Parallel de novo Assembly of Large Genomes from High-Throughput Short Reads

B.G. Jackson†, M. Regennitter†, X. Yang†, P.S. Schnable‡, and S. Aluru†
†*Department of Electrical and Computer Engineering*
‡*Department of Agronomy*
Iowa State University, Ames, IA, USA

***Abstract*—The advent of high-throughput short read technology is revolutionizing life sciences by providing an inexpensive way to sequence genomes at high coverage. Exploiting this technology requires the development of a *de novo* short read assembler, which is an important open problem that is garnering significant research effort. Current methods are largely limited to microbial organisms, whose genomes are two to three orders of magnitude smaller than complex mammalian and plant genomes. In this paper, we present the design and development of a parallel *de novo* short read assembler that can scale to large genomes with high coverage. Our approach is based on the string graph formulation. Input reads are mapped to short paths, and the genome is reconstructed as a superpath anchored by distance constraints inferred from read pairs. Our method can handle a mixture of multiple read sizes and multiple paired read distances. We present parallel algorithms for string graph construction, string graph compaction, graph based error detection and removal, and computing aggregate summarization of paired read links across graph edges. Using this, we navigate the final graph structure to reproduce large contiguous sequences from the underlying genome. We present a validation of our framework on experimental and simulated data from multiple known genomes and present scaling results on IBM Blue Gene/L.**

I. Introduction

Biologists have been sequencing genomes for the past 25 years using Sanger sequencing technology, a process that enables sequencing a DNA molecule of length 700-1000 base pairs (bp) in a single experiment. When applying this technology to a genome that is millions to billions of bases in length, many such *Sanger reads* are sampled, with multiple sequencing machines operating concurrently to shorten the sequencing timeline. These reads are then combined to reconstruct the originating genome using software known as a *genome assembler*. To accurately reconstruct the genome using this technology, one requires at least a fivefold to sevenfold coverage of the genome, where *coverage* is defined as the ratio of the total length of all the reads to the length of the genome. For organisms such as human, mouse, and maize with genomes in the neighborhood of 3 billion bases, this translates to about 30 million reads. Although the cost of a single Sanger read is low, the large number of reads drives the cost of large genome sequencing projects into the tens of millions of dollars, which in turn has limited large genome sequencing to a handful of species.

Recently developed, high-throughput, short read sequencing is proving a disruptive technology as it allows concurrent generation of millions of reads at a significantly lower per base cost, albeit with limitations on read length (35-75 bp typically, with the exception of 454, which can produce up to 400 bp reads). Several short read platforms are now available and seeing rapid adoption in the experimental biology community (454 Life Sciences system [13], Illumina Solexa [1], Applied Biosystems SOLiD [16], and Helicos Biosciences Heliscope [21]). To compensate for the short read length — which makes it difficult to distinguish between overlaps due to genomic co-location, overlaps due to genomic repeats, and overlaps due to random chance — a much higher 100-fold to 300-fold coverage is needed for accurate assembly. Moreover, the number of short reads needed to reach a given coverage level is about 20 times higher than the number of corresponding Sanger reads. These factors contribute to a 300-fold to 1000-fold increase in the number of reads and significantly complicate the assembly problem.

A major challenge in genome assembly is to resolve repeats accurately; a repeat can cause overlaps between reads derived from different parts of the genome, and these must be differentiated from overlaps caused by genomic co-location. To help with this, biologists sequence reads in pairs, from both ends of a longer fragment whose approximate length is known (typically to $\pm 10\%$ accuracy). These approximate distance constraints are an excellent source of information for resolving ambiguity. While their use has been limited to validating assembly in most conventional assemblers, it is increasingly seen as important to use this information to guide the assembly process itself for assembling short reads. Intuitively, paired reads from multiple lengths of fragments (*fragment types*) provide additional useful information; we can associate a group of overlapping reads with paired reads at multiple approximately known distances in either direction along the genome. However, as fragments are collectively prepared and too many fragment sizes can increase the cost and complexity of both fragment preparation and tracking fragment type through the course of sequencing, it is prudent to create a few fragment types, rather than indulge in different lengths for every fragment.

Recent large scale Sanger sequencing projects cost between $30 and $60 million. If we could perform accurate *de novo* assembly from short reads, we could lower the cost of sequencing new genomes of the same complexity to

around one million dollars, a significant financial savings. This makes *de novo* short read assembly an important open research problem, and there has been a flurry of activity in response. Several short read assemblers have been developed in the last few years, each using as a foundation graph based formulations laid in landmark papers by Idury and Waterman [8], and Myers [15]. The assemblers — which include ALLPATHS [2], Euler-SR [3], SHARCGS [4], Shorty [7], Edena [6], Medvedev *et al.*'s assembler [14], SSAKE [20], Velvet [22], and ABySS [19] — differ in many details, such as the types of graphs they construct, the way they handle sequencing errors, and their ability to handle different fragment lengths. With the exception of ABySS, the assemblers are sequential, and this largely limits their applicability to microbial organisms.

In this paper, we report the development of a parallel *de novo* short read assembler which we named YAGA (for Yet Another Genome Assembler). Our work is aimed at using parallel computers to enable large-scale short read assemblies and remove the scaling limitation posed by most other programs. Our development is by and large concurrent with ABySS [19], with our initial development somewhat predating it [10], [12], [11]. Despite their similar focus on parallelism, the methods have important differences. ABySS uses graph reductions to produce contigs while YAGA uses a path walking strategy. Rather than construct and manipulate a directed graph, YAGA constructs and manipulates a bidirected graph, which, while being slightly more complicated, naturally represents paired DNA molecules. This allows us to completely avoid the problem of strand selection, or resolving the strand of DNA from which each read came. Unlike ABySS, YAGA was designed to make use of multiple fragment lengths. A comparison of the various short read assemblers is provided in Table I.

