Assembling Genomes on Large-Scale Parallel Computers

Anantharaman Kalyanaraman[1], Scott J. Emrich[1,2], Patrick S. Schnable[2,3], Srinivas Aluru[1,2]

[1]Department of Electrical and Computer Engineering
[2]Bioinformatics and Computational Biology Program
[3]Departments of Agronomy, and Genetics, Development and Cell Biology
Iowa State University, Ames, IA, USA
{ananthk, semrich, schnable, aluru}@iastate.edu

Abstract

Assembly of large genomes from tens of millions of short genomic fragments is computationally demanding requiring hundreds of gigabytes of memory and tens of thousands of CPU hours. New gene-enrichment sequencing strategies are expected to further exacerbate this situation. In this paper, we present a massively parallel genome assembly framework. The unique features of our approach include space-efficient and on-demand algorithms that consume only linear space, and heuristic strategies that reduce the number of expensive pairwise sequence alignments while maintaining assembly quality. As part of the ongoing efforts in maize genome sequencing, we applied our assembly framework to the largest available collection of maize genomic data. We report the partitioning of more than 1.6 *million fragments of over* 1.25 *billion nucleotides total size into genomic islands in 2 hours on 1,024 processors of an IBM BlueGene/L supercomputer.*

1. Introduction

Each cell in a living organism contains one or more long DNA sequences called *chromosomes*, collectively known as the *genome*. Contained in the genome are DNA sequences called *genes* that encode instructions for producing proteins and RNA molecules, which perform various cellular functions in an organism. Deciphering an entire genome sequence and identifying regions within it that are genes and regulatory elements is of fundamental importance in molecular and functional genomics.

Genomes span multiple length scales — from a few tens of thousands of nucleotides in viruses to millions of nucleotides in microbes to billions of nucleotides in complex eukaryotic organisms such as plants and animals. The biochemical procedure of determining the nucleotide sequence of a DNA molecule is called *sequencing*. Accurate sequencing is experimentally viable only up to hundreds of nucleotides ($\approx$ 500–1,000). To extend the reach of sequencing to genomic scales, long genomic stretches are sampled at uniform random locations by a procedure called *shotgun sequencing*. This results in numerous short DNA fragments that can be sequenced using conventional techniques. If this procedure is directly applied to an entire genome, it is called *Whole Genome Shotgun* (WGS) sequencing. After generating and sequencing such fragments, the target genome is computationally *assembled* from them. The primary information used during assembly is the pairwise overlaps that exist between fragments derived from the same region of the genome. Because such overlaps could also result from fragments derived from different but repetitive parts of the genome, fragments are typically sequenced in pairs from either end of longer DNA sequences (or *sub-clones*) of approximate known length ($\sim$ 5,000 nucleotides). Knowing the distances between paired fragments is useful in detecting repeat-induced overlaps, but only for repeats shorter than sub-clone lengths.

Concomitant with advances in sequencing strategies and the undertaking of numerous genome sequencing projects, many genome assembly programs have been developed: Arachne [3], Atlas [10], CAP3 [11], Celera Assembler [16], Euler [20], GigAssembler [14], PCAP [12], Phrap [9], Phusion [15] and TIGR Assembler [23]. Despite advances in hardware speeds and memory capacities over the same period, assembling genomes from the tens of millions of fragments typical of large sequencing projects places enormous demands on computational resources, with most of the run-time and memory spent in detecting and recording overlaps. It is common for such work to be carried out by specialized teams on workstations with tens of gigabytes of main memory using manual efforts to partition the problem, a week or more of compute time, and disks for storing intermediate results. While this should make genome assembly an ideal application for parallel processing, most assemblers are serial and the few that take advantage of parallel processing do so in a rudimentary fashion — using multiple processors to accelerate one

1-4244-0054-6/06/$20.00

stage of the assembler that deals with computing large numbers of pairwise overlaps and/or manually partitioning the problem and launching multiple jobs on different processors.

Shotgun sequencing has been carried out for increasingly larger sized genomes over the past two decades, starting from the ∼50,000 long genome of the virus bacteriophage λ [22] to the recent sequencing of mouse, human and chimpanzee genomes that are 2.5 to over 3 billion nucleotides long. Current targets for large-scale genome sequencing include economically important plant crops such as maize, sorghum, soybean and wheat. In addition to their large sizes, sequencing and assembly of the genomes of these plants is considered particularly challenging because of the abundance of repeats in them. For instance, repeats are estimated to span 65-80% of the maize genome, which has an estimated size of 2.5–3 billion nucleotides [2]. While the previously sequenced genomes contain repeats albeit at a smaller scale, repeats in maize are much harder to resolve due to very high sequence identity resulting from their short evolutionary history. On the other hand, the genes are estimated to occupy only 10-15% of the genome, mostly outside the repeat content [5]. To meet the goal of deciphering this relatively smaller "gene space" in highly repetitive genomes, biologists have designed experimental techniques such as *Methyl Filtration* in plants [21] and *High-C_0t* sequencing [26] that are expected to bias fragment sampling towards gene-rich regions [17, 25]. Similar gene-enrichment sequencing is also underway for sorghum [4] and loblolly pine [18].

Traditionally, genome assemblers are designed with the expectation that fragments are obtained through uniform sampling. For n fragments, it can be argued that their memory and run-time requirement is $O(n)$ for uniform sampling but is $\Theta(n^2)$ in the worst-case for non-uniform sampling or when a significant fraction of fragments show mutual overlaps due to repeats, though the effect is not as bad in practice. As a concrete illustration, our experiments with the CAP3 assembler on a workstation with 2 GB RAM showed that just 80,000 maize fragments saturated the memory.

