

MUMmerGPU 1.0 stored the reference in a two-dimensional texture, the queries in linear global memory, and the tree in parallel two-dimensional textures after reordering the nodes. The texture cache in G80 series GPUs is described as being optimized for two-dimensional locality, so the node reordering was designed to exploit a two-dimensional cache block. Textures were selected for the reference and tree after preliminary testing suggested this selection had better performance. The tree structs were placed in parallel arrays to simplify addressing. In the following discussion, the naïve control configuration disables all optimization: no query texture, no reference texture, 1D reference, no tree texture, 1D tree, no tree reordering, and parallel arrays for the node and child structs.

We evaluated MUMmerGPU under each of the 128 possible combinations of policy choices on several workloads. Each workload constitutes a small slice of the input a life sciences researcher would provide to MUMmerGPU when mapping reads to a reference genome. The first workload, HSILL, represents a human resequencing project using next generation Illumina technology. The second workload, CBRIGG, represents a large eukaryotic resequencing project using traditional Sanger sequencing technology. The last two workloads, which we call SSUIS and LMONO, represent typical inputs for resequencing bacteria using next generation sequencing technologies from Illumina and 454 Life Sciences. All four workloads are comprised of genuine (non-simulated) sequencing reads, and are large enough to constitute a representative slice of work from the project, but terminate quickly enough to permit testing all 128 configurations. Table 2 presents the additional details about the reference sequences and read sets for each workload.

In a resequencing project, reads from a donor organism are aligned to a reference genome. Errors in the sequencing reads along with genuine variations between the donor and the reference genome will prevent some reads from aligning end-to-end without error. MUMmerGPU allows users to control the amount of error to tolerate by allowing users to specify the minimum match length (l) to report. The full details for choosing a proper value for l are beyond the scope of this chapter, but the choice of l can have a dramatic impact the running time, including determining which CUDA kernel dominates the computation, since smaller values of l produce more MEMs for the print kernel to report. For HSILL, we have chosen l to allow at most 1 difference in an alignment between a read and the reference, 2 differences for SSUIS, and numerous differences for LMONO and CBRIGG.

Table 2. MUMmerGPU 2.0 Workloads.



Workload/organism	Reference size (bp)	Minimum match Length (bp)	Read type	# Of reads
HSILL *H. sapiens*	16,000,000	14	Illumina (29bp)	500,000
CBRIGG *C. briggsae*	13,000,000	100	Sanger (~700bp)	500,000
LMONO *L. monocytogenes*	2,944,528	20	454 (~120bp)	1,000,000
SSUIS *S. suis*	2,007,491	10	Illumina (36bp)	1,000,000

Policy Analysis

For MUMmerGPU 2.0, we looked for a set of policy choices that universally reduced running time, as opposed to workload specific improvement. Ideally, a single configuration would be optimal for all workloads, otherwise, we desired configurations that improve HSILL, since we expect human resequencing projects using short reads will constitute the majority of the read alignment workloads in the near future. To this end, we executed MUMmerGPU with all 128 possible policy combinations on all 4 workloads. The test machine was a 3.0 GHz dual-core Intel Xeon 5160 with 2 GB of RAM, running Red Hat Enterprise Linux release 4 update 5 (32 bit). The GPU was an nVidia GeForce 8800 GTX, using CUDA 1.1. The 8800 GTX has 16 multiprocessors, each with 8 stream processors (128 stream processors total), and 768 MB of on-board RAM, with 8 KB of texture cache per multiprocessor. The data that follows is for the HSILL workload and excludes time spent reading from or writing to disk, as this time was identical within the workload, and only obscures the impact of different policy choices. In the figures, we have isolated the policy choice in question, and each bar represents the percent change in running time for enabling that policy while keeping the policy configuration otherwise the same. A positive value indicates the running time increased after enabling that policy, and negative values indicate the running times decreased. The bars are clustered by which textures are enabled, and are labeled by their non-texture policy choices. The label control indicates the default configuration without any non-texture policies enabled. The full results table for all workloads is available online at http://mummergpu.sourceforge.net.

Two-dimensional reference

Storing the reference string in a two-dimensional layout instead of a one-dimensional string consistently increased the total computation time, but only by an insignificant .4% on average (data not shown). We suspect the extra instructions necessary for addressing 2D memory slowed the overall performance relative to any potential gains by 2D locality. Consequently, only configurations that use a one-dimensional layout for the reference string were considered further. The following sections describe the remaining 64 policy configurations.

Query Texture

Configurations that placed the queries in a texture increased the print kernel time by 6.% and the match kernel time by 3% on average. When the tree texture is not used, the match runtime consistently decreased except in 2 exception cases with dramatically increased running time due to increased register footprint and decreased occupancy (Figure 20). When the tree texture is enabled, we observe what appears to be a small amount of cache competition in the match kernel which tends to slow down those configurations. The print kernel had similar results.

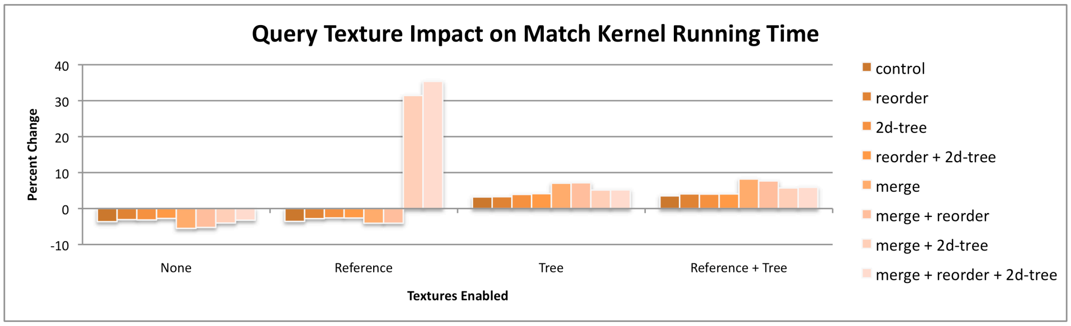


Figure 20. MUMmerGPU 2.0 Query Texture Impact.
Using the query texture with the match kernel generally improves the runtime, except when there is cache competition from the tree texture.

Reference texture

Configurations that placed the reference string in texture memory instead of global memory had significantly different match kernel running times (-12% to +35% change), but had essentially identical print kernel running times. This is as expected, since the print kernel does not access the reference string. Placing the reference string in a texture improved running time for the match kernel by up to 12% (without changing register usage), but only when the tree was not in texture memory as well

(Figure 21). We speculate that the tree, queries, and reference negatively compete for the texture cache in those cases, leading to overall lower performance. The two large increases (35%) in match kernel running time observed when the query texture was also used is a result of an increase in register usage and corresponding decrease in processor occupancy.

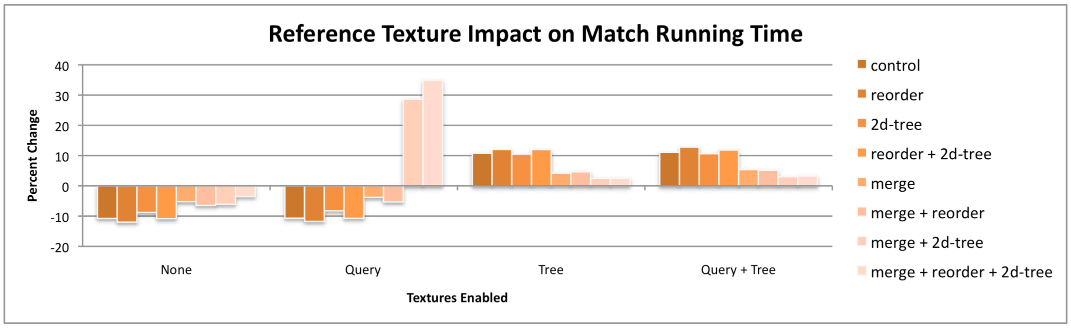


Figure 21. MUMmerGPU 2.0 Reference Texture Impact.
The reference string competes with the suffix tree for the texture cache. Configurations only benefit from placing the reference in texture memory when the tree is not also in texture memory.

Suffix tree texture

Storing the tree in a texture instead of global memory improves print kernel performance in almost all configurations and by 8% on average. The impact of this policy is more complicated for the match kernel (Figure 22). On average, using a texture for the tree improves match kernel performance by 11%, presumably because the cache lowers effective memory latency. In some configurations, though, the tree competes for the cache with the queries or reference and those configurations are generally slower than the equivalent configuration that uses global memory for the tree.

Interestingly, cache competition does not always result in an overall slowdown, especially when the register footprint was improved. In the match kernel, two configurations with multiple data types in texture requires 18 registers, yielding 33% occupancy. However, if these configurations are altered such that the tree is placed in a texture, the match kernel requires only 16 registers, achieving 66% occupancy and an overall speedup. In the print kernel, the control configuration uses 17 registers, and only 16 registers when the tree is placed in a texture and thus has improved occupancy and cache use. We also observed the opposite effect: for some configurations, placing the tree in a texture increased the print kernel register footprint, dropped occupancy, and slowed the overall computation.

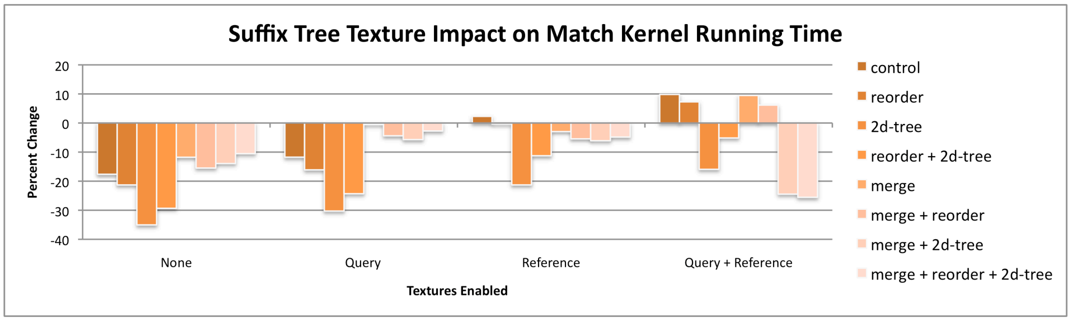


Figure 22. MUMmerGPU 2.0 Suffix Tree Texture Impact.
Placing the suffix tree in texture memory is not universally beneficial, presumably due to cache competition in some configurations.

Two-dimensional tree

Configurations that placed the suffix tree in two-dimensional arrays were on average 15% slower than the configurations that used one-dimensional arrays for total computation time (Figure 23). Placing the tree in a texture appears to mitigate some of the negative impact of using a two-dimensional array for the tree, but not using the

2D layout was faster overall. In addition, some configurations using a two-dimensional tree array increased the register footprint in the print kernel across the threshold from 66% to 33% occupancy and had drastically reduced performance (52% worse).

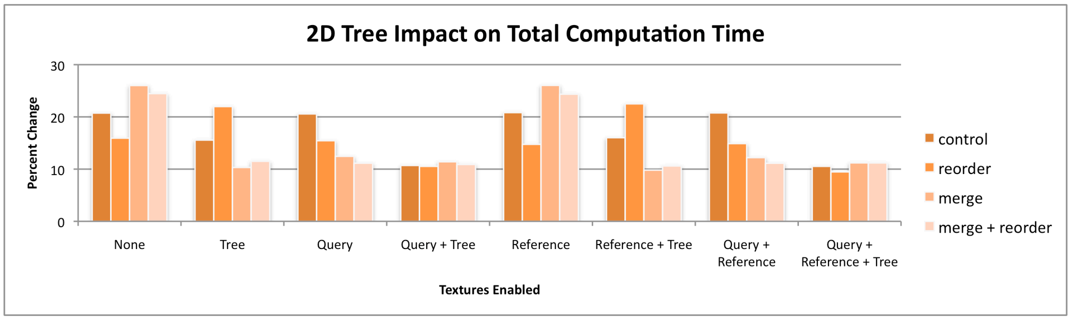


Figure 23. MUMmerGPU 2.0 2D Tree Impact.
Placing the suffix tree in a two-dimensional array universally slows overall computation time.

Tree reordering

In HSILL, configurations that reordered the suffix tree nodes in the GPU memory run significantly and universally faster than the equivalent configurations that do not (between 1% and 11% faster, 5% on average) (Figure 24). This is perhaps the most surprising finding from our benchmark tests, since the reordering is only supposed to improve running time for configurations that use (cached) texture memory for the suffix tree. Furthermore, the node reordering is entirely performed on the CPU, and the GPU kernels are bit-for-bit identical when reordering is enabled over the equivalent configuration with the reordering disabled. However, the actual number of instructions executed by a multiprocessor can vary between invocations based on the access pattern of the kernel. A G80 multiprocessor may serialize execution of threads in a warp if the memory accesses made by threads have significantly different latencies. nVidia offers a profiler that counts events during a kernel execution such as the number of global memory loads and the number of instructions executed. For HSILL, profiling shows a decrease in total instruction count and in the number of instructions due to warp serialization when reordering is enabled.

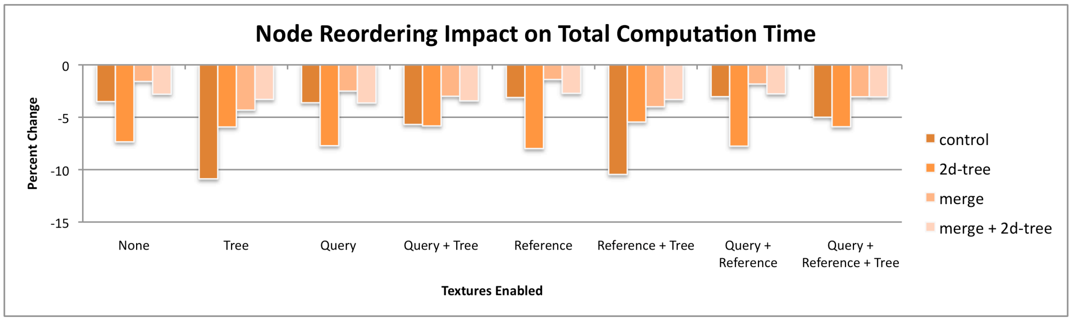


Figure 24. MUMmerGPU 2.0 Node Reordering Impact.
Reordering the tree nodes in GPU memory improves running time not only for configurations that place the tree in cached texture memory, but all configurations.

Merged tree

Merging the two suffix tree arrays into a single array places the two halves of a single tree node adjacent in GPU memory. This policy was originally conceived to exploit a common access pattern in the match kernel where the halves are sequentially accessed. This should have improved cache performance from the increased spatial locality. However, merging the arrays also required slightly more complicated kernel code for addressing nodes. This had a more significant impact by altering the register footprint of both the match and print kernel.

In the print kernel, using a merged array increased running time by 6% on average, including some extreme changes caused by increasing or decreasing the register footprint. (Figure 25). Configurations that placed the suffix tree in a texture or used a two-dimensional array generally suffered when using a merged array. The other configurations saw a reduced footprint, and for a few configurations that reduction boosted the occupancy to 66%. The impact on match kernel time was less dramatic though occupancy differed for some configurations when merged arrays were enabled.

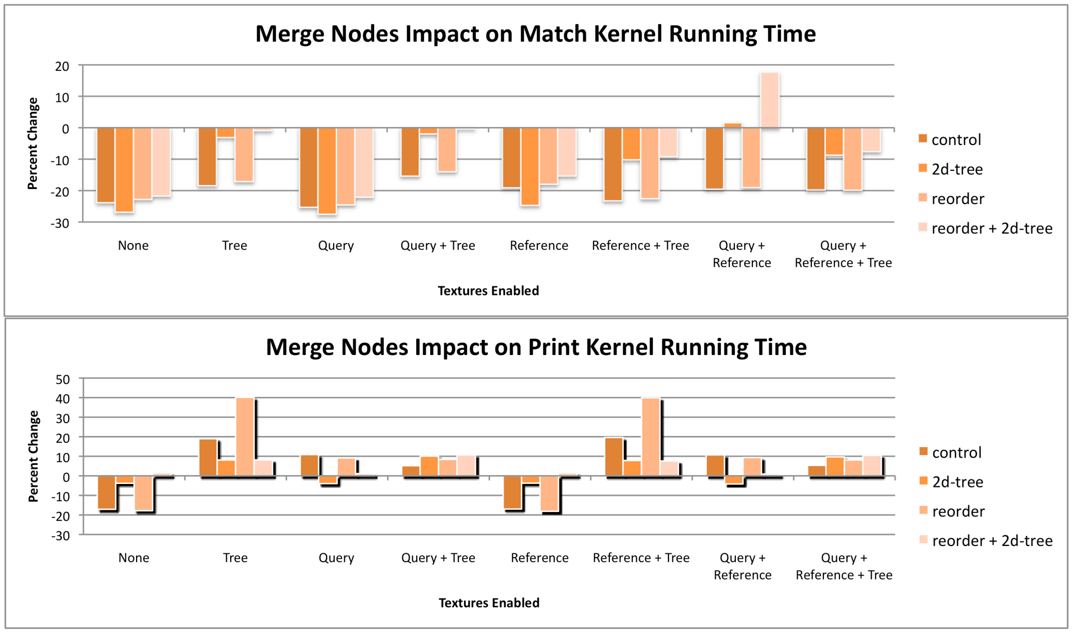
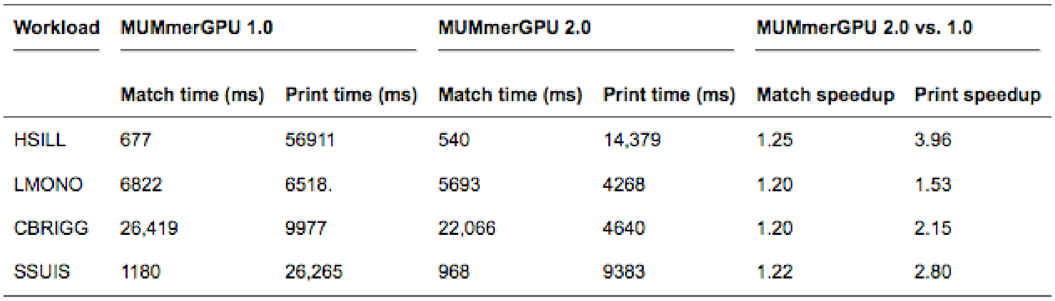


Figure 25. MUMmerGPU 2.0 Merge Node Impact.
(top) The match kernel generally improved in performance with merged nodes enabled. (bottom) The print kernel generally had worse performance using merged nodes. However, a few configurations with merged nodes required only 16 registers, which increasing occupancy to 66% and reduced running time by almost 20%.

Comparison to MUMmerGPU 1.0

Based on the above discussion, the new default policy configuration in MUMmerGPU uses a reordered one-dimensional texture for the suffix tree, global linear memory for the queries and reference, and splits the tree into parallel arrays. This configuration is optimal for HSILL and creates a nearly four-fold speedup in total GPU compute time over MUMmerGPU 1.0. For other workloads, it is not optimal, but this configuration consistently outperforms MUMmerGPU 1.0. For example, in LMONO, CBRIGG and SSUIS, reordering suffix tree nodes generally degrades performance, although a few configurations enjoy a modest speedup. Surprisingly, none of the configurations with increased performance placed the tree in texture memory. In general, the impact of reordering appears very sensitive to the specific access pattern of the kernels for a given input and choice of parameters. However, in all workloads, the new configuration speeds up the match kernel by at least 20%, and the new print kernel is between 1.5x and 4x faster than the CPU based print procedure of MUMmerGPU 1.0 (Table 3). To reach as broad a user base as possible, MUMmerGPU also implements tuned, optimized versions of the matching and print procedures that run on the CPU. For HSILL, MUMmerGPU 2.0's GPU kernels run 13-fold faster than these CPU-based routines.

Table 3. Comparison of MUMmerGPU 1.0 and 2.0 runtimes.



Workload	MUMmerGPU 1.0		MUMmerGPU 2.0		MUMmerGPU 2.0 vs. 1.0	
	Match time (ms)	Print time (ms)	Match time (ms)	Print time (ms)	Match speedup	Print speedup
HSILL	677	56911	540	14,379	1.25	3.96
LMONO	6822	6518.	5693	4268	1.20	1.53
CBRIGG	26,419	9977	22,066	4640	1.20	2.15
SSUIS	1180	26,265	968	9383	1.22	2.80

Discussion

Our exhaustive policy analysis shows occupancy is the single most important factor for the performance of data-intensive applications. This is because higher occupancy allows for more threads to be executed concurrently. Higher occupancy has the added benefit that memory latency can be better hidden with more threads. As such we attempted to improve occupancy of all configurations by reducing their register footprint. In several configurations, we successfully reduced register use to reach 66% occupancy by making small adjustments to the kernel code, such as moving variable declarations to the tightest possible scope, and using bit masks instead of named fields within structs. We also used more aggressive techniques such as using goto's to intentionally disable some compiler optimizations with some success. The CUDA compiler NVCC is actively being developed, and as its register allocation and optimization routines improve, it should be easier to achieve higher occupancy.

Our analysis finds proper use of the texture cache is also critical. Haphazardly placing data in texture memory in the hopes of reducing latency is dangerous. The texture cache is limited to 8KB per multiprocessor, so cache competition can easily hurt overall performance. Furthermore, using textures instead of global memory can reduce occupancy and slow execution. Conversely, proper use of cache that places the most important data in textures can greatly improve performance.

Data reordering can also greatly improve performance. Our results show that it universally improved performance when aligning short reads to a large reference, which we expect to be the most common read mapping use case. For this type of workload, data reordering unexpectedly reduced divergence and warp serialization. Normally used to increase locality and reduce latency, data reordering also is a promising avenue for reducing thread divergence. We plan to explore other reordering strategies in future versions of MUMmerGPU. However, reordering should be used with caution and careful measurement. In MUMmerGPU, it was used to improve cache hit rate in two-dimensional textures, but storing the tree in two-dimensions turned out to be a universally bad choice, despite the claims that the texture cache is optimized for 2D locality. Similarly, merging the node and children halves into the same array was supposed to improve data locality and thus cache performance, but this improvement was lost in most configurations to increased register footprint and reduced occupancy from the more complicated addressing.

These conclusions reflect properties and design decisions concerning the current nVidia graphics processing line, and may fail to hold in the future as the hardware and CUDA evolve. A policy analysis such as the one presented here can help identify high performance policy configurations, and can help "future-proof" an application against rapidly evolving hardware. Ideally, MUMmerGPU would be able to self-tune its policies for an individual system, and we are considering such functionality. In the short term, MUMmerGPU's instrumentation and alternate policy implementations remains in the code, so the application can adapt to new nVidia hardware and CUDA versions as they appear.

Conclusions

MUMmerGPU 2.0 is a significant advance over MUMmerGPU 1.0, featuring improved functionality and higher performance over previous versions of the code. With the new query streaming data model, MUMmerGPU 2.0 can map reads to genomes as large or larger than the human genome. The new GPU-based print kernel post-processes the suffix tree matches into full MEMs, and provides a major performance boost, between 1.5x and 4x, over the serial CPU version in MUMmerGPU 1.0. This kernel required a non-traditional stackless implementation of a depth-first-search of the suffix tree. Its tree-walking technique is applicable to essentially any common tree data structure, and thus we expect many data processing tasks could benefit from running on the GPU. In the future, we plan to extend MUMmerGPU with a second post-processing GPU kernel that computes inexact alignments.

Both the match and print kernels benefited from our exhaustive analysis of seven data organization policies. The impact of individual policies is often surprising and counterintuitive, and we encourage other GPGPU developers to carefully measure their applications when making such decisions. Our analysis shows occupancy is the main determining factor of data-intensive kernel performance. We are optimistic that new versions of the CUDA compiler will simplify reaching high occupancy, but currently, it is imperative that developers monitor their register footprint, and reduce it when possible. The next most important factor is proper use of the textures. The texture cache is very small, and haphazard use of textures will quickly overwhelm it. Instead, applications should only use textures for the most important data. Data reordering can be used to improve locality and cache hit rate, but since different workloads may have different access patterns, developers should select a reordering that is appropriate for the most common workload. Reordering also affects thread divergence, and we recommend that developers consider reordering strategies that reduce divergence, even when not using cached memory.

Data-intensive applications are believed to be less well suited than arithmetic-intensive applications. Nevertheless, our highly data-intensive application MUMmerGPU achieves significant speedup over the serial CPU-based application. A large part of this speedup is due to tuning techniques that may be used in any GPGPU application. The enormous volume of sequencing reads produced by next generation sequencing technologies demands new computational methods. Our software enables individual life science researchers to analyze genetic variations using the supercomputer hidden within their desktop computer.

Chapter 4: Highly Sensitive Read Mapping with MapReduce

Summary of Contribution

This chapter describes the high-throughput sequence alignment program CloudBurst published in the journal Bioinformatics [38]. Similar to MUMmerGPU, CloudBurst aligns large batches of query sequences to a reference genome in parallel. Unlike MUMmerGPU, CloudBurst computes end-to-end alignments of the reads to the reference, allowing for a user specified number of mismatches or differences. In addition, CloudBurst uses a cluster of computers for the alignment using the MapReduce framework to distribute and manage the computation.

CloudBurst uses a seed-and-extend alignment strategy similar to the serial program RMAP [12], except within the MapReduce framework. In the map stage, CloudBurst conceptually constructs an inverted index of k-mers in the reads and k-mers in the reference distributed across all available computers in the cluster. In the reduce function, CloudBurst extends exactly matching k-mer seeds into end-to-end alignments using either scan of the flanking sequences when computing mismatches, or using the Landau-Vishkin k-difference dynamic programming algorithm to also allow for indels. On a cluster with 96 cores, CloudBurst was up to 100 times faster than a serial execution of the program RMAP.

Michael Schatz wrote the software and the manuscript.

Abstract

Next-generation DNA sequencing machines are generating an enormous amount of sequence data, placing unprecedented demands on traditional single-processor read-mapping algorithms. CloudBurst is a new parallel read-mapping algorithm optimized for mapping next-generation sequence data to the human genome and other reference genomes, for use in a variety of biological analyses including SNP discovery, genotyping and personal genomics. It is modeled after the short read-mapping program RMAP, and reports either all alignments or the unambiguous best alignment for each read with any number of mismatches or differences. This level of sensitivity could be prohibitively time consuming, but CloudBurst uses the open-source Hadoop implementation of MapReduce to parallelize execution using multiple compute nodes.

CloudBurst's running time scales linearly with the number of reads mapped, and with near linear speedup as the number of processors increases. In a 24-processor core configuration, CloudBurst is up to 30 times faster than RMAP executing on a single core, while computing an identical set of alignments. Using a larger remote compute cloud with 96 cores, CloudBurst improved performance by >100-fold, reducing the running time from hours to mere minutes for typical jobs involving mapping of millions of short reads to the human genome.

CloudBurst is available open-source as a model for parallelizing algorithms with MapReduce at http://cloudburst-bio.sourceforge.net/.

Introduction

Next-generation high-throughput DNA sequencing technologies from 454 Life Sciences, Illumina, Applied Biosystems and others are changing the scale and scope of genomics. These machines sequence more DNA in a few days than a traditional Sanger sequencing machine could in an entire year, and at a significantly lower cost [41]. James Watson's genome was recently sequenced [8] using technology from 454 Life Sciences in just 2 months, whereas previous efforts to sequence the human genome required several years and hundreds of machines [68]. If this trend continues, an individual will be able to have their DNA sequenced in only a few days and perhaps for as little as $1000.

The data from the new machines consists of millions of short sequences of DNA (25–250 bp) called reads, collected randomly from the target genome. After sequencing, researchers often map the reads to a reference genome to find the locations where each read occurs, allowing for a small number of differences. This information can be used to catalog differences in one person's genome relative to a reference human genome, or compare the genomes of closely related species. For example, this approach was recently used to analyze the genomes of an African [69] and an Asian [9] individual by mapping 4.0 and 3.3 billion 35 bp reads, respectively, to the reference human genome. These comparisons are used for a wide variety of biological analyses including SNP discovery, genotyping, gene expression, comparative genomics and personal genomics. Even a single base pair difference can have a significant biological impact, so researchers require highly sensitive mapping algorithms to analyze the reads. As such, researchers are generating sequence data at an incredible rate and need highly scalable algorithms to analyze their data.

Many of the currently used read-mapping programs, including BLAST [11], SOAP [15], MAQ [7], RMAP [12] and ZOOM [13], use an algorithmic technique called seed-and-extend to accelerate the mapping process. These programs first find sub-strings called seeds that exactly match in both the reads and the reference sequences, and then extend the shared seeds into longer, inexact alignments using a more sensitive algorithm that allows for mismatches or gaps. These programs use a variety of methods for finding and extending the seeds, and have different features and performance. However, each of these programs is designed for execution on a single computing node, and as such requires a long running time or limits the sensitivity of the alignments they find.

CloudBurst is a new highly sensitive parallel seed-and-extend read-mapping algorithm optimized for mapping single-end next generation sequence data to reference genomes. It reports all alignments for each read with up to a user-specified number of differences including both mismatches and indels. CloudBurst can optionally filter the alignments to report the single best non-ambiguous alignment for each read, and produce output identical to RMAPM (RMAP using mismatch scores). As such CloudBurst can replace RMAP in a data analysis pipeline without changing the results, but provides much greater performance by using the open-source implementation of the distributed programming framework MapReduce called Hadoop (http://hadoop.apache.org). The results presented below show that CloudBurst is highly scalable: the running times scale linearly as the number of reads increases, and with near linear speed improvements over a serial execution of RMAP for sensitive searches. Furthermore, CloudBurst can scale to run on large remote compute clouds, and thus map virtually any number of reads with high sensitivity in relatively little time.

MapReduce and Hadoop

MapReduce [36] is the software framework developed and used by GoogleTM to support parallel distributed execution of their data intensive applications. Google uses this framework internally to execute thousands of MapReduce applications per day, processing petabytes of data, all on commodity hardware. Unlike other parallel computing frameworks, which require application developers explicitly manage inter-process communication, computation in MapReduce is divided into two major phases called map and reduce, separated by an internal shuffle phase of the intermediate results (Figure 26), and the framework automatically executes those functions in parallel over any number of processors.

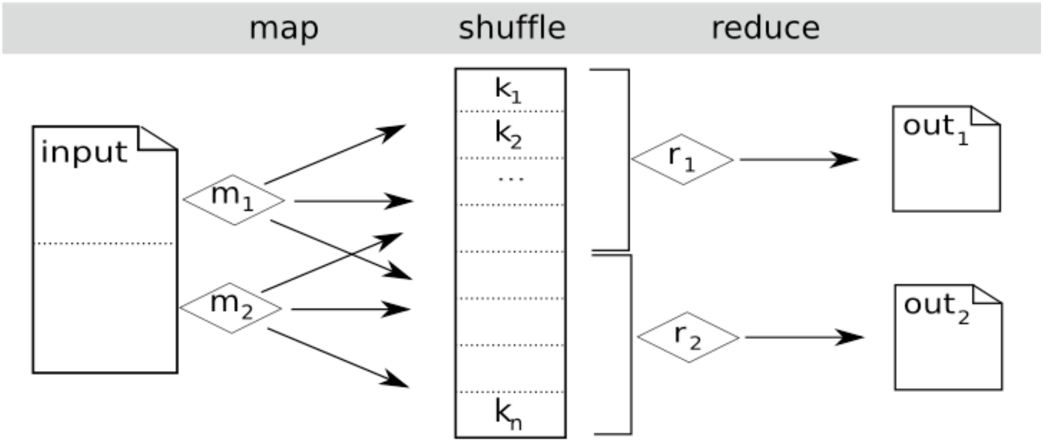


Figure 26. Schematic Overview of MapReduce.
The input file(s) are automatically partitioned into chunks depending on their size and the desired number of mappers. Each mapper (shown here as m1 and m2) executes a user-defined function on a chunk of the input and emits key–value pairs. The shuffle phase creates a list of values associated with each key (shown here as k1, k2 and kn). The reducers (shown here as r1 and r2) evaluate a user-defined function for their subset of the keys and associated list of values, to create the set of output files.

The map function computes key–value pairs from the input data, based on any relationship applicable to the problem, including computing multiple pairs from a single input. For example, the map function of a program that counts the number of occurrences of all length k substrings (k-mers) in a set of DNA sequences could emit the key–value pair (k-mer, 1) for each k-mer. If the input is large, many instances of the map function can execute in parallel on different portions of the input and divide the running time by the number of processors available. Once the mappers are complete, MapReduce shuffles the pairs so all values with the same key are grouped together into a single list. The grouping of key–value pairs effectively creates a large distributed hash table indexed by the key, with a list of values for each key. In the k-mer counter example, the framework creates a list of 1s for each k-mer in the input, corresponding to each instance of that k-mer. The reduce function evaluates a user-defined function on each key–value list. The reduce function can be arbitrarily complex, but must be commutative, since the order of elements in the key–value list is unstable. In the k-mer counting example, the reduce function is called once for each k-mer with its associated list of 1s, and simply adds the 1s together to compute the total number of occurrences for that k-mer. Each instance of the reduce function executes independently, so there can be as many reduce functions executing in parallel as there are distinct keys, i.e. k-mers in the input.

As an optimization, MapReduce allows reduce-like functions called combiners to execute in-memory immediately after the map function. Combiners are not possible in every application because they evaluate on a subset of the values for a given key, but when possible, reduce the amount of data processed in the shuffle and reduce phases. In the k-mer counting example, the combiner emits a partial sum from the subset of 1s it evaluates, and the reduce function sums over the list of partial sums.

Computations in MapReduce are independent, so the wall clock running time should scale linearly with the number of processor cores available, i.e. a 10-core execution should take 1/10th the time of a 1-core execution creating a 10x speedup with complete parallel efficiency. In practice, perfect linear speedup is difficult to achieve because serial overhead limits the maximum speedup possible as described by Amdahl's law [28]. For example, if an application has just 10% non-parallelizable overhead, then the maximum possible end-to-end speedup is only 10x regardless of the number of cores used. High speedup also requires the computation is evenly divided over all processors to maximize the benefit of parallel computation. Otherwise the wall clock running time will be limited to the time for the longest running task, and reduce overall efficiency. MapReduce tries to balance the workload by assigning each reducer ~1/N of the total key space, where N is the number of cores. If certain keys require substantially more time than others, however, it may be necessary to rebalance the workload using a custom partition function or adjusting how keys are emitted.