In this paper, we present the parallel algorithms behind the YAGA short read assembler. In Section II, We begin with a brief description of our problem formulation and the notation used subsequently. Sections III, IV, and V contain our parallel algorithms for graph construction, graph compaction, and error detection and removal (different from error correction, which attempts to change erroneous reads into their correct forms). In Section VI, we present a parallel method for computing aggregate summaries of the distance constraints, which will be subsequently used to direct graph traversal. These steps cause a thousand-fold reduction in the graph size and raw distance constraint information. The resulting graph and aggregate information is then processed sequentially to discover paths corresponding to contiguous stretches of the genome, as outline in Section VII. We present experimental results in Section VIII. Section IX concludes the paper.

II. Bidirected Graph Formulation

For a string α of length $|\alpha| = l$, we denote the i^{th} character of α as $\alpha[i]$, $1 \leq i \leq l$. We denote the substring from $\alpha[i]$ to $\alpha[j]$, inclusive, as $\alpha[i,j]$, $1 \leq i \leq j \leq l$. A *DNA strand* is a string with alphabet $\Sigma = \{a, g, c, t\}$. We call the characters of a DNA strand *bases*. The *complement* of a base $\alpha[i]$, denoted by $\alpha[i]'$, is defined by the following bijection of Σ onto Σ: $\{t \rightarrow a,\ c \rightarrow g,\ a \rightarrow t,\ g \rightarrow c\}$. The *reverse complement* of a DNA strand α, denoted by α', is obtained by reversing α and complementing each base ($\alpha'[i] = \alpha[l-i+1]'$). Note that $\alpha[i] = \alpha[i]''$ and $\alpha = \alpha''$.

A *DNA molecule* is a pair of complementary DNA strands, $m = \{\alpha_m, \alpha'_m\}$. We denote the length of m as $|m| = |\alpha_m| = h$ and call m an h-molecule. We designate the lexicographically larger of the two strands as the positive strand, denoted m^+, and the lexicographically smaller of the two strands as the negative strand, denoted m^-. We choose the ordered tuple $m = \langle m^+, m^- \rangle$ as a canonical representation of the molecule. Note that because we can find m^- from m^+ by computing the reverse complement, we can represent a molecule using only m^+. For this reason we also call m^+ the *representative* strand.

A sub-molecule $m_{[i,j]}$ of molecule m is the molecule $\{m^+[i,j], m^-[|m|-j+1, |m|-i+1]\}$. We denote the positive strand of the sub-molecule as $m^+_{[i,j]}$ and the negative strand as $m^-_{[i,j]}$. Note that either the substrand $m^+[i,j]$ or the substrand $m^-[|m|-j+1, |m|-i+1]$ might be $m^+_{[i,j]}$.

We say that we *extend* a molecule by adding a base to the end of one strand and the complementary base to the beginning of the other strand. If we add a base to the end of the positive strand, we call this operation a *positive extension*. We denote a positive extension as mc, where c is the base appended to the positive strand. If we at the same time remove a base from the beginning of the positive strand, this operation becomes a *positive shift*, denoted $\overrightarrow{mc}$. Correspondingly, we denote a *negative extension* as cm, where c is the base appended to the negative strand. If we at the same time remove from the beginning of the negative strand, we call this operation a *negative shift*, denoted $\overleftarrow{cm}$.

The *genomic DNA* of an organism is a small set of long molecules $\mathcal{M} = \{M_1, M_2, ...M_c\}$ (called *chromosomes*; for example, the human genome consists of 23 chromosomes of total length nearly 3.2 billion). A *read* is a sub-molecule taken from this set, $M_{h[i,j]}$, $1 \leq h \leq c$, $r_{min} \leq j - i + 1 \leq r_{max}$, where r_{min} and r_{max} are the minimum and maximum possible read lengths determined by the experimental process. A sequencing machine does not always sequence the read accurately – approximately 1% error rate is expected with current technology. Short read sequencing errors typically take the form of base miscalls, although insertions and deletions are also possible.

A *bidirected graph* is a graph $G = \{V, E\}$ where edges are of the form $(\langle u, d_u \rangle, \langle v, d_v \rangle)$, where $d_u \in \{out, in\}$ is

Table I
A COMPARISON OF SHORT SEQUENCE ASSEMBLERS BY FUNCTIONALITY AS HAS BEEN DESCRIBED IN THE CITED PAPERS. [1]BY PARALLEL, WE MEAN LARGE SCALE PARALLELISM ACROSS MULTIPLE COMPUTING NODES. [2]THE ALLPATHS ASSEMBLER REQUIRES EXACTLY THREE FRAGMENT LENGTHS: SHORT (FOR EXAMPLE 250 BASES), MEDIUM (FOR EXAMPLE 2000 BASES), AND LONG (FOR EXAMPLE 10,000 BASES).

	Whole Genome	Error Correction	Paired Reads	Single Length	Multiple Lengths	Bidirected Model	Parallel [1]
YAGA [9]	X		X	X	X	X	X
ABySS [19]	X		X	X			X
Medvedev [14]	X		X	X		X	
Myers [15]	X					X	
ALLPATHS [2]	X	X	X		X[2]		
Euler-SR [3]	X	X	X	X			
Velvet [22]	X		X	X			
Shorty [7]	X	X	X	X			
SHARCGS [4]	X						
SSAKE [20]	X						
Edena[6]	X						

Figure 1. The four ways in which nodes are connected in the bidirected de Bruijn graph, with the edges labeled with the initial characters in the bidirected k-string graph (before edge compaction).

the direction of the edge at node u. For sequence assembly, we represent the reads as short paths in a bidirected graph using the de Bruijn graph formulation. We create a graph in which nodes correspond to length k sub-molecules of reads, also called the *k-spectrum* of the reads. The edges correspond to the length $k+1$ sub-molecules of reads, called the *(k+1)-spectrum*. An edge $(\langle u, d_u\rangle, \langle v, d_v\rangle)$ exists between nodes u and v only if there exists a shift that transforms the molecule corresponding to u to the molecule corresponding to v (and v to u). If u is transformed to v by a positive shift, then $d_u = out$. If u is transformed to v by a negative shift, then $d_u = in$. The arrow head at v is defined similarly. The four possible edge types are depicted in Figure 1. Note that the bidirected graph so constructed is a subgraph of the *bidirected de Bruijn graph*, which would contain all possible k-molecules as nodes, and the full set of edges; i.e., the bidirected de Bruijn graph has $|\Sigma|^k$ nodes and $|\Sigma|^{k+1}$ edges.