In this paper, we present the first massively parallel genome assembly framework. Our approach guarantees a worst-case $O(n)$ total space complexity despite gene-enrichment and repeats, and employs heuristic strategies to significantly reduce run-time while arriving at the same solution as any conventional assembler. In November 2005, the NSF, DOE and USDA announced a $32M project for sequencing the maize genome [19]. Our primary role in the maize genome sequencing consortium is to exploit massively parallel distributed memory computers to assemble tens of millions of fragments at a rapid pace. While this project has just begun as of this writing, previously conducted NSF- and DOE-funded pilot projects have produced over 3.1 million gene-enriched and shotgun sequences that in combination total over 2.5 billion nucleotides. Here, we report preliminary results obtained by applying our framework to this data on 1,024 nodes of the IBM BlueGene/L supercomputer. The results demonstrate the effectiveness of our massively parallel framework for the assembly of the maize genome and other impending large-scale genome sequencing projects.

2. Related Work

Many assemblers follow a three phase "overlap-layout-consensus" paradigm. The first phase is the time-dominant phase, in which pairs of "significantly" overlapping fragments are detected. The overlap between a pair of fragments need not be an exact match due to errors in sequencing and natural genetic variations. The standard method for accounting these is to compute an optimal alignment between the fragments using dynamic programming (Chapter 1, [1]). This takes time proportional to the product of the lengths of the fragments being aligned. Given the low rate (∼1-2%) of errors and other variations, any good alignment is expected to contain long exact matching regions, though the converse is not necessarily true. To save run-time, most assemblers first use a faster method to detect pairs that have an exact match of a specified length, say w, and then restrict further consideration to only these pairs. Such pairs are identified using a lookup table constructed for all w-length substrings within each fragment (Chapter 5, [1]). A downside to this approach is that a long exact match of length l reveals itself as $(l - w + 1)$ matches of length w; in practice, there could be many overlaps with matches spanning hundreds of nucleotides, while w is kept as small as 10 or 11 because the size of the lookup table is exponential in w.

In the second phase, a layout consistent with the detected overlaps is constructed. It cannot be guaranteed that each nucleotide in the genome is spanned by one or more fragments. Therefore, the final assembly typically consists of a large number of contiguous stretches called *contigs* interspersed by unsampled regions. During the third phase, contigs are constructed from the layout on a consensus basis and/or by taking the available nucleotide-level sequencing quality values into account. The order and orientation of the contigs is later determined using a process called *scaffolding*. In the layout construction phase, overlaps are first sorted and processed in decreasing order of their qual-

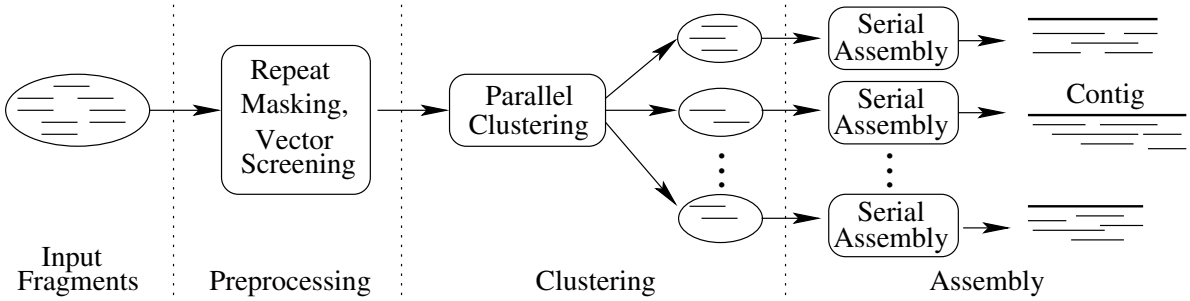

Figure 1. Illustration of our cluster-then-assemble framework. The preprocessed fragments are clustered in parallel. Each resulting cluster is assembled using a serial assembler to generate contigs.

ity. Sorting entails storing all overlaps, implying a linear space complexity only if the fragments uniformly sample the target genome and their repetitive composition is negligible. When gene-enrichment strategies are used on highly repetitively genomes, these assumptions are no longer valid — the gene-enriched fragments correspond to a non-uniform sampling over the genic regions (as demonstrated in [7]), and even the small fraction of repetitive sequences that survive the initial screening is substantial because of their high initial frequency. Under these circumstances, the number of significantly overlapping pairs of fragments is expected to grow quadratically, although the effect is not as bad in practice because a majority of the fragments may contain characterized repeats which can be detected and "masked" in a preprocessing step prior to assembly.

3. Our Clustering-based Parallel Framework for Genome Assembly

Because of selectively sampling the gene-rich portions of the genome and repeat masking, an initial assembly of gene-enriched fragments generates a large number of contigs that correspond to the many sparsely located "genomic islands" from which the fragments were originally derived [7]. Based on this observation, we propose a *cluster-then-assemble* approach that partitions the input fragments into "clusters" corresponding to genomic islands and then assembles the individual genomic islands using any serial assembler of choice. Our main contributions in space optimality, run-time efficiency and parallel methodology lie in the clustering framework. The assembly task is trivially parallelized by distributing the clusters across multiple processors and running multiple instances of a serial assembler in parallel. The space and other limitations of these assemblers will now not be a limiting factor because of the relatively small size of each cluster.

This framework, illustrated in Figure 1, is a divide-and-conquer strategy that reduces the task of assembling one large set of fragments to the task of first identifying clusters containing genomic "neighbors" and then assembling them individually. This approach has allowed us to focus on developing parallel methods while benefiting from and not duplicating the painstakingly built-in biological expertise of current assemblers. Furthermore, this allows one to generate assemblies consistent with what would have been generated by any conventional assembler, except that the problem size reach and speed of the assembler is significantly enhanced.