MapReduce is designed for computations with extremely large datasets, far beyond what can be stored in RAM. Instead it uses files for storing and transferring intermediate results, including the inter-machine communication between map and reduce functions. This could become a severe bottleneck, so Google developed the robust distributed Google File System (GFS) [70] to efficiently support MapReduce. GFS is designed to provide very high-bandwidth for MapReduce by replicating and partitioning files across many physical disks. Files in the GFS are automatically partitioned into large chunks (64 MB by default), which are replicated to several physical disks (three by default) attached to the compute nodes. Therefore, aggregate I/O performance can greatly exceed the performance of an individual memory storage device (e.g. a disk drive), and chunk redundancy ensures reliability even when used with commodity drives with relatively high-failure rates. MapReduce is also ‘data aware’: it attempts to schedule computation at a compute node that has the required data instead of moving the data across the network.

Hadoop and the Hadoop Distributed File System (HDFS) are open source versions of MapReduce and the GFS implemented in Java and sponsored by AmazonTM, YahooTM, Google, IBMTM and other major vendors. Like Google's proprietary MapReduce framework, applications developers need only write custom map and reduce functions, and the Hadoop framework automatically executes those functions in parallel. Hadoop and HDFS are used to manage production clusters with more than 10,000 nodes and petabytes of data, including computation supporting every Yahoo search result. A Hadoop cluster of 910 commodity machines recently set a performance record by sorting 1 TB of data (10 billion 100 bytes records) in 209 s (http://www.hpl.hp.com/hosted/sortbenchmark/).

In addition to in-house Hadoop usage, Hadoop is becoming a de facto standard for cloud computing where compute resources are accessed generically as a service, without regard for physical location or specific configuration. The generic nature of cloud computing allows resources to be purchased on-demand, especially to augment local resources for specific large or time-critical tasks. Several organizations offer cloud compute cycles that can be accessed via Hadoop. Amazon's Elastic Compute Cloud (EC2) (http://aws.amazon.com) contains tens of thousands of virtual machines, and supports Hadoop with minimal effort. In EC2, there are five different classes of virtual machines available providing different levels of CPU, RAM and disk resources with price ranging from $0.10 to $0.80 per hour per virtual machine. Amazon offers preconfigured disk images and launches scripts for initializing a Hadoop cluster, and once initialized, users copy data into the newly created HDFS and execute their jobs as if the cluster was dedicated for their use. For very large datasets, the time required for the initial data transfer can be substantial, and will depend on the bandwidth of the cloud provider. Once transferred into the cloud, though, the cloud nodes generally have very high-internode bandwidth. Furthermore, Amazon has begun mirroring portions of Ensembl and GenBank for use within EC2 without additional storage costs, thereby minimizing the time and cost to run a large-scale analysis of these data.

Read mapping

After sequencing DNA, researchers often map the reads to a reference genome to find the locations where each read occurs. The read-mapping algorithm reports one or more alignments for each read within a scoring threshold, commonly expressed as the minimal acceptable significance of the alignment, or the maximum acceptable number of differences between the read and the reference genome. The algorithms generally allow 1–10% of the read length to differ from the reference, although higher levels may be necessary when aligning to more distantly related genomes, or when aligning longer reads with higher error rates. Read-mapping algorithms can allow mismatch (mutation) errors only, or they can allow insertion or deletion (indel) errors, for both true genetic variations and artificial sequencing errors. The number of mismatches between a pair of sequences can be computed with a simple scan of the sequences, whereas computing the edit distance (allowing for indels) requires a more sophisticated algorithm such as the Smith–Waterman sequence alignment algorithm [71], whose runtime is proportional to the product of the sequence lengths. In either case, the computation for a single pair of short sequences is fast, but becomes costly as the number or size of sequences increases.

When aligning millions of reads generated from a next-generation sequencing machine, read-mapping algorithms often use a technique called seed-and-extend to accelerate the search for highly similar alignments. This technique is based on the observation that there must be a significant exact match for an alignment to be within the scoring threshold. For example, for a 30 bp read to map to a reference with only one difference, there must be at least 15 consecutive bases, called a seed, that match exactly regardless of where the difference occurs. In general, a full-length end-to-end alignment of an m bp read with at most k differences must contain at least one exact alignment of $\lfloor m/(k+1) \rfloor$ consecutive bases [16]. Similar arguments can be made when designing spaced seeds of non-consecutive bases to guarantee finding all alignments with up to a certain numbers of errors [13]. Spaced seeds have the advantage of allowing longer seeds at the same level of sensitivity, although multiple spaced seeds may be needed to reach full sensitivity.

In all seed-and-extend algorithms, regions that do not contain any matching seeds are filtered without further examination, since those regions are guaranteed to not contain any high-quality alignments. For example, BLAST uses a hash table of all fixed length k-mers in the reference to find seeds, and a banded version of the Smith–Waterman algorithm to compute high-scoring gapped alignments. RMAP uses a hash table of non-overlapping k-mers of length $\lfloor m/(k+1) \rfloor$ in the reads to find seeds, while SOAP, MAQ and ZOOM use spaced seeds. In the extension phase, RMAP, MAQ, SOAP and ZOOM align the reads to allow up to a fixed number of mismatches, and SOAP can alternatively allow for one continuous gap. Other approaches to mapping include using suffix trees [14, 17] to quickly find short exact alignments to seed longer inexact alignments, and Bowtie [72] uses the Burrows–Wheeler transform (BWT), to find exact matches coupled with a backtracking algorithm to allow for mismatches. Some BWT-based aligners are reporting extremely fast runtimes, especially in configurations that restrict the sensitivity of the alignments or limit the number of alignments reported per read. For example, in their default high-speed configuration, SOAP2 (http://soap.genomics.org.cn/), BWA (http://maq.sourceforge.net) and Bowtie allow at most two differences in the beginning of the read, and report a single alignment per read selected randomly from the set of acceptable alignments. In more sensitive or verbose configurations, the programs can be considerably slower (http://bowtie-bio.sourceforge.net/manual.shtml).

After computing end-to-end alignments, some of these programs use the edit distance or read quality values to score the mappings. In a systematic study allowing up to 10 mismatches, [12] determined allowing more than two mismatches is necessary for accurately mapping longer reads, and incorporating quality values also improves accuracy. Several of these programs, including RMAPQ (RMAP with quality), MAQ, ZOOM and Bowtie, use quality values in their scoring algorithm, and all are more lenient of errors in the low-quality 3' ends of the reads by trimming the reads or discounting low-quality errors.

Consecutive or spaced seeds dramatically accelerate the computation by focusing computation to regions with potential to have a high-quality alignment. However, to increase sensitivity the length of the seeds must decrease (consecutive seeds) or the number of seeds used must increase (spaced seeds). In either case, increasing sensitivity increases the number of randomly matching seeds and increases the total execution time. Decreasing the seed length can be especially problematic because a seed of length s is expected to occur ~ L/4s times in a reference of length L, and each occurrence must be evaluated using the slower inexact alignment algorithm. Therefore, many of the new short read mappers restrict the maximum number of differences allowed, or limit the number of alignments reported for each read.

Algorithm

CloudBurst is a MapReduce-based read-mapping algorithm modeled after RMAP, but runs in parallel on multiple machines with Hadoop. It is optimized for mapping many short reads from next-generation sequencing machines to a reference genome allowing for a user specified number of mismatches or differences. Like RMAP, it is a seed-and-extend algorithm that indexes the non-overlapping k-mers in the reads as seeds. The seed size $s=\lfloor m/(k+1)\rfloor$ is computed from the minimum length of the reads (m) and the maximum number of differences or mismatches (k). Like RMAP, it attempts to extend the exact seeds to count the number of mismatches in an end-to-end alignment using that seed, and reports alignments with at most k mismatches. Alternatively, like BLAST, it can extend the exact seed matches into end-to-end gapped alignments using a dynamic programming algorithm. For this step, CloudBurst uses a variation of the Landau–Vishkin k-difference alignment algorithm [73], a dynamic programming algorithm for aligning two strings with at most k differences in O(km) time where m is the minimum length of the two strings. See Gusfield's [48] classical text on sequence alignment for more details.

As a MapReduce algorithm, CloudBurst is split into map, shuffle and reduce phases (Figure 27). The map function emits k-mers of length s as seeds from the reads and reference sequences. The shufffle phase groups together k-mers shared between the read and reference sequences. Finally, the reduce function extends the shared seeds into end-to-end alignments allowing both mismatches and indels. The input to the application is a multi-fasta file containing the reads and a multi-fasta file containing one or more reference sequences. These files are first converted to binary Hadoop SequenceFiles and copied into the HDFS. The DNA sequences are stored as the key–value pairs (id, SeqInfo), where SeqInfo is the tuple (sequence, start_offset) and sequence is the sequence of bases starting at the specified offset. By default, the reference sequences are partitioned into chunks of 65 kb overlapping by 1 kb, but the overlap can be increased to support reads longer than 1 kb.

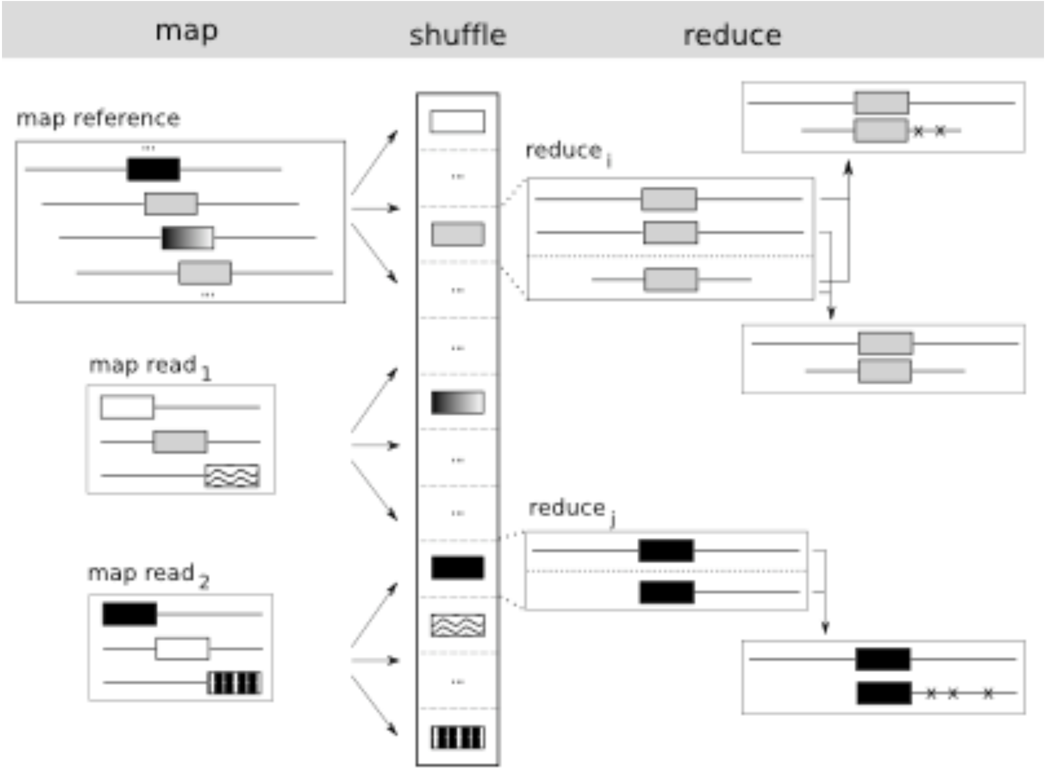


Figure 27. Overview of CloudBurst algorithm.
The map phase emits k-mers as keys for every k-mer in the reference, and for all non-overlapping k-mers in the reads. The shuffle phase groups together the k-mers shared between the reads and the reference. The reduce phase extends the seeds into end-to-end alignments allowing for a fixed number of mismatches or indels. In the figure, 2 grey reference seeds are compared to a single read creating one alignment with 2 errors and 1 alignment with 0 errors, while the black shared seed is extended to an alignment with 3 errors.

Map: extract K-mers

The map function scans the input sequences and emits key–value pairs (seed, MerInfo) where seed is a sequence of length s, and MerInfo is the tuple (id, position, isRef, isRC, left_flank, right_flank). If the input sequence is a reference sequence, then a pair is emitted for every k-mer in the sequence, with isRef = 1, isRC = 0, and position set as the offset of the k-mer in the original sequence. If the given input sequence is a read, then isRef = 0, and a pair is emitted for the non-overlapping k-mers with appropriate position. Seeds are also emitted for the non-overlapping k-mers of the reverse complement sequence with isRC = 1. The flanking sequences [up to (m – s + k) bp) are included in the fields left_flank and right_flank. The seeds are represented with a 2 bit/bp encoding to represent the four DNA characters (ACGT), while the flanking sequences are represented with a 4 bit/bp encoding, which also allows for representing an unknown base (N), and a separator character (.).

CloudBurst parallelizes execution by seed, so each reducer evaluates all potential alignments for approximately 1/N of the 4s seeds, where N is the number of reducers. Overall this balances the workload well, and each reducer is assigned approximately the same number of alignments and runs for approximately the same duration. However, low-complexity seeds (defined as seeds composed of a single DNA character) occur a disproportionate number of times in the read and reference datasets, and the reducers assigned these high-frequency seeds require substantially more execution time than the others. Therefore, CloudBurst can rebalance low-complexity seeds by emitting redundant copies of each occurrence in the reference and randomly assigning occurrences in the reads to one of the redundant copies. For example, if the redundancy is set to 4, each instance of the seed AAAA in the reference will be redundantly emitted as seeds AAAA-0, AAAA-1, AAAA-2 and AAAA-3, and each instance of AAAA from the reads will be randomly assigned to seed AAAA-R with 0≤R≤3. The total number of alignments considered will be the same as if there were no redundant copies, but different subsets of the alignments can be evaluated in parallel in different reducers, and thus improve the overall load balance.

Shuffle: collect shared seeds

Once all mappers have completed, Hadoop shuffles the key–value pairs, and groups all values with the same key into a single list. Since the key is a k-mer from either the read or reference sequences, this has the effect of cataloging seeds that are shared between the reads and the reference.

Reduce: extend seeds

The reduce function extends the exact alignment seeds into longer inexact alignments. For a given seed and MerInfo list, it first partitions the MerInfo tuples into the set R from the reference and set Q from the reads. Then it attempts to extend each pair of tuples from the Cartesian product R x Q using either a scan of the flanking bases to count mismatches, or the Landau–Vishkin k-difference algorithm for gapped alignments. The evaluation proceeds block-wise across subsets of R and Q to maximize cache reuse, and using the bases flanking the shared seeds stored in the MerInfo tuples. If an end-to-end alignment with at most k mismatches or k differences is found, it is then checked to determine if it is a duplicate alignment. This is necessary because multiple exact seeds may be present within the same alignment.

For example, a perfectly matching end-to-end alignment has k +1 exact seeds, and is computed k +1 times. If another exact seed with smaller offset exists in the read the alignment is filtered as a duplicate, otherwise the alignment is recorded. The value for k is small, so only a small number of alignments are discarded.

The output from CloudBurst is a set of binary files containing every alignment of every read with at most k mismatches or differences. These files can be converted into a standard tab-delimited text file of the alignments using the same format as RMAP or post-processed with the bundled tools.

Alignment filtration

In some circumstances, only the unambiguous best alignment for each read is required, rather than the full catalog of all alignments. If so, the alignments can be filtered to report the best alignment for each read, meaning the one with the fewest mismatches or differences. If a read has multiple best alignments, then no alignments are reported exactly as implemented in RMAPM. The filtering is implemented as a second MapReduce algorithm run immediately after the alignments are complete. The map function reemits the end-to-end alignments as key–value pairs with the read identifier as the key and the alignment information as the value. During the shuffle phase, all alignments for a given read are grouped together. The reduce function scans the list of alignments for each read and records the best alignment if an unambiguous best alignment exists. As an optimization, the reducers in the main alignment algorithm report the top two best alignments for each read. Also, the filtration algorithm uses a combiner to filter alignments in memory and reports just the top two best alignments from its subset of alignments for a given read. These optimizations improve performance without changing the results.

Results

CloudBurst was evaluated in a variety of configurations for the task of mapping random subsets of 7.06 million publicly available Illumina/Solexa sequencing reads from the 1000 Genomes Project (accession SRR001113) to portions of the human genome (NCBI Build 36) allowing up to four mismatches. All reads were exactly 36 bp long. The test cluster has 12 compute nodes, each with a 32 bit dual core 3.2 GHz Intel Xeon (24 cores total) and 250 GB of local disk space. The compute nodes were running RedHat Linux AS Release 3 Update 4, and Hadoop 0.15.3 set to execute two tasks per node (24 simultaneous tasks total). In the results below, the time to convert and load the data into the HDFS is excluded, since this time was the same for all tasks, and once loaded the data was reused for multiple analyses.

The first test explored how CloudBurst scales as the number of reads increases and as the sensitivity of the alignment increases. In this test, sub-sets of the reads were mapped to the full human genome (2.87 Gbp), chromosome 1 (247.2 Mbp) or chromosome 22 (49.7 Mbp). To improve load balance across the cores, the number of mappers was set to 240, the number of reducers was set to 48, and the redundancy for low-complexity seeds was set to 16. The redundancy setting was used because the low-complexity seeds required substantially more running time than the other seeds (>1 h compared with <1 min), and the redundancy allows their alignments to be processed in parallel in different reducers. Figure 28 shows the running time of these tasks averaged over three runs, and shows that CloudBurst scales linearly in execution time as the number of reads increases, as expected. Aligning all 7M reads to the full genome with four mismatches failed to complete after reporting ~25 billion mappings due to lack of available disk space. Even allowing zero mismatches created 771M end-to-end perfect matches from the full 7M read set, but most other tools would report just one match per read. Allowing more mismatches increases the runtime superlinearly, because higher sensitivity requires shorter seeds with more chance occurrences. The expected number of occurrences of a seed length s in a sequence of length L is $(L – s + 1)/4^s$, so a random 18 bp sequence (k = 1) is expected to occur ~0.04, ~0.003 and ~0.001 times in the full genome and chromosomes 1 and 22, respectively, while a 7 bp sequence (k =4) is expected to occur >17,500, >15,000 and >3000 times, respectively. Consequently, short seeds have drastically more chance occurrences and correspondingly more running time even though most chance occurrences will fail to extend into end-to-end matches.

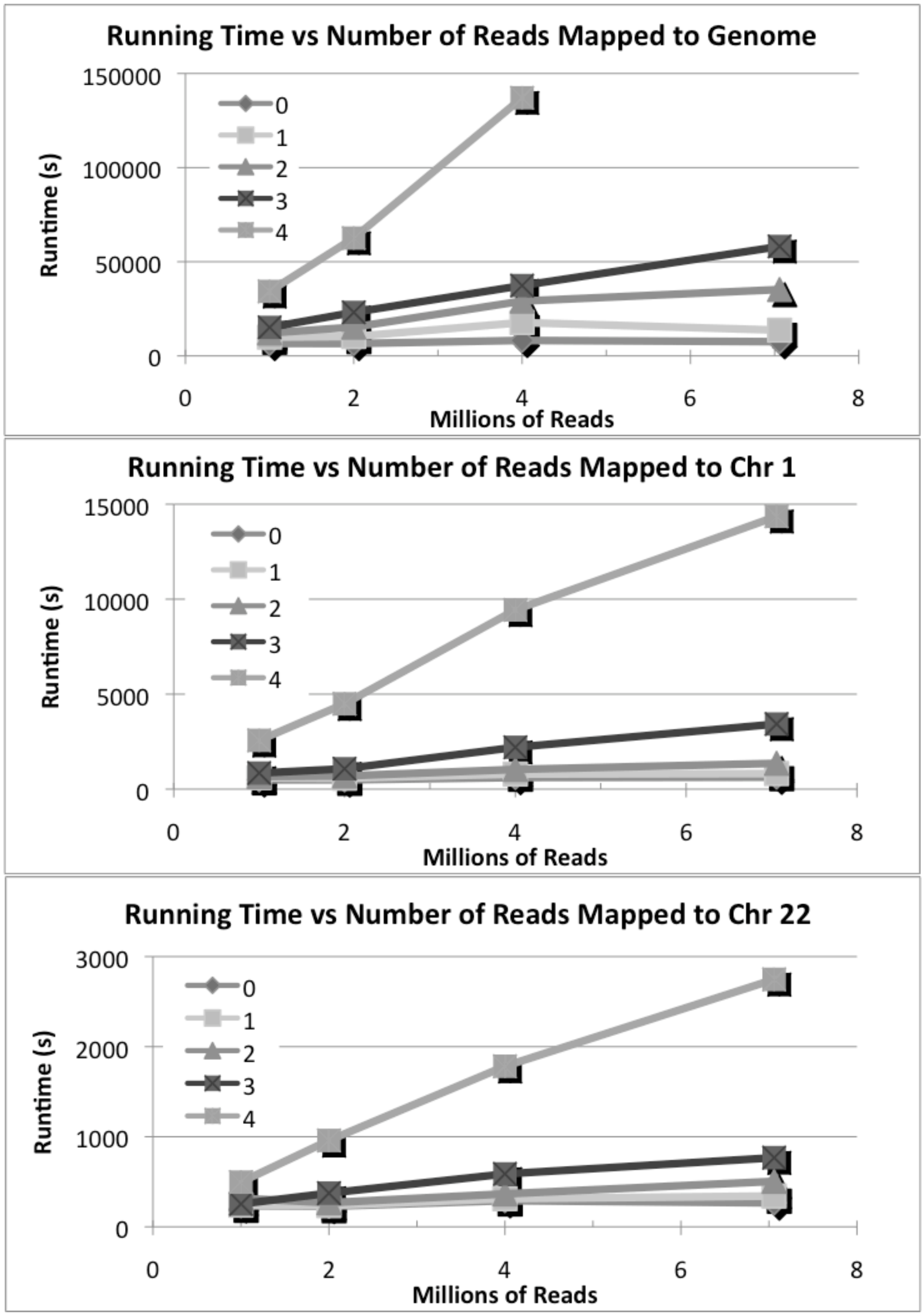


Figure 28. CloudBurst Scaling Performance.
Evaluation of CloudBurst running time while scaling the number of reads and sensitive for mapping to the (A) full human genome; (B) chromosomes 1; and (C) 22 on the local cluster with 24 cores. Tinted lines indicate timings allowing 0 (fastest) through four (slowest) mismatches between a read and the reference. As the number of reads increases, the running time increases linearly. As the number of allowed mismatches increases, the running time increases superlinearly from the exponential increase in seed instances. The four mismatch computation against the full human genome failed to complete due to lack of available disk space after reporting ~25 billion end-to-end alignments.

The second test compared the performance CloudBurst on 24 processor cores with a serial execution of RMAPM (version 0.41) on 1 core with the full read set to chromosomes 1 and 22. RMAP requires a 64 bit operating system, so it was run on 1 core of a 64 bit dual core 2.4 GHz AMD Opteron 250 with 8 GB of RAM running RedHat Enterprise Linux AS Release 3 Update 9. CloudBurst was configured as before, except with the alignment filtration option enabled so only a single alignment was reported for each read identical to those reported by RMAPM. Figure 29 shows the results of the test, and plots the speedup of CloudBurst over RMAP for the different levels of sensitivity. The expected speedup is 24, since CloudBurst runs in parallel on 24 cores, but CloudBurst's speedup over RMAP varies between 2x and 33x depending on the level of sensitivity and reference sequence. At low sensitivity (especially k =0), the overhead of shuffling and distributing the data over the network overwhelms the parallel computation compared with the in-memory lookup and evaluation in RMAP. As the sensitivity increases, the overhead becomes proportionally less until the time spent evaluating alignments in the reduce phase dominates the running time. The speedup beyond 24x for high-sensitivity mapping is due to implementation differences between RMAP and CloudBurst, and the additional compute resources available in the parallel environment (cache, disk IO, RAM, etc.). The speedup when mapping to the full genome did not improve as the level of sensitivity increased because of the increased overhead from the increased data size. This effect can be minimized by aligning more reads to the genome in a single batch, and thus better amortize the time spent emitting and shuffling all of the k-mers in the genome.

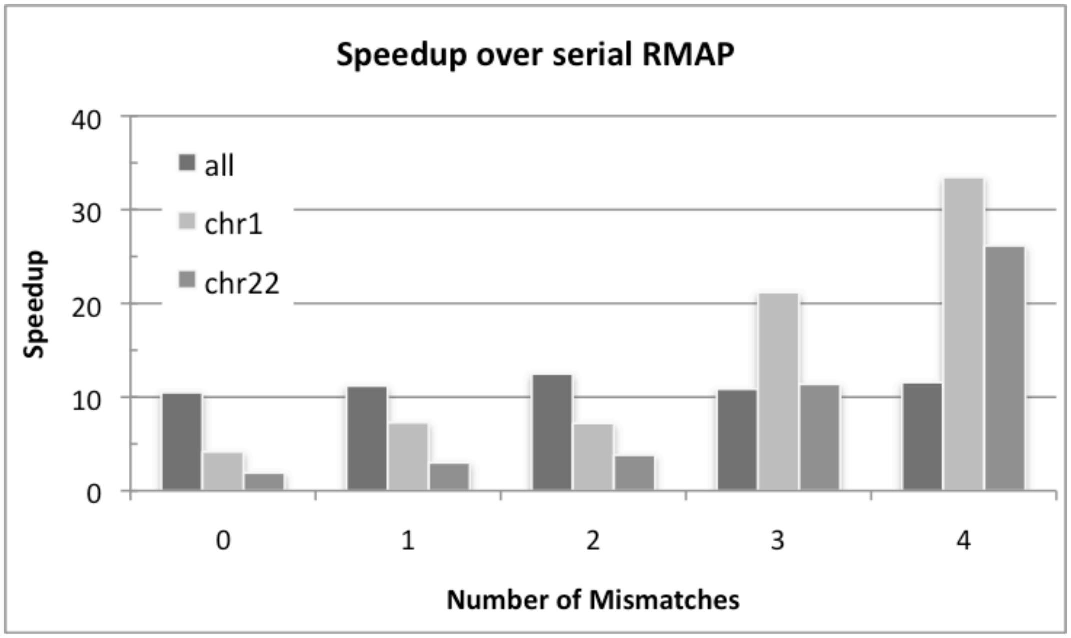


Figure 29. CloudBurst Speedup over RMAP.
CloudBurst running time compared with RMAP for 7M reads, showing the speedup of CloudBurst running on 24 cores compared with RMAP running on 1 core. As the number of allowed mismatches increases, the relative overhead decreases allowing CloudBurst to meet and exceed 24x linear speedup.

The next experiment compared CloudBurst with an ad hoc parallelization scheme for RMAP, in which the reads are split into multiple files, and then RMAP is executed on each file. In the experiment, the full read set was split into 24 files, each containing 294k reads, and each file was separately mapped to chromosome 22. The runtimes were just for executing RMAP, and do not consider any overhead of partitioning the files, remotely launching the program, or monitoring the progress, and thus the expected speedup should be a perfect 24x. However, the runtimes of the different files varied considerably depending on which reads were present, and the corresponding speedup is computed based on the runtime for the longest running file: between 18 and 41 s with a 12x speedup for zero mismatches, 26–67 s with a 14x speedup for one mismatch, 34–98 s with a 16x speedup for two mismatches, 132–290 s with a 21x speedup for three mismatches and 1379–1770 s with a 29x speedup for four mismatches. The superlinear speedup for four mismatches was because the total computation time after splitting the read set was less than the time for the full batch at once, presumably because of better cache performance for RMAP with fewer reads. This experiment shows the ad hoc scheme works well with speedups similar to CloudBurst, but fails to reach perfect linear speedup in most cases because it makes no special considerations for load balance. In addition, an ad hoc parallelization scheme is more fragile as it would not benefit from the inherent advantages of Hadoop: data-aware scheduling, monitoring and restart and the high-performance file system.

Amazon Cloud Results

CloudBurst was next evaluated on the Amazon EC2. This environment provides unique opportunities for evaluating CloudBurst, because the performance and size of the cluster are configurable. The first test compared two different EC2 virtual machine classes with the local dedicated 24-core Hadoop cluster described above. In all three cases, the number of cores available was held constant at 24, and the task was mapping all 7M reads to human chromosome 22 with up to four mismatches, with runtimes averaged over three runs. The first configuration had 24 'Small Instance' slaves running Hadoop 0.17.0, priced at $0.10 per hour per instance and provides one virtual core with approximately the performance of a 1.0–1.2 GHz 2007 Xeon processor. The second configuration had 12 'High-CPU Medium Instance' slaves, also running Hadoop 0.17.0 and priced at $0.20 per hour per instance, but offers two virtual cores per machine and have been benchmarked to have a total performance approximately five times the small instance type. The running time for the ‘High-CPU Medium Instance’ class was 1667 s, and was substantially better per dollar than the ‘Small Instance’ class at 3805 s, and even exceeds the performance of the local dedicated cluster at 1921 s.

The final experiment evaluated CloudBurst as the size of the cluster increases for a fixed problem. In this experiment, the number of ‘High-CPU Medium Instance’ cores varied between 24, 48, 72 and 96 virtual cores for the task of mapping all 7M reads to human chromosome 22. Figure 5 shows the running time with these clusters averaged over three runs. The results show CloudBurst scales very well as the number of cores increases: the 96-core cluster was 3.5 times faster than the 24-core cluster and reduced the running time of the serial RMAP execution from >14 h to ~8 min (>100x speedup). The main limiting factor towards reaching perfect speedups in the large clusters was that the load imbalance caused a minority of the reducers running longer than the others. This effect was partially solved by reconfiguring the parallelization settings: the number of reducers was increased to 60 and the redundancy of the low-complexity seeds was increased to 24 for the 48-core evaluation, 144 and 72 for the 72-core evaluation and 196 and 72 for the 96-core evaluation. With these settings, the computation had better balance across the virtual machines and decreased the wall clock time of the execution.

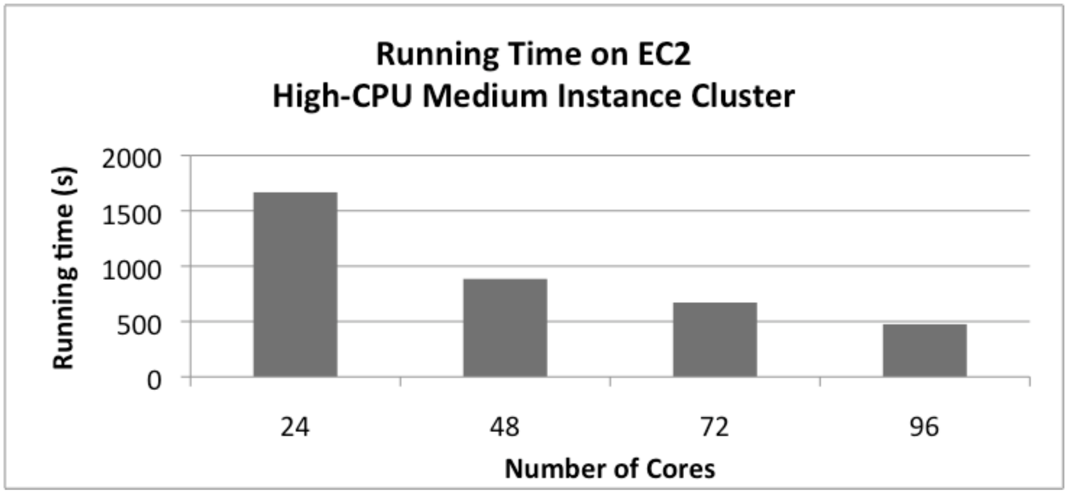


Figure 30. CloudBurst Scaling on EC2.
Comparison of CloudBurst running time (in seconds) while scaling size of the cluster for mapping 7M reads to human chromosome 22 with at most four mismatches on the EC2 Cluster. The 96-core cluster is 3.5x faster than the 24-core cluster.

<u>*Discussion*</u>

CloudBurst is a new parallel read-mapping algorithm optimized for next-generation sequence data. It uses seed-and-extend alignment techniques modeled after RMAP to efficiently map reads with any number of mismatches or differences. It uses the Hadoop implementation of MapReduce to efficiently execute in parallel on multiple compute nodes, thus making it feasible to perform highly sensitive alignments on large read sets. The results described here show CloudBurst scales linearly as the number of reads increases, and with near linear parallel speedup as the size of the cluster increases. This high level of performance enables computation of extremely large numbers of highly sensitive alignments in dramatically reduced time, and is complementary to new BWT-based aligners that excel at quickly reporting a small number of alignments per read.

CloudBurst's superior performance is made possible by the efficiency and power of Hadoop. This framework makes it straightforward to create highly scalable applications with many aspects of parallel computing automatically provided. Hadoop's ability to deliver high performance, even in the face of extremely large datasets, is a perfect match for many problems in computational biology. Seed-and-extend style algorithms, in particular, are a natural fit for MapReduce, and any of the hash-table based seed-and-extend alignment algorithms including BLAST, SOAP, MAQ or ZOOM could be implemented with MapReduce. Future work for CloudBurst is to incorporate quality values in the mapping and scoring algorithms and to enhance support for paired reads. We are also exploring the possibility of integrating CloudBurst into RNA-seq analysis pipeline, which can also model gene splice sites. Algorithms that do not use a hash table, such as the BWT based short-read aligners, can also use Hadoop to parallelize execution and the HDFS.

Implementing algorithms to run in parallel with Hadoop has many advantages, including scalability, redundancy, automatic monitoring and restart and high-performance distributed file access. In addition, no single machine needs to have the entire index in memory, and the computation requires only a single scan of the reference and query files. Consequently, Hadoop based implementations of other algorithms in computational biology might offer similar high levels of performance. These massively parallel applications, running on large compute clouds with thousands of nodes, will drastically change the scale and scope of computational biology, and allow researchers to cheaply perform analyses that are otherwise impossible.

Chapter 5: Searching for SNPs with Cloud Computing

Summary of Contribution

This chapter describes the pipeline Crossbow published in Genome Biology [74] in collaboration with Ben Langmead, Jimmy Lin, Mihai Pop and Steven Salzberg at the University of Maryland. Crossbow is a pipeline for rapid and large scale genotyping, including genotyping entire human genomes from short reads.