A traversal of a bidirected graph must respect edge directions at each node – if entering on an out direction, one must exit on an in direction, and vice versa. Note that a read of length r contains $r-k+1$ constituent k-molecules, and is represented by a path connecting them in a chain, obeying the above traversal rule. We label each edge in the graph with two characters (which will be generalized to strings later) to create a bidirected string graph. Each label corresponds to the first character of each strand of the $(k+1)$-molecule corresponding to the edge. Each character is associated with a traversal direction, again corresponding to the strand directions of the $(k+1)$-molecule (see Figure 1).

Formally, let m_u and m_v denote the molecules labeling u and v, respectively. We define characters labeling the edge connecting them, c_{uv} and c_{vu}, as:

$$c_{uv} = \begin{cases} m_u^+[1] & \text{if } d_u = \text{out} \\ m_u^-[1] & \text{otherwise} \end{cases}$$

$$c_{vu} = \begin{cases} m_v^+[1] & \text{if } d_v = \text{out} \\ m_v^-[1] & \text{otherwise} \end{cases}$$

The key property of the string graph is that each read now can be recovered by traversing the corresponding path in either direction, concatenating the edge labels on the path and adding $(k-1)$-molecule suffix of the last node on the path. As all input reads are derived from chromosomes, each chromosome can now be seen as a long path in this graph containing the paths of the reads that came from this chromosome. In real-life, one would not expect to see continuity in sampling, and the goal is to recover the genome as a large set of *contigs*, or contiguous sequences.

III. PARALLEL GRAPH CONSTRUCTION

Throughout the assembly process, we create and manipulate a bidirected graph represented by a distributed list of edges. We have developed our assembly algorithms on top of a small set of common parallel primitives:

1) Parallel sorting - sorting is used to rearrange edge tuples in various ways to discover and manipulate graph structure [5].
2) Redistribution - redistributing an array of tuples based on keys given in another array. This is equivalent to the Random Access Read (RAR) operation and translates to a many-to-many communication on parallel computers [17].
3) Prefix sums - used to assign unique ranks as needed.

These primitive operations are well studied, and efficient algorithms that scale to large systems are readily available. Our choice to build the assembler on top of these operations decouples it from a specific implementation, environment, or system. For example, one can substitute disk-bound algorithms for these primitives and derive an out-of-core implementation of the assembler.

We construct the graph by directly generating edges corresponding to $(k+1)$-length submolecules of the input reads. Reads are initially distributed such that the total length of short reads on each processor is approximately the same. Each read is scanned to identify all constituent $(k+1)$-molecules and generate all corresponding edges. Note that the same $(k+1)$-molecule may be present in several short reads, and in fact it is expected due to the high coverage. In addition, if the $(k+1)$-molecule is a repeat, its frequency will increase with its multiplicity. Once all of the edges are created, they are sorted to eliminate duplicates while maintaining the frequency count. Thus, one parallel sort accomplishes the task of graph construction.

In an earlier implementation of our method and in the ABySS assembler, the assembly graph is constructed by directly creating nodes in the graph from the observed k-spectrum and then generating edges between pairs of nodes whose corresponding k-molecules obey the shift property described in the previous section. To this end, the four (limited by alphabet size) potential edges corresponding to the four possible shifts starting at a node are created, and then the validity of each potential edge is confirmed by checking the existence of the node at the other end of the edge. Thus, at the cost of additional space requirements (from generating hypothetical edges) and communication time, node based construction might correctly infer a small number of valid edges missed by edge based construction.

The primary problem with node based graph generation is the creation of erroneous edges when length $k-1$ strings are *maximal repeats* in the genome [9]. To illustrate how erroneous edges are created, consider a $(k-1)$ length molecule α that occurs exactly twice in the genome, as a part of $(k+1)$-molecules $A\alpha T$ and $G\alpha A$. Using the node-based construction method described above, edges corresponding to $(k+1)$-molecules $A\alpha A$ and $G\alpha T$ would exist in the graph, even though no such molecules exist in the genome. Thus, all edges that node based construction generates are, with low probability, suspect, and an assembler that relies on node based construction must answer the challenge of validating all edges in the graph at a later phase in assembly. On the other hand, an assembler using edge based construction can assume that, after correcting for sequencing error, all edges are valid and the genome corresponds to a tour of the graph that visits all edges at least once.

In order to further reduce the memory footprint per node, we process the input in batches of reads. Each batch of reads is converted to the corresponding edges, and sorted in conjunction with the existing set of edges to eliminate duplicates and maintain the frequency counts. By iteratively sorting in multiple stages and eliminating duplicates frequently, the storage required is limited to the size of the graph rather than the input short reads. Note that, ignoring the accumulation of errors, the graph size is bounded by the length of the genome, irrespective of the choice of k and no matter what the coverage of the genome is (even if it is infinite).

Due to sequencing errors, invalid edges will accumulate as low frequency edges in our edge list. Ideally the parallel system would have sufficient memory to store all edges generated as we process reads, including edges generated due to errors. However, in the case of very large genomes and very high coverage, it might be necessary to cull these errors after processing only a subset of reads. A simple approach would be to periodically remove single occurrence edges from the list as memory limits are reached, while a more complicated approach would repeatedly employ the compaction and error detection methods described in the next two sections.

We represent an edge using one strand of the corresponding $(k+1)$-molecule. We arbitrarily choose the lexicographically larger of the two strands as the canonical representation. The corresponding complementary strand can be computed on demand. The k-molecules corresponding to the two nodes incident to an edge can be easily derived from the $(k+1)$-molecule for the edge. To further realize efficiencies, we use a $2(k+1)$ bit integer to denote the edge as each base can be represented using two bits.

IV. Parallel Graph Compaction

The first step in the assembly process is to identify chains in the assembly graph with no further branching, and replace them with single edges. The label of the newly created edge is the concatenation of all the edge labels on the path. This is expected to resolve all the unique regions of the genome and the size of the resulting string graph is significantly reduced.