Our strategy is applicable even for conventional whole genome shotgun assembly. This is because gaps invariably occur in sampling, or through repeat masking, or owing to the difficulty in sequencing certain regions of the genome. As a result, an initial assembly is expected to consist of a large number of contigs that are subsequently scaffolded, followed by targeted biological experiments to fill in the gaps. As an example, in the human genome project [24], using whole genome shotgun sequencing resulted in an initial assembly with over 221,000 contigs, and the largest contig spanned only under 2 million nucleotides of the genome.

3.1. Clustering Fragments into Genomic Islands

We formulate the clustering problem as follows: Two fragments are said to *overlap* if there is a "high quality" alignment between a suffix of one and a prefix of the other, also known as *suffix prefix alignment*. Two fragments are said to belong to the same *cluster* if and only if they overlap or there exists a chain of overlaps connecting them. Because of the transitive implication, this formulation may permit two non-overlapping fragments to be clustered as illustrated in Figure 2(a). Resolving such inconsistencies is deferred until assembly. An advantage of allowing transitive clustering is the following observation: regardless of how a set of

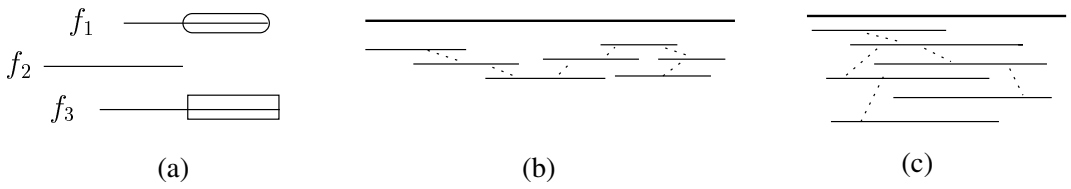

Figure 2. Illustratin of clustering: (a) Three fragments clustered because of transitivity despite not sharing consistent overlaps, i.e., (f_1,f_2) and (f_2,f_3) overlap, but (f_1,f_3) do not overlap as depicted by the oval and rectangular regions. Parts (b) and (c) show genomic regions (shown in thick lines) with uniform and non-uniform sampling, respectively. In either case, a linear number of pairwise overlaps (shown in dotted lines) is sufficient to cluster the fragments. Note that such a combination of overlaps need not be unique.

fragments sample an underlying genomic island, there exists a linear number of overlapping pairs that is sufficient to arrive at their clustering (see Figures 2(b) and 2(c)). While it is not possible to predict these in advance, the heuristic algorithm described below reduces run-time by increasing the likelihood that such pairs are identified early.

A pair of fragments is a *promising pair* if it has a maximal match[1] of length no smaller than a cutoff ψ. The clustering algorithm is as follows: Let n denote the number of genomic fragments. Initially, each fragment is considered to be in a cluster by itself. Promising pairs are generated in the non-increasing (henceforth, "decreasing" for convenience) order of their maximal match lengths. Each generated pair is aligned only if the constituent fragments currently belong to two different clusters. If the alignment test succeeds, then the two clusters are merged into one. Otherwise, the clusters are left intact, and so the alignment effort is considered wasted. The process of merging is continued until all promising pairs are considered.

In the above clustering scheme, the number of merges is no more than $n-1$, though in the worst case a quadratic number of pairs could be aligned before arriving at the final clustering. Generating pairs based on maximal matches, as opposed to fixed length matches using lookup tables, helps in two ways: (i) it limits the number of times a promising pair is generated to the number of distinct maximal matches in it, instead of the considerably larger number of fixed length matches shared by the fragments; and (ii) it provides an effective way to distinguish among promising pairs, in terms of the expected overlap quality — longer the maximal match, higher the likelihood of surviving the alignment test. Therefore, processing pairs in this order is expected to result in early cluster merges, thereby significantly reducing the chance of a pair being selected for alignment work as the execution progresses.

[1] A "maximal match" is an exact match between two fragments that cannot be extended on either side to result in a longer match.

3.2. Generating Promising Pairs

Our algorithm to generate promising pairs uses the generalized suffix tree (or GST; see Chapter 5 of [1]) built on all input fragments and their complementary strands[2]. Complementary strands are included because fragments could have been sequenced from either strand of the genomic DNA. For convenience, we use 'fragment' to refer to both types of sequences. A GST for a set of strings is a compacted trie of all suffixes of all the strings, and occupies space proportional to the input size. Our algorithm generates promising pairs in the desired order without storing them; once generated, each pair can be either discarded or aligned as determined from the current clustering.

Fragments are represented as strings over the alphabet $\Sigma = \{A, C, G, T\}$. Let $s[i]$ denote the character in position i, and $s(i)$ denote the suffix starting from position i of string s. Positions are numbered starting at 1. For a node u in the GST, let *path-label(u)* denote the concatenation of edge labels along the path from root to the node, and *string-depth(u)* denote the length of *path-label(u)*. The main idea behind our pair generation algorithm is the following: Fragments f_i and f_j share a maximal match α if and only if

C1. $\exists$ u such that *path-label(u)*$= \alpha$.

C2. $\exists$ k and l such that $f_i(k)$ and $f_j(l)$ are in subtree rooted at u.

C3. (right maximality) If u is not a leaf, $f_i(k)$ and $f_j(l)$ are in subtrees of different children of u.

C4. (left maximality) If $k \neq 1$ and $l \neq 1$, $f_i[k-1] \neq f_j[l-1]$.