Similar to CloudBurst, Crossbow uses the MapReduce framework to distribute and accelerate computation across a cluster of computers. Crossbow is composed of 3 major stages of computation. The first stage of Crossbow executes the ultrafast short read alignment program Bowtie [72] as the MapReduce map function to align batches of reads to the reference genome. The second stage is the MapReduce shuffle phase groups and sorts the alignments so that all alignments within the same chromosome region are collected and sorted on the same computer. The final stage reuses the tool SOAPsnp [75] as the MapReduce reduce function to scan the multiple alignment of reads to find significant differences between the reference genome and the reads. On a cluster with 320 cores, Crossbow was able to genotype 38 fold coverage of the human genome in just three hours with over 99% concordance with independently generated genotype information.

Michael Schatz implemented the Crossbow pipeline for a local cluster, and executed the whole genome genotyping experiments on the local cluster. Ben Langmead designed the overall pipeline, implemented the Crossbow pipeline on the Amazon cluster, and executed the experiments at Amazon. Jimmy Lin contributed to the algorithm design. Ben Langmead wrote the initial draft of the manuscript, and Michael Schatz made extensive edits to the manuscript. Jimmy Lin, Mihai Pop and Steven Salzberg edited the manuscript and provided guidance on the project.

Abstract

As DNA sequencing outpaces improvements in computer speed, there is a critical need to accelerate tasks like alignment and SNP calling. Crossbow is a cloud-computing software tool that combines the aligner Bowtie and the SNP caller SOAPsnp. Executing in parallel using Hadoop, Crossbow analyzes data comprising 38-fold coverage of the human genome in three hours using a 320-CPU cluster rented from a cloud computing service for about $85. Crossbow is available from http://bowtie-bio.sourceforge.net/crossbow/.

Rationale

Improvements in DNA sequencing have made sequencing an increasingly valuable tool for the study of human variation and disease. Technologies from Illumina (San Diego, CA, USA), Applied Biosystems (Foster City, CA, USA) and 454 Life Sciences (Branford, CT, USA) have been used to detect genomic variations among humans [7, 62, 71-73], to profile methylation patterns [76], to map DNA-protein interactions [77], and to identify differentially expressed genes and novel splice junctions [78, 79]. Meanwhile, technical improvements have greatly decreased the cost and increased the size of sequencing datasets. For example, at the beginning of 2009 a single Illumina instrument was capable of generating 15 to 20 billion bases of sequencing data per run. Illumina has projected (http://investor.illumina.com/) that its instrument will generate 90 to 95 billion bases per run by the end of 2009, quintupling its throughput in one year. Another study shows the per-subject cost for whole-human resequencing declining rapidly over the past year [1], which will fuel further adoption. Growth in throughput and adoption are vastly outpacing improvements in computer speed, demanding a level of computational power achievable only via large-scale parallelization.

Two recent projects have leveraged parallelism for whole-genome assembly with short reads. Simpson et al. [80] use ABySS to assemble the genome of a human from 42-fold coverage of short reads [69] using a cluster of 168 cores (21 computers), in about 3 days of wall clock time. Jackson and colleagues [81] assembled a Drosophila melanogaster genome from simulated short reads on a 512-node BlueGene/L supercomputer in less than 4 hours of total elapsed time. Though these efforts demonstrate the promise of parallelization, they are not widely applicable because they require access to a specific type of hardware resource. No two clusters are exactly alike, so scripts and software designed to run well on one cluster may run poorly or fail entirely on another cluster. Software written for large supercomputers like BlueGene/L is less reusable still, since only select researchers have access to such machines. Lack of reusability also makes it difficult for peers to recreate scientific results obtained using such systems.

An increasingly popular alternative for large-scale computations is cloud computing. Instead of owning and maintaining dedicated hardware, cloud computing offers a 'utility computing' model, that is, the ability to rent and perform computation on standard, commodity computer hardware over the Internet. These rented computers run in a virtualized environment where the user is free to customize the operating system and software installed. Cloud computing also offers a parallel computing framework called MapReduce [82], which was designed by Google to efficiently scale computation to many hundreds or thousands of commodity computers. Hadoop (http://hadoop.apache.org) is an open source implementation of MapReduce that is widely used to process very large datasets, including at companies such as Google, Yahoo, Microsoft, IBM, and Amazon. Hadoop programs can run on any cluster where the portable, Java-based Hadoop framework is installed. This may be a local or institutional cluster to which the user has free access, or it may be a cluster rented over the Internet through a utility computing service. In addition to high scalability, the use of both standard software (Hadoop) and standard hardware (utility computing) affords reusability and reproducibility.

The CloudBurst project [38] explored the benefits of using Hadoop as a platform for alignment of short reads. CloudBurst is capable of reporting all alignments for millions of human short reads in minutes, but does not scale well to human resequencing applications involving billions of reads. Whereas CloudBurst aligns about 1 million short reads per minute on a 24-core cluster, a typical human resequencing project generates billions of reads, requiring more than 100 days of cluster time or a much larger cluster. Also, whereas CloudBurst is designed to efficiently discover all valid alignments per read, resequencing applications often ignore or discount evidence from repetitively aligned reads as they tend to confound genotyping. Our goal for this work was to explore whether cloud computing could be profitably applied to the largest problems in comparative genomics. We focus on human resequencing, and single nucleotide polymorphism (SNP) detection specifically, in order to allow comparisons to previous studies.

We present Crossbow, a Hadoop-based software tool that combines the speed of the short read aligner Bowtie [72] with the accuracy of the SNP caller SOAPsnp [75] to perform alignment and SNP detection for multiple whole-human datasets per day. In our experiments, Crossbow aligns and calls SNPs from 38-fold coverage of a Han Chinese male genome [9] in as little as 3 hours (4 hours 30 minutes including transfer time) using a 320-core cluster. SOAPsnp was previously shown to make SNP calls that agree closely with genotyping results obtained with an Illumina 1 M BeadChip assay of the Han Chinese genome [75] when used in conjunction with the short read aligner SOAP [15]. We show that SNPs reported by Crossbow exhibit a level of BeadChip agreement comparable to that achieved in the original SOAPsnp study, but in far less time.

Crossbow is open source software available from the Bowtie website (http://bowtie-bio.sf.net/crossbow). Crossbow can be run on any cluster with appropriate versions of Hadoop, Bowtie, and SOAPsnp installed. Crossbow is distributed with scripts allowing it to run either on a local cluster or on a cluster rented through Amazon's Elastic Compute Cloud (EC2) (http://aws.amazon.com) utility computing service. Version 0.1.3 of the Crossbow software is also provided as Additional data file 1.

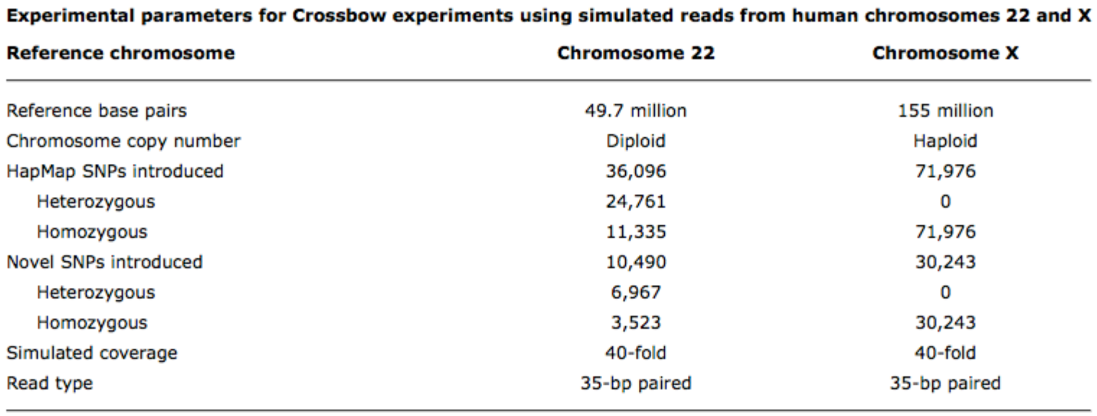
Results

Crossbow harnesses cloud computing to efficiently and accurately align billions of reads and call SNPs in hours, including for high-coverage whole-human datasets. Within Crossbow, alignment and SNP calling are performed by Bowtie and SOAPsnp, respectively, in a seamless, automatic pipeline. Crossbow can be run on any computer cluster with the prerequisite software installed. The Crossbow package includes scripts that allow the user to run an entire Crossbow session remotely on an Amazon EC2 cluster of any size.

Resequencing simulated data

To measure Crossbow's accuracy where true SNPs are known, we conducted two experiments using simulated paired-end read data from human chromosomes 22 and X. Results are shown in Table 4 and Table 5. For both experiments, 40-fold coverage of 35-bp paired-end reads were simulated from the human reference sequence (National Center for Biotechnology Information (NCBI) 36.3). Quality values and insert lengths were simulated based on empirically observed qualities and inserts in the Wang et al. dataset [9].

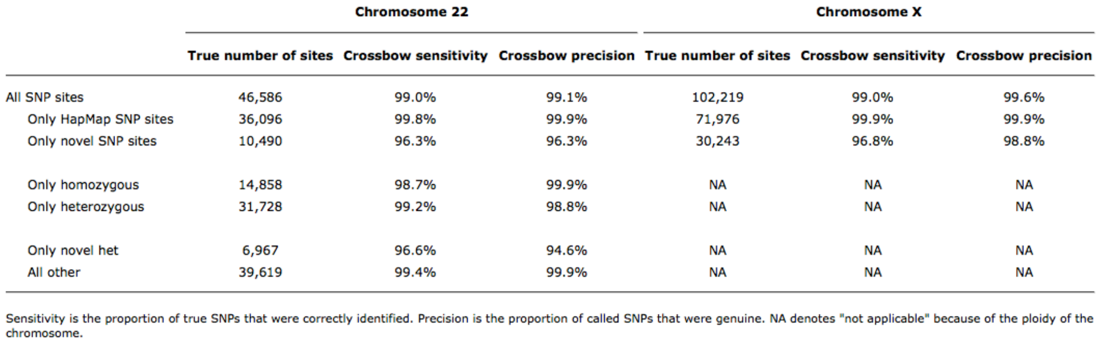
Table 4. Experimental Parameters for Crossbow experiments using simulated reads from human chromosomes 22 and X.



Experimental parameters for Crossbow experiments using simulated reads from human chromosomes 22 and X

Reference chromosome	Chromosome 22	Chromosome X
Reference base pairs	49.7 million	155 million
Chromosome copy number	Diploid	Haploid
HapMap SNPs introduced	36,096	71,976
Heterozygous	24,761	0
Homozygous	11,335	71,976
Novel SNPs introduced	10,490	30,243
Heterozygous	6,967	0
Homozygous	3,523	30,243
Simulated coverage	40-fold	40-fold
Read type	35-bp paired	35-bp paired

Table 5. SNP calling measurements for Crossbow experiments using simulated reads from human chromosomes 22 and X.



	Chromosome 22			Chromosome X		
	True number of sites	Crossbow sensitivity	Crossbow precision	True number of sites	Crossbow sensitivity	Crossbow precision
All SNP sites	46,586	99.0%	99.1%	102,219	99.0%	99.6%
Only HapMap SNP sites	36,096	99.8%	99.9%	71,976	99.9%	99.9%
Only novel SNP sites	10,490	96.3%	96.3%	30,243	96.8%	98.8%
Only homozygous	14,858	98.7%	99.9%	NA	NA	NA
Only heterozygous	31,728	99.2%	98.8%	NA	NA	NA
Only novel het	6,967	96.6%	94.6%	NA	NA	NA
All other	39,619	99.4%	99.9%	NA	NA	NA

Sensitivity is the proportion of true SNPs that were correctly identified. Precision is the proportion of called SNPs that were genuine. NA denotes "not applicable" because of the ploidy of the chromosome.

SOAPsnp can exploit user-supplied information about known SNP loci and allele frequencies to refine its prior probabilities and improve accuracy. Therefore, the read simulator was designed to simulate both known HapMap [83] SNPs and novel SNPs. This mimics resequencing experiments where many SNPs are known but some are novel. Known SNPs were selected at random from actual HapMap alleles for human chromosomes 22 and X. Positions and allele frequencies for known SNPs were calculated according to the same HapMap SNP data used to simulate SNPs.

For these simulated data, Crossbow agrees substantially with the true calls, with greater than 99% precision and sensitivity overall for chromosome 22. Performance for HapMap SNPs is noticeably better than for novel SNPs, owing to SOAPsnp's ability to adjust SNP-calling priors according to known allele frequencies. Performance is similar for homozygous and heterozygous SNPs overall, but novel heterozygous SNPs yielded the worst performance of any other subset studied, with 96.6% sensitivity and 94.6% specificity on chromosome 22. This is as expected, since novel SNPs do not benefit from prior knowledge, and heterozygous SNPs are more difficult than homozygous SNPs to distinguish from the background of sequencing errors.

Whole-human resequencing

To demonstrate performance on real-world data, we used Crossbow to align and call SNPs from the set of 2.7 billion reads and paired-end reads sequenced from a Han Chinese male by Wang et al [9]. Previous work demonstrated that SNPs called from this dataset by a combination of SOAP and SOAPsnp are highly concordant with genotypes called by an Illumina 1 M BeadChip genotyping assay of the same individual [75]. Since Crossbow uses SOAPsnp as its SNP caller, we expected Crossbow to yield very similar, but not identical, output. Differences may occur because: Crossbow uses Bowtie whereas the previous study used SOAP to align the reads; the Crossbow version of SOAPsnp has been modified somewhat to operate within a MapReduce context; in this study, alignments are binned into non-overlapping 2-Mbp partitions rather than into chromosomes prior to being given to

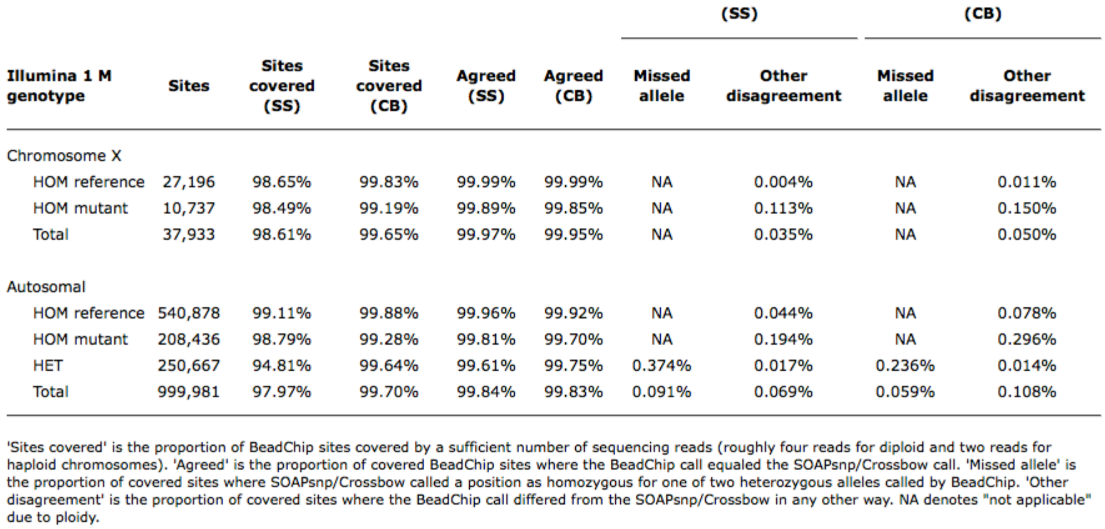
SOAPsnp; and the SOAPsnp study used additional filters to remove some additional low confidence SNPs. Despite these differences, Crossbow achieves comparable agreement with the BeadChip assay and at a greatly accelerated rate.

We downloaded 2.66 billion reads from a mirror of the YanHuang site (http://yh.genomics.org.cn/). These reads cover the assembled human genome sequence to 38-fold coverage. They consist of 2.02 billion unpaired reads with sizes ranging from 25 to 44 bp, and 658 million paired-end reads. The most common unpaired read lengths are 35 and 40 bp, comprising 73.0% and 17.4% of unpaired reads, respectively. The most common paired-end read length is 35 bp, comprising 88.8% of all paired-end reads. The distribution of paired-end separation distances is bimodal with peaks in the 120 to 150 bp and 420 to 460 bp ranges.

Table 6 shows a comparison of SNPs called by either of the sequencing-based assays - Crossbow labeled 'CB' and SOAP+SOAPsnp labeled 'SS' - against SNPs obtained with the Illumina 1 M BeadChip assay from the SOAPsnp study [75]. The 'sites covered' column reports the proportion of BeadChip sites covered by a sufficient number of sequencing reads. Sufficient coverage is roughly four reads for diploid chromosomes and two reads for haploid chromosomes (see Materials and methods for more details about how sufficient coverage is determined). The 'Agreed' column shows the proportion of covered BeadChip sites where the BeadChip call equaled the SOAPsnp or Crossbow call. The 'Missed allele' column shows the proportion of covered sites where SOAPsnp or Crossbow called a position as homozygous for one of two heterozygous alleles called by BeadChip at that position. The 'Other disagreement' column shows the proportion of covered sites where the

BeadChip call differed from the SOAPsnp/Crossbow in any other way. Definitions of the 'Missed allele' and 'Other disagreement' columns correspond to the definitions of 'false negatives' and 'false positives', respectively, in the SOAPsnp study.

Table 6. Coverage and agreement measurements comparing Crossbow (CB) and SOAP/SOAPsnp (SS) to the genotyping results obtained by an Illumina 1 M genotyping assay in the SOAPsnp study.



						(SS)		(CB)	
Illumina 1 M genotype	**Sites**	**Sites covered (SS)**	**Sites covered (CB)**	**Agreed (SS)**	**Agreed (CB)**	**Missed allele**	**Other disagreement**	**Missed allele**	**Other disagreement**
Chromosome X									
HOM reference	27,196	98.65%	99.83%	99.99%	99.99%	NA	0.004%	NA	0.011%
HOM mutant	10,737	98.49%	99.19%	99.89%	99.85%	NA	0.113%	NA	0.150%
Total	37,933	98.61%	99.65%	99.97%	99.95%	NA	0.035%	NA	0.050%
Autosomal									
HOM reference	540,878	99.11%	99.88%	99.96%	99.92%	NA	0.044%	NA	0.078%
HOM mutant	208,436	98.79%	99.28%	99.81%	99.70%	NA	0.194%	NA	0.296%
HET	250,667	94.81%	99.64%	99.61%	99.75%	0.374%	0.017%	0.236%	0.014%
Total	999,981	97.97%	99.70%	99.84%	99.83%	0.091%	0.069%	0.059%	0.108%

'Sites covered' is the proportion of BeadChip sites covered by a sufficient number of sequencing reads (roughly four reads for diploid and two reads for haploid chromosomes). 'Agreed' is the proportion of covered BeadChip sites where the BeadChip call equaled the SOAPsnp/Crossbow call. 'Missed allele' is the proportion of covered sites where SOAPsnp/Crossbow called a position as homozygous for one of two heterozygous alleles called by BeadChip. 'Other disagreement' is the proportion of covered sites where the BeadChip call differed from the SOAPsnp/Crossbow in any other way. NA denotes "not applicable" due to ploidy.

Both Crossbow and SOAP+SOAPsnp exhibit a very high level of agreement with the BeadChip genotype calls. The small differences in number of covered sites (<2% higher for Crossbow) and in percentage agreement (<0.1% lower for Crossbow) are likely due to the SOAPsnp study's use of additional filters to remove some SNPs prior to the agreement calculation, and to differences in alignment policies between SOAP and Bowtie. After filtering, Crossbow reports a total of 3,738,786 SNPs across all autosomal chromosomes and chromosome X, whereas the SNP GFF file available from the YanHaung site (http://yh.genomics.org.cn/) reports a total of 3,072,564 SNPs across those chromosomes. This difference is also likely due to the SOAPsnp study's more stringent filtering.

Cloud performance

The above results were computed on a Hadoop 0.20 cluster with 10 worker nodes located in our laboratory, where it required about 1 day of wall clock time to run. Each node is a four-core 3.2 GHz Intel Xeon (40 cores total) running 64-bit Redhat Enterprise Linux Server 5.3 with 4 GB of physical memory and 366 GB of local storage available for the Hadoop Distributed Filesystem (HDFS) and connected via gigabit ethernet. We also performed this computation using Amazon's EC2 service on clusters of 10, 20 and 40 nodes (80, 160, and 320 cores) running Hadoop 0.20. In each case, the Crossbow pipeline was executed end-to-end using scripts distributed with the Crossbow package. In the 10-, 20- and 40-node experiments, each individual node was an EC2 Extra Large High CPU Instance, that is, a virtualized 64-bit computer with 7 GB of memory and the equivalent of 8 processor cores clocked at approximately 2.5 to 2.8 Ghz. At the time of this writing, the cost of such nodes was $0.68 ($0.76 in Europe) per node per hour.

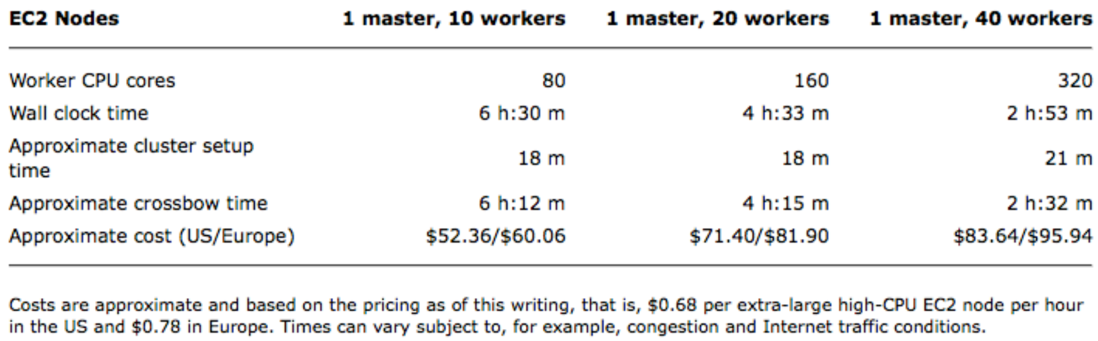
Before running Crossbow, the short read data must be stored on a filesystem the Hadoop cluster can access. When the Hadoop cluster is rented from Amazon's EC2 service, users will typically upload input data to Amazon's Simple Storage Service (S3), a service for storing large datasets over the Internet. For small datasets, data transfers typically complete very quickly, but for large datasets (for example, more than 100 GB of compressed short read data), transfer time can be significant.

An efficient method to copy large datasets to S3 is to first allocate an EC2 cluster of many nodes and have each node transfer a subset of the data from the source to S3 in parallel. Crossbow is distributed with a Hadoop program and driver scripts for performing these bulk parallel copies while also preprocessing the reads into the form required by Crossbow. We used this software to copy 103 gigabytes of compressed short read data from a public FTP server located at the European Bioinformatics Institute in the UK to an S3 repository located in the US in about 1 hour 15 minutes (approximately 187 Mb/s effective transfer rate). The transfer cost approximately $28: about $3.50 ($3.80 in Europe) in cluster rental fees and about $24 ($24 in Europe) in data transfer fees.

Transfer time depends heavily on both the size of the data and the speed of the Internet uplink at the source. Public archives like NCBI and the European Bioinformatics Institute (EBI) have very high-bandwidth uplinks to the >10 Gb/s JANET and Internet2 network backbones, as do many academic institutions. However, even at these institutions, the bandwidth available for a given server or workstation can be considerably less (commonly 100 Mb/s or less). Delays due to slow uplinks can be mitigated by transferring large datasets in stages as reads are generated by the sequencer, rather than all at once.

To measure how the whole-genome Crossbow computation scales, separate experiments were performed using 10, 20 and 40 EC2 Extra Large High CPU nodes. Table 7 presents the wall clock running time and approximate cost for each experiment. The experiment was performed once for each cluster size. The results show that Crossbow is capable of calling SNPs from 38-fold coverage of the human genome in under 3 hours of wall clock time and for about $85 ($96 in Europe).

Table 7. Timing and cost for Crossbow experiments using reads from the Wang et al. study.



EC2 Nodes	1 master, 10 workers	1 master, 20 workers	1 master, 40 workers
Worker CPU cores	80	160	320
Wall clock time	6 h:30 m	4 h:33 m	2 h:53 m
Approximate cluster setup time	18 m	18 m	21 m
Approximate crossbow time	6 h:12 m	4 h:15 m	2 h:32 m
Approximate cost (US/Europe)	$52.36/$60.06	$71.40/$81.90	$83.64/$95.94

Costs are approximate and based on the pricing as of this writing, that is, $0.68 per extra-large high-CPU EC2 node per hour in the US and $0.78 in Europe. Times can vary subject to, for example, congestion and Internet traffic conditions.

Figure 31 illustrates scalability of the computation as a function of the number of processor cores allocated. Units on the vertical axis are the reciprocal of the wall clock time. Whereas wall clock time measures elapsed time, its reciprocal measures throughput - that is, experiments per hour. The straight diagonal line extending from the 80-core point represents hypothetical linear speedup, that is, extrapolated throughput under the assumption that doubling the number of processors also doubles throughput. In practice, parallel algorithms usually exhibit worse-than-linear speedup because portions of the computation are not fully parallel. In the case of Crossbow, deviation from linear speedup is primarily due to load imbalance among CPUs in the map and reduce phases, which can cause a handful of work-intensive 'straggler' tasks to delay progress. The reduce phase can also experience imbalance due to, for example, variation in coverage.

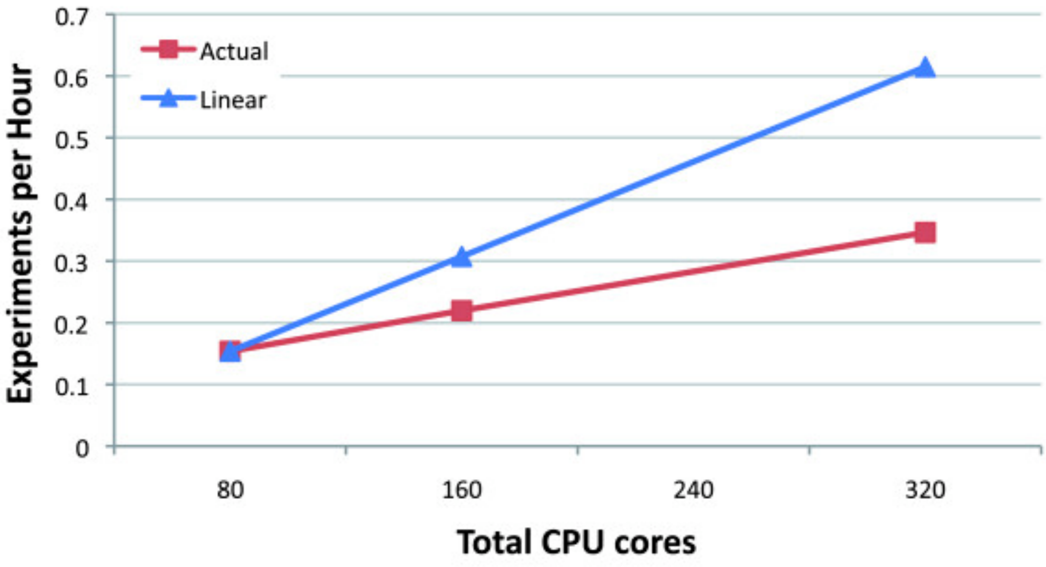


Figure 31. Crossbow Scaling Performance.
Number of worker CPU cores allocated from EC2 versus throughput measured in experiments per hour: that is, the reciprocal of the wall clock time required to conduct a whole-human experiment on the Wang et al. dataset. The line labeled 'linear speedup' traces hypothetical linear speedup relative to the throughput for 80 CPU cores.

Materials and methods

Alignment and SNP calling in Hadoop

Hadoop is an implementation of the MapReduce parallel programming model. Under Hadoop, programs are expressed as a series of map and reduce phases operating on tuples of data. Though not all programs are easily expressed this way, Hadoop programs stand to benefit from services provided by Hadoop. For instance, Hadoop programs need not deal with particulars of how work and data are distributed across the cluster; these details are handled by Hadoop, which automatically partitions, sorts and routes data among computers and processes. Hadoop also provides fault tolerance by partitioning files into chunks and storing them redundantly on the HDFS. When a subtask fails due to hardware or software errors, Hadoop restarts the task automatically, using a cached copy of its input data.

A mapper is a short program that runs during the map phase. A mapper receives a tuple of input data, performs a computation, and outputs zero or more tuples of data. A tuple consists of a key and a value. For example, within Crossbow a read is represented as a tuple where the key is the read's name and the value equals the read's sequence and quality strings. The mapper is generally constrained to be stateless - that is, the content of an output tuple may depend only on the content of the corresponding input tuple, and not on previously observed tuples. This enables MapReduce to safely execute many instances of the mapper in parallel. Similar to a mapper, a reducer is a short program that runs during the reduce phase, but with the added condition that a single instance of the reducer will receive all tuples from the map phase with the same key. In this way, the mappers typically compute partial results, and the reducer finalizes the computation using all the tuples with the same key, and outputs zero or more output tuples. The reducer is also constrained to be stateless - that is, the content of an output tuple may depend only the content of the tuples in the incoming batch, not on any other previously observed input tuples. Between the map and reduce phases, Hadoop automatically executes a sort/shuffle phase that bins and sorts tuples according to primary and secondary keys before passing batches on to reducers. Because mappers and reducers are stateless, and because Hadoop itself handles the sort/shuffle phase, Hadoop has significant freedom in how it distributes parallel chunks of work across the cluster.

The chief insight behind Crossbow is that alignment and SNP calling can be framed as a series of map, sort/shuffle and reduce phases. The map phase is short read alignment where input tuples represent reads and output tuples represent alignments. The sort/shuffle phase bins alignments according to the genomic region ('partition') aligned to. The sort/shuffle phase also sorts alignments along the forward strand of the reference in preparation for consensus calling. The reduce phase calls SNPs for a given partition, where input tuples represent the sorted list of alignments occurring in the partition and output tuples represent SNP calls.

A typical Hadoop program consists of Java classes implementing the mapper and reducer running in parallel on many compute nodes. However, Hadoop also supports a 'streaming' mode of operation whereby the map and reduce functions are delegated to command-line scripts or compiled programs written in any language. In streaming mode, Hadoop executes the streaming programs in parallel on different compute nodes, and passes tuples into and out of the program as tab-delimited lines of text written to the 'standard in' and 'standard out' file handles. This allows Crossbow to reuse existing software for aligning reads and calling SNPs while automatically gaining the scaling benefits of Hadoop. For alignment, Crossbow uses Bowtie [72], which employs a Burrows-Wheeler index [84] based on the full-text minute-space (FM) index [85] to enable fast and memory-efficient alignment of short reads to mammalian genomes.

To report SNPs, Crossbow uses SOAPsnp [75], which combines multiple techniques to provide high-accuracy haploid or diploid consensus calls from short read alignment data. At the core of SOAPsnp is a Bayesian SNP model with configurable prior probabilities. SOAPsnp's priors take into account differences in prevalence between, for example, heterozygous versus homozygous SNPs and SNPs representing transitions versus those representing transversions. SOAPsnp can also use previously discovered SNP loci and allele frequencies to refine priors. Finally, SOAPsnp recalibrates the quality values provided by the sequencer according to a four-dimensional training matrix representing observed error rates among uniquely aligned reads. In a previous study, human genotype calls obtained using the SOAP aligner and SOAPsnp exhibited greater than 99% agreement with genotype calls obtained using an Illumina 1 M BeadChip assay of the same Han Chinese individual [75].

Crossbow's efficiency requires that the three MapReduce phases, map, sort/shuffle and reduce, each be efficient. The map and reduce phases are handled by Bowtie and SOAPsnp, respectively, which have been shown to perform efficiently in the context of human resequencing. But another advantage of Hadoop is that its implementation of the sort/shuffle phase is extremely efficient, even for human resequencing where mappers typically output billions of alignments and hundreds of gigabytes of data to be sorted. Hadoop's file system (HDFS) and intelligent work scheduling make it especially well suited for huge sort tasks, as evidenced by the fact that a 1,460-node Hadoop cluster currently holds the speed record for sorting 1 TB of data on commodity hardware (62 seconds) (http://sortbenchmark.org/).

Modifications to existing software

Several new features were added to Bowtie to allow it to operate within Hadoop. A new input format (option --12) was added, allowing Bowtie to recognize the one-read-per-line format produced by the Crossbow preprocessor. New command-line options --mm and --shmem instruct Bowtie to use memory-mapped files or shared memory, respectively, for loading and storing the reference index. These features allow many Bowtie processes, each acting as an independent mapper, to run in parallel on a multi-core computer while sharing a single in-memory image of the reference index. This maximizes alignment throughput when cluster computers contain many CPUs but limited memory. Finally, a Crossbow-specific output format was implemented that encodes an alignment as a tuple where the tuple's key identifies a reference partition and the value describes the alignment. Bowtie detects instances where a reported alignment spans a boundary between two reference partitions, in which case Bowtie outputs a pair of alignment tuples with identical values but different keys, each identifying one of the spanned partitions. These features are enabled via the --partition option, which also sets the reference partition size.

The version of SOAPsnp used in Crossbow was modified to accept alignment records output by modified Bowtie. Speed improvements were also made to SOAPsnp, including an improvement for the case where the input alignments cover only a small interval of a chromosome, as is the case when Crossbow invokes SOAPsnp on a single partition. None of the modifications made to SOAPsnp fundamentally affect how consensus bases or SNPs are called.

Workflow

The input to Crossbow is a set of preprocessed read files, where each read is encoded as a tab-delimited tuple. For paired-end reads, both ends are stored on a single line. Conversion takes place as part of a bulk-copy procedure, implemented as a Hadoop program driven by automatic scripts included with Crossbow. Once preprocessed reads are situated on a filesystem accessible to the Hadoop cluster, the Crossbow MapReduce job is invoked (Figure 32). Crossbow's map phase is short read alignment by Bowtie. For fast alignment, Bowtie employs a compact index of the reference sequence, requiring about 3 Gb of memory for the human genome. The index is distributed to all computers in the cluster either via Hadoop's file caching facility or by instructing each node to independently obtain the index from a shared filesystem. The map phase outputs a stream of alignment tuples where each tuple has a primary key containing chromosome and partition identifiers, and a secondary key containing the chromosome offset. The tuple's value contains the aligned sequence and quality values. The soft/shuffle phase, which is handled by Hadoop, uses Hadoop's KeyFieldBasedPartitioner to bin alignments according to the primary key and sort according to the secondary key. This allows separate reference partitions to be processed in parallel by separate reducers. It also ensures that each reducer receives alignments for a given partition in sorted order, a necessary first step for calling SNPs with SOAPsnp.