For convenience of understanding, view an edge connecting nodes u and v in the graph as $\langle u, v\rangle$. For many phases of the algorithm, we find it necessary to create for each edge two entries, one labeled $\langle u, v\rangle$, and the other labeled $\langle v, u\rangle$. Both edges have the same edge identifier and other information. For the purpose of graph compaction, we wish to identify multiple chains in the graph simultaneously and compact them simultaneously. First, we sort the edges by

using the first node as the key - i.e, the $\langle u, v\rangle$ representation of the edge is bucketed based on u and the $\langle v, u\rangle$ representation of the same edge is bucketed based on v. This brings together all edges incident to a node. If a node is incident to one edge or more than two edges, it is treated as the termination of a chain. If a node has two incident edges, the directionality of the edges is checked to see if a traversal through the node is possible. If so, the corresponding edges are part of a chain. Thus, all edges that are part of a chain can be identified.

For the purpose of compaction, we view edges in the graph as nodes for the input to a variation of the classic parallel *list ranking problem*, in which a linked list is distributed on processors such that each node in the list has a pointer to the next [18]. It is desired to find the distance of each node to the end of the list. In our case, we are ranking undirected lists and compute the distance from both ends of the lists, operating on multiple lists simultaneously. These generalizations are fairly straightforward and do not alter the complexity of original list ranking algorithm. Hence, they are not described in greater detail.

In order to compact chains using our list ranking generalization, we need a pointer from each edge in a chain to two adjacent edges in the chain, each of which can be found when the edges are sorted by their first node as above. To bring these two adjacencies together, we sort the edge entries a second time, using their smaller node labels as primary key and the larger node labels as the secondary key. This brings both representations $\langle u, v\rangle$ and $\langle v, u\rangle$ of a graph edge together, at which point they can be merged into a single node with adjacency pointers for list ranking. After list ranking, each compacted chain is replaced by a single edge with a longer edge label, along with the average $(k+1)$-molecule frequency of its constituent edges.

V. Identification and Removal of Errors

It is commonly known that an assembler is not well crafted unless it accounts for sequencing errors. To identify errors, we make use of the common assumption that incidence of errors is random, and therefore an error is unlikely to occur repeatedly at the same base. At the same time, each base in the genome is sampled on an average as many times as the coverage number, which is high in case of short read sequencing. The conjunction of these two ideas points to identifying errors based on their comparatively lower frequency compared to correct sampling.

With this idea in mind, we (and others [22]) have developed a method for identification and removal of errors that relies on graph editing. We consider several cases of potential errors in the reads and identify how these errors would manifest in the graph topology. We then look for such topological characteristics and remove them from the graph. The three identifiable motifs are depicted in Figure 2.

To find these motifs, we store enriching information with each edge entry $\langle u, v\rangle$ in the distributed edge list of the compacted graph — the number of in edges and out edges for both u and v. As was done in compaction, this information is added to the edge entry using parallel sorts. First, the entries are sorted by the first node, which brings all edges adjacent to that node together. After this sort, the number of in and out edges at the node is associated with each of the edge entries. Then the edges are sorted canonically to bring $\langle u, v\rangle$ and $\langle v, u\rangle$ together. At this point, the two entries exchange the information generated about the two nodes.

Tips: A commonly occurring error is the misreading of one or more bases towards the end of the short read, such that a misread base is no more than k distance from the end. In this case, the error manifests itself as a tip in the graph. A tip consists of a low frequency edge attached to a node which is on a path with high frequency edges. Essentially, the path with high frequency edges is part of the genomic sequence, while the edge leading to the tip is due to the erroneous base in one read or a small number of reads. A special case of a tip occurs when a read contains errors such that there is no error free portion of the read of length $k+1$. In this case, the read becomes a singleton, unconnected edge in the graph. We identify all tips and singletons in the graph using the information gathered above and remove them.

Bubbles: A bubble in the graph occurs when the error is the same as described above but occurs in the middle of a short read. In this case, a sequence of $(k+1)$-molecules that contain this erroneous base are corrupted and together form a chain, while the correct sequence of $(k+1)$-molecules that come from the many error-free reads form another chain. Due to chain compaction carried out previously, both chains are now edges. Note that even though the two edges are incident to the same pair of nodes, they are still different edges due to differences in the strings labeling the edges. It is important to differentiate bubbles that arise due to error and bubbles that arise from nearly identical repeats, which are regions of the genome that are nearly the same. Thus to remove a bubble edge, we require that the edge have a very low frequency. Bubbles can be identified by sorting edges by the first endpoint, thus bringing low frequency and high frequency edges together.

Spurious Links: A spurious link occurs when an erroneous $(k+1)$-molecule happens to be identical to a legitimate $(k+1)$-molecule from elsewhere in the genome. This is manifested as a low frequency link between two nodes both of which are on a high frequency path. Removal of spurious links is very important, as a path that traverses them will erroneously jump from one part of the genome to another. Once again, spurious links can be identified using the information gathered above.

Because removing error motifs might expose additional error motifs that had been previously masked, we perform

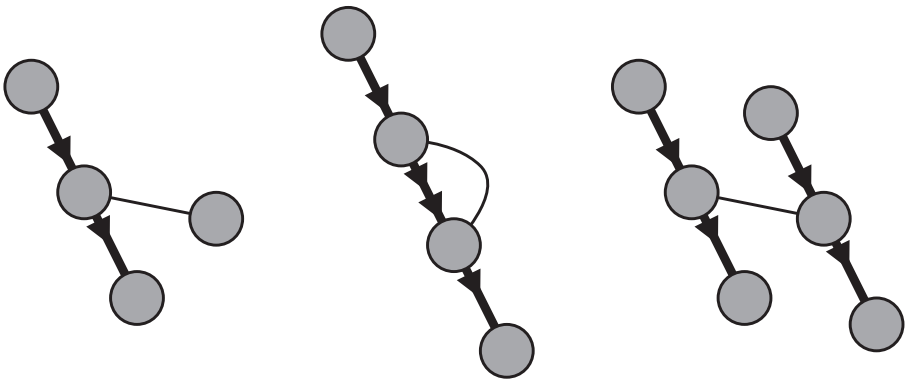

Figure 2. Motifs used to identify errors with frequency indicated by edge thickness. From left to right: a tip, a bubble, and a spurious link.

the error removal described above iteratively until no error motifs are detected. After each round of error removal, the graph is again compacted, as removing edges might create new chains. We have found that in practice only a few such iterations are sufficient before the method terminates.

