Characterizing and Optimizing the Memory Footprint of De Novo Short Read DNA Sequence Assembly

Jeffrey J. Cook[1] Craig Zilles[2]

[1]*Department of Electrical and Computer Engineering* [2]*Department of Computer Science*
University of Illinois at Urbana-Champaign
{*jjcook, zilles*}*@illinois.edu*

Abstract

In this work, we analyze the memory-intensive bioinformatics problem of "de novo" DNA sequence assembly, which is the process of assembling short DNA sequences obtained by experiment into larger contiguous sequences. In particular, we analyze the performance scaling challenges inherent to de Bruijn graph-based assembly, which is particularly well suited for the data produced by "next generation" sequencing machines.

Unlike many bioinformatics codes which are computation-intensive or control-intensive, we find the memory footprint to be the primary performance issue for de novo sequence assembly. Specifically, we make four main contributions: 1) we demonstrate analytically that performing error correction before sequence assembly enables larger genomes to be assembled in a given amount of memory, 2) we identify that the use of this technique provides the key performance advantage to the leading assembly code, Velvet, 3) we demonstrate how this pre-assembly error correction technique can be subdivided into multiple passes to enable de Bruijn graph-based assembly to scale to even larger genomes, and 4) we demonstrate how Velvet's in-core performance can be improved using memory-centric optimizations.

1. Introduction

Sequence assembly is the bioinformatics problem of composing short DNA sequences (obtained experimentally) to reconstruct large contiguous sequences from an original genomic sequence (see Figure 1(a)). These large sequences are then used in many genetic, genomic, and phylogenetic research activities, both in computational bioinformatics and in directing future laboratory experiments [7].

In this work, we focus on a variant of this problem called *de novo* sequence assembly, which performs an assembly using only the experimentally-derived short DNA sequences, *without* using a reference genome as a guide. As we will discuss, sequence assembly can be a computationally challenging problem, due to the size of the chromosomes being assembled.

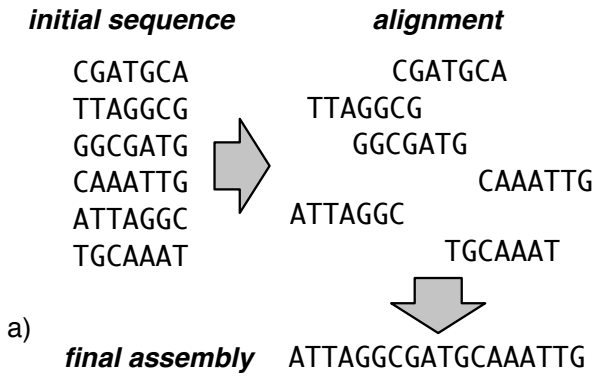
Historically, sequencing DNA has been very expensive. To minimize the amount of sequence data that was necessary to collect, sequencing and assembly were performed hierarchically. The DNA to be sequenced was preprocessed to subdivide it into 40-200 kilo-base pair segments and these segments were experimentally mapped to their relative positions in the original sequence. Then these segments could be sequenced and assembled independently and "scaffolded" into larger assemblies [10].

In the past couple years, the cost of sequencing has dropped by orders of magnitude with the development of new high-throughput sequencing machines, but the wet lab work for hierarchical assembly remains time consuming and costly. As a result, there has been a significant shift away from hierarchical assembly to the whole genome shotgun (WGS) approach, where the entire set of DNA is sequenced as a single batch experiment. This produces a single, large data set without an obvious way to sub-divide the assembly problem.

The assembly problem is compounded by two factors. First, these new high-throughput sequencing machines generate shorter fragments of sequence, which means that more sequence fragments are required (increasing the assembly effort) to construct a genome of a given size. Second, the sequence data includes a non-trivial number of errors, which must be tolerated by the assembly algorithms. In part to distinguish true sequence from the errors, each part of the DNA is sequenced many times, further exacerbating data set size.

These characteristics of high-throughput sequencing machines have led to the development of a number of de Bruijn graph-based de novo assembly codes, which are the focus of our paper. These codes construct a de Bruijn graph from the input sequence data, which can then be post-processed using domain-specific algorithms to eliminate erroneous sequence data from the graph. We describe de Bruijn graph-based assembly in Section 2.

We then explore the analytical characteristics of de Bruijn graph-based de novo assembly in Section 3. In particular, we investigate the selectivity and sensitivity of the node length parameter k. We also demonstrate the effect that high coverage has in the presence of erroroneous data. We then characterize the behavior of the leading de Bruijn graph-based de novo as-



a)

b)

Organism	Approx. Genome Size
Viruses	10 thousand bases
Bacteria	1-10 million bases
Yeast	10 million bases
Nemotode	100 million bases
Rice	400 million bases
Chicken	1 billion bases
Human	3 billion bases
Corn	3 billion bases

Figure 1: **Genetic Sequence Assembly.** a) At a very high-level sequence assembly involves aligning overlapping fragments of sequence to compose a larger, contiguous piece of sequence. b) Example genome sizes.

sembly application, Velvet [14], and demonstrate that memory is the key bottleneck in the graph construction process. Specifically, we show the advantage of performing some error removal *before* graph construction, as the erroneous data can produce a graph 8 times larger than is necessary for a genome. We also perform a scalar performance analysis and demonstrate a memory-centric optimization that improves Velvet's in-core assembly performance. In Section 5, we then identify four memory-dictated performance domains for de Bruijn graph-based assembly. In particular, we demonstrate why Velvet does not scale beyond the second performance domain. We then propose and evaluate a widely-applicable assembly algorithm transformation to scale Velvet and other de Bruijn graph algorithms into the next performance domain. We then conclude in Section 6.

2. Background

In this section, we first describe the general sequencing problem and the characteristics of the sequence data that is the input of the assembly problem. Then we overview the "overlap-layout-consensus" sequence assembly method (in Section 2.2), which has been successful for the data produced by the previous generation of sequencing machines in the hierarchical sequencing method discussed in the introduction. We then formulate the de Bruijn graph assembly problem in Section 2.3.