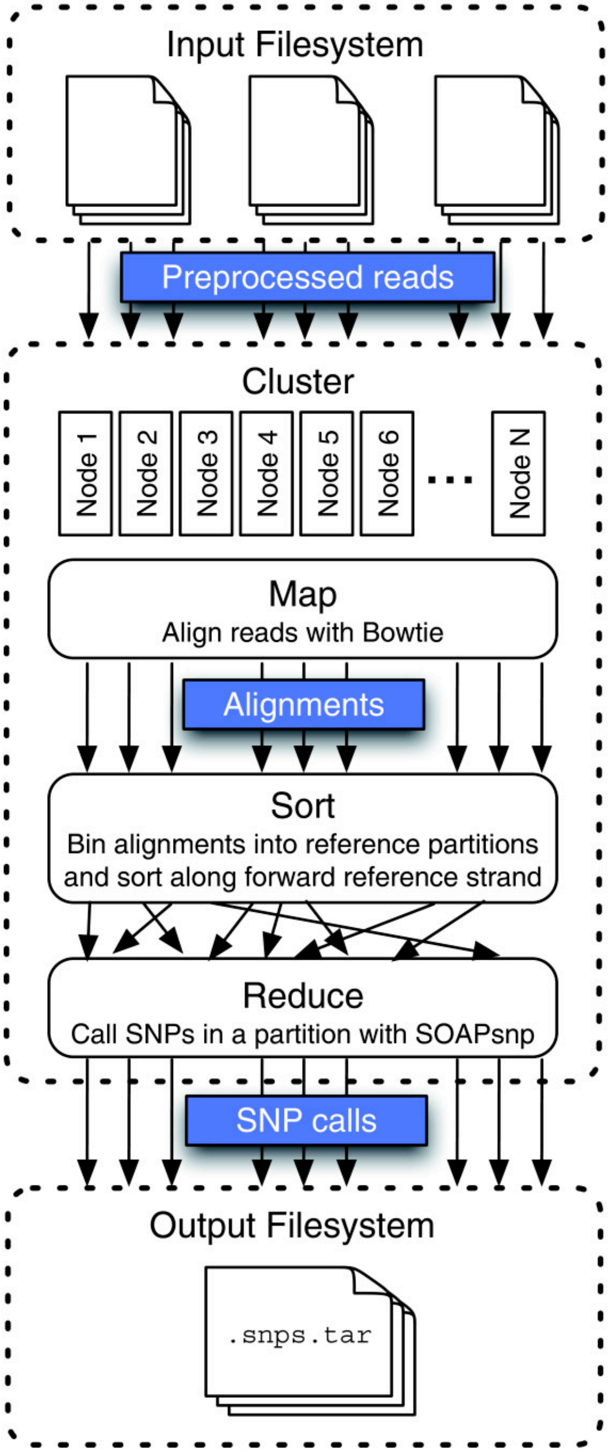


Figure 32. Crossbow workflow.
Previously copied and pre-processed read files are downloaded to the cluster, decompressed and aligned using many parallel instances of Bowtie. Hadoop then bins and sorts the alignments according to primary and secondary keys. Sorted alignments falling into each reference partition are then submitted to parallel instances of SOAPsnp. The final output is a stream of SNP calls made by SOAPsnp.

The reduce phase performs SNP calling using SOAPsnp. A wrapper script performs a separate invocation of the SOAPsnp program per partition. The wrapper also ensures that SOAPsnp is invoked with appropriate options given the ploidy of the reference partition. Files containing known SNP locations and allele frequencies derived from dbSNP [86] are distributed to worker nodes via the same mechanism used to distribute the Bowtie index. The output of the reduce phase is a stream of SNP tuples, which are stored on the cluster's distributed filesystem. The final stage of the Crossbow workflow archives the SNP calls and transfers them from the cluster's distributed filesystem to the local filesystem.

Cloud support

Crossbow comes with scripts that automate the Crossbow pipeline on a local cluster or on the EC2 utility computing service. The EC2 driver script can be run from any Internet-connected computer; however, all the genomic computation is executed remotely. The script runs Crossbow by: allocating an EC2 cluster using the Amazon Web Services tools; uploading the Crossbow program code to the master node; launching Crossbow from the master; downloading the results from the cluster to the local computer; and optionally terminating the cluster, as illustrated in Figure 33. The driver script detects common problems that can occur in the cluster allocation process, including when EC2 cannot provide the requested number of instances due to high demand. The overall process is identical to running on a local dedicated cluster, except cluster nodes are allocated as requested.

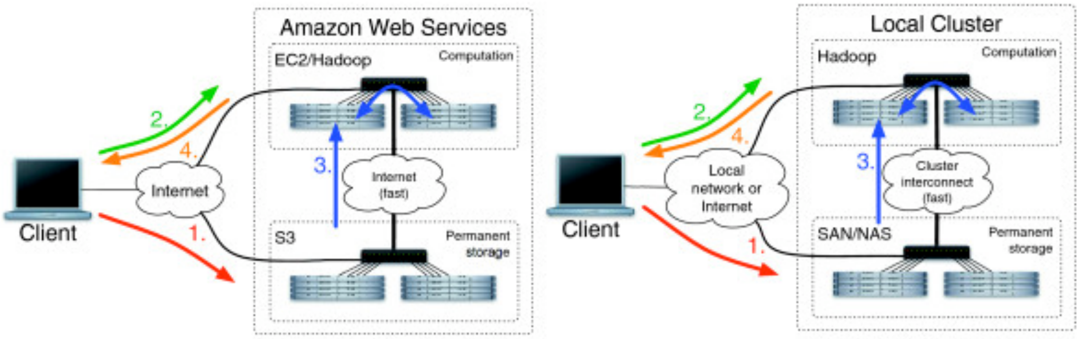


Figure 33. Four basic steps to running the Crossbow computation.
Two scenarios are shown: one where Amazon's EC2 and S3 services are used, and one where a local cluster is used. In step 1 (red) short reads are copied to the permanent store. In step 2 (green) the cluster is allocated (may not be necessary for a local cluster) and the scripts driving the computation are uploaded to the master node. In step 3 (blue) the computation is run. The computation download reads from the permanent store, operates on them, and stores the results in the Hadoop distributed filesystem. In step 4 (orange), the results are copied to the client machine and the job completes. SAN (Storage Area Network) and NAS (Network-Attached Storage) are two common ways of sharing filesystems across a local network.

Genotyping experiment

We generated 40-fold coverage of chromosomes 22 and X (NCBI 36.3_ using 35-bp paired-end reads. Quality values were assigned by randomly selecting observed quality strings from a pair of FASTQ files in the Wang et al. dataset (080110_EAS51_FC20B21AAXX_L7_YHPE_PE1). The mean and median quality values among those in this subset are 21.4 and 27, respectively, on the Solexa scale. Sequencing errors were simulated at each position at the rate dictated by the quality value at that position. For instance, a position with Solexa quality 30 was changed to a different base with a probability of 1 in 1,000. The three alternative bases were considered equally likely.

Insert lengths were assigned by randomly selecting from a set of observed insert lengths. Observed insert lengths were obtained by aligning a pair of paired-end FASTQ files (the same pair used to simulate the quality values) using Bowtie with options '-X 10000 -v 2 --strata --best -m 1'. The observed mean mate-pair distance and standard deviation for this subset were 422 bp and 68.8 bp, respectively.

Bowtie version 0.10.2 was run with the '-v 2 --best --strata -m 1' to obtain unique alignments with up to two mismatches. We define an alignment as unique if all other alignments for that read have strictly more mismatches. SOAPsnp was run with the rank-sum and binomial tests enabled (-u and -n options, respectively) and with known-SNP refinement enabled (-2 and -s options). Positions and allele frequencies for known SNPs were calculated according to the same HapMap SNP data used to simulate SNPs. SOAPsnp's prior probabilities for novel homozygous and heterozygous SNPs were set to the rates used by the simulator (-r 0.0001 -e 0.0002 for chromosome 22 and -r 0.0002 for chromosome X).

An instance where Crossbow reports a SNP on a diploid portion of the genome was discarded (that is, considered to be homozygous for the reference allele) if it was covered by fewer than four uniquely aligned reads. For a haploid portion, a SNP was discarded if covered by fewer than two uniquely aligned reads. For either diploid or haploid portions, a SNP was discarded if the call quality as reported by SOAPsnp was less than 20.

Whole-human resequencing experiment

Bowtie version 0.10.2 and a modified version of SOAPsnp 1.02 were used. Both were compiled for 64-bit Linux. Bowtie was run with the '-v 2 --best --strata -m 1' options, mimicking the alignment and reporting modes used in the SOAPsnp study. A modified version of SOAPsnp 1.02 was run with the rank-sum and binomial tests enabled (-u and -n options, respectively) and with known-SNP refinement enabled (-2 and -s options). Positions for known SNPs were calculated according to data in dbSNP [86] versions 128 and 130, and allele frequencies were calculated according to data from the HapMap project [83]. Only positions occurring in dbSNP version 128 were provided to SOAPsnp. This was to avoid biasing the result by including SNPs submitted by Wang et al. to dbSNP version 130. SOAPsnp's prior probabilities for novel homozygous and heterozygous SNPs were left at their default values of 0.0005 and 0.001, respectively. Since the subject was male, SOAPsnp was configured to treat autosomal chromosomes as diploid and sex chromosomes as haploid.

To account for base-calling errors and inaccurate quality values reported by the Illumina software pipeline [87, 88], SOAPsnp recalibrates quality values according to a four-dimensional matrix recording observed error rates. Rates are calculated across a large space of parameters, the dimensions of which include sequencing cycle, reported quality value, reference allele and subject allele. In the previous study, separate recalibration matrices were trained for each human chromosome; that is, a given chromosome's matrix was trained using all reads aligning uniquely to that chromosome. In this study, each chromosome is divided into non-overlapping stretches of 2 million bases and a separate matrix is trained and used for each partition. Thus, each recalibration matrix receives less training data than if matrices were trained per-chromosome. Though the results indicate that this does not affect accuracy significantly, future work for Crossbow includes merging recalibration matrices for partitions within a chromosome prior to genotyping.

An instance where Crossbow reports a SNP on a diploid portion of the genome is discarded (that is, considered to be homozygous for the reference allele) if it is covered by fewer than four unique alignments. For a haploid portion, a SNP is discarded if covered by fewer than two unique alignments. For either diploid or haploid portions, a SNP is discarded if the call quality as reported by SOAPsnp is less than 20. Note that the SOAPsnp study applies additional filters to discard SNPs at positions that, for example, are not covered by any paired-end reads or appear to have a high copy number. Adding such filters to Crossbow is future work.

Discussion

In this chapter we have demonstrated that cloud computing realized by MapReduce and Hadoop can be leveraged to efficiently parallelize existing serial implementations of sequence alignment and genotyping algorithms. This combination allows large datasets of DNA sequences to be analyzed rapidly without sacrificing accuracy or requiring extensive software engineering efforts to parallelize the computation.

We describe the implementation of an efficient whole-genome genotyping tool, Crossbow, that combines two previously published software tools: the sequence aligner Bowtie and the SNP caller SOAPsnp. Crossbow achieves at least 98.9% accuracy on simulated datasets of individual chromosomes, and better than 99.8% concordance with the Illumina 1 M BeadChip assay of a sequenced individual. These accuracies are comparable to those achieved in the prior SOAPsnp study once filtering stringencies are taken into account.

When run on conventional computers, a deep-coverage human resequencing project requires weeks of time to analyze on a single computer by contrast, Crossbow aligns and calls SNPs from the same dataset in less than 3 hours on a 320-core cluster. By taking advantage of commodity processors available via cloud computing services, Crossbow condenses over 1,000 hours of computation into a few hours without requiring the user to own or operate a computer cluster. In addition, running on standard software (Hadoop) and hardware (EC2 instances) makes it easier for other researchers to reproduce our results or execute their own analysis with Crossbow.

Crossbow scales well to large clusters by leveraging Hadoop and the established, fast Bowtie and SOAPsnp algorithms with limited modifications. The ultrafast Bowtie alignment algorithm, utilizing a quality-directed best-first-search of the FM index, is especially important to the overall performance of Crossbow relative to CloudBurst. Crossbow's alignment stage vastly outperforms the fixed-seed seed-and-extend search algorithm of CloudBurst on clusters of the same size. We expect that the Crossbow infrastructure will serve as a foundation for bringing massive scalability to other high-volume sequencing experiments, such as RNA-seq and ChIP-seq. In our experiments, we demonstrated that Crossbow works equally well either on a local cluster or a remote cluster, but in the future we expect that utility computing services will make cloud computing applications widely available to any researcher.

Chapter 6: Assembly of Large Genomes using Second-Generation Sequencing.

Summary of Contribution

This chapter is a review of current genome assembly methods and results, written in collaboration with Steven Salzberg and Arthur Delcher at the University of Maryland, and is currently under review for publication. The review is written as an introduction to the challenges of genome assembly, especially in consideration of the current second-generation sequencing technologies.

Michael Schatz wrote the initial draft of the review of the assembly methods and challenges, and made the figures (including the necessary computations). Steven Salzberg wrote the initial draft of the introduction, and the recommendations for future sequencing projects. Arthur Delcher wrote the initial draft of the review of the current assembly results.

Abstract

Current second-generation sequencing technology can now be used to sequence an entire human genome in a matter of days and at low cost. Sequence read lengths, initially very short, have rapidly increased since the technology first appeared, and we now are seeing a growing number of efforts to sequence large genomes *de novo* from these short reads. In this Perspective, we describe the issues associated with short-read assembly, the different types of data produced by second-

gen sequencers, and the latest assembly algorithms designed for these data. We also review the genomes that have been assembled recently from short reads, and make recommendations for sequencing strategies that will yield a high-quality assembly.

Introduction

As genome sequencing technology has evolved, methods for assembling genomes have changed with it. Genome sequencers have never been able to "read" more than a relatively short stretch of DNA at once, with read lengths gradually increasing over time. Reconstructing a complete genome from a set of reads requires an assembly program, and a variety of genome assemblers have been used for this task. In 1995, when the first bacterial genome was published (*Haemophilus influenzae*), read lengths were approximately 460 base pairs (bp), and that whole-genome shotgun sequencing project generated 24,304 reads [5]. The human genome project required approximately 30 million reads, with lengths up to 800 bp, using Sanger sequencing technology and automated capillary sequencers [68, 89]. This corresponded to 24 billion bases (Gb), or approximately 8-fold coverage of the 3-Gb human genome. Redundant coverage, in which on average every nucleotide is sequenced many times over, is required to produce a high-quality assembly. Another benefit of redundancy is greatly increased accuracy compared to a single read: where a single read might have an error rate of 1%, 8-fold coverage has an error rate as low as 10^{-16} when eight high-quality reads agree with one another. High coverage is also necessary to fully and accurate sequence polymorphic alleles within diploid or polyploid genomes.

Current second-generation sequencing (SGS) technologies produce read lengths ranging from 35-400 bp, at far greater speed and much lower cost than Sanger sequencing. However, as reads get shorter, coverage needs to increase to compensate for the decreased connectivity and produce a comparable assembly. Certain problems cannot be overcome by deeper coverage: if a repetitive sequence is longer than a read, then coverage alone will never compensate, and all copies of that sequence will produce gaps in the assembly. These gaps can be spanned by paired reads—consisting of two reads generated from a single fragment of DNA and separated by a known distance—as long as the pair separation distance is longer than the repeat. Paired-end sequencing is available from most of the SGS machines, although it is not yet as flexible or as reliable as paired-end sequencing using traditional methods.

After the successful assembly of the human [68] and mouse [90] genomes by whole-genome shotgun (WGS) sequencing, most large-scale genome projects quickly moved to adopt the WGS approach, which has subsequently been used for dozens of eukaryotic genomes. Today, thanks to changes in sequencing technology, a major question confronting genome projects is, can we sequence a large genome (>100 Mbp) using short reads? If so, what are the limitations on read length, coverage, and error rates? How much paired-end sequencing is necessary? And what will the assembly look like? In this perspective we take a look at each of these questions and describe the solutions available today. Although we provide some answers, we have no doubt that the solutions will change rapidly over the next few years, as both the sequencing methods and the computational solutions improve.

Overview of SGS technologies

The two leading sequencing technologies today produce reads with decidedly different characteristics. The pyrosequencing approach, embodied in the 454 sequencer from Roche, produces read lengths approaching 400 bp, and in a single 1-day run generates several hundred million nucleotides. This technology sequences DNA by sequentially flowing bases in a predetermined order across templates that are captured on microscopic beads contained in tiny wells. A single cycle will incorporate multiple bases whenever the template sequence has a homopolymer run. Base calling is done by measuring the fluorescence intensity at each well, with greater intensity corresponding to multiple bases. Read lengths and error rates have steadily improved since this method was introduced in 2005 [91] and 800 bp reads are expected in the near future. At that point, pyrosequencing read lengths will match those of Sanger sequencing.

The alternative approach produces shorter reads, but at much higher throughput. This approach is embodied in several different commercial sequencers, including those from Illumina, Applied Biosystems, and Helicos. The shared theme is to incorporate only one base per cycle, using specially modified bases that include both a fluorescent tag and a terminator [92]. After reading the base with a laser, the tag and terminator are removed so that the template can be extended by one more base. These machines operate at much higher densities, produce 20-30 Gb per run, although a single run takes 5-10 days depending on the machine. Read lengths have grown over the past two years from 25-30 bases to >100 bases today on some platforms. The overall cost per run is similar to pyrosequencing, yielding a much lower per-base cost.

All these platforms offer some form of paired-end sequencing, but thus far the reliability of paired ends is not nearly as good as it is for Sanger sequencing. In conventional Sanger sequencing, a “long” paired-end protocol starts with DNA templates ranging from 5000 to 35,000 bp. These fragments are cloned into a vector, which is then amplified in *E. coli* prior to sequencing. The vectors are subsequently extracted and then both ends of the vector inserts are sequenced. One drawback to this traditional method is that the *E. coli* cloning step introduces a bias, making it difficult to capture some regions of a genome.

Paired-end protocols for SGS avoid the use of a bacterial cloning step. Instead, they generally start with DNA fragments of the desired size, and then try to sequence both ends by circularizing the DNA, using a special tag or linker to connect the ends. By sequencing fragments containing the tag, both ends of the original fragment will be captured. Although this sounds straightforward, experience to date has indicated that it is very difficult to get DNA to circularize efficiently, and problems increase as the fragments get longer [93]. As a result, many paired-end libraries contain too little DNA, and the paired-end sequences fail to cover the genome at the required depth. These protocols are also currently limited to relatively short insert sizes compared to those available with Sanger sequencing, and very long range fosmid or BAC-end sequence data are not currently available at all. This has significant implications for genome assembly, as we discuss below.

Overview of assembly methods

Current genome sequencing technology can only sequence a tiny portion of genome in a contiguous read. Nevertheless, just as a jigsaw puzzle can be assembled from small puzzle pieces, a complete genome sequence can be assembled from short reads. Unlike jigsaw puzzle pieces that precisely lock together, DNA sequence reads may fit together in more than one way because of repetitive sequences within the genome. Assembly methods aim to create most complete reconstruction possible without introducing errors.

The central challenge of genome assembly is resolving repetitive sequences. The magnitude of the challenge depends on the sequencing technology, because the fraction of repetitive reads depends on the length of reads themselves. At one extreme, if the reads were just one base long, every read would be repetitive; at the other extreme, if we could simply read an entire chromosome from one end to the other, repeats would pose no problem at all. In between these extremes, the fraction of unique sequences increases as the read length increases, until eventually every sequence in the genome is unique. If DNA sequences were random (which they are not), then the expected number of occurrences of any sequence would decrease exponentially as the length of the sequence increases, and a modest increase in read length could dramatically reduce the number of repeats in the genome. However, real genomes have complicated repeat structures making some sequences nearly impossible to assembly correctly.

To illustrate the variability in repetitiveness among species, Figure 34 shows the uniqueness ratio constructed using the tallymer tool [94] for varying read lengths plotted for six genomes: fruit fly (*Drosophila melanogaster*), grapevine (*Vitis vinifera*), chicken (*Gallus gallus*), dog (*Canis familiaris*), human (*Homo sapiens*), and the single-celled parasite *Trichomonas vaginalis*. The figure shows how much of each genome would be covered by k-mers (reads) that occur exactly once. Among the multi-cellular species, dog and chicken are the least repetitive while fly is the most repetitive. The percentage of a genome covered uniquely increases rapidly as read length increases to 50 bp and above, but the rate of increase varies due to the variable repeat lengths in different species.

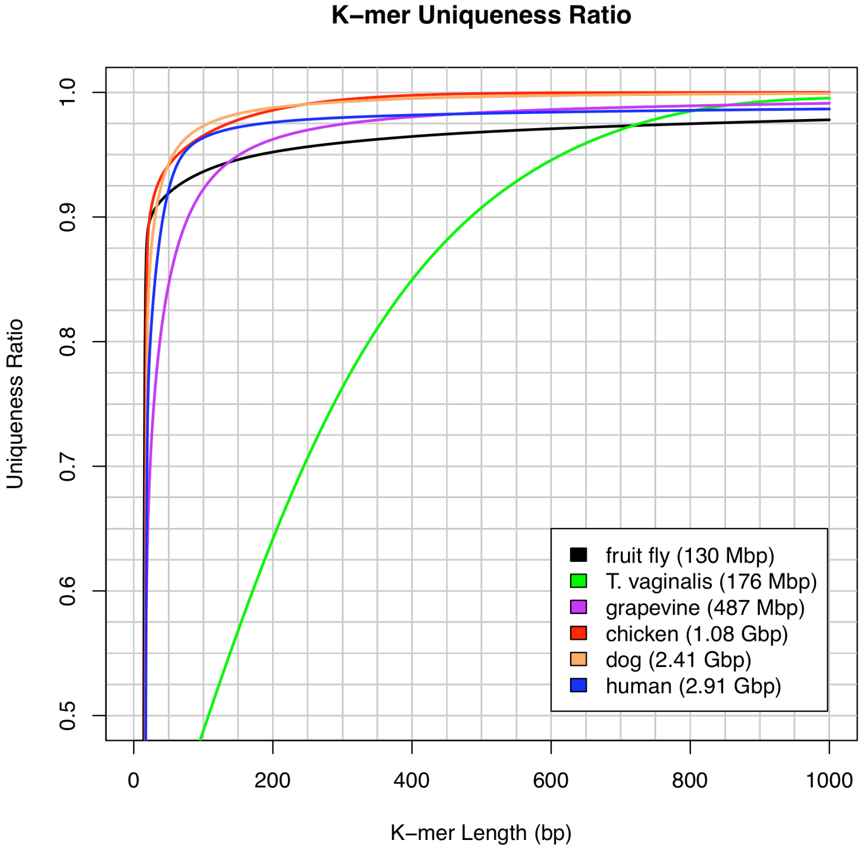


Figure 34. The K-mer uniqueness ratio for five well-known organisms and one single-celled human parasite.
The ratio is defined here as the percentage of the genome that is covered by unique sequences of length K or longer. The horizontal axis shows the length in base pairs of the sequences. For example, ~92.5% of the grapevine genome is contained in unique sequences of 100 bp or longer.

Early genome assemblers used a simple “greedy” algorithm, in which all pairs of reads are compared to each other, and the ones that overlap most are merged first. To allow for sequencing errors, assembler compute these overlaps with a variant of the Smith-Waterman algorithm [71], which allows for a small number of differences in the overlapping sequence, typically 1%-10%. Once all overlaps are computed, the reads with the longest overlap are concatenated to form a contig (contiguous sequence). The process then repeats, each time merging the sequences with the longest overlap until all overlaps are used.

This simple merging process will accurately reconstruct the simplest genomes, but fails for repetitive sequences longer than the read length. The greedy algorithm will assemble all copies of a repeat into a single instance, because all reads with the repetitive sequence overlap equally well. The problem is that the greedy algorithm cannot tell how to connect the unique sequences on either end of a repeat, and it can easily assemble together distant portions of the genome into mis-assembled, “chimeric” contigs. Beginning in the 1990’s, assembly of bacterial genomes required development of more sophisticated methods to handle repetitive sequences. Assembly of large eukaryotic genomes required further innovations, not only in the handling of repeats, but also in the computational requirements for memory and processing time. If these issues are not handled in a sophisticated way, then the enormous data sets comprising mammalian genome projects will simply overwhelm even the largest computers.

Large-scale shotgun assembly

Several assemblers have been developed to assemble large, repetitive genomes from long ("Sanger") reads, including the Celera Assembler [21], Arachne [22, 23], and PCAP [95]. More recently the Newbler assembler [91] was designed to handle shorter 454 reads, which have a different error profile from Sanger reads. Unlike simple greedy assemblers, these algorithms assemble the reads in two or more distinct phases, with separate processing of repetitive sequences. First they assemble reads with unambiguous overlaps, creating contigs that end on the boundaries of repeats. (Myers et al. call these "unitigs" [21].) Then in a second phase, they assemble the unambiguous contigs together into larger sequences, using mate-pair constraints to resolve repeats.

As with earlier methods, these large-scale assemblers begin by computing overlaps between all pairs of reads. One technique for saving memory, used by Celera Assembler (CABOG), is to construct an *overlap graph* where each read is a node in the graph, and weighted edges connect overlapping reads. These assemblers also attempt to correct sequencing errors by using overlapping reads to confirm each other. These error correction methods can be very effective when coverage is deep, as it often is with newer short-read sequencing projects.

The scaffolding phase of assembly focuses on resolving repeats by linking the initial contigs into scaffolds, guided by mate-pair data. Mate pairs constrain the separation distance and the orientation of contigs containing mated reads. A scaffold is a collection of contigs linked by mate pairs, in which the gaps between contigs may represent either repeats, in which case the gap can in theory be filled with one or more copies of the repeat, or true gaps in which the original sequencing project did not capture the sequence needed to fill the gap. If the mate pair distances are long enough, they permit the assembler to link contigs across almost all repeats.

Assemblers vary in their strategies for calling a contig repetitive, but most of them rely on some combination of the length of the contig and number of reads it contains. If a contig contains too many reads, then it is flagged as a repeat. High copy-number repeats are easy to identify, because the coverage statistics make it obvious that they are repetitive; in contrast, 2-copy repeats are the most difficult to identify using statistical methods.

After flagging repeats, an assembler can build scaffolds by connecting unique contigs using mate-pair links. If the contigs in a scaffold overlap, the assembler can merge them at this point. Otherwise, the assembler will record a gap of approximately known size within the scaffold. Assemblers can also include repetitive contigs in these scaffolds, as long as the repeats are connected by mate pairs to unique contigs.

Short Read Assembly

In principle, assemblers created for long reads should also function for short reads. The principles of detecting overlap and building contigs are no different. In practice, initial attempts to use existing assemblers with very short reads either failed or performed very poorly, for a variety of reasons. Some of these failures were mundane: for example, assemblers impose a minimum read length, or they require a minimum amount of overlap that is too long for a short-read sequencing project. Other failures are caused by more fundamental problems.

The computation of overlaps is one of the most critical steps in any assembly algorithm. Short-read sequencing projects require that this step be re-designed to make it computationally feasible, especially since many more short reads than long reads are needed to achieve the same level of coverage. (Coverage is defined as the average number of reads that contain any nucleotide; thus 8X coverage implies that the genome is sequenced eight times over.) As such, the number of overlaps to compute will increase, and any per-read or per-overlap overhead will be greatly magnified. This problem is exacerbated by the fact that short-read projects compensate for read length by obtaining deeper coverage, and it is not unusual to see SGS projects at 30, 40, or 50X coverage rather than the 8X coverage that is typical of Sanger sequencing projects.

The parameters used for computing overlaps have to be carefully tuned to accommodate shorter read lengths. Genome assemblers such as CABOG and Arachne do not compute the overlap between all pairs of reads, but instead use a seed-based strategy to identify reads that are likely to overlap. With this approach, short fixed length substrings of the reads, k-mers, are used as an index, and only pairs of reads that share a seed are evaluated further. The choice of seed length is critical, and depends on the length of the read, the amount of sequencing error, and the size of the genome. If the seed is too long, legitimate overlaps will be missed, thereby fragmenting the assembly, but if the seed length is too short, the computation time increases dramatically, so much that the computation may no longer be feasible. In addition to adjusting the seed length for short reads, the amount of error varies among SGS technologies, meaning that assemblers may have to be fine-tuned separately for each sequencing technology.

For these reasons and others, a new generation of genome assemblers has been developed specifically to address the challenges of assembling very short reads. These assemblers include Velvet [96, 97], ALLPATHS [98, 99], ABySS [80] and SOAPdenovo [100]. Rather than using an overlap graph, all of these assemblers use a *de Bruijn graph* algorithm, first described for the EULER assembler [101]. In this approach, the reads are decomposed into k-mers that in turn become the nodes of a de Bruijn graph. A directed edge between nodes indicates that the k-mers on those nodes occur consecutively in one or more reads. These k-mers take the place of the seeds used for overlap computation in other assemblers (Figure 35).

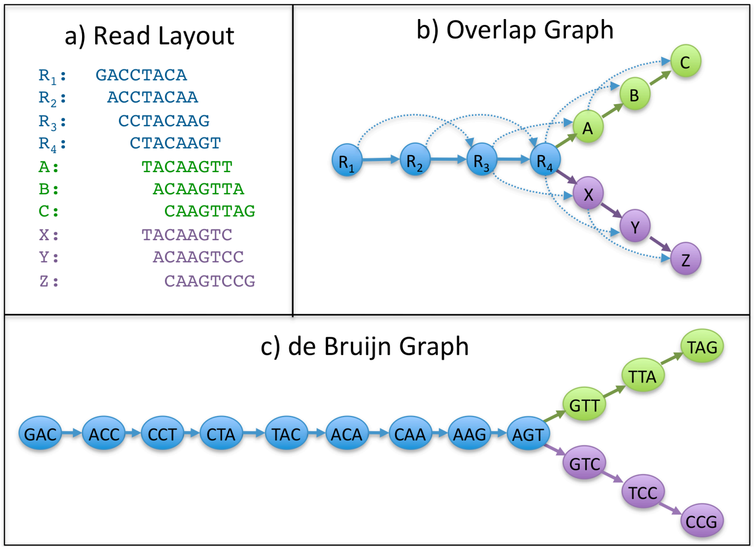


Figure 35. Comparison of the overlap graph and a de Bruijn graph for assembly.
Based on the set of ten 8 bp reads in (a), we can build an overlap graph (b) in which each read is a node and overlaps >5 bp are indicated by directed edges. Transitive overlaps, which are implied by other longer overlaps, are shown as dotted edges. In a de Bruin graph (c), a node is created for every k-mer in all the reads; here the k-mer size is 3. Edges are drawn between every pair of successive k-mers in a read, where the k-mers overlap by k-1 bases. In both approaches, repeat sequences create a fork in the graph. Note here we have only considered the forward orientation of each sequence to simplify the figure.

Unambiguous stretches of sequence form non-branching paths in the de Bruijn graph, making it easy to "read off" contigs by walking these paths. Overlaps between reads are implicitly captured by the graph, rather than explicitly computed, saving a substantial amount of computing time. Similar to the overlap graph approach, all copies of a repeat will initially be represented by a single high coverage node. Repeat boundaries and sequencing errors show up as branch points in the graph, and complex repeats appears as densely connected "tangles."

Sequencing error complicates the de Bruijn graph, but many errors are easily recognized by their structure in the graph. For example, errors at the end of a read usually create k-mers that occur only once, and therefore form dead-end "tips" in the graph. Errors in the middle of a read create alternate paths called "bubbles" that terminate at the same node. De Bruijn graph assemblers search for these localized graph structures in an error correction phase, and remove the error nodes and other low coverage nodes. Mate-pair information can be used to resolve ambiguity, using the coverage at each node to indentify repeats, and by searching for unique paths through the graph consistent with the mate-pairs.

The main drawback to the de Bruijn approach is the loss of information caused by decomposing a read into a path of k-mers. Compared to conventional assemblers, where a read is a single node in the overlap graph, de Bruijn assemblers initially create multiple nodes for each read, and these nodes may not form a linear path once edges from other reads are added. Furthermore, unlike the overlap graph, the de Bruijn graph is not *read coherent* [102], meaning there may be paths through the graph that form a sequence that is not supported by the underlying reads. For example, if the same k-mer occurs in the middle of two reads, but the reads do not otherwise overlap, the corresponding de Bruijn graph for those reads contains a branching node instead of two separate paths. Short repeats of this type can be resolved, but they require additional processing and therefore additional time.

Another potential drawback of the de Bruijn approach is the de Bruijn graph can require an enormous amount of computer space (random access memory, or RAM). Unlike conventional overlap computations, which can be easily partitioned into multiple jobs with distinct batches of reads, the construction and analysis of a de Bruijn graph is not easily parallelized. As a result, de Bruijn assemblers such as Velvet and ALLPATHS, which have been used successfully on bacterial genomes, do not scale to large genomes. For a human-sized genome, these programs would require several terabytes of RAM to store their de Bruijn graphs, which is far more memory than is available on most computers.

To date, only two de Bruijn graph assemblers have been shown to have the ability to assemble a mammalian-sized genome. ABySS [80] assembled a human genome in 87 hours on a cluster of 21 8-core machines each with 16GB of RAM (168 cores, 336 GB of RAM total). SOAPdenovo assembled a human genome in 40 hours using a single computer with 32 cores and 512GB of RAM [100]. Although these types of computing resources are not widely available, they are within reach for large-scale scientific centers.

In theory, the size of the de Bruijn graph depends only on the size of the genome, including polymorphic alleles, and should be independent of the number of reads. However, because sequencing errors create their own graph nodes, increasing the number of reads inevitably increases the size of the de Bruijn graph. In the *de novo* assembly of human from short reads, SOAPdenovo reduced the number of 25-mers from 14.6 billion to 5.0 billion by correcting errors before constructing the de Bruijn graph [100]. Its error correction method first counts the number of occurrences of all k-mers in the reads, and replaces any k-mers occurring <3 times with the highest frequency alternative k-mer.

Choice of assembler and sequencing strategy

Only de Bruijn graph assemblers have demonstrated the ability to successfully assemble very short reads (<50bp). For longer reads (>100bp), overlap graph assemblers have been quite successful and have a much better track record overall. A de Bruijn graph assembler should function with longer reads as well, but a large difference between the read length and the k-mer length will result in many more branching nodes than in the simplified overlap graph. The precise conditions under which one assembly method is superior to the other remain an open question, and the answer may ultimately depend on the specific assembler and genome characteristics.