After compaction and error removal, we are left with a graph where each node is either an end node incident to one edge or is incident to three or more edges (Figure 3). Ideally, each edge in the graph is a maximal repeat or a unique region and all that is left is to identify which traversal of the graph reconstructs the genome, with edges corresponding to a repeats traversed multiple times. The frequency of a repeat edge should grow in accordance with the number of times the repeat occurs in the genome, assuming the short reads constitute a uniformly random sampling of the genome. Thus, we can deduce the approximate traversal counts for each edge. Myers pioneered this approach [15] where he provides a fairly accurate traversal count for each edge by transforming this problem to a min-cost network flow. In our case, we assign upper bounds and lower bounds for traversal count of an edge simply based on its frequency and rely upon paired distances to guide us to accurate traversal.

VI. Summarizing Clusters of Paired Reads

In the sequencing process, the length of each fragment is selected by a physical process that produces lengths that lie within a predefined range (mean $\pm 10\%$). We assign all pairs coming from fragments sharing the same length range a single *pair type.* For short read assembly projects, multiple fragment libraries are created, sometimes with differing ranges (with *differing types* in our nomenclature).

Each pair of reads thus gives us an approximate distance constraint that we can use when traversing the assembly graph; if the paired reads map to two edges in the graph, we know that those two edges are separated by an approximate distance. For YAGA, we demonstrate that by considering multiple observations concurrently, we create a much better estimate of the distance between edges. To this end, we group paired reads that carry complementary and redundant information in a process we call $(k+1)$-*pair clustering*, summarizing each cluster as an interval. Each summary is orders of magnitude smaller than its corresponding cluster; thus this process greatly reduces the problem size. We use these clusters, which provide reasonably tight distance constraints between edges, to guide our traversal of the assembly graph.

A. Generating summary distance constraints

We denote a *position* in the bidirected graph G as $p = \langle e, f \rangle$, where e is an edge identifier and f is a position within the edge label in an arbitrarily assigned forward direction. By construction of the string graph, there is a bijection between valid $(k+1)$-molecules in the input and the set of all positions within all edge labels in the graph. We use $p(m)$ to denote the position corresponding to $(k+1)$-molecule m, and $p(m).e$ and $p(m).f$ to denote the corresponding fields.

We denote a *read pair* as $\langle R_1, R_2, z \rangle$, where R_1 and R_2 are the reads and z is the pair type. Each read pair contains multiple $(k+1)$-*pairs*, of the form $\pi = \langle m_1, m_2, z \rangle$, where m_1 and m_2 are molecules taken from R_1 and R_2 respectively. We denote the set of all $(k+1)$-pairs in the data as $\Pi = \{\pi_1, \pi_2, ..., \pi_M\}$.

An *edge traversal* $t = \langle e, d \rangle$ is composed of an edge e and a traversal direction $d \in \{F, R\}$, with F corresponding to traversing the edge in the forward direction. A *path* is a sequence of edge traversals: $T = \langle t_1, t_2, ..., t_l \rangle$. In general, edges in the graph can be traversed multiple times; there could exist t_i and t_j, $i \neq j$ and $e_i = e_j$. A path must also be a valid traversal of the string graph.

Consider some $\pi_x = \langle m_{1x}, m_{2x}, z_x \rangle$ and traversal T. Let $\mathcal{L}_x = \{t_i | e_i = p(m_{1x}).e\}$ be the set of edge traversals in T to which m_{1x} maps. Let $\mathcal{R}_x = \{t_j | e_j = p(m_{2x}).e\}$ be the set of all edge traversals to which m_{2x} maps.

Definition 1: For each $(t_i, t_j) \in \mathcal{L}_x \times \mathcal{R}_x$, $i < j$, the

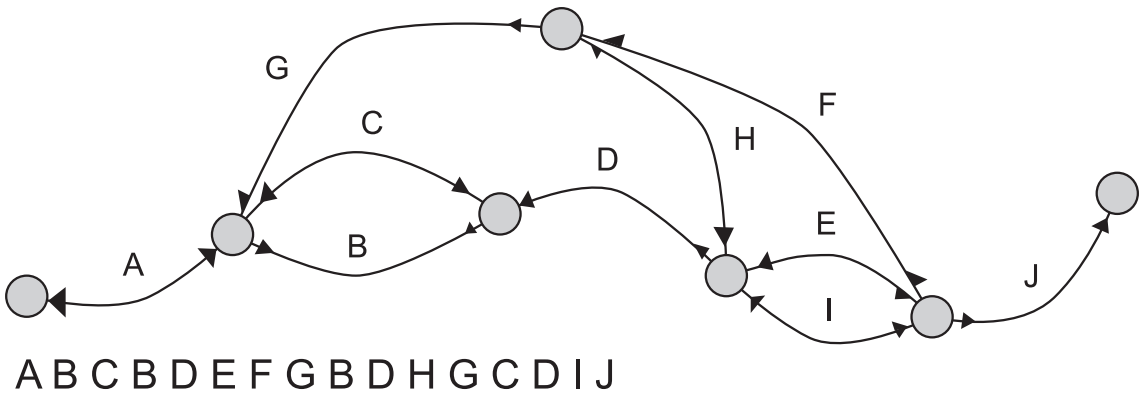

Figure 3. An example genome with repeats and the resulting bidirected k-string graph. The genome is given as a sequence of maximal repeat or unique regions, each labeled with a letter from the English alphabet. We draw the graph nodes as gray circles and label the edges using the corresponding letters.

observed distance of π_x is:

$$d(\pi_x, t_i, t_j) = \sum_{h=i+1}^{j-1} \|e_h\| + \sigma_i + \sigma_j$$

$$\sigma_i = \begin{cases} p(m_{1x}).f & \text{if } d_i = R \\ \|p(m_{1x}).e\| - p(m_{1x}).f & \text{if } d_i = F \end{cases}$$

$$\sigma_j = \begin{cases} p(m_{2x}).f & \text{if } d_j = F \\ \|p(m_{2x}).e\| - p(m_{2x}).f & \text{if } d_j = R \end{cases}$$

We use $\lceil z \rceil$ to denote the largest possible distance between observed $(k+1)$-molecules when reading the ends of a fragment of type z and $\lfloor z \rfloor$ to denote the smallest possible distance. We say that π_x *supports* T using t_i and t_j if and only if $\lfloor z_x \rfloor \leq d(\pi_x, t_i, t_j) \leq \lceil z_x \rceil$, and call this support weak because the true distance between t_i and t_j can differ from the observed distance by as much as $\lceil z_x \rceil - \lfloor z_x \rfloor$.