2.1. Sequencing and its Data

DNA sequencing is the biochemical process of determining the series of nucleotide bases (adenine (A), guanine (G), cytosine (C), and thymine (T)) for a segment of DNA. The chemical processes involved in sequencing prevent it from being economically feasible to sequence a whole chromosome in one pass. Instead, a fragment of the DNA is sequenced to identify its sequence of bases, what is referred to as a *read*.

Traditional *Sanger* sequencing technology [12], which has been in use for over thirty years, produces reads of approximately 800-1000 nucleotide bases in length. With this moderately long read length and a relatively low experimental error rate, a *coverage* value (the average number of reads that contain each base, *i.e.*, $(\#reads * read\ length)/genome\ length$) of less than 10 is generally sufficient to statistically ensure that each base in the genome is contained within at least one read. The main drawback of Sanger sequencing is its high cost so it is used only sparingly.

In the past couple of years, a number of new sequencing technologies have been commercialized that dramatically reduce the cost of producing sequence data. These technologies, collectively referred to as "next generation" sequencers, produces much shorter reads: about 36 bases long for Illumina machines [3] and about 250 bases for 454 machines [1]. Also, the error rates for these technologies are much higher, requiring significantly increased coverage to enable error detection and correction. **As a result, next generation sequencing requires the production of an order of magnitude more data than Sanger sequencing, and the number of reads increases by two orders of magnitude.** The virtue of these next generation (which we will abbreviate as *nextgen*) technologies is that they can generate 30x coverage reads for a genome for orders of magnitude less money than Sanger sequencing.

Two kinds of errors can be present in the reads. First, a read can have an extra base inserted into or deleted from anywhere in the read; such errors are called *indel*s. Alternatively, one of the bases could be mislabelled, either due to the machine incorrectly identifying the proper base (*misread*) or correctly reading data from an experiment containing multiple individuals that exhibit a *single nucleotide polymorphism* (SNP), a genetic variation that only affects a single base.

2.2. Overlap Layout Consensus

The assembly method used for traditional Sanger read data has been the "overlap-layout-consensus" approach, where three phases are performed. In the first phase, the pair-wise alignment (overlap) between each read is determined to detect which reads might align on either side of a given read. After comput-

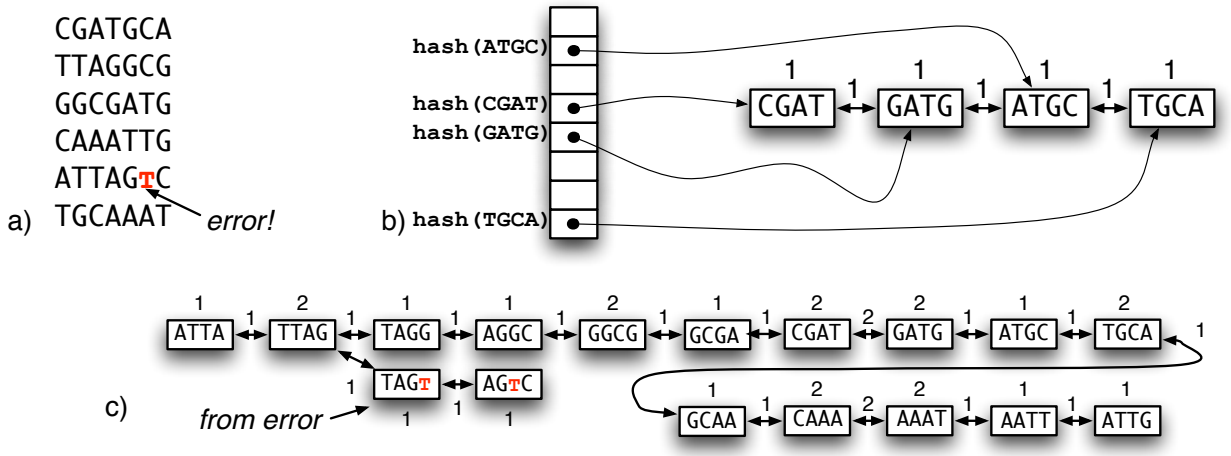


Figure 2: **de Bruijn-graph Sequence Assembly.** a) An example set of reads containing an error, b) after processing the first read, we have a graph containing 4 k-mers ($k = 4$ in this example) and a hash table to find the nodes for those k-mers, c) when assembly is complete, the resulting graph is constructed; with higher coverage of the true sequence, we can post-process to eliminate errors.

ing the overlaps, a graph is formed, where each node is a read and each weighted arc represents the overlap between the pair of reads. The layout phase then computes the Hamiltonian path in the graph, which is an NP-complete problem and, in general, has run time exponential in the number of nodes. The final step, consensus, compares the overlaps that don't perfectly agree and decides (calls) the most probable base for each base position of the overlap. In practice, this method can be successful for the nextgen sequencing of small genomes, but its poor scaling properties and relative intolerance to errors has lead de Bruijn graph-based assembly to become the dominant approach for nextgen sequence assembly.

2.3. Eulerian Assembly via de Bruijn Graph

Eulerian assembly by de Bruijn graph [9] is based on the concept of a *k-mer*: a word consisting of k nucleotides. Each read (of length L) is decomposed into $L+1-k$ overlapping k-mers. The graph is formed by creating a node for each unique k-mer and connecting two k-mers if they are adjacent in a read with a k-1 base overlap, as shown in Figure 2. For example, the nodes for $CGAT$ and $GATG$ are connected in Figure 2(b) because they represent the 1st and 2nd kmers from the first read $CGATGCA$. As we shall discuss in Section 3.1, typical values of k range from 19 to 27.

In Eulerian assembly, the procedure is to form the graph and then form an Eulerian path. The graph can be formed in expected time linear with total number of k-mers in the reads, providing that we appropriately size the hash table to prevent excessive chaining. This condition can be achieved by dedicating memory to the hash table proportional to the size (*i.e.*, number of unique k-mers) of the graph; in practice, because the hash table entries are smaller than the graph nodes this requires only a few percent of memory used. Unlike a Hamiltonian path, an Eulerian path can found in linear time.

As shown in Figure 2(c), when the input data contains errors, the resulting de Bruijn graph contains "split ends" and can also contain "hammocks" where paths in the graph diverge and re-converge. These errors can be eliminated by domain-specific heuristics to eliminate the likely erroneous nodes and edges, typically biasing toward regions with larger coverage. We find, however, that this post-processing doesn't contribute meaningfully to execution, taking roughly 1% of the time to form the graph itself.