As Figure 36 illustrates, there is a direct and dramatic tradeoff among read length, coverage, and expected contig length in a genome assembly. The figure shows the theoretical expected contigs length, based on the Lander-Waterman model [103], in an assembly where all overlaps have been detected perfectly. This model was widely applied for predicting assembly quality using traditional sequencing, and shows under ideal conditions 710 bp reads should require only 3X coverage to produce an expected 4Kbp expected contig size, while 30bp reads require 28X coverage. In practice, the model is incomplete for modeling very short reads: the figure also shows the actual contig sizes for the dog genome, assembled with 710 bp reads, and the panda genome, assembled with 52 bp reads. The dog assembly tracked closely to the theoretical prediction, while the panda assembly has contig sizes that are many times lower than predicted by the model. The large discrepancy between predicted and observed assembly quality is because simplifying assumptions in the model are problematic for shorter reads, especially the assumptions that the genome is free of repeats and the reads uniformly sample the genome with uniform error rates. As seen in Figure 34, a larger proportion of a genome is repetitive at short read length, and consequently the assembler will be forced to end contigs more often at repeat boundaries. Furthermore, second-generation sequencing is known to have sequence dependent coverage biases and non-uniform error rates [88]. These sequence irregularities will cause unexpectedly low coverage regions and consequently end contigs more often than expected. Fortunately, many of these limitations can be overcome by additional oversampling of the genome to boost the low coverage regions. Future work remains to enhance the Lander-Waterman model to capture more of these effects.

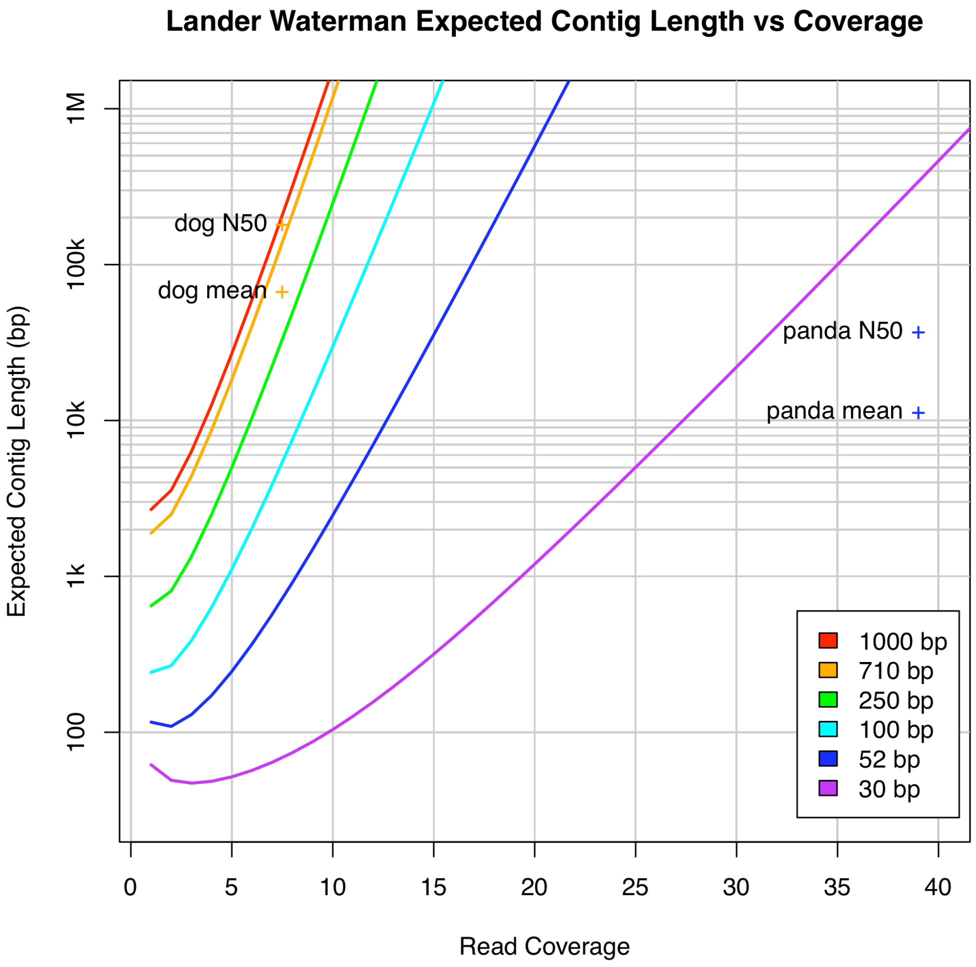


Figure 36. Expected average contig length by read length and coverage.
Also shown are the average contig lengths and N50 lengths for the dog genome, assembled with 710-bp reads, and the panda genome, assembled with 52-bp reads.

The figure also shows that even for longer Sanger reads, the theoretical model is a better predictor of N50 contig sizes than of mean contig lengths. An N50 contig size of N means that 50% of the assembled bases are contained in contigs of length N or larger. N50 sizes are often used as a measure of assembly quality because they capture how much of the genome is covered by relatively large contigs.

A good compromise solution to the problem of assembling a genome with short reads using today's technology, is to create a hybrid assembly using a mix of short and long reads. One strategy that we have used with some success is to assemble the short reads with a de Bruijn graph method such as Velvet, and then treat the resulting contigs as reads. The Velvet contigs together with the longer reads can then be assembled with CABOG or another overlap graph assembler.

Another strategy is to assemble the short and long reads using a single de Bruijn graph assembler. In this approach, the long reads are primarily used to disambiguate short repeats. This can work well, although overlap graph assemblers (CABOG and Arachne) are more mature than the new short read assemblers, and generally produce much better assemblies, especially because of their more sophisticated use of mate pairs. Using an overlap graph assembler with a combination of long and short reads requires that the assembler is carefully tuned to accommodate the shorter reads and potentially higher error rates.

By far the best approach is to use a reference genome sequence, which the assembler will use as a guide to resolve repeats. This is known as comparative assembly [104], and the assemblers that can perform this are a special subclass of assemblers. Most human re-sequencing efforts have followed this approach, if they attempted assembly at all, because it produces a far better result. However, the obvious drawback is that comparative assembly is simply not possible unless the species has already been sequenced and assembled previously. Furthermore, purely comparative techniques can not resolve large insertions or structural variations, and they face the same challenges of read quality, especially if there are significant coverage biases or high error rates.

Genome coverage and gaps

As coverage increases, the fraction of the genome sequenced increases while the number of gaps decreases. However, each sequencing technology has its own biases that produce gaps in coverage. Conventional Sanger sequencing uses cloning steps that amplify the genome in *E. coli*, which does not amplify all sequences equally well. SGS technologies avoid cloning in *E. coli*, but they too seem to have biases. Therefore any genome sequenced with just one technology, regardless of the depth of coverage, is liable to contain gaps due to bias. One way to overcome these biases and to close many gaps is to generate deep coverage in two or more sequencing technologies [105].

For Sanger sequencing projects, the point of diminishing returns, where additional sequencing yields little additional genomic sequence, falls at ~8X coverage. For very short reads (<50bp), higher coverage is clearly necessary, but the optimal depth of coverage has been a rapidly moving target over the past several years. Below we describe a number of SGS projects that have used different read lengths, depths of coverage, and assembly algorithms, with a mixture of results.

Read Length and Insert Size

In the ideal case, the quality of an assembly will be determined by the read lengths, mate-pair distances, and by the repeat structure of the genome. In general, longer reads make better assemblies because they span more repeats. Similarly, longer insert sizes (mate-pair distances) will increase scaffold sizes, but longer inserts will not always improve contig sizes. For an assembler to close a gap within a scaffold, it must find a set of reads that form an unambiguous path between the flanking contigs. With large gaps, multiple alternative paths through the overlap or de Bruijn graph are much more likely.

For this and other reasons, using a mixture of insert sizes can be very effective. The shortest inserts are used to resolve the small repeats, and longer inserts can resolve progressively longer repeats. In practice, long inserts tend to be less reliable, with a much higher variance in their length distribution.

Published SGS Genome Assemblies

In this section we survey short-read assembly results that have been published or recently announced. A summary of the *de novo* short-read assemblies is contained in Table 8, which gives general characteristics of the assemblies. Specific values can vary in how they are computed; e.g., the number of contigs depends on the minimum contig length included in the published assembly.

Human Genomes

Initial assembly results with SGS technology consisted primarily of mapping reads to a reference genome. This was the case with several human assemblies, including that of James Watson [8], which was sequenced with 454 unpaired reads. Genomes from African (Yoruba) [106], Asian (Han) [107] and Korean [108] individuals were all sequenced with Illumina technology and mapped to the reference human sequence. For the Asian genome, 487 million reads that did not map successfully were assembled using Velvet, but only a small portion of these (0.36%) assembled into contigs >100bp. The Korean genome included sequencing of targeted BACs in addition to WGS sequencing.

The above-mentioned African genome data were later assembled *de novo* to test the ABySS assembler [80]. The assembly of the 3.5 billion paired-end reads (lengths 35-46 bp from DNA sequence fragments of ~210 bp) yielded an astounding 2.76 million contigs with an N50 length of only 1499 bp. These contigs covered only 68% of the human reference genome. The assembly took almost 4 days using a 168-core compute cluster. This same dataset was later assembled in 40 hours on a 32-core 512 GB RAM supercomputer by SOAPdenovo showing an improved N50 contig length of 4.6 kbp and covering 85% of the human reference genome. The current best published *de novo* assembly of the human genome was also assembled using SOAPdenovo on a total of 90x coverage of an Asian individual [100] producing an N50 contig length of 7.4kbp, These assemblies were computed from older Illumina sequence data (average read length <40bp), and may not be representative of what would be possible using improved technology today (average read length >75bp). However, the lack of paired reads from long (>10kbp) fragments combined with the short read lengths present a very difficult assembly problem given the repetitive nature of the human genome.

Combinations of Sanger and SGS Reads

Several large draft genomes have been published that used a combination of Sanger and short-read sequencing. The draft assembly of grapevine (*Vitis vinifera*, genome size ~500Mb) reported in [109] combined Sanger and 454 sequencing. An initial assembly of the 6.5X coverage Sanger data was created, and the additional 4.2X coverage of 454 sequence was used to correct errors and fill gaps.

The draft genome sequence of cucumber, *Cucumis sativus*, was obtained using a combination of Sanger and Illumina sequencing [110]. Illumina reads represented 68X coverage by pairs from fragment sizes 200, 400 and 2000 bp; while Sanger reads represented coverage of 4X coverage using pairs with insert sizes 2, 4, 6, 40, and 150 Kb. Results for 3 different assemblies—Illumina only, Sanger only, and combined—were reported with the best results obtained, as expected, using the combined data set: N50 contig and scaffold sizes of 19.8 Kb and 1.14 Mb, respectively, and totals of 227 Mb in contigs and 244 Mb in scaffolds. It is interesting, however, that although the N50 sizes of the Sanger-only assembly were much smaller (2.6 Kb contigs and 19 Kb scaffolds), the coverage of the Sanger-only assembly was rather good—204 Mb in contigs and 238 Mb in scaffolds—and better than the Illumina-only assembly (190 Mb in contigs and 200 Mb in scaffolds). The entire genome is estimated to be ~360Mb, indicating that something hampered the assembly, possibly a large number of repeats, or problems with the assembler itself, or with the laboratory protocols. The assembly was accomplished using the authors' own software to assemble the Illumina reads first, and then RePS2 [111] was used to merge the Illumina scaffolds with the Sanger reads.

Panda

The first *de novo*, exclusively SGS assembly of a novel, large genome, that of the giant panda, *Ailuropoda melanoleura*, was recently published by the Beijing Genome Institute [112]. This assembly used only Illumina reads and was done with the SOAPdenovo assembler. 37 paired-end libraries were constructed, with fragment sizes ranging from 150 bp to 10 Kb, totaling 176 Gb of sequence (73X coverage of the 2.4 Gb genome). After filtering out low-quality and redundant reads, 134 Gb (56X coverage) of reads were used in the actual assembly. The final assembly contained 200,604 contigs (of length at least 100bp) totaling 2.25 Gb (93.8% of the genome), with impressive N50 contig and scaffold sizes of 36,728 bp and 1.22 Mb, respectively. There were 5,201 multi-contig scaffolds comprised of 124,336 contigs, and a total of 119,135 gaps with mean gap size of only 455 bp. Thus the total span of all contigs and scaffolds (including gaps) was 2.30 Gb, 95.8% of the genome. The remarkably good quality of this assembly is in large part due to the very high depth of sequence coverage, particularly by long-pairs, and the fact that the genome is much less repetitive than primate and rodent genomes.

An interesting comparison is the dog genome, which has a nearly identical genome size (estimated to be 2.45Gb) and is used for several evolutionary comparisons in the panda paper. The dog genome was assembled at the Broad Institute in 2005 using 7.5X coverage by Sanger sequence data [113]. The N50 contig size for the dog assembly was 180 Kb, and the N50 scaffold size was an impressively large 45 Mb. This rather significant advantage of the dog assembly over the panda assembly is likely due to three factors:

1. Longer Sanger reads—there are many very short gaps in the panda assembly that undoubtedly would be closed by the Sanger reads, which averaged 770bp long.

2. Longer insert libraries—the sequence available for the dog assembly included 2.2 million reads from a 40 Kb fosmid library and 302,000 BAC ends – that cannot be sequenced using current SGS technology.

3. More mature assembly software—the dog assembly paper reported that improvements to the Arachne assembler alone increased contig N50 size from 123 Kb to 180 Kb.

It is interesting to note that the panda download site includes several "gene scaffolds," indicating locations where a gene spans separate scaffolds in the assembly. This information could have been used to combine scaffolds and improve the scaffold N50 value.

Announced but unpublished SGS assemblies

A number of draft SGS assemblies have been announced but have not been published. We describe them here to give a sense of the various strategies currently being used to assemble large genomes.

Cod

An assembly of the cod genome (*Gadus morhua*, genome size ~800 Mb) [http://www.genomeweb.com/sequencing/norwegian-consortium-assembles-annotates-cod-genome-454-data?page=show] was generated from ~27X coverage of

454 reads and included paired libraries from 2, 3, 8, and 20 Kb fragments. Additional Sanger sequencing of BAC ends was also used to confirm the assembly. The N50 scaffold size is reportedly 571 Kb and the scaffolds cover 618 Mb of the genome. The relatively low scaffold coverage and difficulty in accurately estimating the genome size are largely due to the presence of copious repeats in the sequence.

Strawberry

The announced draft assembly of the wild strawberry genome, *Fragaria vesca*, was obtained using a combination of 454, Illumina and ABI SOLiD sequence data (http://strawberry.vbi.vt.edu). The assembly was created by first using CABOG to assemble the 454 data. Then SOLiD pairs were added to grow scaffolds, using the scaffolder within CABOG. Finally a Velvet assembly of the Illumina data was done, and the contigs were mapped to the 454/SOLiD assembly to fill gaps and correct homopolymer SNP errors. The resulting N50 sizes of contigs and scaffolds were 28 Kb and 1.44 Mb, respectively, for this ~220 Mb genome. There are plans to improve the assembly by incorporating data from a restriction digest of a BAC library.

Turkey

The draft assembly announced for the turkey genome (*Meleagris gallopavo*, genome size ~1.1 Gb) was created primarily from a combination of 454 and Illumina sequencing. The 454 sequences included 4 million read pairs from 3 Kb and 20Kb fragments plus 13 million unpaired reads. Illumina sequencing included 400 million 74bp reads from both paired and unpaired. 40,000 Sanger BAC-end sequences also were used in the assembly, which was done with CABOG. The N50 contig and scaffold sizes of the assembly were 12.6 Kb and 1.5 Mb, respectively, with the longest contig being 90 Kb and the longest scaffold 9 Mb. These values are substantially smaller than the corresponding ones for the chicken genome, done with Sanger sequencing: N50 contig 36Kb, N50 scaffold 7.1 Mb, longest contig 442 Kb, longest scaffold 7.1 Mb. On the other hand, the sequencing costs for turkey were estimated to be less than 2.5% of those of chicken. It is also interesting that the average sequence coverage in contigs in the turkey assembly was 17x, even though the overall level of sequence coverage was >30x, indicating that this version of the assembly had difficulty incorporating all available sequence data.

Recommendations for SGS sequencing

The above results make it clear that assemblies using SGS reads alone are substantially worse than what can be done using Sanger sequencing. The two-to-three orders of magnitude cost advantage of SGS, however, will continue to make it much more appealing, and for many genomes it may be the only affordable option. The assembly results now being obtained with SGS sequencing, such as the pioneering panda genome assembly, are scientifically useful: they cover most of the genome and they produce contigs and scaffolds long enough for comprehensive gene-annotation efforts. These results will continue to improve as SGS read lengths grow, paired-end protocols improve, and assembly software innovations appear.

The keys to good assembly results include deep coverage by reads with lengths longer than common repeats, and paired-end reads from moderate (2-5Kb) and large (>8Kb) DNA fragments. Using currently available sequencing technology, the most cost-effective way to obtain sequence coverage with what are effectively 200-300bp reads, is to use paired-end Illumina reads from 200-300bp fragments. With at least 20X coverage in such reads, assemblers using either de Bruijn graphs or overlap graphs should be able to assemble contigs that cover the unique regions of a large genome. To obtain large scaffolds and fill in repeat-induced gaps, a sequencing project should generate a large set of reliable paired-end reads. As long as both ends of a pair map uniquely to contigs, the pair can be used for scaffolding. To fill in scaffold gaps, we need paired reads in which one read is anchored in a contig and its mate falls in the gap. The gap read must be long enough to be assembled with other reads to fill the gap. Thus for paired reads, longer reads have a distinct advantage. For this reason, paired 454 reads will likely provide (today) the most cost-effective type of long-range paired sequences, particularly when 800-bp physical reads become available.

More important than the read length of paired reads, however, is the number of distinct, non-chimeric pairs produced. Protocols to generate paired reads are still being refined, and we have seen sequencing runs that suffered from having very few distinct pairs in them, from having numerous redundant pairs (the same pairs occurred repeatedly), and from having chimeric pairs (the paired sequence were not at the expected separation and orientation in the genome). Until the protocols become standardized, sequencing projects will need to identify experienced laboratories that have demonstrated an ability to generate these sequences.

Sequencing technology is a rapidly advancing field, and third-generation sequencing technologies have been announced for release this year that advertise even longer read lengths and insert sizes than were possible with first generation Sanger sequencing. When these technologies are available, our recommendations and associated cost analysis are likely to change. However, given the extremely high throughput and low cost of the current second-generation sequencing technolgies, we suspect hybrid assemblies composed of second and third generation sequencing technologies will be the norm for years to come.

Table 8. Results from several second-generation sequencing projects.
Summary of inputs and assembly results of recent genome assemblies using SGS reads. Status indicates when the assembly was published; "announced" assemblies have been described publicly but not yet published. Read Cov is the number of estimated genome size units contained in the sum of read lengths. Pair Cov is the same value for the sum of lengths of fragments from which paired reads were sequenced. NR: not reported. GA: Illumina Genome Analyzer.

Key to Notes column:
a. Contig total greater than scaffold total is largely attributable to "single haplotype contigs"
b. Assembly of only Sanger reads
c. Assembly of only GA reads
d. # of scaffolds includes single-contig scaffolds. There were 5,201 multi-contig scaffolds.

		Input Sequence						Assembly								
								Contigs				Scaffolds				
Organism/ Genome Size	Assembler/ Status	Type	Pair Size	Avg Read(bp)	# Reads	Read Cov	Pair Cov	#	N50	Max	Total	#	N50	Max	Total	Note
Human *H. sapiens* 3.0Gb	ABYSS Pub 2009	GA	210bp	35-46	3.5B	45x	120x	2.76M	1.5Kb	18.8Kb	2.18Gb	NR	NR	NR	NR	
Grapevine *V. vinifera* 500Mb	Myriad Pub 2007	Sanger	2-10Kb	579	5.95M	6.9x	21x									
		Sanger	40Kb	460	144K	0.13x	4.4x	58,611	18.2Kb	238Kb	531Mb	2,093	1.33Mb	7.8Mb	421Mb	a
		Sanger	120Kb	369	68K	0.02x	4.2x									
		454	none	169	12.5M	4.2x	-									
Cucumber *C. sativus* 367Mb	RePS2 Pub 2009	Sanger	2-6Kb	439	2.08M	3.35x	9.9x	62,412	19,807	NR	226Mb	47,837	1.15Mb	NR	244Mb	
		Sanger	40Kb	496	339K	0.46x	16.7x									
		Sanger	140Kb	551	33.2K	0.04x	5.6x	NR	2.6Kb	NR	204Mb	NR	19Kb	NR	238Mb	b
		GA	200bp	42	282M	32.5x	76.8x									
		GA	400bp	44	173M	20.6x	94.4x	NR	12.5Kb	NR	190Mb	NR	172Kb	NR	200Mb	c
		GA	2Kb	53	105M	15.3x	286x									
Panda *A.melanoleura* 2.4Gb	SOAP denovo Pub 2010	GA	150	45	1.31B	24.5x	43.3x									
		GA	500	67	917M	25.5x	90.2x	200,604	36,728	434,635	2.25Gb	81,469	1.22Mb	6.05Mb	2.30Gb	d
		GA	2Kb	71	397M	11.8x	192x									
		GA	5Kb	38	505M	8.0x	533x									
		GA	10Kb	35	254M	3.7x	571x									
Strawberry *F. vesca* 220Mb	CABOG & Velvet Announced	454	none	209	7.73M	7.3x	-									
		454	none	368	787M	13.2x	-	16,487	28,072	215,349	202Mb	3,263	1.44Mb	4.1Mb	214Mb	
		454	2.5Kb	193	2.39M	2.1x	6.9x									
		454	20Kb	236	1.58M	1.7x	20x									
		GA	none	76	36M	12.4x	-									
		SOLiD	2Kb	25	1.30M	0.14x	6.4x									
Turkey *M. gallopavo* 1.1Gb	CABOG Announced	454	3Kb	180	6M	1x	8x									
		454	20Kb	195	2M	0.3x	18x	128,271	12,594	90Kb	931Mb	26,917	1.5Mb	9Mb	NR	
		454	none	366	13M	4x	-									
		GA	180bp	74	200M	13x	16x									
		GA	none	74	200M	13x	-									

Chapter 7: Genome Assembly Validation and Visualization

Summary of Contribution

This chapter describes the genome assembly validation and visualization program Hawkeye published in Genome Biology [25], in collaboration with Adam Phillippy, Ben Shneiderman, and Steven Salzberg at the University of Maryland. Hawkeye is an interactive tool for validating and visualizing a genome assembly, which is critical for assessing the quality of an assembly beyond size statistics.

Hawkeye allows a user to interactively explore all levels of an assembly, from high level size and quality statistics, the relationships between contigs within scaffolds, and down to the relationships of reads within contigs, and the supporting evidence for the reads themselves. In addition, Hawkeye computes many quality statistics of an assembly such as the depth of coverage and the compression/expansion of the mate pairs at each position in the assembly, which are essential for revealing potential mis-assemblies. Mis-assembly signatures were further explored in a follow up publication describing the mis-assembly detection pipeline AMOSValidate [20].

Michael Schatz implemented most of Hawkeye and wrote the first draft of the manuscript. Adam Phillippy implemented several features within Hawkeye, especially for enhancing the scaffold view, and contributed to the manuscript. Ben Shneiderman and Steven Salzberg edited the manuscript, and provided guidance for the project.

Abstract

Genome sequencing remains an inexact science, and genome sequences can contain significant errors if they are not carefully examined. Hawkeye is our new visual analytics tool for genome assemblies, designed to aid in identifying and correcting assembly errors. Users can analyze all levels of an assembly along with summary statistics and assembly metrics, and are guided by a ranking component towards likely mis-assemblies. Hawkeye is freely available and released as part of the open source AMOS project http://amos.sourceforge.net/hawkeye.

Rationale

Since the DNA of the first free living organism was sequenced in 1995 [5] using the whole-genome shotgun (WGS) technique [114], hundreds of other organisms, including the human genome [68, 89] and numerous model organisms, have been sequenced using WGS. The relatively low cost and high speed of the WGS method have made it the preferred method of genome sequencing for the past decade. However, achieving results of the highest quality often requires expensive manual analysis with tools that provide only a limited view of the data.

Traditional WGS projects consist of three main steps, namely sequencing, assembly, and finishing. The first stage is highly automated, whereas the latter require painstaking manual curation. In the sequencing stage, fragments of the genome are sequenced by high-throughput laboratory protocols that randomly shear the original DNA molecules into short fragments that are then sequenced. In the assembly stage, sophisticated computer algorithms operated by a human assembly team assemble these short sequences back together into a partially complete 'draft' genome sequence. Finally, in what is usually the most time-consuming stage, human 'finishers' curate the assembly to correct sequencing and assembly errors, and run additional sequencing reactions to fill in the unsequenced gaps. The result of this three-stage process is a high-quality reconstruction of the genome. However, the high cost of the finishing stage, both in terms of time and money, makes it economically unfeasible to finish any genome completely, other than relatively small ones (bacteria and viruses) and the most important model organisms (yeast, nematode, fruit fly, and human). Instead, most genomes are left in the draft stage, where some of the genome remains unsequenced and where even the assembled portions may contain significant errors.

Our primary goals are to reduce the cost of finishing genomes and to increase the quality of draft genomes by providing genome assembly teams and finishers with a visual tool to aid the identification and correction of assembly errors. In addition to these primary goals, our tool - Hawkeye 1.0 - supports numerous other analytical genome tasks, such as consensus validation of potential genes, discovery of novel plasmids, and various other quality control analyses.

Hawkeye blends the best practices from information and scientific visualization to facilitate inspection of large-scale assembly data while minimizing the time needed to detect mis-assemblies and make accurate judgments of assembly quality. Wherever possible, high-level overviews, dynamic filtering, and automated clustering are provided to focus attention and highlight anomalies in the data. Hawkeye's effectiveness has been proven in several genome projects, in which it was used to both to improve quality and to validate the correctness of complex genomes.

Hawkeye can be used to inspect assemblies of all sizes and is compatible with most widely used assemblers, including Phrap [18], ARACHNE [22, 23], Celera Assembler [21], AMOScmp [40], Newbler [91], and assemblies deposited in the National Center for Biotechnology Information (NCBI) Assembly Archive [115].

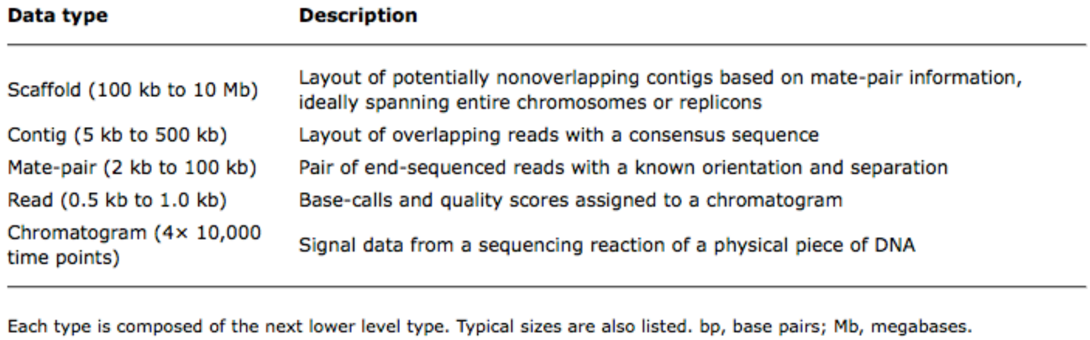
Genome assembly

The need to assemble genomes has inspired many innovative algorithms that have been described in detail elsewhere [18, 21-23, 40, 91, 95, 116]. One of the fundamental steps in any assembly algorithm is to detect how the individual sequences ('reads') overlap one another. The assembler can then use these overlaps to merge reads together, building up longer contiguous stretches ('contigs') of DNA and eventually reconstructing entire chromosomes. More than anything else, repeated sequences in the genome complicate the assembly problem beyond the ability of modern assembly algorithms, and introduce the chance of significant mis-assembly. A repetitive element can be unambiguously assembled using just overlaps only if it is spanned by an entire read. This problem motivated the development of the double-barreled shotgun sequencing approach [117], in which both ends of large fragments are sequenced, creating pairs of sequencing reads with known orientation and separation. A set of these larger fragments of similar size is called a library, and typical sizes range from 2 to 100 kilobases (kb). The end-paired reads, or mate-pairs, can be treated as a large pseudo-read with unknown interior sequence.

State-of-the-art assemblers such as ARACHNE [22, 23], Celera Assembler [21], PCAP [95], and Phusion [116] depend on mate-pairs to untangle false overlaps and bridge unsequenced portions of the genome to form 'scaffolds' of ordered and oriented contigs. Nevertheless, even with high quality reads and mate-pairs, repeat-induced mis-assemblies are common and range from a single incorrect base to large chromosomal rearrangements [24]. Independent validation efforts [118] and additional finishing work [119] for the intensively curated human genome sequence has identified and corrected thousands of mis-assemblies. If the human genome had been left in a draft state, future attempts to identify structural polymorphisms (for example, between human and mouse) would have been difficult if not impossible. The nature and magnitude of mis-assemblies in other genomes is largely unknown, but mis-assemblies are likely to be present in all but the most carefully scrutinized genomes.

Identifying mis-assemblies, as well as avoiding mis-assembly in the first place, is a difficult problem, mostly because of the complexity of the underlying data. The data are not only voluminous and subject to statistical variation, but also error prone because of laboratory error, machine error, and biochemical complications. Consequently, complications can occur at any level of the assembly data hierarchy (Table 9), and therefore all levels of this hierarchy must be collected and analyzed together to verify an assembly effectively. Ignoring even one level of the hierarchy can lead to false assumptions, just as an assembler that ignores mate-pair evidence risks mis-assembly in repetitive regions. Hawkeye is the first analysis tool that enables users to navigate the assembly hierarchy easily, and thus enables a complete and accurate analysis of the assembly.

Table 9. Hierarchy of assembly data types.



Data type	Description
Scaffold (100 kb to 10 Mb)	Layout of potentially nonoverlapping contigs based on mate-pair information, ideally spanning entire chromosomes or replicons
Contig (5 kb to 500 kb)	Layout of overlapping reads with a consensus sequence
Mate-pair (2 kb to 100 kb)	Pair of end-sequenced reads with a known orientation and separation
Read (0.5 kb to 1.0 kb)	Base-calls and quality scores assigned to a chromatogram
Chromatogram (4× 10,000 time points)	Signal data from a sequencing reaction of a physical piece of DNA

Each type is composed of the next lower level type. Typical sizes are also listed. bp, base pairs; Mb, megabases.

Assembly visualization and analysis

Prior work on genome assembly visualization has focused on three different levels of assembly artifacts. The first focuses on the raw signals emitted by sequencing machines as exemplified by the four-color chromatograms displayed at the NCBI Trace Archive [120]. The second is visualization by tools such as Consed [121], which focus on the overlaps and alignment of reads within contigs and allow for detailed inspection of the consensus sequence and its support. The third highlights the mate-pair relationships either between or within contigs, and is commonly displayed as linked arrows or line segments as in the NCBI Assembly Archive [115].

Mate-pair visualization most directly addresses the validation of an assembly by highlighting discrepancies between expected and observed read placements. Clusters of mated reads that are statistically too close together or too far apart are signatures of deletion and insertion mis-assemblies, whereas occurrences of mis-oriented mate-pairs, or reads whose mate-pair are missing, are indicative of other types of mis-assembly. Tools such as Celamy (http://wgs-assembler.sf.net), BACCardI [122], and the clone-middle diagrams proposed by Huson and coworkers [123] effectively highlight these 'unhappy' mates. TAMPA extends this idea further, and provides a positional bound for the mis-assembly event [124].

After a genome is sequenced and assembled, various meta-data, such as gene predictions, are computed and attached to particular intervals on the sequence. Genome browsers such as Ensembl [125], GBrowse [126], CGView [127], and the UCSC Genome Browser [128], lay the features out on either a linear or circular coordinate system as a set of arrows. Additional continuous information, such as GC content or alignment similarity, is often plotted as well. This type of view is widely popular among biologists because it brings multiple sources of evidence into a single display and can be made available over the web. However, these tools are poorly suited for assembly visualization because they cannot capture underlying sequence and assembly data, in part because of the large datasets involved.