In general, multiple $(k+1)$-pairs with the same type can support a pair of edge traversals on a path. We wish to formalize this support expectation.

Definition 2: The **maximum distance expectation** for t_i and t_j and some type z, denoted by $\lceil (t_i, t_j, z) \rceil$, is calculated as $\min\left(\lceil z \rceil, \sum_{h=i}^{j} \|e_h\|\right)$.

Definition 3: The **minimum distance expectation** for t_i and t_j and some type z, denoted by $\lfloor (t_i, t_j, z) \rfloor$, is calculated as $\max\left(\lfloor z \rfloor, \sum_{h=i+1}^{j-1} \|e_h\|\right)$.

Definition 4: A $(k+1)$**-pair cluster** is a set of observed distances for t_i, t_j, and z that have been grouped together via a clustering algorithm (which we shall describe shortly). We summarize a cluster using the interval $\alpha(t_i, t_j, z) = [min, max]$, with min being the minimum observed distance in the cluster and max the maximum observed distance. We say that t_i and t_j are *supported by* a $(k+1)$-pair cluster $\alpha(t_i, t_j, z)$ if $\alpha_{min} < \lfloor (t_i, t_j, z) \rfloor + T$ and $\alpha_{max} > \lceil (t_i, t_j, z) \rceil - T$, with T a sensitivity parameter.

For efficiency reasons, we want to know if t_i and t_j are supported without having to consider the entire set Π when analyzing a particular path. To achieve this, we preprocess the raw paired reads to extract all clusters without any *a priori* knowledge of the nature of the eventual traversal T. In other words, we know neither the distance between pairs of edges nor their relative orientations at the time of summarization.

We notice that any clustering algorithm that relies exclusively on the difference between observed distances $d(\pi_1, t_i, t_j)$ and $d(\pi_2, t_i, t_j)$ can produce a clustering without such *a priori* knowledge. This is because any difference remains invariant when changing the traversal direction of both t_i and t_j concurrently, adding an edge between t_i and t_j, removing and edge between t_i and t_j, or changing the traversal direction of an edge between t_i and t_j. Proof of these invariants comes easily by considering *Def. 1*. This leaves only a single case, changing the traversal direction of only one of t_i and t_j, that affects such a clustering algorithm.

Thus, for clustering, we can safely ignore the $\sum_{h=i+1}^{j-1} \|e_h\|$ component from *Def. 1*, and must track intervals of $\sigma_i + \sigma_j$ for two of the four possible orientations. We choose [ff_{min}, ff_{max}] and [rf_{min}, rf_{max}], where *ff* denotes forward traversal on both edges *rf* denotes reverse traversal on the first edge and forward on the second. The value for rr_{min} is $\|e_i\| + \|e_j\| - ff_{max}$, and we can compute rr_{max}, fr_{min}, and fr_{max} similarly.

Definition 5: A **partial** $(k+1)$**-pair cluster** is a summarization of a set of observed partial sums $\sigma_i + \sigma_j$ for edges e_i, e_j and type z, denoted as $\hat{\alpha}(e_i, e_j, z) = \langle O, [min, max] \rangle$, with $O \in \{ff, rf\}$ denoting the orientation of the two edges.

While many clustering methods could be conceived, we describe a specific clustering method that can be incrementally applied to the interval representation of the clusters. We construct the $(k+1)$-pair clusters starting from all single element sets taken from Π and proceeding in two phases of merging. In the first phase, we perform the equivalent of single linkage clustering, merging two sets $\alpha_x(t_i, t_j, z)$ and $\alpha_y(t_i, t_j, z)$ if and only if their orientations

are the same and their ranges overlap or nearly overlap, specifically iff $(max_x + R > min_y) \wedge (min_x - R < max_y)$, for some parameter R. In the second stage, we order all clusters $\alpha_x(t_i, t_j, z)$ by $\langle O, min \rangle$, and then, considering all consecutive pairs $(\alpha_x(t_i, t_j, z), \alpha_y(t_i, t_j, z))$ in this ordered set, merge if clusters share the same orientation and $max_y - min_x < \lceil z \rceil$. This heuristic attempts to compensate for incomplete data.

B. Summarizing clusters in parallel

We will now describe a parallel algorithm for computing all partial $(k+1)$-pair clusters from Π, using the parallel computational primitives described in *Section III*. We have as the result of the list ranking stage of the pipeline the position of all $(k+1)$-molecules $\langle m, p(m) \rangle$, sorted according to molecule representative. We create an array $\mathcal{C}$ of partial $(k+1)$-pair clusters of the form $\langle e_i, e_j, z, O, min, max \rangle$. In this representation, a canonical ordering of each pair of edges is chosen (by choosing as e_i the edge with smaller identifier) in order to eliminate duplicate entries.

As was the case in graph construction, we process the reads in multiple rounds, in order to reduce the memory consumption of the assembler. For each read pair $\langle R_1, R_2, z \rangle$ we construct all possible $(k+1)$-pairs $\langle m_i, m_j, z \rangle$. These pairs are distributed first by m_i, and then by m_j, in order to find the position of each molecule in the graph, $p(m_i)$ and $p(m_j)$. From these positions, we create for each pair a $(k+1)$-pair cluster for each of the two orientations, correctly recording ranges [ff_{min},ff_{max}] and [rf_{min}, rf_{max}] for partial clusters consisting of this single pair. We add these two clusters to the array $\mathcal{C}$.