[2] Complementary strand of a DNA fragment is obtained by reversing it and substituting $A \leftrightarrow T$ and $C \leftrightarrow G$. DNA is a double stranded molecule where the two strands are related as above due to opposite directionality.

Maximal matches can be identified by considering each node in the GST and examining pairs of suffixes in the node's subtree that satisfy C3 and C4. To generate maximal matches in decreasing length order, we sort the nodes in GST in decreasing order of the lengths of their path-labels using radix sort, and process them in that order. Instead of checking C3 and C4 for each pair, we generate maximal matches in amortized $O(1)$ time per pair as follows: For node u and $c \in \Sigma$, let $\ell_c(u) = \{f_i(j) \mid f_i(j)$ is in subtree of $u; j > 1; f_i[j-1] = c\}$, and $\ell_\lambda(u) = \{f_i(1) \mid f_i(1)$ is in subtree of $u\}$. These are collectively known as *lsets* at u. The *lsets* at leaves are computed directly. For an internal node u and $c \in \Sigma \cup \{\lambda\}$, $\ell_c(u) = \bigcup_{u'} \ell_c(u')$ over all children u' of u. The *lsets* are maintained as linked lists to allow constant time union operations.

Consider pair generation at internal node u corresponding to *path-label(u)* as the maximal match. At this stage, pair generation at u's children would have been completed and their *lsets* are known. The set of pairs at u are obtained by computing $\bigcup \ell_c(u') \times \ell_{c'}(u'')$, where u' and u'' are two different children of u (to satisfy C3), and $c \neq c'$ or $c = c' = \lambda$ (to satisfy C4). After pair generation at u is finished, its *lsets* are computed from the *lsets* of its children. At a leaf u, right maximality is automatically satisfied. Hence, pairs are generated as $\bigcup \ell_c(u) \times \ell_{c'}(u)$, where $c \neq c'$ or $c = c' = \lambda$.

The above scheme generates all maximal matches (of length $\geq \psi$) between each pair of fragments. This is needed if pairwise alignment computations are anchored to the maximal matches. If arbitrary suffix prefix alignments are computed, then it is wasteful to generate the same pair multiple times. In such a case, the algorithm can be modified to reduce the number of duplicate generations of the same fragment pair, while still guaranteeing $O(1)$ generation time per pair [13]. In practice, fragments contain unspecified or masked nucleotides, denoted by the character 'N'. Pair generation algorithm is modified so that N is not part of a match. We omit the details for brevity.

3.3 Parallel Generalized Suffix Tree Construction

There are no provably optimal and practically efficient parallel algorithms for suffix tree construction suited for distributed memory parallel computers. We developed the following algorithm that works well in practice. Let N denote the total length of the fragments and p denote the number of processors.

The first step is to sort all suffixes based on their w-length prefixes, where $w \leq \psi$. Partition the fragments such that each processor has $\sim \frac{N}{p}$ nucleotides. Through a linear scan, each processor partitions the suffixes of its fragments into $|\Sigma|^w$ buckets based on their first w characters. The suffixes are then globally redistributed such that those belonging to the same bucket are in the same processor, and the number of suffixes per processor is $\sim \frac{N}{p}$. While adversarial input such that only one bucket contains all N suffixes can be easily constructed, this poses no difficulty in practice. Empirically, a value of $w = 11$ was found appropriate for maize data and for the range of processors tested (up to 1,024 processors). This generates over 4 million buckets, sufficient to distribute them in a load balanced manner even for thousands of processors.

The next phase consists of constructing for each bucket, a compacted trie of all its suffixes. Each of these represents a subtree in GST rooted at a node with string depth $\geq w$. We construct each trie in a depth-first manner as follows: Partition all suffixes in the bucket into at most $|\Sigma|$ sub-buckets based on their respective $(w+1)^{th}$ characters. This is recursively applied for each sub-bucket by examining characters in subsequent positions until all suffixes are separated or their lengths exhausted. In the worst case, this procedure visits all suffixes to their full lengths, resulting in a run-time of $O\left(\frac{N \times l}{p}\right)$, where l is the average length of an input fragment. We now have a distributed representation of the GST as a collection of subtrees containing all nodes at depth $\geq w$. The top portion of the GST is not needed for pair generation.

The main challenge in this scheme is acquiring the fragments required to construct the local subtrees. Storing all fragments with suffixes in local buckets requires $O\left(min\{\frac{N \times l}{p}, N\}\right)$ space in the worst case, which is not a scalable solution. Space can be reduced by constructing one subtree at a time, and loading all fragments required for a subtree from disk prior to its construction. Given that disk latencies are in the millisecond range for random accesses as required here, we developed an alternative to take advantage of the high bandwidth interconnection network of BlueGene/L.

Each processor partitions its buckets into variable-sized batches, such that the fragments required to construct all buckets in each batch would occupy $\Theta\left(\frac{N}{p}\right)$ space. Before constructing a batch, all fragments needed for its construction are fetched through two collective communication steps — the first to request the processors that have the required fragments, and the second to service the request. The processor that has a given fragment is determined in constant time by recalling the initial distribution of the fragments. A processor may exhaust all its batches, in which case it continues to participate in the remaining communication rounds to serve requests from other processors.