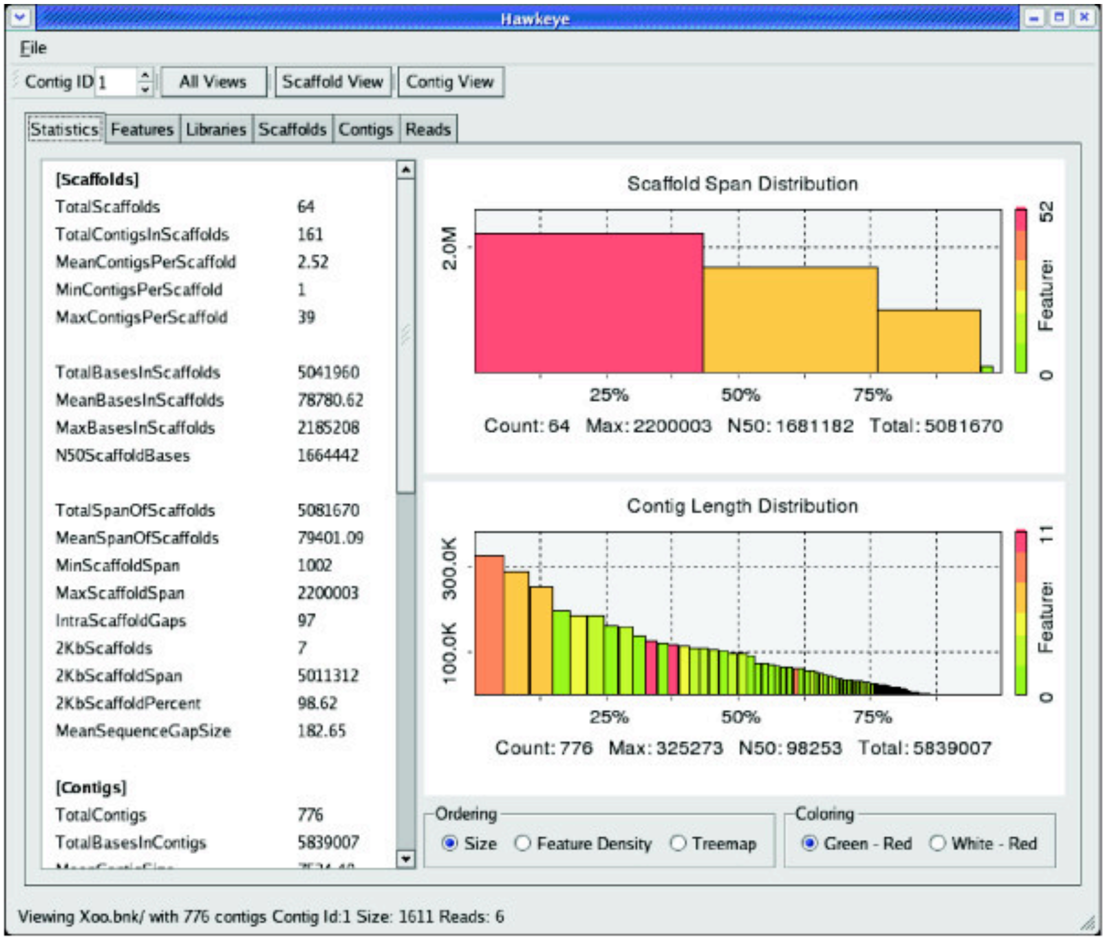
In addition to visualizations, various statistics have been described for the validation of read layouts. The A-statistic [21] compares the distribution of individual reads against a statistical model of random read coverage to detect contigs whose coverage is too deep, suggesting a collapsed repeat. Another measure, the Compression-Expansion (CE) statistic [129], developed by Roberts and coworkers at the University of Maryland IPST Genome Assembly Group, quantifies the degree of compression or expansion for the set of mate-pairs spanning any particular position in the assembly. It is computed on a per library basis as the mean of the insert sizes spanning a position minus the mean value of the library divided by the standard error (the library standard deviation multiplied by the square root of the number of inserts at the position). The expected value of the CE statistic is zero, which occurs when inserts spanning a position have a size distribution that matches the global library distribution. CE values far from 0 outside the interval [-3, +3] indicate an unexpected distribution of insert sizes at that location. Certain mis-assemblies, such as collapsed repeats, generate characteristic insert size distributions with large negative CE values, whereas insertion mis-assemblies produce large positive CE values.

The Hawkeye interface

Launch Pad

Effective overview, ranking, and navigation components are the keys to exploring large data spaces, just as sightseeing is more effective with a map, tour guide, and car. The Hawkeye Launch Pad is the first view presented to the user and it is designed to address these three needs as well as answer the first questions any analyst has about an assembly: 'How big are the contigs?' and 'How good is it?'

To answer these initial questions graphically, Launch Pad displays two N-plots in its initial view: one for contigs and another for scaffolds. An N-plot is a bar graph based on the popular N50 assembly metric (Figure 37). Each bar represents a contig (or scaffold), where the height of the bar represents its length in base pairs and the width represents its length as a percentage of the genome size. This plot gives immediate feedback on both the size and number of contigs contained within the assembly. A few wide steps covering most of the x-axis indicates that the assembly contains a small number of large contigs, whereas many steps of the same size indicate a fragmented assembly. In addition to N-plots, contig and scaffold sizes also can be visualized as a space-filling Treemap [130]. Various other assembly statistics are presented in text-based tables for detailed inspection of high-level assembly quality.



```
[Scaffolds]
TotalScaffolds             64
TotalContigsInScaffolds    161
MeanContigsPerScaffold     2.52
MinContigsPerScaffold      1
MaxContigsPerScaffold      39

TotalBasesInScaffolds      5041960
MeanBasesInScaffolds       78780.62
MaxBasesInScaffolds        2185208
N50ScaffoldBases           1664442

TotalSpanOfScaffolds       5081670
MeanSpanOfScaffolds        79401.09
MinScaffoldSpan            1002
MaxScaffoldSpan            2200003
IntraScaffoldGaps          97
2KbScaffolds               7
2KbScaffoldSpan            5011312
2KbScaffoldPercent         98.62
MeanSequenceGapSize        182.65

[Contigs]
TotalContigs               776
TotalBasesInContigs        5839007
```

Count: 64 Max: 2200003 N50: 1681182 Total: 5081670

Count: 776 Max: 325273 N50: 98253 Total: 5839007

Ordering ● Size ○ Feature Density ○ Treemap

Coloring ● Green - Red ○ White - Red

Viewing Xoo.bnk/ with 776 contigs Contig Id:1 Size: 1611 Reads: 6

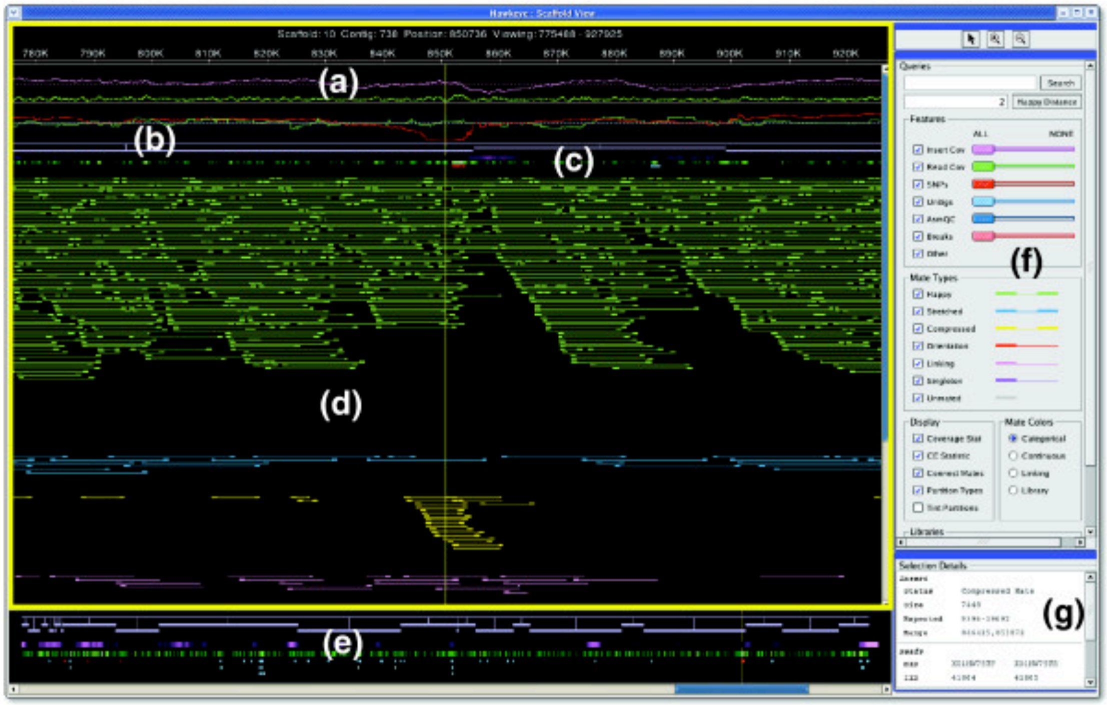
Figure 37. The Hawkeye Launch Pad.
Scaffolds and Contigs are plotted so that the size of the scaffold represents the size of the object. The color of the rectangle indicates the number of mis-assembly features. Details and other abstract visualizations are available through the tabbed interface.

Seo and Shneiderman [131] advocate a generalized rank-by-feature framework for the exploration of multivariate data sets to guide exploration and expedite the discovery process. Hawkeye employs a ranking strategy for contigs and scaffolds that was inspired by the rank-by-feature framework. The first ranking criterion is size, which is implicit in the N-plot described above. The second ranking criterion focuses on contig or scaffold quality, and is encoded in the N-plot by color. Contigs and scaffolds with a high density of mis-assembly signatures (those likely to be mis-assembled) are shaded red in the N-plot, whereas contigs and scaffolds with a low density (those less likely to be mis-assembled) are shaded green. Mis-assembly signatures are regions in the assembly with characteristics indicative of a mis-assembly, such as a cluster of compressed mate-pairs, which suggests a collapsed repeat. Utilities bundled with the software pre-compute some useful mis-assembly indicators such as read polymorphism, alignment breakpoints, and regions with poor insert 'happiness', although users can easily load new metrics via an XML-like interface as additional assembly metrics are invented. Short descriptions of the included metrics are given below in the discussion of the interface components.

Ranking scaffolds and contigs by size and feature density guides users directly to the regions that require the most attention. This minimizes the time needed to pinpoint potential trouble, and provides the ability to drill down to either the scaffold or contig level to examine interesting objects and features in greater detail. Users simply double click in the N-plot to display a new window with the selected contig or scaffold in the more detailed scaffold or contigs views described below. In addition, users can click on other tabs in the Launch Pad to display sortable tables of scaffold, contig, read, library, and feature information. Histograms of insert sizes, GC content, and other attributes are also available that permit quality inspection of other aspects of the assembly.

Scaffold View

The Scaffold View provides an abstract graphical view of the assembly, and is often the most natural view to pursue after identifying an item of interest in the Launch Pad. This view displays the read layout on a per scaffold basis, along with integrated assembly statistics and feature information. The view consists of three panels: the Overview Panel, the Insert Panel, and the Control Panel (Figure 38).



Queries: 2

Features:
- ☑ Insert Cov
- ☑ Read Cov
- ☑ SNPs
- ☑ Unitigs
- ☑ AsmQC
- ☑ Breaks
- ☑ Other

Mate Types:
- ☑ Happy
- ☑ Stretched
- ☑ Compressed
- ☑ Orientation
- ☑ Linking
- ☑ Singleton
- ☑ Unmated

Display:
- ☑ Coverage Stat
- ☑ CE Statistic
- ☑ Connect Mates
- ☑ Partition Types
- □ Sort Partitions

Mate Colors:
- ● Categorical
- ○ Continuous
- ○ Linking
- ○ Library

Selection Details

Inserts

Status	Compressed Mate
Size	7448
Expected	[illegible]
Range	[illegible]

Reads

[illegible]	[illegible]	[illegible]
[illegible]	41864	41863

Figure 38. The Hawkeye Scaffold View.
The scaffold view displays the insert panel, outlined with a yellow border, consisting of (a) plots of statistical information, (b) scaffolded contigs, (c) feature tracks, and (d) inserts. Also displayed are the (e) overview panel, (f) control panel, and (g) details panel. The insert panel displays the details and individual inserts for regions of the scaffold selected in the overview panel, whereas unselected regions are grayed out in the overview. By default, inserts are colored by category (green→happy, blue→stretched, yellow→compressed, purple→singleton). The eye is drawn to the cluster of compressed mates towards the bottom of the insert panel.

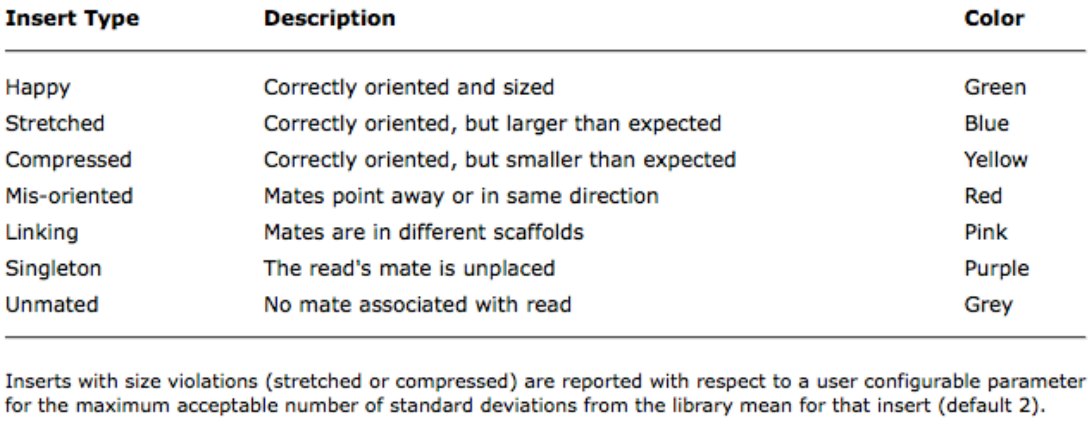
The Overview Panel (Figure 38e) displays the entire current scaffold as a linear ordering of connected contigs along the x-axis, with the assembly features displayed below. The width of the contig boxes and the gaps between them are proportional to the length and separation of contigs, respectively, and contigs are 'scaffolded' together by conjoining lines. Assembly features are laid out below the contigs in multiple tracks. The first two tracks are heat map plots of insert and read depth of coverage that color code coverage regions significantly above or below the mean value. Positions in the assembly with a coverage level near the mean are shaded to blend with the background, whereas positions significantly deviating from the mean, such as in collapsed repeats, are given a contrasting color to the background. Interval features are displayed in additional tracks below the coverage tracks. These discrete features are preloaded with the assembly data and represent arbitrary regions of interest, such as regions with mis-assembly signatures, or sequence characteristics such as gene models, and so on. Large features or clusters of different feature types demand attention and take precedence over small, isolated features. All feature tracks can be filtered by value (score or size), allowing users to focus their attention on the most egregious or interesting features.

The Insert Panel (Figure 38d) provides a detailed look of the region selected in the Overview Panel. Users select regions to investigate in the Insert Panel with a magnifying glass tool, or by adjusting the scroll bars beneath the overview. At the top of the Insert Panel, statistical line plots (Figure 38a) display the depth of read (green) and insert coverage (purple) along with the CE statistic value for each library along the scaffold. The coverage tracks will vary from 0 to the maximum depth of coverage, but the CE statistic track is fixed to display values in the range [-6,6] because the CE statistic value will be near 0 except in mis-assembled regions. Users can read the precise coverage or CE values by clicking on the plot that displays the value in the details panel. Extreme values or variation in any of the statistical tracks can indicate mis-assembly or other assembly issues and encourages users to look at statistically anomalous regions more thoroughly.

A plot of the depth of k-mer coverage is optionally plotted overlaying the read and insert coverage. It displays the number of occurrences in the set of reads, of the substring of length k starting at each position along the contig consensus sequences. K-mer coverage spikes reveal the repeat structure of the genome and highlights regions of potential mis-assembly. Correctly assembled unique sequence has k-mer coverage approximately equal to the read coverage, whereas repeat sequences have k-mer coverage that is a function of the number of copies of the repeat, regardless of whether the repeat has been correctly assembled.

Below the contig and feature tracks lies the layout of the sequencing reads (Figure 38d). The reads are drawn as colored boxes connected to their mate by a thin line. If it is not possible to connect a read with its mate because of misplacement or other issues, a thin line is drawn proportional to the expected size of the insert. Using a size threshold based on the standard deviation of the library (called 'happiness' within the interface), and the orientation constraints of the mate-pair relationship, inserts are categorically grouped to enhance visibility and emphasize clusters of unexpected sizing or inconsistent mate-pair orientation (Table 10). Unfortunately, subtle mis-assemblies can be overlooked if most of the mis-assembled inserts fall within the happiness threshold, and so an alternative continuous coloring scheme is available. In this scheme, happy inserts are shaded into the background to make them less visible, while stretched and compressed mates are given brighter colors corresponding to how compressed or expanded they are. Positions spanned by inserts that are even slightly skewed will show as clusters of bright, similarly colored inserts, indicating a possible problem (Figure 39). This view is more sensitive than setting arbitrary thresholds and has proven to be quite effective for identifying mis-assemblies missed by categorical analysis.

Table 10. Categorization of insert happiness.



Insert Type	Description	Color
Happy	Correctly oriented and sized	Green
Stretched	Correctly oriented, but larger than expected	Blue
Compressed	Correctly oriented, but smaller than expected	Yellow
Mis-oriented	Mates point away or in same direction	Red
Linking	Mates are in different scaffolds	Pink
Singleton	The read's mate is unplaced	Purple
Unmated	No mate associated with read	Grey

Inserts with size violations (stretched or compressed) are reported with respect to a user configurable parameter for the maximum acceptable number of standard deviations from the library mean for that insert (default 2).

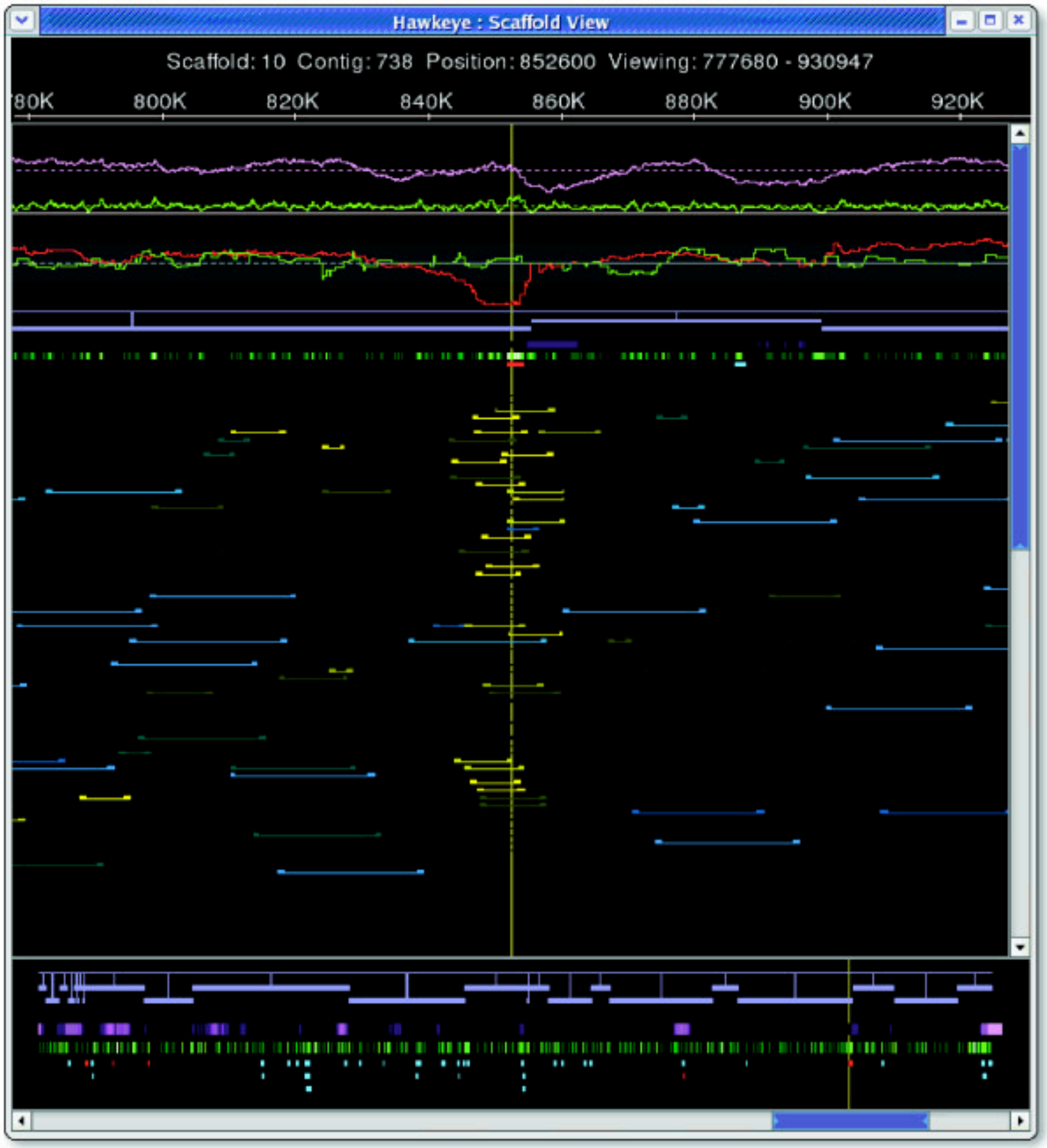
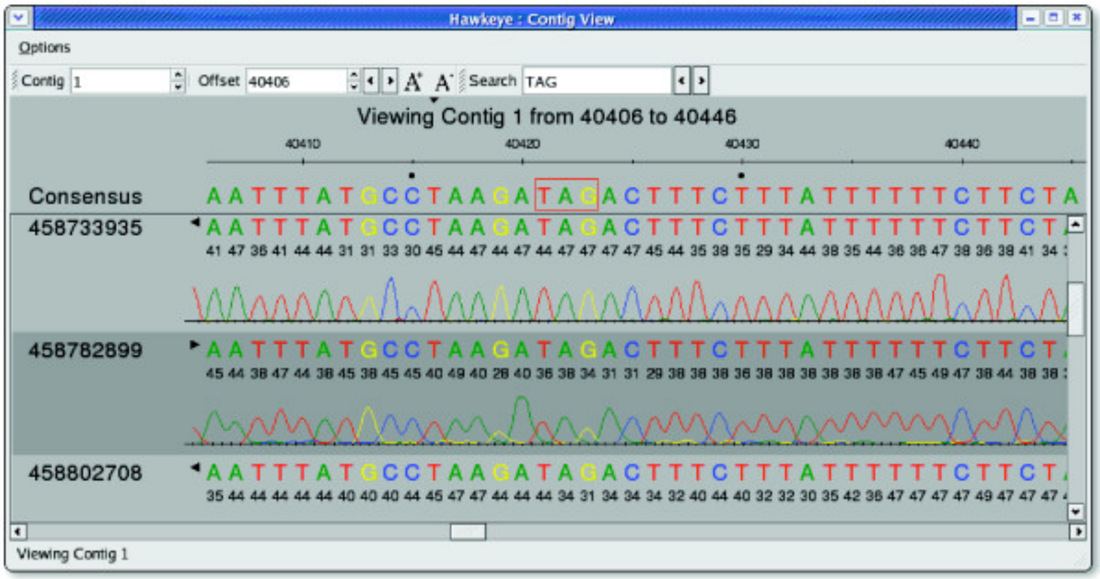

Hawkeye : Scaffold View
Scaffold: 10 Contig: 738 Position: 852600 Viewing: 777680 - 930947
780K
800K
820K
840K
860K
880K
900K
920K

Figure 39. Mis-assembly detection in Scaffold View.
Continuous coloring in the Scaffold View displaying a region of Xanthamonas oryzæ. Slightly compressed mate-pairs are colored increasingly bright yellow as they deviate from the mean. Slightly expanded pairs are also visible in blue, but are uncorrelated and most likely caused by inexact library sizing.

The coordination of multiple forms of evidence combined with user interaction is the key to the Scaffold View's effectiveness. Statistical spikes, feature clusters and contrasting insert colors combine to guide users to the important areas of the assembly. However, the underlying DNA sequences and chromatogram traces are absent from this view, and so another level of detail is required. This is handled by the Contig View, which is essentially a vertical slice of the Scaffold View displaying the read tiling in full detail with base-calls and chromatogram traces. The two views are synchronized, so that a user click in the background of the Insert Panel centers the Contig View to that position.

Contig View

Similar to the Scaffold View, the Contig View also displays the read tiling, except the abstract rectangles from the Scaffold View are replaced with the actual strings of base-calls for each read (Figure 40). The reads supporting the consensus at each position are arranged so that their individual bases are aligned vertically, including gaps inserted by the assembler to maintain the alignment. Consensus positions in which the underlying reads disagree are marked, and dissenting base-calls are highlighted.



Contig 1 Offset 40406 Search TAG

Viewing Contig 1 from 40406 to 40446

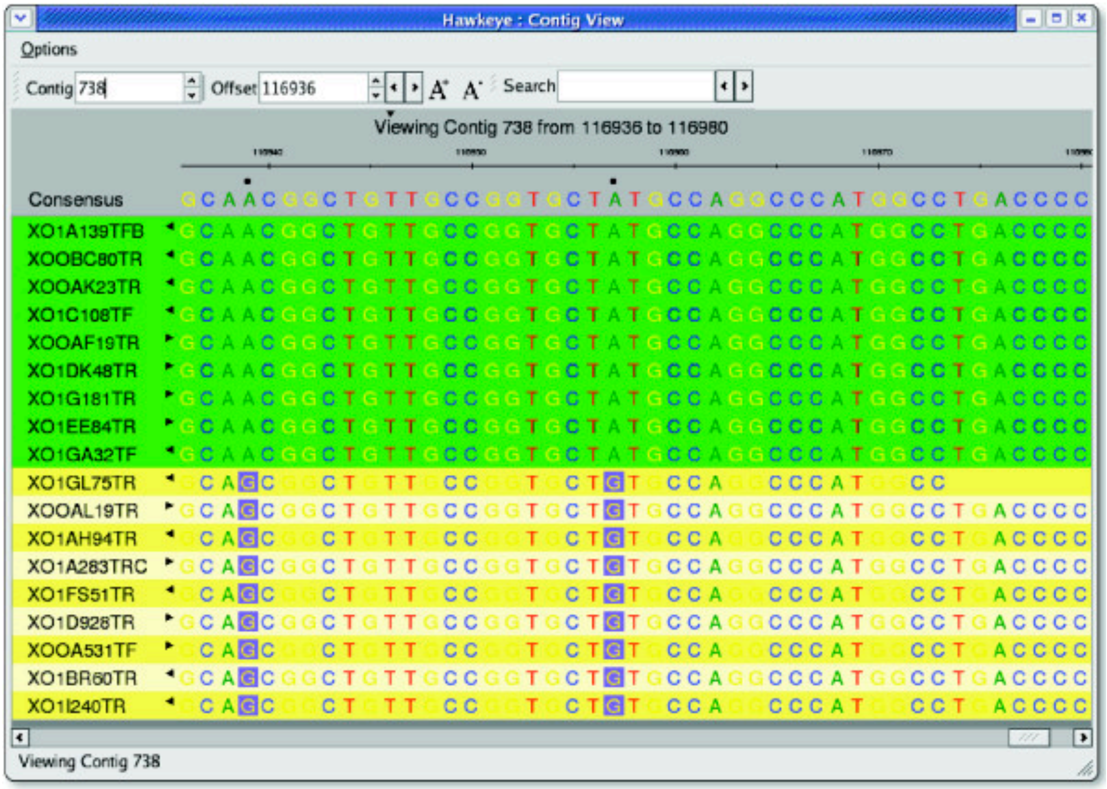
```
Consensus    AATTTATGCCTAAGATAGACTTTCTTTATTTTTTCTTCTA
458733935   ◂AATTTATGCCTAAGATAGACTTTCTTTATTTTTTCTTCT
             41 47 36 41 44 44 31 31 33 30 45 44 47 44 47 44 47 47 47 47 45 44 35 38 35 29 34 44 38 35 44 36 36 47 38 36 38 41 34
458782899   ▸AATTTATGCCTAAGATAGACTTTCTTTATTTTTTCTTCT
             45 44 38 47 44 38 45 38 45 45 40 49 40 28 40 36 38 34 31 31 29 38 38 38 36 38 38 38 38 38 38 47 45 49 47 38 44 38 38
458802708   ◂AATTTATGCCTAAGATAGACTTTCTTTATTTTTTCTTCT
             35 44 44 44 44 44 40 40 40 44 45 47 47 44 44 44 34 31 34 34 34 32 40 44 40 32 32 30 35 42 36 47 47 47 47 49 47 47 47
```

Viewing Contig 1

Figure 40. The Hawkeye Contig View.
Quality values and chromatograms are displayed on demand in the Contig View to confirm a potential stop codon outlined in red in the consensus.

The Contig View can also display base-call quality values and chromatogram traces (if available) to examine discrepancies in more detail. Quality values are loaded with the assembly data, and the traces are either loaded from the file system or downloaded on-the-fly directly from NCBI Trace Archive or other archives. In the Contig View, the chromatograms may be compressed or expanded to ensure consistency between the reads, but double-clicking on a read displays the undistorted chromatogram for the selected read in a new window. Human examination of the trace data is often necessary to confirm conflicting base-calls as sequencing error or genuine single nucleotide polymorphisms (SNPs). False SNPs caused by sequencing or base-calling errors are quite common and can be largely ignored, whereas SNPs supported by the chromatogram or occurring in multiple reads at the same position must be examined more closely.

When two or more reads share a discrepancy from the multi-alignment, we call this a correlated SNP. Because most SNPs are caused by random sequencing error, it is highly unlikely that a random error in two separate experiments will occur at exactly the same position, especially if those bases have high quality values. Although biological or biochemical explanations can sometimes account for this correlated error, it is commonly caused by mis-placed reads from different positions in the genome, especially for haploid organisms. One very common cause of a correlated SNP is the collapse of two near-identical copies of a repeat into a single copy by the assembler. Because both copies of the repeat should have been sampled evenly, the same number of reads should be present for each copy, and the reads will partition into two equally sized groups distinguished by the differences in the multiple alignment. In addition to flagging these regions in the Scaffold View, the Contig View supports the separation of these groups via on-the-fly clustering of correlated discrepancies. Clicking the consensus base in question sorts the underlying reads into groups based on the base-calls at that position (Figure 41).



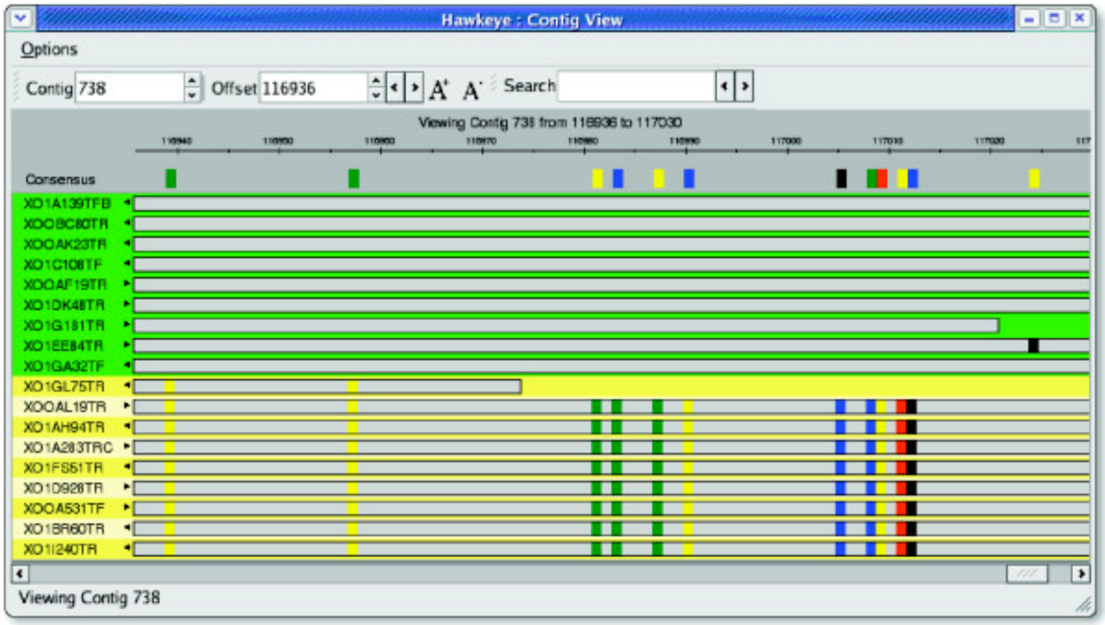
```
Viewing Contig 738 from 116936 to 116980

Consensus    GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XO1A139TFB  ◄GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XOOBC80TR   ◄GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XOOAK23TR   ◄GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XO1C108TF   ◄GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XOOAF19TR   ►GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XO1DK48TR   ►GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XO1G181TR   ►GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XO1EE84TR   ►GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XO1GA32TF   ◄GCAACGGCTGTTGCCGGTGCTATGCCAGGCCCATGGCCTGACCCC
XO1GL75TR   ◄GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCC
XOOAL19TR   ►GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCCTGACCCC
XO1AH94TR   ◄GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCCTGACCCC
XO1A283TRC  ►GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCCTGACCCC
XO1FS51TR   ◄GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCCTGACCCC
XO1D928TR   ►GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCCTGACCCC
XOOA531TF   ►GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCCTGACCCC
XO1BR60TR   ◄GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCCTGACCCC
XO1I240TR   ◄GCAGCGGCTGTTGCCGGTGCTGTGCCAGGCCCATGGCCTGACCCC

Viewing Contig 738
```

Figure 41. SNP sorted reads in the Contig View.
Clicking in the consensus automatically clusters the reads into correlated groups by sorting and coloring the reads by their base at that position. SNP, single nucleotide polymorphism.

In addition to SNPs correlated by row, they also can be correlated across multiple columns of the multi-alignment. In this case, it can be difficult to fit all the correlated columns on the screen at once, and so the Contig View employs a semantic zooming mechanism for viewing large regions of the multi-alignment simultaneously. Zooming out reduces the size of the base-calls until the text becomes unreadable. At this point, the view switches to a 'SNP barcode' view, inspired by the software DNPTrapper [132]. In this view, agreeing bases are blended with the background to remove them from view, and only the disagreeing bases are colored (Figure 42). Reads that share the same pattern of SNPs are quickly identified and can be clustered together as before.



Contig 738

Offset 116936

Viewing Contig 738 from 116936 to 117030

Viewing Contig 738

Figure 42. Semantic zooming in the Contig View.
Semantic zooming shifts from displaying the individual base pairs in reads to a compact abstract SNP-Barcode in which only bases that disagree with the consensus are colored thus displaying a wider range of a contig. SNP, single nucleotide polymorphism.

Results

We designed Hawkeye to enhance understanding of genome assemblies and to assist in the detection and correction of assembly errors. Below we outline a sample of analysis tasks possible with Hawkeye.

Assembly validation

We applied Hawkeye to inspect potential mis-assemblies systematically in the draft assembly of a recent genome sequencing project for the bacterium Xanthamonas oryzæ pv.oryzicola [133]. The 4.8 megabase (Mb) genome was sequenced in 62,229 end-paired shotgun reads representing approximately 9× coverage of the genome. The reads were assembled with Celera Assembler using default parameters. Over 96% of the assembly was contained in three large scaffolds, each over 1 Mb in size. Hawkeye uncovered a number of mis-assemblies that were present in the draft assembly.

One mis-assembly was discovered near the end of a contig in the third largest scaffold. The evidence for the mis-assembly was threefold: elevated read coverage, the presence of compressed mate-pairs, and correlated SNPs within the reads. As explained above, this combination of evidence suggests that the reads from two or more instances of a repeat have been collapsed into a single instance.