After we have created all singleton clusters for this input round, $\mathcal{C}$ is reduced by first sorting by $\langle e_i, e_j, t, O, min \rangle$ while forcing all tuples with the same $\langle e_i, e_j, t, O \rangle$ to the same processor. Once this sort is complete, we merge all clusters in a single local linear scan of the sorted array, using the single linkage rule described above, updating all minimums and maximums appropriately.

After all rounds of processing have completed, and we have processed all reads, we merge partial $(k+1)$-pair clusters in accordance with the phase two merging rule described above, updating all minimums and maximums. This is accomplished by first sorting all clusters for a given $\langle e_i, e_j, t, O \rangle$ using min as the primary key and max as the secondary key, and then merging with a greedy, linear-time algorithm.

VII. Graph Traversal

The final stage of the algorithm is a graph traversal aided by the partial $(k+1)$-pair clusters that produces a set of final assembled contigs. At this stage, the size of the graph is typically three orders of magnitude smaller than the original graph and the reduction of paired read distance information by aggregating it as partial $(k+1)$-clusters brings about a reduction of five to six orders of magnitude when compared to the set of $(k+1)$-pairs, Π. As a result of this reduction, this phase can be carried out sequentially, even for large mammalian and plant genomes. We briefly describe the procedure here.

We begin contig construction at "long" edges in the graph, where long is taken to be larger than the longest fragment length. Such an edge serves as a starting (or restarting) point in the traversal process because no distance constraint can span it, provided it is correct. Starting from such an edge, the contig is extended in both directions using path extension.

At any point in this process, we have a partially constructed path and there exist a number of possible candidate edges that might be used to extend the path. We limit the candidates by considering both the graph structure and the traversal constraints described at the end of *Section V*. We then consider each candidate in turn. We consider each pair constructed by taking the candidate and each path edge within a set distance from the end of the path. For each pair, we calculate an expected cluster interval for each fragment type. We convert the partial $(k+1)$ clusters into correct observed lengths based on the knowledge of the path, and check to see if a cluster exists that matches (supports) the expectation. If a cluster supports the expectation (see *Defs. 2, 3,* and *4*), it provides evidence that this edge is the correct path extension.

For each edge, a score is produced by taking the ratio of the score of all met expectations to all expectations. If, for an extension candidate, this score is close to 1 and is significantly higher than the score for other candidates, this extension is considered good and unambiguous, and it is chosen.

Note that there is some degree of inherent parallelism in this procedure as contig construction can simultaneously start from each long edge used as a seed. This can be the basis for developing a parallel algorithm. However, this step of the algorithm is neither compute nor memory intensive, prompting us to settle for a sequential execution at present.

VIII. Experimental Results

Using the method outlined earlier, we developed a parallel short read assembler named *YAGA* (for Yet Another Genome Assembler). Our software is written in C++ and MPI and is about 12,000 lines of code. We make heavy use of C++ templates. A C++ template allows the specialization of a particular class or function when combined with different external components, as long as those components adhere to an interface contract. We choose templates over other options such as base classes and inheritance for three primary reasons. One, templates provide performance improvements at the cost of executable size due to a specialized version of the template class being created for each new component with which it is used. Two, templates allow the choice of either functors (objects acting and functions) or function

Table II

RUN-TIME OF THE PARALLEL ASSEMBLER IN SECONDS, BROKEN DOWN BY STAGE OF THE ALGORITHM. FROM LEFT TO RIGHT THE COLUMNS ARE: p: THE NUMBER OF PROCESSORS, **Init**: INITIALIZATION TIME, FROM PROGRAM STARTUP TO INITIAL READ, **Read**: READ THE $(k+1)$-MOLECULES FROM THE DATA FILE, **Con**: CONSTRUCT GRAPH TUPLES AND COMPACT EDGES IN THE GRAPH, **Wr**: WRITE GRAPH INFORMATION, **Clean**: PERFORM ERROR REMOVAL BY GRAPH EDITING, **Wr**: WRITE GRAPH INFORMATION, **Pairs**: READ PAIRED INFORMATION AND CREATE CLUSTERS, **Wr**: WRITE CLUSTERS, **Tot**: TOTAL RUNNING TIME, **-Wr**: TOTAL RUNNING TIME WITHOUT WRITE PHASES, AND **Per**: PERFECT SPEEDUP.

p	**Init**	**Read**	**Con**	**Wr**	**Clean**	**Wr**	**Pairs**	**Wr**	**Tot**	**-Wr**	**Per**
16	5	469	49	106	23	30	3369	3	4,054	3,915	4,054
32	11	294	26	155	22	45	1760	10	2,323	2,113	2,027
64	11	179	14	62	89	88	910	1	1,354	1,203	1,013
128	9	120	7	59	37	32	390	1	655	563	507
256	13	101	3	104	4	70	190	11	496	311	253

pointers. Three, templates allow seamless integration with the C++ Standard Template Library (STL).

We evaluated the assembler using the *E. coli* Illumina data set from short read archive at NCBI (Accession SRX000429). The data consists of 20.8 million paired *E. coli* reads of 36 bp length, all with a fragment length of approximately 200 bases. The length of this genome is approximately 4.64 million bases. Although our work is primarily aimed at scaling to large genomes, we use this microbial organism for two reasons - the data set is actual short reads produced by the Illumina genome analyzer, as opposed to synthetically generated data. Second, as our goal is to demonstrate the scalability of our method and it is harder to use a larger number of processors efficiently for a relatively smaller data set, it allows us to better gauge the scalability. Our assembly generated 217 contigs of length at least 200, with an average contig size over 21 kbp. At least 50% of the genome could be covered using contigs no smaller than 45,592 bp, and the assembly spanned 98.47% of the genome. In keeping with the scope of the conference, our emphasis here will be on presenting parallel performance results to demonstrate the efficiency and scalability of the proposed approach.