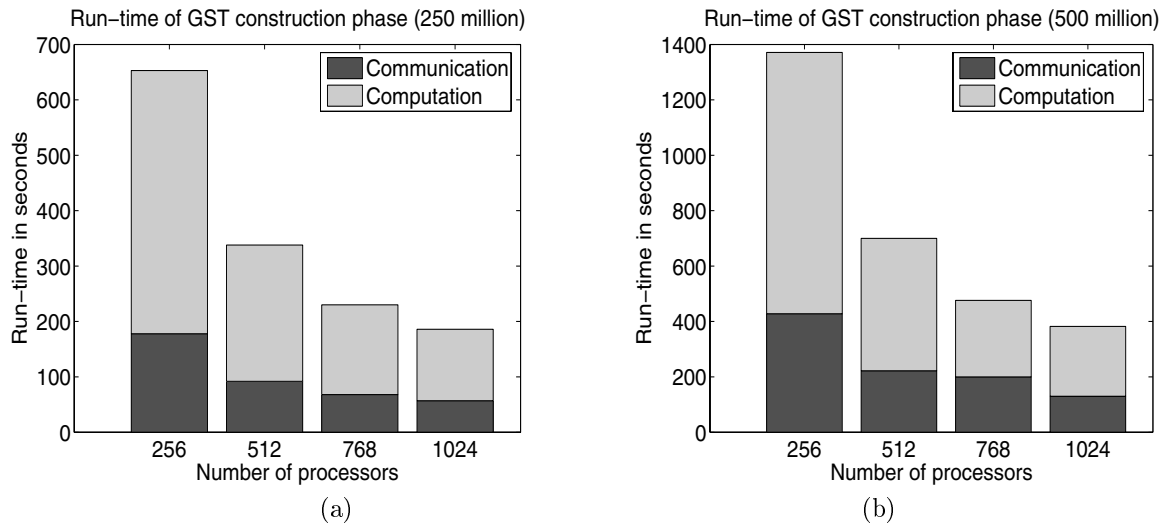

Figure 3. Parallel run-times for constructing GST on inputs of sizes: (a) 250 million, and (b) 500 million nucleotides.

In the above communication based solution, each processor receives $O(\frac{N}{p})$ characters from all other processors per communication step. However, the size of the buffer used to send fragments to other processors may exceed $O(\frac{N}{p})$. This is because requests from different processors may intersect, in the worst case over all of $O(\frac{N}{p})$ local data; the likelihood of this scenario increases with the number of processors. We resolved this issue by implementing a *customized Alltoallv*, which ensures $O(\frac{N}{p})$ size for the buffers by doing $p-1$ sends and receives instead of one collective communication.

Experimental Results

We studied the performance of our GST construction algorithm by varying the number of processors from 256 to 1,024. Each dual-processor node of the BlueGene/L system was used in co-processor mode, i.e., one processor was used for computation and the other processor was used for communication. Experiments were conducted on two subsets of the maize data, with sizes 250 and 500 million nucleotides that comprised 322,009 and 649,957 fragments, respectively. Figure 3 shows the parallel run-times and their breakdown into communication and computation times, all of which show linear scaling with both processor and input sizes.

3.4. Detecting Overlaps And Managing Clusters In Parallel

Once the GST for all input fragments is constructed in parallel, each processor can generate promising pairs from its portion of the GST using the algorithm described in Section 3.2. As pairs are generated, they need to be checked against the current clustering, allocated for alignment if necessary, and the alignment results interpreted to update the current clustering. To implement these tasks in parallel, we designed an iterative solution with one master and $p-1$ worker processors, and with responsibilities as shown in Figure 4.

Initially, the master processor creates a cluster for each input fragment. As the execution progresses, it updates the clusters using pairs that succeed alignment tests. Managing clusters entails two operations: finding the cluster that contains a given fragment, and merging two clusters whenever an alignment succeeds. We use the union-find data structure that allows each operation to complete with an amortized cost given by the inverse of Ackermann's function, which is a constant for all practical purposes.

The worker processors are responsible for computing alignments, while the master processor is responsible for allocating the alignments. Each promising pair generated by a worker processor is sent to the master processor, which allocates it for alignment only if the constituent fragments are in different clusters. To make better use of the network bandwidth, generated pairs and alignment allocations are communicated in batches of sizes that are determined dynamically by the master processor.

In addition to the load balancing concerns typical in a single master multiple worker setup such as keeping all the worker processors busy and the master processor available most of the time, our master-worker model presents other unique challenges. The worker processors in our model, in addition to processing the work (by aligning pairs), also generate work (by gen-

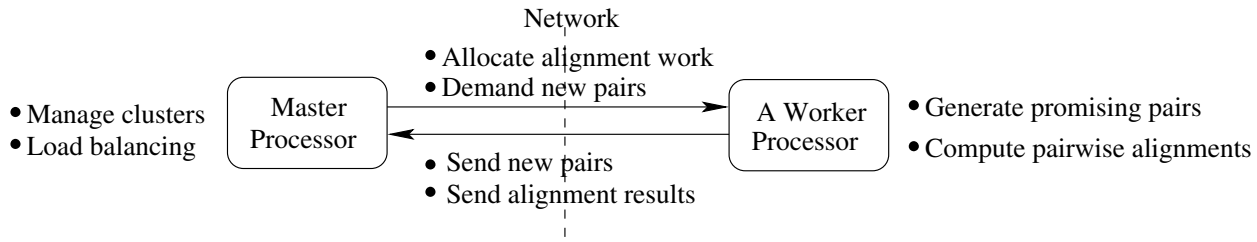

Figure 4. A "single master multiple workers" design for detecting overlaps and clustering in parallel, with responsibilities designated as shown. Arrows indicate the direction of communication.

erating pairs). Thus, care must be taken that the rate of work generation is neither too fast to result in a memory overflow (because a batch of pairs needs to be stored until the master processor decides if they should be aligned) nor too slow to result in unnecessary processor wait times. Moreover, as not all generated pairs are necessarily selected for alignment, it is important to regulate the rate of pair generation in order to maintain a steady rate in alignment computation. Another concern may arise when processors start to run out of pairs to generate from their portion of the GST as execution progresses. Not only is it necessary to keep such processors busy computing alignments, but it is beneficial for the master processor to allocate pending alignment computations to these processors before allocating any to a processor that still has pairs to generate.