3. Analysis of De Bruijn-graph-based Assembly

We find that the key to characterizing de Bruijn graph-based sequence assembly is in understanding its memory usage. To that end, this section explores the characteristics of such an assembly, considering required k-mer size (Section 3.1), required read coverage (Section 3.2), the characteristics of the k-mer space (Section 3.3), and the resulting number of unique true (*i.e.*, those present in the actual genome) and erroneous k-mers and their coverages (Section 3.4).

In this paper, we use four reference genomes, Streptococcus suis strain P1/7 (2Mbases in length), Escherichia coli strain K-12 (4.5Mb), Saccharomyces cerevisiae (yeast) (12Mb), and Caenorhabditis elegans (round worm) (100Mb). Of these, we use actual Illumina data for S. suis at 48x coverage and E. coli at 40x coverage.

To permit sensitivity analysis for other input sets, we use a read synthesizer [11] which takes a reference genome and generates reads whose characteristics roughly approximate those of data produced by some nextgen sequencing machines.

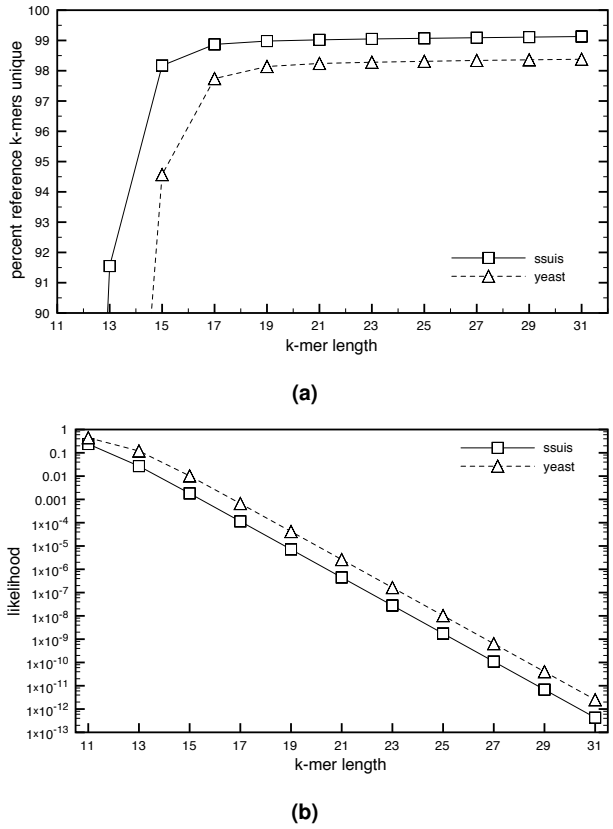


Figure 3: **Discrimination and Error Tolerance as a Function of K-mer Length** a) the fraction of k-mers in the S. Suis P1/7 and yeast genome that are unique as a function of k, and b) the likelihood that an erroneous k-mer will map to a k-mer from the S. Suis P1/7 and yeast reference genome as a function of k.

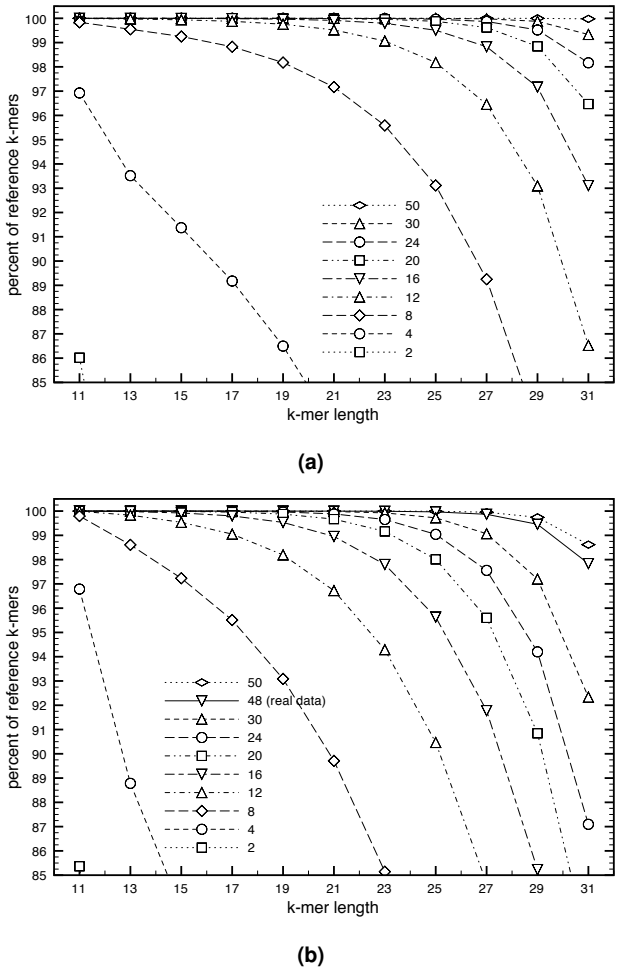


Figure 4: **Coverage requirements as a function of k.** Our goal is to get reads that include every k-mer in the genome; the Y-axis shows how close we come to that goal as a function of k (X-axis) and the average coverage (the lines). Data shown (a) for error-free reads, and (b) for reads with errors representative of nextgen processes.

3.1. K-mer Size

When decomposing reads into k-mers, the question of what size k to choose arises; practically speaking, current practice involves running the assembly process at multiple values of k and selecting the "best" assembly. Nevertheless, a relatively small range of k values are interesting. If the k value is too small, then some k-mers will show up in many places in the genome, which introduces unnecessary loops in the de Bruijn graph; for example, if we use a k value of 2 for the short sequence in Figure 1(a), the k-mer AT shows up three times.

For each genome, there is a threshold k value at which the bulk of such aliasing is eliminated. Figure 3(a) plots the fraction of k-mers in a reference genome that are unique. For these organisms, the k-mer space becomes large enough at $k = 17$ to eliminate most of the coincidental aliasing between two regions of a genome. The k at which this occurs is a weak function of genome size; larger genomes require slightly larger k values. Past this value, the fraction of unique k-mers levels off until very large values of k, due to large (*i.e.*, > 1 kilo-base) repeated regions in the genome.