The parallel run-time on the *E. coli* data is shown in Table II. For further details, the run-time is split into several components to provide the time for each phase of the algorithm. The tests were run on a Blue Gene/L system with 512 MB of memory per node. This demonstrates that the software is highly parallel and can function with small amounts of memory per node. As seen in the table, reasonable scaling is achieved on all factors in the above decomposition, save parallel file write, which seemed to slow significantly as we increased the number of processors. This is reflective of the lack of a parallel file system on our experimental platform, rather than a weakness of the algorithm. The stages of the algorithm not involved with file writing achieved a respectable 12.6 speedup in our testing, when the number of processors is increased by a factor of 16 from 16 to 256.

To demonstrate the applicability of our software to much larger data sets, we generated synthetic short reads from the genome of *Drosophila Melanogaster*, about 120 million bases in length. We generated 900 million paired reads to approximate 300X coverage of the genome. A 1% error rate was used in generating the short reads, with 0.7% substitution rate and 0.3% insertion/deletion rate. The data was used to test parallel performance, as well as to test assembly quality by aligning the produced contigs back to the reference genome.

The assembler completed all the parallel phases of the run in about an hour and 40 minutes on 512 nodes of the Blue Gene/L. The last phase of the assembler which does sequential path walking to produce contigs from the reduced graph and summarization of paired read distance constraints ran in about 20 minutes on a Pentium workstation with 2GB memory. The largest contig size produced was 855 kb. There were 1,687 contigs of length greater than 10 kb, covering 91.2% of the genome with greater than 99.9% match to the reference genome. A coverage of 50% of the genome is achieved using contig lengths greater than 102 kb, 75% coverage is achieved using contigs of length greater than 43 kb, and 90% coverage is achieved using contigs of length greater than 12 kb. The assembly software made 1.5 misassemblies per million bases, where a misassembly is defined as the production of a contig that does not align with the reference genome in totality (i.e., it indicates a wrong assembly consisting of reads that come from different parts of the genome).

IX. CONCLUSIONS

In this paper, we presented a parallel method for *de novo* assembly of unknown genomes from high coverage short read sequencing. This is an important open problem necessitated by advances in high-throughput sequencing and a number of research groups are actively pursuing it. The work presented here is focused on utilizing parallel computers in order to scale to the largest genomes and highest coverage data sets that would be generated in genome sequencing projects. It can be further developed in a number of directions, some of which we are currently pursuing. Of high priority is developing more sophisticated models of error correction, as this will most likely have the largest

impact on the quality of the assembler. Another direction is to parallelize the final phase, the graph traversal algorithm, such that the entire assembly runs in parallel. To augment the traversal algorithm, it might be worthwhile to develop a maximum likelihood formulation to accurately estimate edge traversal counts, and use a combination of frequency and edge traversal count information with paired read information to improve the accuracy of assembly. An important functionality that remains to be added is scaffolding – the problem of ordering and orienting the assembled contigs along the genome. The paired read links that connect the contigs are useful in generating a scaffold, but a variety of other types of data, when available, can also be useful. A scaffolded assembly can guide finishing efforts and improve the quality of the final draft genome.

ACKNOWLEDGEMENTS

This research is funded in part by the Iowa State University Plant Sciences Institute Innovation Research Grants program.

REFERENCES

[1] S. Bennet. Solexa ltd. *Pharmacogenomics*, 5(4):433–438, 2004.

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

[3] M.J. Chaisson and P.A. Pevzner. Short fragment assembly of bacterial genomes. *Genome Research*, pages 18:324–330, 2008.

[4] J.C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Research*, 17:1697–1706, 2007.

[5] D.R. Helman, J. Ja'Ja', and D.A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. Technical Report CS-TR-3670 and UMIACS-TR-96-54, College Park, MD, 1996.

[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 Research*, 18:802–809, 2008.

[7] S. Hossain, N. Azimi, and S. Skiena. Crystallizing short-read assemblies around lone Sanger reads. *Bioinformatics*, 2009.

[8] R.M. Idury and M.S. Waterman. A new algorithm for DNA sequence assembly. *Journal of Computational Biology*, 2:291–306, 1995.

[9] B.G. Jackson. *Parallel methods for short read assembly.* PhD thesis, Iowa State University, August 2009.

[10] B.G. Jackson and S. Aluru. Parallel construction of bidirected string graphs for genome assembly. In *Proc. 37th International Conf. on Parallel Processing*, pages 346–353, 2008.

[11] B.G. Jackson, P.S. Schnable, and S. Aluru. Assembly of large genomes from paired short reads. In *Proc. 1st International Conference on Bioinformatics and Computational Biology*, volume 5462, pages 30–43, 2009.

[12] B.G. Jackson, P.S. Schnable, and S. Aluru. Parallel short sequence assembly of transcriptomes. *BMC Bioinformatics*, 10:S14, 2009.

[13] M. Margulies and M. Egholm. Genome sequencing in open microfabricated high density picoliter reactors. *Nature*, 437(7054):376–380, 2005.

[14] P. Medvedev and M. Brudno. Ab initio whole genome shotgun assembly with mated short reads. In *Lecture Notes in Computer Science*, volume 4955, pages 50–64, 2008.

[15] E.W. Myers. The fragment assembly string graph. *Bioinformatics*, 21:ii79–ii85, 2005.

[16] V. Pandey, R.C. Nutter, and E. Prediger. *Applied Biosystems SOLiD System: Ligation-Based Sequencing*. Wiley, 2008.

[17] R.V. Shankar and S. Ranka. Random data accesses on a coarse-grained parallel machine. II. one-to-many and many-to-one mappings. *Journal of Parallel and Distributed Computing*, 44(1):24–34, 1997.

[18] J.F. Sibeyn, F. Guillaume, and T. Seidel. Practical parallel list ranking. *Journal of Parallel and Distributed Computing*, 56:156–180, 1999.

[19] J.T. Simpson, K. Wong, S.D. Jackman, J.E. Schein, S.J. Jones, and I. Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, Preprint, 2009.

[20] R.L. Warren, G.G. Sutton, S.J.M. Jones, and R.A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23:500–501, 2007.

[21] Business Wire. Helicos biosciences enters molecular diagnostics collaboration with renowned research center to sequence cancer-associated genes. *Genetic Engineering and Biotechnology News*, 2008.

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