With the above goals in mind, we devised the following iterative solution, in which the master and worker processors interact iteratively until all promising pairs are generated and all alignments identified as necessary have been computed. In each iteration, the worker processor generates as many new promising pairs as requested by the master processor and sends them in a message along with the results of the latest alignments it computed. While waiting for the master to reply, it computes a fresh batch of alignments allocated by the master processor for the current iteration, effectively masking the wait time with computation. If the alignments are computed before the master processor replies, then the worker processor resumes from its earlier state of pair generation and generates new pairs until either a reply arrives or its fixed size queue in which the pairs are stored is full.

The master processor polls for messages from other processors. When a message arrives from processor p_i, it first updates the union-find data structure using pairs that succeeded the alignment test, scans the batch of newly generated pairs from p_i, and appends to a fixed size "work queue" only those pairs for which computing alignments is necessary. It then repeatedly extracts a fixed number of pairs, say b, from the work queue, dispatching each batch to an "idle" processor that has run out of pairs to generate. Another queue of size $p-1$ is used to keep track of all such idle processors at any stage of the execution. If on exhausting this queue, there are still pairs in the work queue, then the next b pairs are sent to processor p_i. With it, the master processor also piggybacks the number of new pairs, say r, that it expects to receive from p_i in its next correspondence; r is analytically calculated based on the ratio of the number of pairs received from p_i in the current iteration, say k, and the number of these pairs inserted into the work queue, say $k' \leq k$. It is given by $r = \frac{p-1}{p_a} \times min\{b \times \frac{k}{k'}, \frac{W_c - W_s}{p-1}\}$, where p_a denotes the number of processors that still have pairs to generate, and W_c and W_s are the capacity and current size of the work queue, respectively, measured in number of pairs.

Experimental Results

We studied the performance of our master worker implementation on the BlueGene/L (see Figure 5). The results show a better scaling for the larger (500 million) input than the smaller (250 million) input. Upon increasing the number of processors from 256 to 1,024, we observe relative speedups of 2.6 for the 250 million input and 3.1 for the 500 million input. Further investigation revealed that the percentage average idle time for the processors increased from 16% on 256 processors to 26% on 1,024 processors on the 250 million input, and from 9% to 16% for the 500 million input — indicating that more processors can be deployed while maintaining efficiency if the problem size is larger. Note that a full sequencing project will generate over 22 billion nucleotides (30 million fragments each about 750 nucleotides long), on which tens of thousands of processors can be utilized with our scheme.

Figure 5b shows the number of promising pairs generated as a function of the input size. This figure also shows the effectiveness of our clustering heuristic in significantly reducing the number of alignments com-

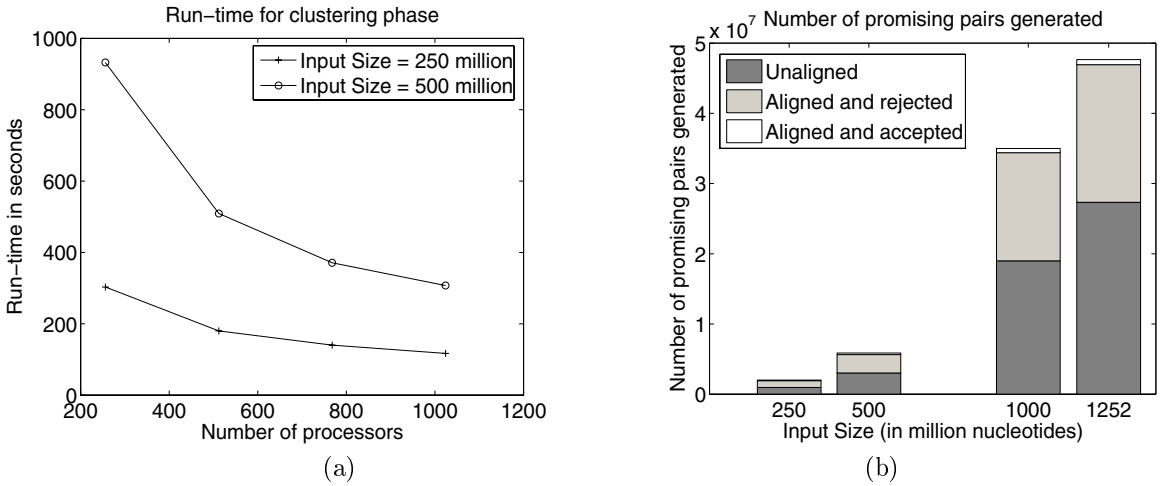

Figure 5. (a) Total parallel run-time for the entire clustering algorithm excluding that of GST construction. (b) The number of pairs generated, aligned, and accepted as a function of input size.

puted. For the entire maize data, which has 1,607,364 fragments of total size 1.252 billion nucleotides, only about 40% of the pairs generated are aligned. However, less than 1% of the pairs aligned contributed to merging of clusters, indicating the presence of numerous medium-sized ($\sim$ 100) repetitive elements that survived initial screening procedures. Growth in the number of promising pairs is a direct reflection of the expected worst-case quadratic growth in the maize data. The number of promising pairs generated and the relative savings in the alignment work are highly data sensitive. For example, we observed that only 22% of generated pairs were aligned on a different data set.