Similarly, increasing the k-mer size increases our ability to distinguish errors from true sequence data. With larger k-mers, it is also less likely that an erroneous read will include k-mers that alias to another k-mer present in another read. As shown in Figure 3(b), the likelihood of aliasing decreases rapidly, becoming negligible in the same $k = 19$ to $k = 21$ region.

3.2. Required Read Coverage

While increasing k-mer size has benefits for error tolerance and eliminating aliasing, it comes at a cost of requiring higher coverage. Because we have little control over which fragments of DNA are sequenced, ensuring that we get at least a coverage of 1 on every k-mer in a genome requires an average coverage significantly above 1.

Figures 4(a) and (b) show the relationship between k-mer length, average coverage, and the fraction of the S. Suis P1/7

reference's k-mers we get at least once for reads without errors and with errors, respectively. As can be seen in both graphs, increasing k-mer size requires an increase in the average coverage to observe the same fraction of reference k-mers. Without errors, increasing k-mer length from 15 to 23 requires increasing average coverage from 12x to 20x to observe all of a genome's k-mers. Furthermore, the introduction of errors necessitates even larger average coverage; while a 16x average coverage is sufficient for a complete genome for $k = 21$, it requires 30x average coverage when errors are considered.

Empirically, for the genomes used in this paper, k values around 21 are frequently used as good trade-offs between accuracy and required coverage.

3.3. Size of the K-mer Space

The implications of using a k value of 21 may not be immediately obvious. Because there are 4 possible nucleotides, the k-mer space for $k = 21$ is 4^{21}, which is the same as 2^{42}. This is a rather large space with respect to memory sizes in contemporary commodity computers. A naïve implementation that counts the frequency of each k-mer in a set of reads by allocating a counter for each potential k-mer would require 4 Terabytes, even if only 1 byte were allocated for each counter. Clearly, we need to use data structures whose size is proportional to the number of distinct k-mers actually observed.

3.4. Distinct K-mers and their Coverage

The number of distinct k-mers observed has two components: 1) for all interesting values of k, the number of true k-mers is roughly equal to the length of the genome, and 2) because the errors are mostly independent and each error creates distinct k-mers with high likelihood, the number of erroneous k-mers tends to grow linearly with coverage. These trends can be seen from synthesized read data in Figure 5(a); furthermore, the number of erroneous k-mers (the difference between the two lines) dominates the overall number of unique k-mers for necessary levels of average coverage (*e.g.*, 30-50).

As previously noted, however, most of these erroneous k-mers can be detected relatively easily. As shown in Figure 5(b), true k-mers have much higher coverage than erroneous k-mers. Thus, we can generally distinguish erroneous and true k-mers by their coverage; the specific heuristics are beyond the scope of this paper.

The relative simplicity of pruning erroneous k-mers from a fully formed graph suggests a straightforward 2-step approach to assembly: step 1) construct a full graph using all observed k-mers; step 2) post-process the graph to eliminate the erroneous k-mers to produce the final assembly. The drawback of this simplistic approach is its peak memory requirement. As suggested by Figure 5(a), the de Bruijn graph including all of the erroneous k-mers is roughly 7 times larger than the error free graph. For a relatively small genome like S. Suis P1/7, this is the difference between a 100-200 MB and 1 GB of memory;

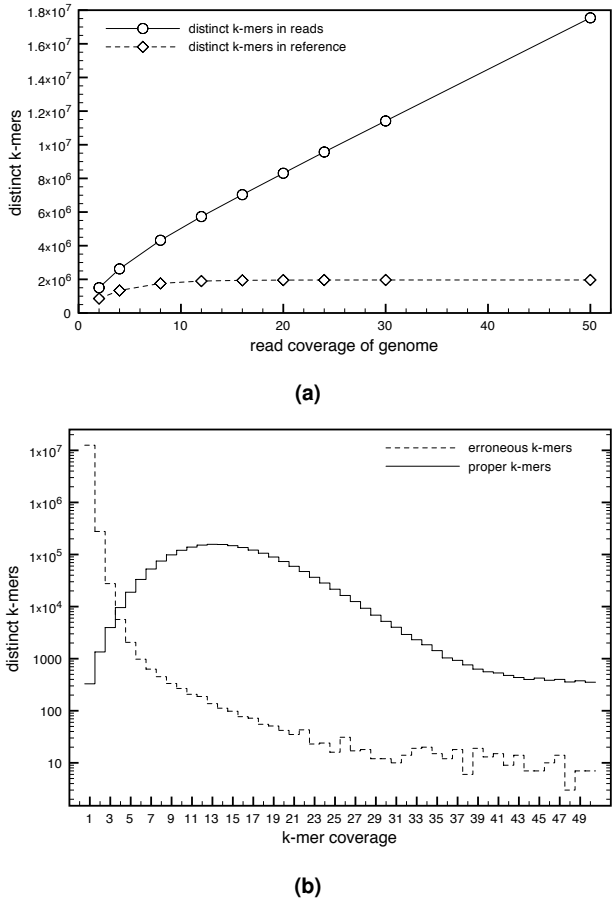


Figure 5: **Distinct K-mers and their Coverage** a) the number of distinct k-mers from the reference genome is limited by the size of the genome, but the number of erroneous k-mers is roughly linear with the amount of sequence data, and b) the distribution of coverages for erroneous and proper k-mers are largely non-overlapping; to first order, all coverage 1 k-mers are errors. Data: S. suis P1/7, 48x coverage real data-set, k=21.

for larger genomes this can delineate feasibility from infeasibility. In the next section, we consider a de Bruijn assembly code that mitigates peak memory usage by eliminating some erroneous k-mers before constructing the graph.