The Scaffold View has strong support for the hypothesis of a collapse. It includes a spike in read coverage in this region, to more than twice the mean (Figure 39). In the default categorical view, only one mate-pair is classified as compressed using a threshold of three standard deviations from the mean. However, the continuous insert coloring reveals a cluster of moderately compressed mates in this region (colored yellow). Furthermore, clicking in the CE statistic plot shows the CE statistic for this region falls to -6.36, which is well below the threshold of -3.0 for likely compression type mis-assembly. Finally, the red features spanning the area indicate a high level of read polymorphism. The coordinated Contig View shows two distinct clusters of reads, probably representing the two repeat copies that were collapsed together (Figure 42).

Following our discovery, we created a second assembly using just the reads and mates from the collapsed region with stricter parameters for the assembler, which required a greater degree of similarity between overlapping reads. This local assembly was inspected, and did not have any mis-assembly signatures. A contig alignment dot plot generated by Nucmer [14] revealed that the collapsed repeat did not occur exactly in tandem, but contained an additional approximately 500 base pairs of unique sequence between the two repeat copies that was missing from the original assembly. The mis-assembled region was replaced with the corrected local assembly using the AMOS tool stitchContigs (http://amos.sf.net), providing an accurate consensus sequence for gene annotation.

Assembly diagnostics

Hawkeye also has proved useful for improving assemblies globally by explaining why assemblies are worse than expected. The initial assembly for the Bacillus megaterium sequencing project (Ravel J, personal communication) had a surprisingly large number of small scaffolds given the expected read and insert coverage levels. The genome size was estimated at about 5 Mb, and the 74,000 shotgun reads should have provided 12× read coverage and nearly 50× insert coverage of the genome. Despite adequate sequencing, the assembly had on average less than 10× read coverage and no scaffold larger than 1 Mb. Furthermore, over 12% of the reads were left out of the assembly (called 'singletons').

We explored the source of the fractured assembly by inspecting the largest scaffold. We quickly discovered a high percentage of singleton mates (reads in the scaffold whose mates were singletons). Clusters of singleton mates can be caused by deletion mis-assemblies, but the singleton mates in this assembly were distributed evenly throughout the scaffold, and were not correlated with other mis-assembly features. Another likely cause of singleton mates is low read quality, below what the assembler will tolerate. For example, with default parameters, Celera Assembler will not assemble together reads if they disagree by more than 1.5%. To test for low read quality, we examined the largest contig using Hawkeye's SNP barcode view with a quality value heat map. As suspected, the ends of the reads were lower quality than the interior, but we were surprised to find clusters of differences near the ends of individual reads. Furthermore, these differences were not correlated and all were deletion events.

This combination of evidence suggested that the base-caller systematically missed peaks near the ends of chromatograms. These missed peaks fell in relatively low quality regions, so we re-trimmed the reads with more aggressive parameters, and re-assembled the genome. This re-trimming reduced the number of singleton reads to fewer than 2% and greatly improved scaffold and contig sizes. In a follow-up investigation, we discovered that the base-calling software in the sequencing pipeline had been updated recently, but the trimming software had not been appropriately recalibrated.

Discovery of novel plasmids

The assembly of Bacillus megaterium also was interesting because the organism was thought to have seven plasmids in addition to the main chromosome of the organism. The complete sequence for four plasmids was previously available, but the sequences for the others were not. After assembly, we inspected the scaffolds using Hawkeye to find the novel plasmids by searching for circular scaffolds. In a linear version of a circular scaffold, reads near each end of the scaffold will be oriented such that their mate would fall outside the scaffold, while instead those mates will appear within the scaffold at the opposite end. In addition, these mates will appear in Hawkeye as mis-oriented mates occurring on the ends of the scaffold without the presence of other mis-assembly evidence. We identified seven scaffolds with this structure, and four matched the known plasmid sequence. The additional circular scaffolds are the three novel plasmids (laboratory confirmation is pending).

Consensus validation

During the genome sequencing and annotation of the 160 Mb parasite Trichomonas vaginalis [134] a large number of 'split genes' were identified. In a split gene, two adjacent open reading frames (ORFs) are separated by a stop codon, but in other organisms' homologous genes the entire region is a single ORF forming a single functional gene.

We attempted to confirm the correctness of these split genes by ruling out the possibility of mis-assembly and confirming the accuracy of the consensus sequence. The split gene annotations were loaded as features into Hawkeye. We then systematically checked for potential mis-assemblies near these genes in the Scaffold View, but found only happy inserts and no evidence of mis-assembly. In the Contig View, we examined the chromatograms and quality values for base-calls in these regions, looking particularly for mis-calls that would have introduced frame shifts or false stop codons. After finding no consensus discrepancies or signs of mis-assembly, we concluded the sequence was correct, and the genes had not been mis-assembled. The reads in this region came from several different genomic libraries, providing further evidence that the split genes are not an artifact of library construction.

Discussion

Cognitive psychologist and computer science researcher Herbert Simon stated, 'Solving a problem simply means representing it so that the solution is obvious' [135]. In this spirit, Hawkeye strives to provide a visual, manipulable interface to help finishers understand and reason about complex assembly data. In addition to providing a useful interface for the examination of assembly data, Hawkeye further supports the analytical process by providing statistical and computational data analysis, enabling users both to reduce data complexity and to form accurate judgments.

Hawkeye addresses the issues of scale and complexity by guiding users to the most likely areas of mis-assembly, and adhering to the visual information seeking mantra: overview first, zoom and filter, then details-on-demand [136]. The main application window, or 'Launch Pad', acts as a global overview by displaying summary assembly statistics, along with graphs and sortable tables of assembly information. The ranking component of this display encourages users to inspect regions of the assembly in order of importance: largest to smallest and low quality to high quality. The more detailed 'Scaffold View' is capable of displaying an entire contig or scaffold and its underlying reads on a single screen for scaffolds spanning 10+ Mb of sequence and 100,000+ reads. Alternatively, users can zoom in and filter the display to focus on particular regions of interest. Finally, the lowest level assembly information is displayed in the coordinated 'Contig View', displaying the consensus sequence, read-tiling, base-calls, and supporting data. Coordination among these three views - Launch Pad, Scaffold View, and Contig View - allows for very efficient top-down analysis of even the largest assemblies. It leads the user to a natural analytic progression: discern high-level quality from statistics and features; examine a poorly scoring scaffold for mis-assembly at the clone-insert level, looking for uneven insert distribution and improperly sized or mis-oriented mate-pairs; examine possible mis-assemblies in more detail at the base-call and chromatogram level, looking for correlated discrepancies supported by chromatogram traces; and confirm or refute hypothesis of mis-assembly.

After confirming the presence of mis-assemblies, users have a choice of methods for correcting the assembly. If there are numerous or systematic errors, the best solution is often to reassemble the genome after adjusting the assembler parameters, such as adjusting the read trimming to be more conservative, or requiring a higher degree of similarity between overlapping reads to correct for collapsed repeats. If the errors are more localized, such as collapsed repeats or mis-placed reads, users can correct the individual mis-assemblies with the companion AMOS tools (http://amos.sf.net) or with other third party tools. Other assembly complications, such as high levels of sequencing error, can be automatically corrected with tools such as AutoEditor [137].

Hawkeye combines computational predictors with interactive visualizations to enable efficient and accurate human inspection of assembly data, resulting in decreased verification costs and higher quality data for the scientific community. We have utilized its ranking component to detect the presence of localized mis-assemblies in various genome assemblies, and have used its abilities to verify the correctness of reassemblies. We have also used it to improve genome assemblies globally by identifying systematic problems with read trimming, which had fragmenting assemblies. Finally, we have positively identified biologically interesting phenomena such as novel plasmid sequences, and demonstrated how Hawkeye can be used to confirm the base-call level consensus sequence of contigs to verify the accuracy of unusual gene structure.

Hawkeye 1.0 emphasizes visual presentation, but future versions should include capability to edit individual bases, manipulate contigs, and interactively mark regions for further attention. We also plan to improve visualizations for new sequencing technologies such as the display of flowgrams used in 454 sequencing. Finally, we also plan to improve support for gene annotation tasks, including displaying the translated amino acid sequence in addition to the DNA sequence and enhanced support for displaying gene models with introns.

Hawkeye is a desktop GUI application written in C++, and requires the Qt graphics library, which is freely available from Trolltech (http://www.trolltech.com/products/qt/). Otherwise, users can load and analyze assemblies without any other dependencies on Linux/Unix, Microsoft Windows (with Cygwin), and Mac OS X based computers. Desktop machines with 1 GB of RAM will easily accommodate small to mid-sized assemblies (<200,000 reads), whereas more RAM may be necessary for larger assemblies to remain responsive. The user manual and source code for Hawkeye are available from the Hawkeye website (http://amos.sf.net/hawkeye).

Chapter 8: De novo Assembly of Large Genomes using Cloud Computing

Summary of Contribution

This chapter describes the new de novo genome assembly program Contrail, created in collaboration with Dan Sommer, David Kelley and Mihai Pop at the University of Maryland. It is currently unpublished, but we hope it will ultimately become a published result.

Contrail addresses the problem that few assemblers are capable of assembling large genomes from short reads, primarily because large genome assembly requires tremendous computational resources. Addressing this limitation, Contrail uses MapReduce to parallelize computation across a cluster of commodity computers. Contrail's assembly methods are based on existing assembly algorithms, but required inventing entirely novel and non-trivial parallel algorithms to manipulate an extremely large de Bruijn graph of the read sequences. As a result, Contrail is perhaps the third assembler created that is possible of assembling a human genome from short reads and is probably the only assembler that can do so on commodity resources.

Michael Schatz designed and implemented the entire Contrail system described here, except for the read correction method developed by David Kelley. Michael Schatz wrote the manuscript and performed the experiments, except for the error correction of the E. coli dataset that David Kelley executed. Dan Sommer contributed in many discussions to the design, and is developing an overlap-graph based component of Contrail that is not described here. Mihai Pop contributed in many design discussions, and provided guidance to the overall system.

Abstract

Current DNA sequencers can sequence the equivalent of multiple copies of an entire human genome in a few days and at low cost, but analyzing these data remains a difficult challenge. In particular, de novo genome assembly is essential to many sequencing projects, but to date requires compute resources out of reach for most researchers when assembling large mammalian sized genomes. Addressing this critical need, we have developed a new genome assembler Contrail that harnesses the power of cloud computing to scale genome assembly to large genomes on commodity resources. Contrail is available open-source at http://contrail-bio.sf.net.

Introduction

Sequencing the genome is fundamental for many biological analyses, and has become a standard technique for unlocking the genetic content of an organism. Current DNA sequencing technology is limited to sequencing relatively tiny fragments of DNA ranging from 25bp to 1000bp, although many billions of fragments can be sequenced in high throughput and at low cost from random positions in the genome [1]. Consequently, a genome can be reconstructed from short sequences by computationally assembling the sequences originating from overlapping positions in the genome [138].

Genome assembly is challenging because of errors and other technical artifacts of the preparation and sequencing. More fundamentally, repeated sequences in the genome complicated assembly by obscuring which sequences should be assembled together [20]. Large mammalian-sized genomes are especially challenging because in addition to the tremendous volume of data required, they contain highly repetitive sequences that are nearly impossible to disambiguate. As such, except for the very smallest and simplest genomes, a genome assembly will not contain a single contiguous sequence (contig) for the entire genome or for each chromosome, but will instead consist of a set of contigs for the regions that could be resolved. The success of an assembly is therefore typically evaluated by the size and accuracy of the contigs produced.

Several genome assemblers were developed to assemble large repetitive genomes from traditional Sanger sequencing reads, including the Celera Assembler [21] and Arachne [22, 23], and later for 454 reads such as the assembler Newbler [91]. These assemblers assemble genomes in three major phases. In the first phase, the assembler constructs an overlap graph, where nodes represent reads, and weighted edges connect overlapping reads. In the second phase, these assemblers analyze the overlap graph, and conservatively assemble unambiguous regions of the genome into relatively small, but correctly assembled contigs. In the final phase, these assemblers analyze the mate-pairs, which are pairs of reads generated from opposite ends of a single fragment, to resolve ambiguities and link contigs into scaffolds. Mate-pairs constrain the order and orientation of contigs, and the assembler uses these relationships to construct a linear scaffold. If the contigs in the scaffold overlap, the scaffolder will merge the contigs together. Otherwise, the sequence between the contigs will be unknown, but the size of the gap will be approximately known.

The massive volume of data and short read lengths from the current second generation DNA sequencing machines sold by Illumina (http://www.illumina.com), Applied Biosystems (http://www.appliedbiosystems.com), and Helicos (http://www.helicosbio.com) has spurred development of a new class of genome assemblers specifically tuned to accommodate their data characteristics. Many of the new assemblers, such as Velvet [96, 97], ABySS [80], and SOAPdenovo [100] attempt to reconstruct the genome by constructing and simplifying the de Bruijn graph of the read sequences. Nodes in the de Bruijn graph represent substrings of the reads, and directed edges connect nodes of consecutive substrings. Genome assembly is then modeled as finding an Eulerian tour through the graph, or failing that because there are multiple Eulerian tours, finding unambiguous paths within the graph representing individual contigs. After the initial contigs are assembled from the de Bruijn graph, the assemblers resolve ambiguities and build scaffolds using the mate-pairs much like how the overlap graph based assemblers operated.

The de Bruijn graph framework has several advantages over the overlap graph framework when used to assemble short reads, including efficient computation of overlapping reads and robust handling of sequencing errors [96]. As such, these assemblers have successfully assembled many small genomes from short reads. However, the de Bruijn graph is also more challenging to analyze computationally, and usually requires the entire graph is available in main memory [96, 97]. Consequently, these assemblers have had limited adoption assembling larger mammalian-sized genomes because the de Bruijn graph of genomes of this size contain billions of nodes and billions of edges. For example, the recently published de novo assembly of the panda and human genomes required a “supercomputer” with 512 GB of main memory [100]. Furthermore, the parallel assembly of 120Mbp *D. melanogaster* genome completed in just 4 hours [81], but required an extremely high end BlueGene/L supercomputer with 512 nodes, not available to most researchers.

Addressing this limitation, we have developed a new open source genome assembler Contrail that harnesses the power of cloud computing for the de novo assembly of large genomes from short sequencing reads. Contrail uses the open source implementation of MapReduce [82] called Hadoop (http://hadoop.apache.org) for parallel genome assembly, including on clusters of commodity computers leased from a 3rd part commercial cloud providers.

For the initial assembly, Contrail builds contigs using the de Bruijn graph framework, using many operations to simplify the graph and remove spurious nodes and edges introduced by sequencing errors from the graph. After the initial contigs are constructed, Contrail then uses mate-pairs to resolve ambiguities and build scaffolds. Unlike the older programs for assembling large genomes, which require large servers or RAM resources, Contrail uses Hadoop to efficiently transform the graph across dozens or even hundreds of computers, using minimal system memory per machine. Contrail’s contigs are of similar size and quality to those generated by other leading assemblers when applied to small (bacterial) genomes, but provides vastly superior scaling capabilities when applied to large genomes. Contrail is available open source at http://contrail-bio.sf.net.

Results

Bacterial Assembly

We first evaluated Contrail by assembling E. coli K12 substrain MG1655 using 20.8 million paired-end 36 bp Illumina reads (accession no. SRX0000429). Contrail created 300 contigs ≥ 100bp long, and a N50 contig size of 54,807 bp using an initial node size of 27bp. The assembly was computed on a relatively small cluster of 10 dual core computers hosted at the University of Maryland, providing a total of 20 3.2 GHz Intel Xeon cores and 3.5 TB of local disk.. The runtime was approximately 8 hours (including ~.5 hours for error correction), although much of the time was dominated by overhead that is amortized on a larger genome.

This exact dataset has been assembled by several other recent assemblers including ABySS [80], SOAPdenovo [100], Velvet [96], EULER-SR [139], SSAKE [140] and Edena [141] (reported in [80, 100]), so the relative assembly quality could be evaluated, as shown in Table 11. The contig N50 size is a standard metric for evaluating contig sizes, and is the size such that 50% of the genome is present in contigs of this size or larger. The complete reference genome is available (RefSeq accession no. NC_000913), so we also assessed the accuracy of the contigs by aligning the contigs to the reference using the program nucmer [14] (using option --maxmatch) and then filtering the alignments using delta-filter –q to find the best mapping for each of Contrail's contigs. Following the thresholds used in previous studies, contigs that aligned for less than 95% of their length or at less than 95% identity were considered to be incorrect (1 contig spans the origin and is not considered incorrect). By these criterions, Contrail's contigs are of similar size and quality to other leading assemblers.

Table 11. Comparison of recent E. coli assemblies.

Assembler	Contigs ≥ 100bp	N50 (bp)	Incorrect contigs
Contrail PE	300	54,807	4
Contrail SE	529	20,062	0
SOAPdenovo PE	182	89,000	5
ABySS PE	233	45,362	13
Velvet PE	286	54,459	9
EULER-SR PE	216	57,497	26
SSAKE SE	931	11,450	38
Edena SE	680	16,430	6

SE indicates single end data only, before any scaffolding. PE indicates paired-end analysis, including scaffolding. Total length indicates total length of contigs at least 100bp.

Human Genome Assembly

Next we evaluated Contrail by assembling the genome of an African male individual (HapMap DNA identifier NA18507) (International HapMap Consortium 2003, 2007), using 3.5 billion reads downloaded from the NCBI short read archive (accession no. SRA000271). The read lengths ranged from 36 to 42 bp with a median fragment size of 210 bp. All together, the data consist of an average 42X coverage of the human genome in ~176 GB of compressed sequence data (gzipped fastq format). Previous studies of this data set found 72% of the reads aligned perfectly to the reference genome, and estimated the per-base error rate at 1.4% [80].

Given the large volume of data, the assembly was executed on the much larger cluster managed by the NSF Cluster Exploratory (CLuE) program. This cluster consists of approximately 450 nodes, each with a dual core 2.8 GHz Xeon processor, 8 GB of RAM, and two 400 GB disks. However, 5-10% of the machines are offline for maintenance at any given time, and the cluster is shared with many other users. During the assembly, Contrail executed on as many as 500 cores in parallel, but the number of cores used varied dramatically depending on the number of other users using the cluster. The assembly was interrupted after ~65 hours after building the initial graph and correcting deadend tips and bubble popping described below (preassembly read error correction was also not performed). An earlier version of Contrail assembled this dataset up to scaffolding in 74 hours on a dedicated cluster with 188 cores at the University of Wisconsin. By comparison, ABySS required approximately 96 hours on a cluster of 168 cores connected by a high speed interconnect, and SOAPdenovo required approximately 40 hours on 40 cores, but also had peak memory usage over 140GB of RAM.

Table 12. Contrail human genome assembly statistics

Stage	MR Cycles	Contigs ≥ 100 bp	N50 (bp)
Construction & Compression	23	192,073	<100
Error Correction			
- Tip Removal	73	5,080,285	650
- Bubble Popping	36	4,285,080	923

The initial de Bruijn graph used a k-mer length of 27bp, and consisted of more than 10 billion nodes. The initial compression reduced this to ~1.05 billion nodes, ranging in size from 27bp to 303 bp, and 192 thousand nodes were at least 100bp long. Trimming dead end tips removed 65 million tips less than 54bp long, and bubble popping removed 1.5 million bubbles within 5% sequence identity. This error corrected assembly had a total of 94.9 million nodes, although the majority of these were less than 100bp long. Additional assembly statistics are listed in Table 12

As shown in Table 13, the partial assembly statistics are comparable to both the ABySS and SOAPdenovo assemblies of this exact same dataset. In particular, Contrail's contig N50 size is ~5% larger than either SOAPdenovo or ABySS at the contigging stage before using the paired-end data to build scaffolds. Future work remains to complete the assembly, and analyze the contigs for accuracy and also novel insertions not present in the reference human genome.

Table 13. Comparison of human genome assemblies.

Assembler	Contigs ≥ 100bp	N50 (bp)	Total Length (Gbp)
Contrail SE	4,285,080	923	2.13
SOAPdenovo PE	NA	4,611	2.63
SOAPdenovo SE	NA	886	2.10
ABySS PE	2,762,173	1,499	2.18
ABySS SE	4,348,132	870	2.10

SE indicates single end data only, before any scaffolding. PE indicates paired-end analysis, including scaffolding. Total length indicates total length of contigs at least 100bp.

Discussion

DNA sequencing costs have fallen by several orders of magnitude over the last decade, and are projected to continue on this trend towards realizing a $1000 human genome within the next few years. This dramatic shift has created a massive increase in the scale and scope of DNA sequencing, and sequencing projects are now underway to sequence organisms from all corners of the globe and across the entire tree of life. De novo assembly is obviously fundamental to each of these projects, but is useful even when a high quality reference genome is available. In these cases, de novo assembly can be used for resolving large-scale polymorphisms and structural variations that are difficult or impossible to resolve with purely comparative techniques.

Given the huge demand for assembly, including the huge demand for assembly of large genomes, scalable methods for de novo assemble are paramount. Contrail directly addresses this need as a scalable solution for de novo assembly of large genomes. The sequences assembled by Contrail today are comparable to those produced by other leading assemblers, and does so on a cluster of commodity machines without requiring any special high end resources. In the current implementation, Contrail's runtime is comparable to ABySS, but is considerably slower than SOAPdenovo, mainly because SOAPdenovo can execute the assembly in main memory instead of transferring data across the network. Future work remains to improve the runtime performance, such as by removing tips and compressing linear paths in a single cycle, and also to improve the assembly quality. In particular, new network flow methods have recently been proposed to improve the assembly of repetitive sequences [102]. Conceptually these proposed methods are compatible with Contrail, and will be incorporated in future releases, as will specializations for de novo transcriptome and metagenome assembly.

More generally, as sequencing and sequence analysis moves out of the large sequencing centers and into individual labs, there is a corresponding need for scalable methods for all forms of sequence analysis. MapReduce and cloud computing may be the enabling technologies for realizing the democratization of sequencing, as it enables researchers to easily tap the power of many hundreds commodity servers, and scale up their analyses on demand to using just the resources required for the task at hand.

Methods

MapReduce Overview

MapReduce [82] is a framework for computation invented at Google consisting of 3 major stages called map, shuffle, and reduce. In the map stage, a user defined map function scans each record of the input, and emits zero or more intermediate key-value pairs per record. In the shuffle phase, the intermediate key-value pairs are routed and sorted so that all key-value pairs with the same key are collected into a single list of values. Then the user defined reduce function is executed once for each key, using the entire list of values associated with that key to compute the final output(s).

The power of the MapReduce is the computation is evaluated in parallel on all available computers, including clusters with many hundreds or thousands of nodes. Furthermore, a new application need only implement the map and reduce functions, and the system automatically manages the large distributed sort necessary for the shuffle, and also provides all of the services necessary to guarantee reliable and efficient computation on large datasets.

Not every algorithm benefits from MapReduce, nor is it even possible to implement every algorithm in the framework. In particular the map and reduce functions must be stateless and streaming, and each execution of the map or reduce function must by entirely independently from all others, and the functions cannot store the entire dataset in memory. Despite these limitations, MapReduce has been extremely successful accelerating many large data processing applications, including constructing inverted indices and processing log files [82], but also short read mapping [38] and genotyping [74].

The relatively simple applications listed above operate using a single cycle of map, shuffle, and reduce for their algorithms, but MapReduce can also be chained together across multiple cycles, where the output from the reducer becomes the input to the mapper in the next cycle. In this way, iterative MapReduce expands the class of algorithms that can be efficiently implemented, especially for computations on graphs. For example, an iterative MapReduce graph algorithm could propagate information from all nodes to their immediate neighbors in the first MapReduce cycle, and then to their 2-hop (or more distant) neighbors in the second, and so forth. This technique has been used for efficiently computing a breath-first search or the spanning tree of a graph in a few MapReduce cycles [37].

Computations in MapReduce are restricted to executing on key-value pairs, but this requirement lends itself well to analyzing large graphs. The id of the node is used as the key and the adjacency list of edges and other node information is stored a "node-tuple" as the value. Then computation on nodes and edges is implemented using a form of message passing to exchange information between adjacent nodes as follow. The map function iterates over the graph stored as node-id, node-tuple pairs, and immediately emits the same node-id, node-tuple pairs. At the same time, the map function also emits messages for the neighboring nodes using the node id of the neighbor as the key, and a "message-tuple" with the content of the message as the value. The shuffle stage then sorts all of the key-value pairs, and collects all tuples with the same key (node-id) into a single list. Finally, the reduce function evaluates once for each node in the graph, using both the node-tuple of the current node-id and any message-tuples from neighboring nodes. The reduce function processes the messages to compute the new state of the node-tuple, and stores the updated graph as a node-id,node-tuple pairs for the next cycle.

Contrail Overview

Contrail is implemented as an iterative MapReduce algorithm that runs in parallel on many computers starting from the initial set of unassembled reads, and finally outputting a set of contigs. All stages of the algorithm run in parallel using Hadoop across all available computers, and at no time is the entire assembly loaded into memory. The exact number of MapReduce cycles depends on the dataset and cluster configuration, but the algorithm executes in three major phases for (1) pre-assembly read error correction, (2) contig construction, and (3) scaffolding, as explained below. Contrail can also assemble small genomes on a single machine without Hadoop using the unix sort command to simulate the MapReduce shuffle as disk space permits, but is likely to be slower than an assembler that operates entirely in memory.

1. Read Error Correction

The first phase of Contrail trims very low quality 3' ends and corrects apparent sequencing errors in the reads. First, MapReduce is used to count the number of occurrences of every k-mer present in the reads. Sequencing errors are detected as k-mers in a read that occur less than a threshold, which is computationally determined using a histogram of k-mer coverages. K-mers occurring more than this threshold are classified as “trusted” (to be in the genome), while the remaining low frequency k-mers are classified as “untrusted” (caused by sequencing error). The pattern of untrusted k-mers defines the region of the read in which the sequencing error(s) must have occurred. Using the read’s quality values, we can assign a likelihood to a set of possible corrections to the read in this region. Sets of corrections are considered in decreasing order of their likelihood, and if they make every k-mer in the read “trusted”, we correct the read accordingly.

2. Contig Construction

2.1. De Bruijn Graph Construction

The de Brujin graph is constructed using MapReduce by scanning each read in the mapper and emitting the key-value pair (u, v) to encode the edge between consecutive k-mers u and v in the read. The reversed key-value pair (v,u) is also emitted and appropriately annotated to construct reverse edges in the de Bruijn graph. K-mers and their reverse complement are represented by a single value and the edges are bidirectional, with annotations to indicate the orientation of the associated nodes.

After the map function completes, the internal shuffle phase collects key-value pairs with the same key, which effectively collects edges with the same source k-mer. The reduce function saves the graph structure as key-value pairs, with the sequence of the k-mer as the key, and the adjacency list of oriented edges and other node information in a node-tuple as the value. In the following discussion, forward-to-forward edges are considered out-links, as are forward-to-reverse and reverse-to-forward links when the first node has node-id lexicographically less than the second node. Otherwise, the links are considered in-links. Note that reverse-to-reverse links are stored as forward-to-forward links.

2.2. Compression

After the initial construction, the error-free regions of the genome between repeat boundaries form non-branching paths of nodes (Figure 43, top). Since these paths are unambiguous, they can be safely compressed into single nodes with a longer sequence without risk of mis-assembly (Figure 43, bottom). In particular, nodes u and v can be compressed together if the only outlink from node u connects to node v, and the only inlink into node v originates from node u. The compression expands the node label on u with the last length(v)-k-1 characters from v, replaces u's outlinks with v's outlinks and removes node v. By repeating this pairwise merging, any number of nodes can be compressed into individual nodes forming contigs of unlimited length.

Path compression in Velvet and other serial de Bruijn graph based assemblers use an iterative serial algorithm that compresses unambiguous paths into individual nodes entirely in memory. In contrast, a parallel implementation must simultaneously compress different regions of the graph, all while keeping the graph in a consistent state for the next iteration. This is challenging to implement within MapReduce because adjacent nodes are not directly accessible, and may even be stored on the physically different machines. Nevertheless, it is possible to propagate updates between nodes stored on different machines using the MapReduce message passing technique described above.

A naïve MapReduce path compression algorithm iteratively compresses the first node from a linear path until no compressible nodes remain. It first tests if a given node u is the first node in a linear path by examining the in- and out-links stored in the node-tuple. If node u is the first node in a path, it is compressed with its sole successor v and updates the appropriate edges of its neighbors using the message passing technique. Then this process repeats until no nodes are merged, which occurs when every linear path in the graph is maximally compressed. The naïve approach correctly compresses the graph, but requires P iterations to compress a graph where P is the length of the longest linear path in the graph. For large genomes such as the human genome, linear paths may span tens of thousands of nodes, and would be computationally prohibitive to compress large graphs using this approach.

Instead Contrail uses a novel MapReduce algorithm inspired by randomized parallel list ranking to merge long linear paths [142]. Consider a linear path of 8 nodes shown in Figure 43 (top). If every other node is merged with its successor, the length of the linear path is cut in half (Figure 43, round 1). This merging could then be repeated a total of O(log p) rounds to compress a linear path of length p into a single node (Figure 43, bottom). However, in a parallel setting there is no information available to determine which set of nodes to compress, as every compressible node is symmetrical to every other except for those at the extreme ends of a linear path.

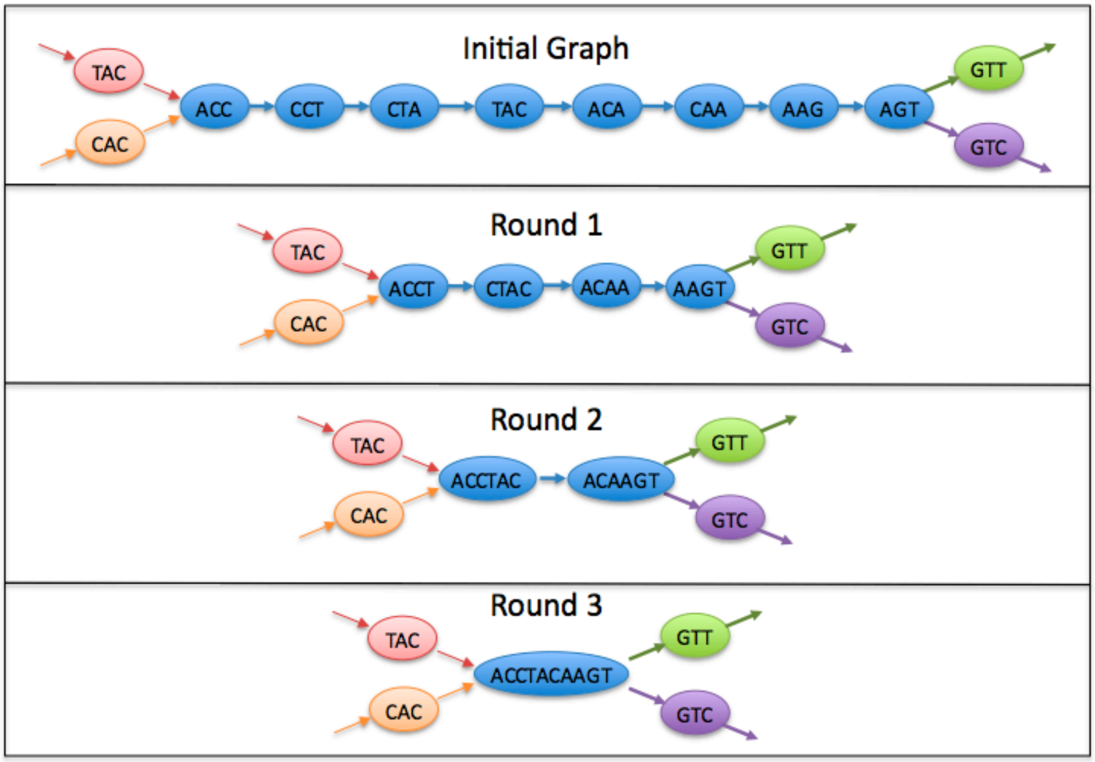


Figure 43. A linear path of 8 nodes compressed in 3 rounds.
In each round of compression the number of nodes in the linear path is cut in half by merging every other node with its successor.

This problem can be overcome by selecting an independent set of edges to compress using a randomized parallel algorithm [142]. First, each compressible node is randomly assigned the label head or the label tail with equal probability. Then edges between head nodes and tail nodes are compressed together using the pairwise merging algorithm described above. The process then repeats until there are no compressible nodes remaining as before. In each iteration, 25% of the edges are expected to be identified and resolved, thereby requiring O(log *p*) total iterations with high probability [142]. In practice, Contrail commonly compresses 50% - 60% of compressible edges in a single iteration, because Contrail also considers the edges of the de Bruijn graph are bidirectional, and also because Contrail applies additional rules to compress edges at the extreme edges of the linear path. Each iteration of compression requires 2 MapReduce cycles to correctly perform the appropriate message passing, but the total number of MapReduce cycles is O(log *p*). In addition, the overall computational complexity of algorithm is work optimal, because the total number amount of parallel work performed (merges executed) is exactly the number compressible edges.

In addition, Contrail uses an additional fast in-memory compression mode to reduce the number of MapReduce cycles necessary to compress all paths. After each iteration, Contrail tests if there are less than M total compressible nodes remaining, and if so, Contrail partitions the graph so that all compressible nodes are in the same partition. These nodes are then compressed in memory by a single reducer, and the remaining non-compressible nodes are randomly assigned to a random partition. This is implemented by emitting all compressible nodes tagged with the same key C in the mapper, while all non-compressible nodes are tagged with a random key X. The reducer then temporarily stores all values with the compressible key in memory, and performs the compression entirely in memory including chains of *m* nodes long. This optimization skips up to O(log *m*) MapReduce cycles, but requires the value of *m* is sufficiently small so that *m* nodes can all be stored in the local memory of a single machine.

2.3. Topological Error Correction

Sequencing errors in the reads create false k-mers that do not exist in the true genome sequence. The corresponding nodes in the de Bruijn graph will therefore have lower coverage or distorted topology. Similar to SOAPdenovo, Contrail recognizes error induced graph topologies, and removes the error nodes from the graph.

If the sequencing error occurs within k bp from the end of the read, the error creates low coverage “tips” in the compressed de Brujin graph connected to the rest of the graph by a single edge (Figure 44, top). Contrail recognizes this topology using MapReduce, and removes the appropriate nodes and edges from the graph. If there are only tips as in- or out- links from a given node in the graph, the longest such tip is kept.

