Available online at www.sciencedirect.com

ScienceDirect

J. Parallel Distrib. Comput. 67 (2007) 1240–1255

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Assembling genomes on large-scale parallel computers

A. Kalyanaraman[a], S.J. Emrich[b,c], P.S. Schnable[c,d], S. Aluru[b,c,*]

[a]*School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164, USA*
[b]*Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA*
[c]*Bioinformatics and Computational Biology Graduate Program, Iowa State University, Ames, IA 50011, USA*
[d]*Departments of Agronomy, and Genetics, Development and Cell Biology, Iowa State University, Ames, IA 50011, USA*

Received 7 July 2006; received in revised form 1 May 2007; accepted 7 May 2007
Available online 9 June 2007

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. The advent of high throughput sequencing technologies, new gene-enrichment sequencing strategies, and collective sequencing of environmental samples further exacerbate this situation. In this paper, we present the first massively parallel genome assembly framework. The unique features of our approach include space-efficient and on-demand algorithms that consume only linear space, and strategies to reduce the number of expensive pairwise sequence alignments while maintaining assembly quality. Developed as part of the ongoing efforts in maize genome sequencing, we applied our assembly framework to genomic data containing a mixture of gene enriched and random shotgun sequences. We report the partitioning of more than 1.6 million fragments of over 1.25 billion nucleotides total size into genomic islands in under 2 h on 1024 processors of an IBM BlueGene/L supercomputer. We also demonstrate the effectiveness of the proposed approach for traditional whole genome shotgun sequencing and assembly of environmental sequences.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Computational biology; Genome assembly; Genome sequencing; Parallel algorithms; Suffix trees

1. Introduction

Each cell in a living organism contains one or more long DNA sequences called *chromosomes*, collectively known as the *genome*. Contained within the genome are DNA sequences called *genes* that code for 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. Genome sequencing also forms the basis for the rapidly expanding field of comparative genomics, which attempts to study genome evolution and unravel genome structure through cross-genome comparisons.

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. Because DNA is double stranded, its length is measured in units called *base pairs*, denoted *bp*. 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 base pairs (≈ 500–1000 bp). 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. Pairwise overlaps are detected by computing alignments between the corresponding pairs of fragments using standard dynamic programming approaches. Because such overlaps could also result from fragments derived from different but repetitive parts of the genome, fragments are typically sequenced in pairs

[*] Corresponding author. Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA.
E-mail address: aluru@iastate.edu (S. Aluru).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.05.014

from either end of longer DNA sequences (or *sub-clones*) of approximate known length (~ 5000 bp). Knowledge of the distances between paired fragments, known as *clone mate* information, 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 [4], Atlas [13], CAP3 [14], Celera Assembler [19], Euler [26], GigAssembler [17], PCAP [15], Phrap [11], Phusion [18] and TIGR Assembler [31]. 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 overlaps using alignment algorithms and recording them. 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 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. For example, the human genome assembly by Venter et al. [32] took 20,000 CPU hours for ~ 27 million fragments using ten 4-processor SMP clusters each with 4 GB RAM, along with a 16-processor NUMA machine with 64 GB shared memory.

Shotgun sequencing has been carried out for increasingly larger sized genomes over the past two decades, starting from the $\sim 50{,}000$ long genome of the virus bacteriophage λ [29] 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 are 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 [3]. 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 [6]. 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 [27] and *High-*C_0t sequencing [35] that are expected to bias fragment sampling towards gene-rich regions [23,34]. Similar gene-enrichment sequencing is also underway for sorghum [5] and loblolly pine [25].

Traditionally, genome assemblers have been designed with the expectation that fragments are obtained through uniform random sampling. For n fragments, it can be argued that their memory and run-time requirement is $\Theta(n)$ 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. Like other assemblers, our method generates a selected set of pairs of fragments to which alignment computations are restricted. However, we generate such pairs in $O(1)$ time per pair and do so in a prioritized order, which is used to drive a heuristic strategy that significantly reduces the number of pairwise alignments computed without affecting quality. We demonstrate the effectiveness, scalability, and biological validity of our approach using data generated from diverse sequencing approaches: (a) random shotgun sequencing data from the fruit fly *Drosophila psuedoobscura* [28], (b) a mixture of shotgun, gene enriched, and bacterial artificial chromosome (BAC) derived fragments from maize, and (c) shotgun fragments from thousands of bacterial genomes from the Sargasso Sea environmental sample [33]. We present detailed experimental analysis for all three cases using a 1024 node BlueGene/L supercomputer.