4. Velvet

In this section, we describe and provide insight into Velvet [14], the open-source short read de novo assembler with the highest performance to date; other short read de novo assemblers include Euler-SR [5], Edena [6], Allpaths [4], and Abyss [2]. We begin in Section 4.1 by describing Velvet's phases, and follow in Section 4.2 with a brief analysis of its scalar bottlenecks. As we will find, in addition to being memory footprint limited, Velvet is also often severly stalled on cache and TLB misses due to the inherently random accesses of its large data-structures. Guided by this analysis, we perform

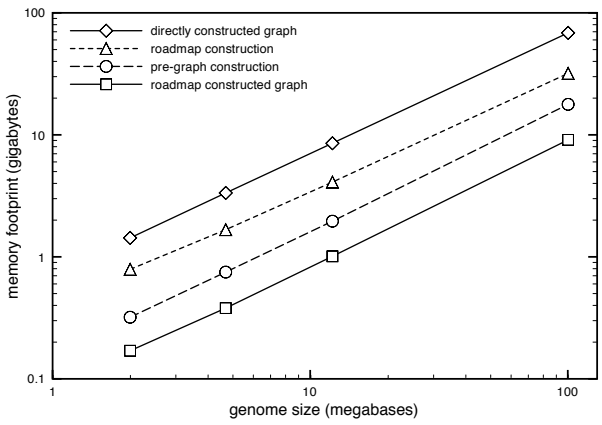


Figure 6: **Peak memory requirements for de Bruijn graph formation.** Results for S. suis, E. coli, S. cerevisiae (yeast), and C. elegans (round worm): simulated reads with coverage of 50x and k-mer size of 21.

memory-centric optimizations in Section 4.3.

4.1. Phases of Velvet

Velvet has seven distinct phases from a performance analysis perspective. In the first phase, *data-set format normalization*, the input data-set is processed from one of many file formats into a canonical format used by subsequent phases of Velvet; this simplifies the subsequent phases, and provides an implicit canonical (integer) identifier for each read.

The second phase, *roadmap construction*, processes this canonical data set into a set of annotations called the *Roadmap*, which enables the third phase, *pre-graph construction*, to reduce the size of the initial de Bruijn graph. In Figure 6, we compare the size of directly constructed de Bruijn graphs for several genomes (diamonds) to the size of the roadmap-constructed graphs (squares); the roadmap-constructed graphs are roughly a factor of 8 smaller, independent of genome size.

Although the peak memory footprint of the initial de Bruijn graph is reduced, the in-memory data structures for roadmap construction and pre-graph construction contribute to Velvet's (albeit smaller) peak memory footprints, as can be seen as the triangles and circles in Figure 6. We discuss roadmap construction in detail as we later focus our optimization efforts on this phase.

Roadmap construction builds a set of annotations for each read; when post-processed by the pre-graph construction phase, these annotations: 1) delineate which k-mers in the read are the canonical instances of those k-mers in the data-set; 2) identify the reads that reference those canonical k-mers; 3) identify the reads that contain the canonical k-mers for those k-mers in the read that are not canonical; 4) discern groups of k-mers that are consecutive in every read in which the k-mers are present.

With this information, pre-graph construction reduces the peak de Bruijn graph size using two properties: 1) those in-

```
1:  splay tree Y ← ∅
2:  for all reads R in data-set D do
3:    for all k-mer K in R do
4:      if K exists in splay tree Y then
5:        if K_previous maps to same read I'_{read−index} as K
          then
6:          memoize: consecutive to K_previous
7:        else
8:          if previous roadmap entry exists then
9:            emit: previous roadmap entry
10:         end if
11:         memoize: new roadmap entry
12:       end if
13:     else
14:       Y ← (map : K → I_{read−index}, P_{kmer−postion})
15:        where I_{read−index} is the integer index of the read
16:        where P_{kmer−position} is the position in the read
          the k-mer occurs
17:     end if
18:   end for
19: end for
```

Figure 7: **Pseudo-code for Velvet's "roadmap" algorithm.** Each k-mer occurrence is tested for presence in a splay tree; if not found, an entry records the location this occurrence (the canonical occurrence) came from; if it is found, then a roadmap entry is created to link this read's k-mer to the canonical k-mer.

read canonical k-mers which are not referenced by other reads (single copy k-mers) and which are not adjacent to referenced k-mers will not be connected to the rest of the de Bruijn graph, and thus, are erroneous; 2) the consistent groups of consecutive k-mers can be merged prior to graph formation, requiring only a single node to represent each group. We find that the first effect dominates, typically reducing the number of k-mers by a factor of 3, while the latter forms nodes of on average 2.4 k-mers.

The algorithm for roadmap formation is outlined in Figure 7. For each read R from the data-set D, and for each k-mer K at position P in R, a hash table of splay trees is queried to identify if K is the first occurrence in D. If it is, it creates an annotation with the index I of R in D and the position P, thus identifying a canonical location for each distinct k-mer. Like the directly constructed de Bruijn graph, the size of the splay trees is proportional to the number of distinct k-mers in D, but is smaller by a empirical factor of approximately two, as the splay trees use smaller nodes and do not track k-mer adjacency.

In the third phase, *pre-graph construction*, the roadmap is used in combination with the sequences from the input data-set to construct an initial de Bruijn graph, called the pre-graph. At this point, many pairs of nodes are each other's only neighbor on a side, and thus, by definition, can be merged with their neighbor into a single node. In addition, "tips" (short, low coverage growths off the graph (*e.g.*, the erroneous k-mers in Figure 2(c)) are removed. These two operations tend to reduce

Phase	Time	Component	Stalls
1 - Normalization	22%		computation intensive
2 - Roadmap Const.	25%	splay tree lookups (70%)	4% L1 miss, 51% L2 miss, 23% TLB miss, 14% mispredictions
		reverse complement (10%)	computation intensive
		output roadmap (20%)	
3 - Pre-graph Const.	28%	file I/O (20%)	64% I/O stalls
		graph construction (73%)	35% TLB miss, 32% mispredictions, 8% L1 and L2 misses
4,5 - Graph Con/Ann.	19%	binary search (80%)	50% TLB miss, 42% L2 miss, 6% mispredictions

Table 1: **Breakdown of major hot spots.** For the phases with non-negligible execution time, we report the major activities in these phases and the sources of stalls they incur.

the size of the graph by a factor of 10 to 100.

In the fourth phase, *full-graph construction*, the pre-graph is copied to construct an identical graph using larger node data structures, which are then populated with additional information in the fifth phase, *graph annotation*, with additional information from the input reads. This additional information is used by the *domain specific* phase, to perform heuristic transformations on the graph to further remove ambiguity due to errors. While this phase is important for the quality of Velvet's solution, its run time is negligible. Lastly, the *emit* phase computes the Eulerian path for each island in the graph, producing the output set of assembled contiguous sequences.