If instead the error occurs in the middle of the read, then the sequencing error creates a “bubble” where 2 nodes have the same in-links and out-links and nearly identical sequences except a small number of differences (Figure 44, bottom). Contrail detects and pops (resolves) bubbles in two MapReduce steps. The first finds bubbles based on network topology and their sequence, and the second removes the bubbles from the graph keeping just the variant with higher coverage.

Finally, sequencing error may also create chimeric reads connecting distant regions of the genome, but with low coverage. Contrail recognizes and removes these low coverage nodes using a single MapReduce cycle.

Each of the error correction operations is iteratively applied, because removing errors may reveal additional opportunities for correction. For example, popping a bubble may reveal another bubble.

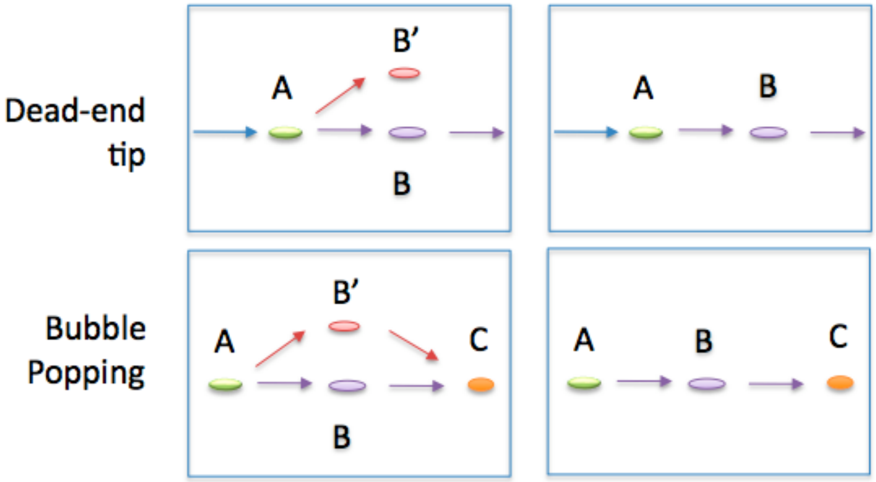


Figure 44. Topological Error Correction.
(top, left) Dead end tips shorter than the threshold with only in- or out- links are removed (top, right). (bottom, left)) Bubbles with identical neighbors and nearly identical sequence are removed (bottom, right).

2.4. Resolve Short Repeats

Despite its advantages for recognizing errors and overlap efficiency, the de Bruijn graph approach to genome assembly suffers the limitation that it is not *read coherent*, meaning the graph allows paths that are not supported by any read [102]. This is because the graph is constructed from k-mers that may be considerably shorter than the full sequences of the reads. For example, if two reads share a small repetitive k-mer, but otherwise do not overlap, then the de Bruijn graph will contain a branching node (in degree and out degree = 2) for that k-mer, but only 2 of the 4 possible paths are supported by the read sequences (Figure 45). Contrail therefore annotates each edge with the ids of the spanning reads, and resolves branching nodes that are entirely spanned by multiple reads (default 5) by making separate, non-branching copies of the branching node for each confirmed path. This in effect resolves the ambiguities introduced by repeats shorter than the read length.

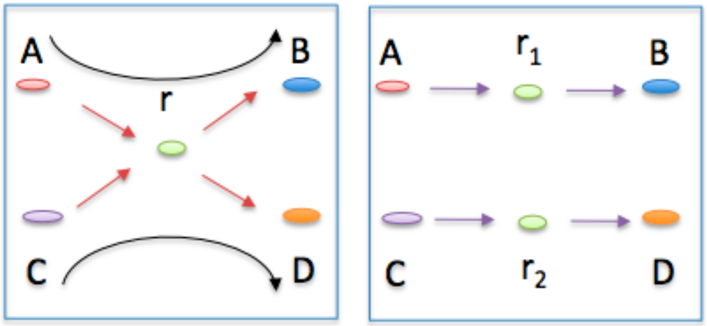


Figure 45. Resolve short repeats.
Contrail annotates each branching edge with the ids of the reads spanning that edge, and then splits the edges appropriately. Here all of the reads spanning the edge from A->r exit along the edge r->B, and all of the reads spanning the edge C->r exit along the edge r->D.

3. Scaffolding

The output of Contrail's second phase is an assembly graph consisting of error-corrected contigs, with edges between overlapping contigs. Each contig in the graph terminates either because there was a gap in coverage and overlaps no other contigs, or because of ambiguity in how the overlapping contigs should be connected. Coverage gaps generally require additional sequencing to resolve, but Contrail can resolve many ambiguities by finding unique paths through the assembly graph consistent with the mate-pair constraints.

3.1 Mate Bundling

The first step of scaffolding determines which contigs are linked by mate-pairs, and their relative orientation and separation. By convention, mated reads have the same name except for their suffix (either _1 or _2). Contrail therefore finds all mate-linked contigs using a single MapReduce cycle by emitting from the mapper mate messages consisting of the read name without the suffix as the key, and the contig name, read orientation, and read offset as the value. The reduce function scans these mate messages and saves contig link messages that contain the id of the two contigs that are linked and their expected separation and relative orientation. A second MapReduce stage emits the contig link messages and the assembly graph in the mapper, and then bundles together link messages between the same pair of contigs in the reducer (Figure 46, left). Mate-pairs between repetitive contigs are discarded, using a threshold on contig depth of coverage to filter repetitive contigs.

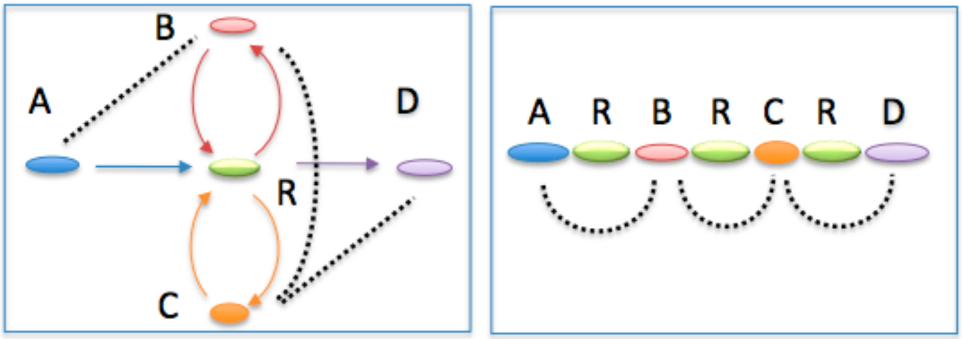


Figure 46. Contrail mate-pair bundling and resolution.
(left) Mate-pair bundles constrain the orientation and separation of unique nodes A, B, C & D with respect to repeat node R. (right) Contrail finds a linear path that satisfies the mate-pair constraints.

3.2 Bundle Resolution

Once the mates are bundled, Contrail searches the graph for paths of contigs consistent with bundles supported by multiple mate pairs (default 5). If there are multiple such bundles extending from the forward or reverse of a contig, then only the bundle to the nearest contig is considered. More distant connections are considered in subsequent rounds of scaffolding. A path is consistent if the separation between contigs implied by the path of overlapping contigs is within the expected distance recorded in the bundle, and their relative orientation matches the relative orientation implied by the mate pairs. If a unique path is found to be consistent with the bundle, it merges the contigs along that path into a single contig (Figure 46, right).

For this, Contrail uses a variation of breath-first frontier search [143] from all unique nodes with bundles. In the first MapReduce cycle, all paths of length 1 are explored, using message passing between the unique nodes and their immediate neighbors. In the second cycle, all paths of length 2 are explored using message passing from the 1-hop neighbors in the first cycle. The process repeats for n cycles (default 20), iteratively exploring more distant neighbors. Each hop adds at least one extra base to the candidate path, but will usually extend the path by a much larger stride, depending on the contig size. In each cycle, paths longer (in total base-pairs) than the expected distance to the mate-linked contig are pruned from further consideration. Paths ending with the correct separation and orientation at the mate-linked contig are stored. If after n cycles, there is only a single path consistent with the bundled mate pairs, the path of contigs connecting those contigs are resolved into individual larger contigs using a variation of the repeat resolution method described above. In short, repetitive contigs along the path are split into multiple copies, depending on how many paths contain them, and then the linear path compression routine merges the path of now non-branching contigs into a single contig. The scaffolding process then repeats bundling and merging nodes until no more merges occur.

3.3 Assembly Finalization

After scaffolding, the assembly is converted using MapReduce from the internal formats into a traditional multifasta file with the contig sequences, and a contig layout file storing the position of each read in each contig. In addition, after scaffolding and periodically throughout the assembly processes, Contrail will compute contig size statistics, such as the average or N50 contig size using MapReduce.

Chapter 9: Summary of Contributions

The research in this dissertation enables accelerated alignment, genotyping, and de novo assembly of large genomes from short reads by bridging research in computation biology with research in high performance computation. This combination is essential in light of the large data sets involved, and has the potential to unlock discoveries of critical magnitude.

Whereas the published analysis of the African and Asian human individuals required over 1000 CPU hours to analyze the ~100GB of compressed sequence data, the Crossbow pipeline can reproduce their results in just a few hours. CloudBurst demonstrates how to efficiently parallelize classic seed-and-extend algorithms from computational biology to achieve 100 fold improvements in runtime. MUMmerGPU lays the foundation for utilizing GPGPU technologies in computational biology.

In addition to these advances for short read alignment, Contrail is a highly scalable genome assembler, capable of assembling the human genome de novo on a cluster of commodity computers. This is essential for understanding the large numbers of complex organisms that have never been sequenced before, and will directly contribute to new biological knowledge.

Future work remains to develop additional scalable algorithms for other sequence based assays. For example, the alignment-shuffle-scan framework used by Crossbow for genotyping could be naturally extended to analyze RNAseq [78], CHIPseq [77], or Methyl-seq [76] assays. Contrail could be extended to with specializations for metagenomics or transcriptome assembly. The assembly validation methods used by Hawkeye and the genome forensics pipeline would benefit from MapReduce technologies to scale to larger genomes.

As demonstrated in this dissertation, computational biology can greatly benefit from research in parallel algorithms and parallel systems. Further research in these fields is also needed. MUMmerGPU demonstrates it is possible to use GPUs for analyzing tree structures, and Contrail demonstrates that it is possible to use MapReduce for analyzing graph structures, but further research is necessary to improve support for these types of algorithms, especially for analyzing irregular data structures. In particular, MapReduce is extremely inefficient when a small number of messages are exchanged relative to the size of the graph, since the entire graph is rewritten in every MapReduce cycle.

Research Highlights

1. Implemented GPGPU-based parallel suffix tree alignment program MUMmerGPU, featuring a novel space filling curve to reorder memory accesses and maximize cache performance, and a novel stackless depth-first-search algorithm for traversing a suffix tree using a constant amount of space.

2. Developed a MapReduce-based parallel short read mapping program CloudBurst and short read genotyping pipeline Crossbow capable of quickly and accurately genotyping an entire human genome in an afternoon.

3. Developed visualization and analysis tools used to improve the quality of the assembly of several genomes including cow [144], papaya [145], the rice pathogen *Xanthomonas oryzae* [133], the honey bee fungus *Nosema ceranae* [146], multiple species of fruit flies [147], mosquito [148] and the human pathogen *Trichonomas vaginalis* [134].

4. Developed a MapReduce-based short read assembler Contrail capable of assembling the human genome using a cluster of commodity computers, featuring a novel work-optimal parallel algorithm for path compression.

Bibliography

1. Pushkarev, D., N.F. Neff, and S.R. Quake, *Single-molecule sequencing of an individual human genome.* Nat Biotechnol, 2009. **27**(9): p. 847-52.
2. Brown, T.A., *Genomes 3*. 3rd ed. 2006: Garland Science.
3. Sanger, F., S. Nicklen, and A.R. Coulson, *DNA sequencing with chain-terminating inhibitors.* Proc Natl Acad Sci U S A, 1977. **74**(12): p. 5463-7.
4. Mardis, E.R., *The impact of next-generation sequencing technology on genetics.* Trends Genet, 2008. **24**(3): p. 133-41.
5. Fleischmann, R.D., et al., *Whole-genome random sequencing and assembly of Haemophilus influenzae Rd.* Science, 1995. **269**(5223): p. 496-512.
6. Lander, E.S. and M.S. Waterman, *Genomic mapping by fingerprinting random clones: a mathematical analysis.* Genomics, 1988. **2**(3): p. 231-9.
7. Li, H., J. Ruan, and R. Durbin, *Mapping short DNA sequencing reads and calling variants using mapping quality scores.* Genome Res, 2008.
8. Wheeler, D.A., et al., *The complete genome of an individual by massively parallel DNA sequencing.* Nature, 2008. **452**(7189): p. 872-6.
9. Wang, J., et al., *The diploid genome sequence of an Asian individual.* Nature, 2008. **456**(7218): p. 60-5.
10. Delcher, A.L., et al., *Alignment of whole genomes.* Nucleic Acids Res, 1999. **27**(11): p. 2369-76.
11. Altschul, S.F., et al., *Basic local alignment search tool.* J Mol Biol, 1990. **215**(3): p. 403-10.
12. Smith, A.D., Z. Xuan, and M.Q. Zhang, *Using quality scores and longer reads improves accuracy of Solexa read mapping.* BMC Bioinformatics, 2008. **9**: p. 128.
13. Lin, H., et al., *ZOOM! Zillions Of Oligos Mapped.* Bioinformatics, 2008.
14. Kurtz, S., et al., *Versatile and open software for comparing large genomes.* Genome Biol, 2004. **5**(2): p. R12.
15. Li, R., et al., *SOAP: short oligonucleotide alignment program.* Bioinformatics, 2008. **24**(5): p. 713-4.
16. Baeza-yates, R.A. and C.H. Perleberg, *Fast and practical approximate string matching.* In Combinatorial Pattern Matching, Third Annual Symposium, 1992: p. 185-192.
17. Schatz, M.C., et al., *High-throughput sequence alignment using Graphics Processing Units.* BMC Bioinformatics, 2007. **8**: p. 474.
18. Green, P. *Phrap User Documentation*. [cited; Available from: http://www.phrap.org/phredphrap/phrap.html.
19. Sutton, G.G., et al., *TIGR Assembler: A new tool for assembling large shotgun sequencing projects.* Genome Science and Technology., 1995. **1(1)**(Jan 01): p. 9-19.
20. Phillippy, A.M., M.C. Schatz, and M. Pop, *Genome assembly forensics: finding the elusive mis-assembly.* Genome Biology, 2008. **9**(3): p. R55.

21. Myers, E.W., et al., *A whole-genome assembly of Drosophila.* Science, 2000. **287**(5461): p. 2196-204.
22. Batzoglou, S., et al., *ARACHNE: a whole-genome shotgun assembler.* Genome Res, 2002. **12**(1): p. 177-89.
23. Jaffe, D.B., et al., *Whole-genome sequence assembly for mammalian genomes: Arachne 2.* Genome Res, 2003. **13**(1): p. 91-6.
24. Salzberg, S.L. and J.A. Yorke, *Beware of mis-assembled genomes.* Bioinformatics, 2005. **21**(24): p. 4320-1.
25. Schatz, M.C., et al., *Hawkeye: an interactive visual analytics tool for genome assemblies.* Genome Biol, 2007. **8**(3): p. R34.
26. Thain, D., T. Tannenbaum, and M. Livny, *Distributed Computing in Practice: The Condor Experience.* Concurrency and Computation: Practice and Experience, 2005. **17**(2-4): p. 323-356.
27. Darling, A., L. Carey, and W. Feng. *The Design, Implementation, and Evaluation of mpiBLAST.* in *4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with ClusterWorld Conference & Expo*. 2003.
28. Krishnaprasad, S., *Uses and abuses of Amdahl's law.* J. Comput. Small Coll., 2001. **17**(2): p. 288-293.
29. Kruskal, J.B. *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.* in *Proceedings of the American Mathematical Society*. 1956.
30. Shiloach, Y. and U. Vishkin, *An O(log n) parallel connectivity algorithm.* Journal of Algorithms, 1982. **3**: p. 57-67.
31. Jaja, J., *An Introduction to Parallel Algorithms*. 1992: Addison-Wesley.
32. Dongarra, J.J., et al., *A message passing standard for MPP and workstations.* Communications of the ACM, 1996. **39**(7): p. 84-90.
33. van Emde Boas, P., *Machine models and simulations*, in *Handbook of theoretical Computer Science (Vol. A): Algorithms and Complexity*, J.v. Leeuwen, Editor. 1990, MIT Press: Cambridge, MA. p. 1-66.
34. Cook, S., C. Dwork, and R. Reischuk, *Upper and Lower Time Bounds for Parallel Random Access Machines without Simultaneous Writes.* SIAM J. Computing, 2986. **15**(87).
35. Trapnell, C. and M.C. Schatz, *Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment.* Parallel Computing, 2009. **35**(8-9): p. 429-440.
36. Jeffrey, D. and G. Sanjay, *MapReduce: simplified data processing on large clusters.* Commun. ACM, 2008. **51**(1): p. 107-113.
37. Karloff, H., S. Suri, and S. Vassilvitskii. *A Model of Computation for MapReduce.* in *SODA*. 2010. Austin, TX.
38. Schatz, M.C., *CloudBurst: highly sensitive read mapping with MapReduce.* Bioinformatics, 2009. **25**(11): p. 1363-9.
39. Delcher, A.L., et al., *Fast algorithms for large-scale genome alignment and comparison.* Nucleic Acids Res, 2002. **30**(11): p. 2478-83.
40. Pop, M., et al., *Comparative genome assembly.* Brief Bioinform, 2004. **5**(3): p. 237-48.

41. Shaffer, C., *Next-generation sequencing outpaces expectations.* Nat Biotechnol, 2007. **25**(2): p. 149.

42. Pearson, W.R. and D.J. Lipman, *Improved tools for biological sequence comparison.* Proc Natl Acad Sci U S A, 1988. **85**(8): p. 2444-8.

43. Brudno, M., et al., *LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA.* Genome Res, 2003. **13**(4): p. 721-31.

44. Hohl, M., S. Kurtz, and E. Ohlebusch, *Efficient multiple genome alignment.* Bioinformatics, 2002. **18 Suppl 1**: p. S312-20.

45. Kurtz, S., et al., *REPuter: the manifold applications of repeat analysis on a genomic scale.* Nucleic Acids Res, 2001. **29**(22): p. 4633-42.

46. Weiner, P. *Linear pattern matching algorithms. .* in *14th IEEE Symposium on Switching and Automata Theory*. 1979.

47. Ukkonen, E., *On-line construction of suffix-trees.* Algorithmica, 1995. **14**: p. 249-260.

48. Gusfield, D., *Algorithms on strings, trees, and sequences: computer science and computational biology*. 1997: Cambridge University Press. 534.

49. Owens, J.D., et al., *A Survey of General-Purpose Computation on Graphics Hardware.* Computer Graphics Forum, 2007. **26**(1): p. 80-113.

50. Govindaraju, N., et al. *A memory model for scientific algorithms on graphics processors*. in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*: ACM Press.

51. Harris, M., et al. *Physically-Based Visual Simulation on Graphics Hardware*. in *Proc 2002 SIGGRAPH/Eurographics Workshop on Graphics Hardware*. 2002.

52. Juekuan, Y., W. Yujuan, and C. Yunfei, *GPU accelerated molecular dynamics simulation of thermal conductivities.* J Comput Phys, 2007. **221**(2): p. 799-804.

53. Dally, W., et al. *Merrimac: Supercomputing with Streams*. in *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. 2003. Washington DC: IEEE Computer Society.

54. Buck, I., *Taking the Plunge into GPU computing*, in *GPU Gems 2*, P. M, Editor. 2005, Addison-Wesley. p. 509-519.

55. Liu, W., et al. *Bio-Sequence Database Scanning on a GPU.* in *20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006) (HICOMB Workshop)*. 2006. Rhode Island.

56. Charalambous, M., P. Trancoso, and A. Stamatakis. *Initial Experiences Porting a Bioinformatics Application to a Graphics Processor.* in *Proceedings of the 10th Panhellenic Conference on Informatics (PCI 2005)*. 2005. Volos, Greece: Springer LNCS.

57. nVidia (2007) *nVidia Compute Unified Device Architecture (CUDA) Programming Guide, version 1.0.* **Volume**,

58. Mellor-Crummey, J., D. Whalley, and K. Kennedy, *Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings.* International Journal of Parallel Programming, 2001. **29**(3): p. 217-247.

59. Stein, L.D., et al., *The genome sequence of Caenorhabditis briggsae: a platform for comparative genomics.* PLoS Biol, 2003. **1**(2): p. E45.

60. AMD (2008) *ATI CTM Guide, Technical Reference Manual*. **Volume**,

61. Chi, K.R., *The year of sequencing.* Nat Methods, 2008. **5**(1): p. 11-4.

62. Popov, S., et al., *Stackless KD-tree traversal for high performance GPU ray tracing.* Computer Graphics Forum, 2007. **26**(3): p. 415-424.

63. nVidia (2009) *NVIDIA GeForce 9 Series*. **Volume**,

64. Michalakes, M.V.J. *GPU Acceleration of numerical weather prediction*. in *Workshop on Large Scale Parallel Processing, IEEE International Parallel & Distributed Processing Symposium*. 2008.

65. Stone, J.E., et al., *Accelerating molecular modeling applications with graphics processors.* J Comput Chem, 2007. **28**(16): p. 2618-40.

66. Manavski, S.A. *CUDA compatible GPU as an efficient hardware accelerator for AES cryptography*. in *Proceedings IEEE International Conference on Signal Processing and Communication*. 2007.

67. Lewin, B., *Genes IX*. 2007: Jones & Bartlett.

68. Venter, J.C., et al., *The sequence of the human genome.* Science, 2001. **291**(5507): p. 1304-51.

69. Bentley, D.R., et al., *Accurate whole human genome sequencing using reversible terminator chemistry.* Nature, 2008. **456**(7218): p. 53-9.

70. Ghemawat, S., H. Gobioff, and S.-T. Leung, *The Google file system*, in *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, ACM: Bolton Landing, NY, USA.

71. Smith, T.F. and M.S. Waterman, *Identification of common molecular subsequences.* J Mol Biol, 1981. **147**(1): p. 195-7.

72. Langmead, B., et al., *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.* Genome Biol, 2009. **10**(3): p. R25.

73. Landau, G.M. and U. Vishkin, *Introducing efficient parallelism into approximate string matching and a new serial algorithm*, in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986, ACM: Berkeley, California, United States.

74. Langmead, B., et al., *Searching for SNPs with cloud computing.* Genome Biol, 2009. **10**(11): p. R134.

75. Li, R., et al., *SNP detection for massively parallel whole-genome resequencing.* Genome Res, 2009. **19**(6): p. 1124-32.

76. Lister, R. and J.R. Ecker, *Finding the fifth base: genome-wide sequencing of cytosine methylation.* Genome Res, 2009. **19**(6): p. 959-66.

77. Tuteja, G., et al., *Extracting transcription factor targets from ChIP-Seq data.* Nucleic Acids Res, 2009. **37**(17): p. e113.

78. Mortazavi, A., et al., *Mapping and quantifying mammalian transcriptomes by RNA-Seq.* Nat Methods, 2008. **5**(7): p. 621-8.

79. Trapnell, C., L. Pachter, and S.L. Salzberg, *TopHat: discovering splice junctions with RNA-Seq.* Bioinformatics, 2009. **25**(9): p. 1105-11.

80. Simpson, J.T., et al., *ABySS: a parallel assembler for short read sequence data.* Genome Res, 2009. **19**(6): p. 1117-23.

81. Jackson, B., P. Schnable, and S. Aluru. *Assembly of large genomes from paired short reads*. in *Proceedings of the 1st International Conference on Bioinformatics and Computational Biology*. 2009. New Orleans, LA: Springer-Verlag.
82. Dean, J. and S. Ghemawat. *MapReduce: simplified data processing on large clusters.* in *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*. 2004. San Francisco, California, USA: ACM.
83. Frazer, K.A., et al., *A second generation human haplotype map of over 3.1 million SNPs.* Nature, 2007. **449**(7164): p. 851-61.
84. Burrows, M. and D. Wheeler, *A Block-sorting Lossless Data Compression Algorithm.*, in *Technical Report 124*. 1994, Digital Systems Research Center: Palo Alto, CA.
85. Ferragina, P. and G. Manzini. *Opportunistic data structures with applications.* in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science.* 2000. Los Alamitos, California USA: IEEE Computer Society.
86. Sherry, S.T., et al., *dbSNP: the NCBI database of genetic variation.* Nucleic Acids Res, 2001. **29**(1): p. 308-11.
87. Bravo, H.C. and R.A. Irizarry, *Model-Based Quality Assessment and Base-Calling for Second-Generation Sequencing Data.* Biometrics, 2009.
88. Dohm, J.C., et al., *Substantial biases in ultra-short read data sets from high-throughput DNA sequencing.* Nucleic Acids Res, 2008. **36**(16): p. e105.
89. Lander, E.S., et al., *Initial sequencing and analysis of the human genome.* Nature, 2001. **409**(6822): p. 860-921.
90. Waterston, R.H., et al., *Initial sequencing and comparative analysis of the mouse genome.* Nature, 2002. **420**(6915): p. 520-62.
91. Margulies, M., et al., *Genome sequencing in microfabricated high-density picolitre reactors.* Nature, 2005. **437**(7057): p. 376-80.
92. Schuster, S.C., *Next-generation sequencing transforms today's biology.* Nat Methods, 2008. **5**(1): p. 16-8.
93. Collins, F.S. and S.M. Weissman, *Directional cloning of DNA fragments at a large distance from an initial probe: a circularization method.* Proc Natl Acad Sci U S A, 1984. **81**(21): p. 6812-6.
94. Kurtz, S., et al., *A new method to compute K-mer frequencies and its application to annotate large repetitive plant genomes.* BMC Genomics, 2008. **9**: p. 517.
95. Huang, X., et al., *PCAP: a whole-genome assembly program.* Genome Res, 2003. **13**(9): p. 2164-70.
96. Zerbino, D.R. and E. Birney, *Velvet: algorithms for de novo short read assembly using de Bruijn graphs.* Genome Res, 2008. **18**(5): p. 821-9.
97. Zerbino, D.R., et al., *Pebble and rock band: heuristic resolution of repeats and scaffolding in the velvet short-read de novo assembler.* PLoS One, 2009. **4**(12): p. e8407.
98. Butler, J., et al., *ALLPATHS: de novo assembly of whole-genome shotgun microreads.* Genome Res, 2008. **18**(5): p. 810-20.

99. Maccallum, I., et al., *ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads.* Genome Biol, 2009. **10**(10): p. R103.

100. Li, R., et al., *De novo assembly of human genomes with massively parallel short read sequencing.* Genome Res, 2010. **20**(2): p. 265-72.

101. Pevzner, P.A., H. Tang, and M.S. Waterman, *An Eulerian path approach to DNA fragment assembly.* Proc Natl Acad Sci U S A, 2001. **98**(17): p. 9748-53.

102. Myers, E.W., *The fragment assembly string graph.* Bioinformatics, 2005. **21 Suppl 2**: p. ii79-85.

103. Lander, E.S. and M.S. Waterman, *Genomic mapping by fingerprinting random clones: a mathematical analysis.* Genomics, 1988. **2**: p. 231-239.

104. Pop, M., et al., *Comparative genome assembly.* Brief Bioinform, 2004. **5**(3): p. 237-248.

105. Goldberg, S.M., et al., *A Sanger/pyrosequencing hybrid approach for the generation of high-quality draft assemblies of marine microbial genomes.* Proc Natl Acad Sci U S A, 2006. **103**(30): p. 11240-5.

106. Bentley, D.R., et al., *Accurate whole human genome sequencing using reversible terminator chemistry.* Nature, 2008. **456**(7218): p. 53-9.

107. Wang, J., et al., *The diploid genome sequence of an Asian individual.* Nature, 2008. **456**(7218): p. 60-5.

108. Kim, J.I., et al., *A highly annotated whole-genome sequence of a Korean individual.* Nature, 2009. **460**(7258): p. 1011-5.

109. Velasco, R., et al., *A high quality draft consensus sequence of the genome of a heterozygous grapevine variety.* PLoS One, 2007. **2**(12): p. e1326.

110. Huang, S., et al., *The genome of the cucumber, Cucumis sativus L.* Nat Genet, 2009. **41**(12): p. 1275-81.

111. Wang, J., et al., *RePS: a sequence assembler that masks exact repeats identified from the shotgun data.* Genome Res, 2002. **12**(5): p. 824-31.

112. Li, R., et al., *The sequence and de novo assembly of the giant panda genome.* Nature, 2010. **463**(7279): p. 311-7.

113. Lindblad-Toh, K., et al., *Genome sequence, comparative analysis and haplotype structure of the domestic dog.* Nature, 2005. **438**(7069): p. 803-19.

114. Sanger, F., et al., *Nucleotide sequence of bacteriophage lambda DNA.* J Mol Biol, 1982. **162**(4): p. 729-73.

115. Salzberg, S.L., et al., *The genome Assembly Archive: a new public resource.* PLoS Biol, 2004. **2**(9): p. E285.

116. Mullikin, J.C. and Z. Ning, *The phusion assembler.* Genome Res, 2003. **13**(1): p. 81-90.

117. Edwards, A. and T. Caskey, *Closure strategies for random DNA sequencing.* Methods: A Companion to Methods in Enzymology, 1991. **3**(1): p. 41-47.

118. Istrail, S., et al., *Whole-genome shotgun assembly and comparison of human genome assemblies.* Proc Natl Acad Sci U S A, 2004. **101**(7): p. 1916-21.

119. The International Human Genome Sequencing Consortium, *Finishing the euchromatic sequence of the human genome.* Nature, 2004. **431**(7011): p. 931-45.

120. Wheeler, D.L., et al., *Database resources of the National Center for Biotechnology Information.* Nucleic Acids Res, 2006. **34**(Database issue): p. D173-80.

121. Gordon, D., C. Abajian, and P. Green, *Consed: a graphical tool for sequence finishing.* Genome Res, 1998. **8**(3): p. 195-202.

122. Bartels, D., et al., *BACCardI--a tool for the validation of genomic assemblies, assisting genome finishing and intergenome comparison.* Bioinformatics, 2005. **21**(7): p. 853-9.

123. Huson, D., et al. *Comparing assemblies using fragments and mate-pairs*. in *First International Workshop on Algorithms in Bioinformatics*. 2001. London: Springer-Verlag.

124. Dew, I.M., B. Walenz, and G. Sutton, *A tool for analyzing mate pairs in assemblies (TAMPA).* J Comput Biol, 2005. **12**(5): p. 497-513.

125. Birney, E., et al., *An overview of Ensembl.* Genome Res, 2004. **14**(5): p. 925-8.

126. Stein, L.D., et al., *The generic genome browser: a building block for a model organism system database.* Genome Res, 2002. **12**(10): p. 1599-610.

127. Stothard, P. and D.S. Wishart, *Circular genome visualization and exploration using CGView.* Bioinformatics, 2005. **21**(4): p. 537-9.

128. Kent, W.J., et al., *The human genome browser at UCSC.* Genome Res, 2002. **12**(6): p. 996-1006.

129. Zimin, A.V., et al., *Assembly reconciliation.* Bioinformatics, 2008. **24**(1): p. 42-5.

130. Bederson, B., B. Shneiderman, and M. Wattenberg, *Ordered and quantum treemaps: making effective use of 2D space to display hierarchies.* ACM Trans Graph, 2002. **21**: p. 833-854.

131. Seo, J. and B. Shneiderman, *A rank-by-feature framework for interactive exploration of multidimensional data.* Information Visualization, 2005. **4**: p. 96-113.

132. Arner, E., et al., *DNPTrapper: an assembly editing tool for finishing and analysis of complex repeat regions.* BMC Bioinformatics, 2006. **7**: p. 155.

133. Salzberg, S.L., et al., *Genome sequence and rapid evolution of the rice pathogen Xanthomonas oryzae pv. oryzae PXO99A.* BMC Genomics, 2008. **9**: p. 204.

134. Carlton, J.M., et al., *Draft genome sequence of the sexually transmitted pathogen Trichomonas vaginalis.* Science, 2007. **315**(5809): p. 207-12.

135. Simon, H., *The Sciences of the Artificial*. 3rd Edition ed. 1996, Cambridge, MA: MIT Press.

136. Shneiderman, B. *The eyes have it: a task by data type taxonomy for information visualizations*. in *Proceedings of the 1996 IEEE Symposium on Visual Languages*. 1996. Los Alamitos, CA: IEEE Computer Society.

137. Gajer, P., M. Schatz, and S.L. Salzberg, *Automated correction of genome sequence errors.* Nucleic Acids Res, 2004. **32**(2): p. 562-9.

138. Schatz, M.C., A.L. Delcher, and S.L. Salzberg, *Assembly of large genomes using second generation sequencing.* Manuscript Under Review, 2010.

139. Chaisson, M.J. and P.A. Pevzner, *Short read fragment assembly of bacterial genomes.* Genome Res, 2008. **18**(2): p. 324-30.

140. Warren, R.L., et al., *Assembling millions of short DNA sequences using SSAKE.* Bioinformatics, 2007. **23**(4): p. 500-1.

141. Hernandez, D., et al., *De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer.* Genome Res, 2008. **18**(5): p. 802-9.

142. Vishkin, U., *Randomized speed-ups in parallel computation.* Annual ACM Symposium on Theory of Computing, 1984: p. 230-239.

143. Reinefeld, A. and T. Schütt. *Out-of-Core Parallel Heuristic Search with MapReduce.* in *High-Performance Computing Symposium*. 2009. Kingston, Ontario.

144. Zimin, A.V., et al., *A whole-genome assembly of the domestic cow, Bos taurus.* Genome Biol, 2009. **10**(4): p. R42.

145. Ming, R., et al., *The draft genome of the transgenic tropical fruit tree papaya (Carica papaya Linnaeus).* Nature, 2008. **452**(7190): p. 991-6.

146. Cornman, R.S., et al., *Genomic analyses of the microsporidian Nosema ceranae, an emergent pathogen of honey bees.* PLoS Pathog, 2009. **5**(6): p. e1000466.

147. Clark, A.G., et al., *Evolution of genes and genomes on the Drosophila phylogeny.* Nature, 2007. **450**(7167): p. 203-18.

148. Nene, V., et al., *Genome sequence of Aedes aegypti, a major arbovirus vector.* Science, 2007. **316**(5832): p. 1718-23.