As part of the on-going NSF/DOE/USDA [1] -funded maize genome sequencing consortium [21], we are applying the methods presented in this paper to assemble the maize genome. As a preliminary test to size up the complexity of this sequencing project, NSF- and DOE-funded pilot projects produced over 3.1 million gene-enriched and shotgun fragments containing over 2.5 billion bp [27,35]. Here, we report extensively on the application of our assembly framework to this data. The results demonstrate the effectiveness of our approach for large-scale sequencing projects such as the maize genome sequencing project.

The rest of the paper is organized as follows: in Section 2, we present an overview of conventional genome assembly strategies. Section 3 contains a high-level overview of our parallel assembly framework. The algorithmic and implementation details of our framework and scaling results using maize data are presented in Sections 4–7. Application of our framework to maize gene enriched sequence data, which resulted in the first publicly available assembly of this data, is presented in Section 8. Section 9 contains additional validation using random shotgun sequencing and environmental sample sequencing. Section 10 contains our concluding remarks.

2. Genome assembly

Many assemblers follow a three phase "overlap–layout–consensus" paradigm. The first phase is the time-dominant

[1] National Science Foundation, Department of Energy, United States Department of Agriculture.

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, if multiple individuals are selected in sequencing. The standard method for accounting these is to compute an optimal alignment between the fragments, allowing for insertions, deletions and substitutions. This is achieved using a dynamic programming algorithm [10,22,30] that takes time proportional to the product of the lengths of the fragments being aligned. When the number of fragments is in the millions, it is computationally impractical to apply this algorithm to all pairs of fragments.

Given that the fragments are small ($\approx$ 700–800 bp), and sampled from the much larger genome (10^4–10^7 times larger), only a small subset of the pairs will actually overlap. To exploit this, assembly algorithms use filters to generate a reduced subset of pairs that satisfy a specified criterion, and limit alignment computation to those pairs. A key property of a filter is that every pair of fragments that pass the alignment test should also be generated by the filter. Another important property is to directly generate pairs of fragments instead of testing every pair to see if it satisfies the specified criterion. All of the filters designed have been based on the identification of exact matches.

Given the low rate ($\sim$ 1–2%) of errors, sequencing artifacts, and other variations, any good alignment is expected to contain long exactly matching regions, though the converse is not necessarily true. The most frequently used filter is to generate pairs that have one or more exact matches of a specified length, say w. Such pairs are easily identified using a lookup table constructed for all w-length substrings within each fragment [24]. 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. The extent to which a genome is sampled in a sequencing project is indicated by *coverage*, which is the ratio of the total length of all the fragments to the estimated size of the genome. The coverage denotes the average number of fragments that contain a nucleotide. Typically a five to seven fold coverage (denoted 5X to 7X) is used for very large genomes. However, it cannot be guaranteed that each nucleotide in the genome is spanned by one or more fragments due to the randomness in sampling and cloning difficulties inherent in certain genomic regions. Therefore, the final assembly typically consists of a large number of contiguous stretches called *contigs* interspersed by unsampled regions. As an example, in the human genome project [32], using whole genome shotgun sequencing resulted in an initial assembly with over 221,000 contigs, and the largest contig spanned only under 2 million bp of the genome. 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 along the chromosomes is later determined using a process called *scaffolding*. Programs developed for this purpose are called *scaffolders*. Targeted biological experiments are then used to "fill" the gaps in sequencing, and this process is known as *finishing*.

In the layout construction phase of the assembler, overlaps are typically sorted and processed in decreasing order of their quality. Sorting entails storing all overlaps, implying a linear space complexity only if the fragments uniformly sample the target genome and spurious alignments due to repeats can be successfully avoided. When gene-enrichment strategies are used on highly repetitive genomes, these assumptions are no longer valid—the gene-enriched fragments correspond to a non-uniform sampling over the genic regions (as demonstrated in [8]), 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 that can be detected and "masked" prior to assembly.

3. Our clustering-based parallel framework for genome assembly

As mentioned before, sequencing gaps and regions difficult to sequence result in hundreds of thousands of contigs. Taking advantage of this, we propose a *cluster-then-assemble* approach that partitions the input fragments into "clusters" such that each cluster contains fragments constituting one or more contigs. The goal of clustering is to form as many clusters as possible while making sure that fragments that belong to the same contig are never split across clusters. Ideally, there should be as many clusters as contigs. Thus, clustering can be viewed as decomposing the assembly problem into a large number of much smaller, independent assembly problems. This framework is illustrated in Fig. 1.

Our cluster-then-assemble approach has several advantages: if we develop parallel methods for clustering, the subsequent assembly tasks are trivially parallelized by distributing the clusters across multiple processors and running multiple instances of a serial assembler in parallel. 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. The space and other limitations of these assemblers will now not be a limiting factor because of the relatively small size of each cluster. 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 are significantly enhanced.