4.2. Scalar Performance Analysis

We used the Intel Performance Tuning Utility (PTU, the successor to Intel VTune) version 3.1 Update 3 to analyze the performance bottlenecks in Velvet version 0.7.09. We utilize both statistical sampling and interleaved performance counter data to identify the hot spots in the code and the reasons for stalls. Using Intel's suggested methodology [8], we find L2 data cache misses, TLB misses, and branch misprediction penalties to be the significant sources of stall time.

Our performance measurements were collected on a dual-core Intel Core 2 Duo 6700 2.66GHz with 2GB memory running RHEL 5.2 x86_64. Velvet was compiled with the Intel C++ Compiler v10.1.018 with full optimizations, but excluding feedback-directed optimization and inlining (for easing the attribution of performance data). As we will soon discuss, Velvet's poor data memory behavior meant that eliding inlining had little impact on this analysis.

To create a reproducible environment, we prefix each experiment run with a `sync`, flushing the Linux file cache (`/proc/sys/vm/drop_caches`) and reading a priming data-set from disk to ensure each benchmark run forces the input to be read off disk and to ensure no deferred writes to disk are pending. Furthermore, for overall timing, after completion of a run, Linux is forced to `sync` to ensure all written data is flushed from main memory. These conditions more closely simulate the conditions under larger inputs, presuming the application memory footprint does not begin to actively utilize the disk for virtual memory.

The breakdown of the performance characteristics of the phases that take non-negligible time are shown in Table 1. While the first phase takes a non-trivial amount of time, this is largely due to redundancy that is straight forward to eliminate, so we will no further discuss this phase. More interesting are the other hot spots which are dominated by the data memory system and branch stalls.

In phase 2, the bulk of the time is spent on splay tree accesses. As this splay tree becomes 100's of MB even for relatively small genomes, the lack of locality in the accesses result in L2 cache and TLB misses on most accesses. Phase 3 is similar, in that a substantial fraction of its computation is stalling roughly 75% of the time due to accesses of large data structures, and Phases 4 and 5 are even worse; with upwards of 80% of their execution time performing binary searches in a very large sorted array.

4.3. Memory-centric Optimization

In the previous section, we found that most phases of Velvet are almost entirely dominated by L2D cache misses and TLB misses, all involving structures that map k-mers to some information. Counter-productively, these structures organize the k-mers according to their integer value, which tends to scatter de Bruijn sequence k-mers randomly, and hence have poor locality. Despite this, TLB misses can be reduced by using support for large pages. Intel Core 2 Duo processors have support for 2MB pages, which enable a 500-fold increase in TLB reach with respect to the standard 4KB pages.

By reserving a number of large pages when the machine is booted and modifying Velvet to allocate these large data structures using these large pages, we reduce the number of TLB miss-induced stalls in the execution. Specifically, for roadmap construction, we allocate large pages for the hash table and the bucket allocator used for splay tree nodes; this reduces the average TLB miss rate on the hash table and splay tree nodes by 31%, resulting in a net 9.5% performance improvement for that phase. As the splay tree nodes are allocated in the order they

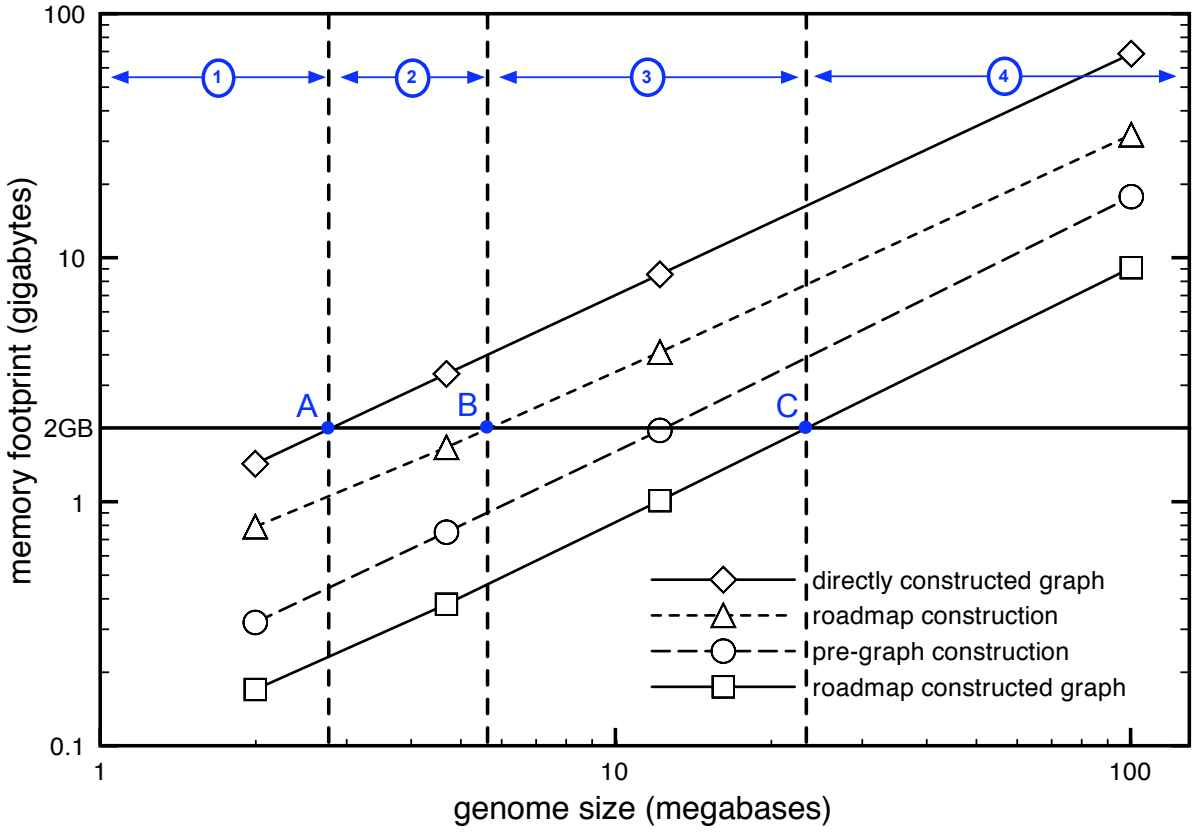


Figure 8: **Four performance domains of de Bruijn graph formation.** Example for a system with 2GB of memory. In region (1) direct graph construction does not oversubscribe memory; in (2), pre-processing to reduce node count (e.g., Velvet's roadmap construction) does not oversubscribe memory; in (3), pre-processed graph does not oversubscribe memory but pre-processing data-structure does; in (4), pre-processed graph oversubscribes system memory.

are observed in the input sequences, and later revisited in the same order, fewer TLB miss inducing page crossings occur.