In our current single-master design, the master processor is designed to handle one request at a time. Messages arriving concurrently from multiple processors are therefore buffered at the MPI level on the master node. Message sizes can range from tens to hundreds of kilobytes depending on the requests made by the master processor, implying that the MPI buffer at the master node can potentially overflow for larger number of processors. To avoid message losses, our implementation uses *MPI_Ssend* that sends a message to the master processor only after a corresponding receive has been posted. Using *MPI_Ssend*, however, indicated a performance degradation of about 30% as opposed to using *MPI_Isend* or *MPI_Send* both on the BlueGene/L and IBM xSeries Myrinet cluster. An alternative is to change the underlying design to allow scaling the number of master processors with processor size. This should also drastically improve the amount of time a master processor is available to its worker processors. With the current single-master implementation, we observed a gradual decrease in its availability (idle time) from 90% to 70% when the processor size was increased from 256 to 1,024. Operating on multiple master processors would, however, necessitate broadcasting cluster merges among master processors, and the number of such merges may no longer be linear.

4. Maize Genome Assembly

The maize genomic data are composed of 3,124,130 fragments with total length over 2.5 billion nucleotides. This includes 852,838 Methyl-Filtrated (MF) [21] and High-C_0t (HC) [26] fragments. The MF strategy is based on the elimination of bacterial colonies containing methylated sub-clones, which are typically non-genic regions in plants. The HC strategy utilizes hybridization kinetics to enrich for lower copy sequences, which in case of maize are mostly genic regions. Also available are fragments from WGS sequencing and another strategy called *Bacterial Artificial Chromosome* (BAC) sequencing, in which long genomic sequences ($\sim$150,000–200,000) are cloned in bacterial vectors, and their ends and internal regions are individually sampled through sequencing. A summary of the entire maize data is provided in the first three columns of Table 1.

As with any other assembler, the first step in our framework is to "preprocess" the input fragments: raw fragments obtained from sequencing strategies can be contaminated with foreign DNA elements known as *vectors*, which are removed using the program *Lucy* [6]. In addition, we designed a database of known and statistically-defined repeats [7] and screened all fragments against it. The matching portions are masked with special symbols such that our clustering method can treat them appropriately during overlap detection.

	Before Preprocessing		After Preprocessing	
Fragment Type	Number of Fragments	Total length (in millions)	Number of Fragments	Total length (in millions)
MF	411,654	335	349,950	288
HC	441,184	357	427,276	348
BAC	1,132,295	964	425,011	307
WGS	1,138,997	870	405,127	309
Total	3,124,130	2,526	1,607,364	1,252

Table 1. Maize genomic fragment data types and size statistics: Methyl-filtrated (MF), High-C_0t (HC), Bacterial Artificial Chromosome (BAC) derived, and Whole Genome Shotgun (WGS).

The last two columns in Table 1 show the results of preprocessing the data using our repeat masking and vector screening procedures. As expected, preprocessing invalidates a significant number of shotgun fragments ($\approx$ 60-65%) because of repeats, while most of the fragments resulting from gene-enrichment strategies are preserved. An efficient masking procedure is important because unmasked repeats cause spurious overlaps that cannot be resolved in the absence of paired fragments spanning multiple length scales. Furthermore, it provides a computational means to preferentially assemble non-repetitive regions of the genome that may be gene-enriched.

The results of applying our parallel genome assembly framework on the entire maize data is as follows: Preprocessing the 3,124,130 fragments downloaded from GenBank took 1 hour by trivially parallelizing on 40 processors of an IBM xSeries cluster with 1.1 GHz Pentium III processors and 1GB RAM per processor. Our clustering method partitioned the resulting 1,607,364 fragments (over 1.25 billion nucleotides) in 102 minutes on 1,024 nodes of the BlueGene/L, with the GST construction taking only the first 13 minutes. We used CAP3 [11] for assembling the fragments in each resulting cluster. This assembly step finished in 8.5 hours on 40 processors of the IBM xSeries cluster through trivial parallelization.

Our assembly resulted in a total of 163,390 maize genomic islands (or contigs) formed by two or more input fragments, and 536,377 singletons. Singletons are fragments that do not assemble with any other fragment because of sharing no overlap and/or having a high repetitive content that was masked during preprocessing. On an average, each cluster assembled into 1.1 contigs; given that the CAP3 assembly is performed with a higher stringency, this result indicates the high specificity of our clustering method and its usefulness in breaking the large assembly problem into disjoint pieces of easily manageable sizes for conventional assemblers. The overall size of our contigs is about 268 million nucleotides, which is roughly 10% of the entire maize genome. Upon validation using independent gene finding techniques, we confirmed that our contigs span a significant portion ($\sim$ 96%) of the estimated gene space [8]. The average number of input fragments per contig is 6.55, while the maximum is 2,435. To more accurately assess non-uniformity within these data, coverage throughout the entire maize assembly was analyzed. The mean coverage of 3.24 was larger than the expected 1.0 coverage provided in the input. Moreover, 1.34 million nucleotides of this assembly have sequence coverage of 25 or higher and may correspond to unmasked repeats and/or biases from the gene-enrichment approach. The results of our assembly can be graphically viewed at *http://www.plantgenomics.iastate.edu/maize*. For further biological details on our on-going effort to assemble the maize genome and a thorough discussion of the results on an earlier version of maize data with less than a million fragments, see [8].