To further improve the locality of roadmap construction, we explored buffering the queries to each of N regions of the hash table, choosing N such that the hash table and splay tree nodes for that region could reside in cache, minimizing L2D cache and TLB misses (for a sufficiently long buffer with repeated k-mers). In doing so, we eschewed the sequential query order that the original roadmap construction algorithm assumes. This caused the Roadmap output file to grow in our experiments by a factor of 4 to 5, because: 1) most consecutive k-mers did not collapse into a single annotation as they had; 2) extra overhead per annotation: each must include the subject sequence identifier, where originally this identifier only appeared once for all the sequence's annotations as they were consecutive in the Roadmap file.

This buffering optimization reduced the overall L2D cache miss rate by 65%, branch mispredictions by 89%, and TLB misses by 7%, but due to the factor of 4 to 5 growth in the Roadmap's file size, resulted in a net performance loss of over 35%. Although this loss might be alieviated by a high-performance disk system (where memory stalls would again limit performance), the corresponding growth in the number of annotations per read sequence still caused the memory footprint of the subsequent pre-graph construction phase to also quadruple, eliminating the memory footprint benefit of Roadmap-based de Bruijn graph construction.

5. Performance Scaling

Up to this point, we have identified two performance domains for de Bruijn assembly. The first performance domain corresponds to Region 1 in Figure 8, where the amount of system memory (here chosen to be 2GB) is larger than the memory required to perform a direct formation of the initial de Bruijn graph. When this is the case, it is more time efficient to directly build the graph and later remove erroneous nodes; in fact, the direct parallel formation is highly scalable and is likely to only be constrained by the sequential disk bandwidth of loading the data set for near-term multi-core processors.

The second performance domain (Region 2 in Figure 8) rep-

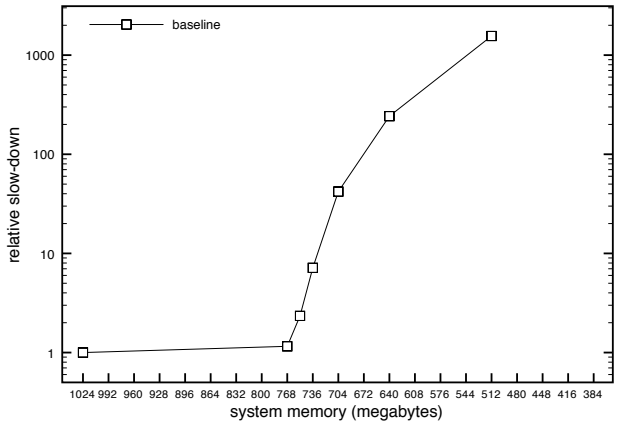


Figure 9: **Velvet begins to thrash when the k-mer splay trees do not fit in physical memory.** At less than 768MB the k-mer splay trees no longer fit in memory, as the memory size is further decreased, Velvet's performance drops precipitously.

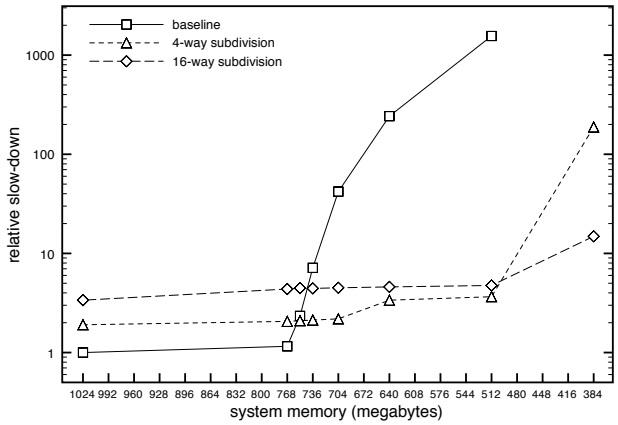


Figure 10: **Performance advantage of sub-dividing the k-mer space.** By subdividing the k-mer space into the largest chunks that will fit into physical memory we achieve a more graceful scaling of performance as we increase the genome size relative to memory. This scaling extends the scalability to the point where the initial filtered graph no longer fits in physical memory.

resents the case where the direct formation requires too much memory (crosses point A), but yet the pre-processing of the data set does not oversubscribe memory. Velvet is designed to excel in this region, with the roadmap construction reducing the peak memory footprint by roughly 2x, and thus enabling genomes of size 2x larger to be assembled efficiently.

When the genome size increases beyond a certain threshold, however, even the roadmap structure does not fit in memory (Region 3). In this region, the system still has sufficient memory for the to-be-formed initial de Bruijn graph in the third phase to fit, but the memory requirement for the roadmap splay trees is too high. In Section 5.1, we demonstrate the severe penalty for operating in this region, and then demonstrate (in Section 5.2) an algorithmic transformation to scale Velvet throughout Region 3, with only linear performance overhead.

Finally, in Region 4, even the roadmap-optimized initial de Bruijn graph is too large to fit into memory. In our future work, we hope to create new algorithms to efficiently subdivide the formation of the de Bruijn graph to further support larger and larger genomes.

5.1. Performance penalty of Region 3

In Region 3 of Figure 8, the number of unique k-mers becomes large enough that Velvet's roadmap data-structure (the splay trees) do not fit entirely into memory. When this occurs, we find an steep overt performance penalty as the virtual memory system pages the splay trees to disk. Figure 9 demonstrates this observation, where we scale the amount of system memory (using the kernel's `mem` parameter) down from 1GB to 512MB. For this data set, the splay trees are 560MB, and other (infrequently accessed) Velvet structures consume 360MB. At 768MB, most of the unused portion of the 360MB "other" has been paged to disk, incurring a minimal 20% penalty. However, as soon as the splay trees begin to page to disk, the performance penalty increases exponentially. This is due to the fact that the splay tree accesses have effectively no locality, and thus the cost of loading a virtual memory page from disk cannot be amortized across a sufficient number of accesses.

Once system memory is scaled to 512MB, we find that Velvet has a slowdown factor of 1558x. This corresponds to a 32 second execution now requiring over 14 hours! In the following section, we describe and demonstrate a technique to mitigate the scaling penalty of Region 3.

5.2. Subdividing the k-mer Space

A key insight to improving performance in Region 3 is recognizing that operations performed for a given k-mer must be performed *in order*, but that they are independent from those performed on other k-mers. The independence of these operations allows us to divide the k-mer space into a set of smaller sub-spaces that we can process iteratively. By dividing the k-mer space into N sub-spaces, we reduce the required size of the k-mer splay trees by a factor of N, but must now process the inputs N times; in addition, a linear time combination phase is also required to re-integrate the results. To be clear, if subdividing the space four ways, one only processes those k-mers that are prefixed with the nucleotide adenine (A) during the first iteration, then cytosine (C) the next iteration, and so on.

While both the original and the subdivision techniques will be disk limited for large genomes, the key is that the subdivision approach will have much better disk access locality. While paging the k-mer splay trees will introduce random accesses to the disk, repeatedly iterating through the reads consists of only sequential file I/O that should translate into mostly sequential disk accesses. The lack of seeks for these sequential accesses

and the operating system's ability to pre-fetch even when seeks are required would permit us to use close to the disk's full bandwidth.

In Figure 10, we show this technique implemented in Velvet; we plot the performance of 4- and 16-way subdivision of the k-mer space. In the regime where the un-divided k-mer splay trees fit in physical memory, the subdivided scheme is slower because it introduces additional passes through the reads; there is roughly a factor of two slowdown for each quadrupling of the passes through the input. When the k-mer splay trees exceed the size of physical memory, however, the performance of the subdivision k-mer splay trees remains stable, with the 4-way subdivision providing more than 400 times speedup at 512 MB physical memory. Past that point, however, a 4-way subdivision still results in k-mer splay trees too large for physical memory and performance drops off accordingly, but the 16-way subdivision continues to execute with more modest (factor of 15) slowdowns.

Clearly, because each configuration has a region in which it provides the best performance, the subdivision of k-mer splay trees needs to be done in an input-dependent manner. Fortunately, using the relations discussed in Section 3, it is relatively easy to approximately predict the number of distinct k-mers based on the size of the input data. With such a prediction and an estimate of the available memory, selecting an appropriate but conservative subdivision factor is near trivial.

It is important to re-emphasize, however, that subdividing the k-mer splay trees only addresses the peak memory requirements of the k-mer splay trees; it has no impact on the size of the initial filtered graph. Thus, for larger genomes the initial graph will exceed physical memory and serve as an bottleneck, effectively limiting the size of genomes that can be assembled.

6. Conclusion

In this paper, we have studied the state of the art approach to assembling sequence data from "next generation" high-throughput sequencing machines. Unlike many other bioinformatics codes which are computation- or control-intensive, the fundamental bottleneck of de novo sequence assembly is in its memory requirements. The focus of this paper is on characterizing and optimizing the memory requirements and behavior of these programs. Specifically, we characterized the relationship between genome size and memory requirements, demonstrated that filtering reads to remove errors before assembly can reduce the peak memory requirements, demonstrated the profitability of memory-centric optimizations, and proposed novel techniques for permitting these codes to scale to larger genomes.

While this work reduces the time to assemble a given genome and increases the size of genomes that a given workstation can assemble, future work must strive to efficiently extend de Bruijn graph-based assembly in to Region 4 of Figure 8. While a few attempts have been made to parallelize these assembly codes across a cluster of workstations to leverage the aggregate size of their memories [2, 13], these techniques are not particularly efficient because they randomly distribute the graph nodes across the clusters. As a result, frequent communication is required over the interconnect leading to run times that are 2 or 3 orders of magnitude longer than Velvet's in-core performance, in spite of their parallel execution.

Seemingly, the key would be the development of a technique that could partition the k-mers *prior to assembly* into groups that make up sub-assemblies, so that each sub-assembly could be constructed largely independently. Such a technique would enable both an efficient parallel assembly algorithm, as well as a single node assembly implementation that could efficiently assemble graphs much larger than it can hold in physical memory.

Acknowledgment

This research was supported in part by a gift from the Intel corporation, NSF CCF-0702501, and NSF CAREER award CCF-0347260.

References

[1] 454, http://www.454.com.

[2] Abyss, http://www.bcgsc.ca/platform/bioinfo/software/abyss.

[3] Illumina, http://www.illumina.com.

[4] J. Butler, I. Maccallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe. Allpaths: De novo assembly of whole-genome shotgun microreads. *Genome Research*, 18(5):810–820, 2008.

[5] M. J. Chaisson and P. A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Research*, 18(2):324–330, 2008.

[6] D. Hernandez, P. Francois, L. Farinelli, M. Osteras, and J. Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Res*, 18(5):802–809, 2008.

[7] N. C. Jones and P. A. Pevzner. *An Introduction to Bioinformatics Algorithms (Computational Molecular Biology)*. The MIT Press, August 2004.

[8] D. Levinthal. Cycle accounting analysis on Intel Core 2 processors.

[9] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci U S A*, 98(17):9748–9753, 2001.

[10] M. Pop, S. L. Salzberg, and M. Shumway. Genome sequence assembly: algorithms and issues. *Computer*, 35(7):47–54, July 2002.

[11] M. S. R. Schmid, S.C. Schuster and D. Huson. Readsim- a simulator for sanger and 454 sequencing. in preparation, 2008.

[12] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proc Natl Acad Sci U S A*, 74(12):5463–5467, 1977 Dec.

[13] W. Shi and W. Zhou. A parallel euler approach for large-scale biological sequence assembly. In *Information Technology and Applications*, 2005.

[14] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829, 2008.