5. Conclusions

We presented the design and development of an efficient clustering-based framework for genome assembly on massively parallel distributed memory machines using gene-enriched fragments, and reported its application on the largest publicly available maize genomic data using the BlueGene/L supercomputer. The results of our assembly are publicly available, and are being frequently used by many plant scientists. Experiments indicate that the run-time behavior of our clustering solution shows good scaling. Our key contributions in space-optimality and a heuristic-based clustering scheme to significantly reduce alignment computations will play a crucial role in the large-scale applicability of our framework in the context of the maize genome and many other complex genomes of economically important plant crops. To give a perspective — our current implementation requires 80 bytes for every

input nucleotide, implying that we can scale up to ≈ 8 million fragments for every 1,024 BlueGene/L nodes (each with 512 MB). This would enable us to cluster 30 million fragments on $\sim$ 4K nodes. Moreover, we conducted a few preliminary experiments on 8K nodes and the scaling results are encouraging. We believe that a continued improvement of our algorithmic techniques on large-scale parallel computers such as the BlueGene/L will provide a robust and efficient platform for many impending large-scale genome projects such as for sorghum and pine, which also involve gene-enrichment sequencing.

Acknowledgments

We thank Sam Ellis, Kurt Pinnow and Brian Smith of IBM Rochester, for facilitating our access to the BlueGene/L supercomputer and commenting on the manuscript. This research was supported in part by NSF grant DBI-0527192 and an IBM Ph.D. Fellowship.

References

[1] S. Aluru (Editor). *Handbook of Computational Molecular Biology*. Chapman & Hall/CRC Press Computer and Information Science Series, 2005.

[2] K. Arumuganathan and E.D. Earle. Nuclear DNA content of some important plant species. *Plant Molecular Biology Reporter*, 9(3):211–215, 1991.

[3] S. Batzoglou, D.B. Jaffe, K. Stanley *et al.* Arachne: A whole-genome shotgun assembler. *Genome Research*, 12(1):177–189, 2002.

[4] J.A. Bedell, M.A. Budiman, A. Nunberg *et al.* Sorghum genome sequencing by methylation filtration. *Public Library of Science*, 3(1):e13, 2005.

[5] J. L. Bennetzen, V. L. Chandler, and P. S. Schnable. National Science Foundation-sponsored workshop report. Maize genome sequencing project. *Plant Physiology*, 127:1572–1578, 2001.

[6] H. Chou and M.H. Holmes. DNA sequence quality trimming and vector removal. *Bioinformatics*, 17(12):1093–1104, 2001.

[7] S.J. Emrich, S. Aluru, Y. Fu *et al.* A strategy for assembling the maize (*Zea mays* L.) genome. *Bioinformatics*, 20(2):140–147, 2004.

[8] Y. Fu, S.J. Emrich, L. Guo *et al.* Quality assessment of Maize Assembled Genomic Islands (MAGIs) and large-scale experimental verification of predicted novel genes. *Proceedings of the National Academy of Sciences USA*, 102:12282–12287, 2005.

[9] P. Green. Phrap - the assembler. *http://www.phrap.org*, 1994.

[10] P. Havlak, R. Chen, K.J. Durbin *et al.* The Atlas genome assembly system. *Genome Research*, 14:721–732, 2004.

[11] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome Research*, 9(9):868–877, 1999.

[12] X. Huang, J. Wang, S. Aluru *et al.* PCAP: A whole-genome assembly program. *Genome Research*, 13(9):2164–2170, 2003.

[13] A. Kalyanaraman, S. Aluru, V. Brendel, and S. Kothari. Space and time efficient parallel algorithms and software for EST clustering. *IEEE Transactions on Parallel and Distributed Systems*, 14(12):1209–1221, 2003.

[14] W.J. Kent and D. Haussler. GigAssembler: an algorithm for initial assembly of the human working draft. *Genome Research*, 11(9):1541–1548, 2001.

[15] J.C. Mullikin and Z. Ning. The Phusion assembler. *Genome Research*, 13(1):81–90, 2003.

[16] E.W. Myers, G.G. Sutton, A.L. Delcher *et al.* A whole-genome assembly of *Drosophila*. *Science*, 287:2196–2204, 2000.

[17] L.E. Palmer, P.D. Rabinowicz, A.L. O'Shaughnessy *et al.* Maize genome sequencing by methylation filtration. *Science*, 302:2115–2117, 2003.

[18] D. Peterson. Accelerating pine genomics through development and utilization of molecular and cytogenetic resources. *http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0421717*, 2004.

[19] NSF. NSF, USDA and DOE Award $32 Million to Sequence Corn Genome. *http://www.nsf.gov/news/news_summ.jsp?cntn_id=104608&org=BIO&from=news*, 2005.

[20] P.A. Pevzner, H. Tang, and M.S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences USA*, 98(17):9748–9753, 2001.

[21] P.D. Rabinowicz, K. Schutz, N. Dedhia *et al.* Differential methylation of genes and retrotransposons facilitates shotgun sequencing of the maize genome. *Nature Genetics*, 23(3):305–308, 1999.

[22] F. Sanger, A.R. Coulson, G.F. Hong *et al.* Nucleotide sequence of bacteriophage lambda DNA. *Journal of Molecular Biology*, 162(4):729–773, 1982.

[23] G. Sutton, O. White, M. Adams, and A. Kerlavage. TIGR Assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1(1):9–19, 1995.

[24] J.C. Venter, M.D. Adams, E.W. Myers *et al.* The sequence of the human genome. *Science*, 291(5507):1304–1351, February 2001.

[25] C.A. Whitelaw, W.B. Barbazuk, G. Pertea *et al.* Enrichment of gene-coding sequences in maize by genome filtration. *Science*, 302(5653):2118–2120, 2003.

[26] Y. Yuan, P.J. SanMiguel, and J.L. Bennetzen. High-C_0t sequence analysis of the maize genome. *The Plant Journal*, 34(2):249–255, 2003.