Our main contributions in space optimality, run-time efficiency and parallel methodology lie in the clustering framework, and are described in detail in the following sections. The following attributes are essential for success in the clustering strategy:

Correctness: Fragments that should be part of the same contig should not be split across clusters because the subsequent assembly step is carried out independently on each cluster, and

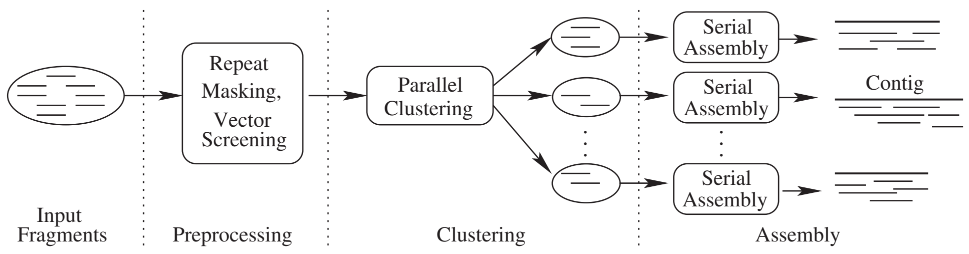

Fig. 1. Illustration of our cluster-then-assemble framework. The preprocessed fragments are clustered in parallel. Each resulting cluster is assembled using a serial assembler to generate contigs.

there is no way of combining fragments from multiple clusters. For this reason, a less stringent overlap criterion is used during clustering than during final assembly. This way, fragments belonging to the same contig are never separated into different clusters. However, this could also cause fragments from multiple contigs to be combined into a cluster. Even so, this is not a problem because the serial assembly on each individual cluster will detect such discrepancies, thereby guaranteeing the correctness of the overall assembly results.

Speed: To effectively utilize a large number of processors during the assembly phase, the fragments must be split into as many clusters as possible. Some measures of interest are the average number of contigs per cluster and the size of a largest cluster. While the former is purely a measure of clustering effectiveness the latter highly depends on the input fragments. As the maximum cluster size determines parallelism in the assembly phase, it is important to experimentally demonstrate that it will not be a limiting factor.

While the large number of initial contigs in shotgun sequencing projects provides sufficient rationale for the proposed strategy, other sequencing techniques should result in even larger number of clusters. Gene-rich sequencing should generate a large number of contigs that correspond to the many sparsely located "genomic islands" from which the fragments were originally derived. Similarly, collective sequencing of thousands of bacterial genomes from environmental samples should further magnify the number of clusters.

4. Clustering fragments

Keeping in mind the goals of clustering, 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 sequence of overlaps connecting them. Because of the transitive implication, this formulation may permit fragments with inconsistent overlaps to be clustered as illustrated in Fig. 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 fragments sample an underlying genomic island, there exists a linear number of overlapping pairs that is sufficient to arrive at their clustering (see Figs. 2(b) and (c)). While it is not possible to predict these in advance, the algorithm described below reduces run-time by increasing the likelihood that such pairs are identified early, without affecting the correctness of the final clustering.

We use the term *promising pair* to denote a pair of fragments that has a maximal match[2] 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.

Pair generation heuristic: Promising pairs are generated in a non-increasing (henceforth, "decreasing" for convenience) order of their maximal match lengths.

Alignment strategy: 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 wasted. This process is continued until all promising pairs are considered. Fig. 3 outlines our clustering strategy.

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 ($\Theta(n^2)$) 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 (as proved in Section 5), 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.

Our clustering scheme significantly reduces the number of pairs aligned in practice. Note that the final clustering remains unchanged regardless of the order in which the pairs are processed because of the transitive closure property. Therefore, our heuristic only reduces run-time but does not affect the correctness of the algorithm.

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

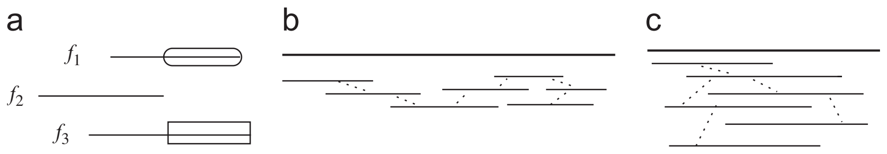

Fig. 2. Illustration of clustering: (a) three fragments clustered because of transitivity despite not sharing consistent overlaps, i.e., (f_1,f_2) and (f_2,f_3) overlap, but (f_1,f_3) do not overlap as depicted by the oval and rectangular regions. Parts (b) and (c) show genomic regions (shown in thick 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.

Algorithm 1 *Fragment Clustering*

```
Input: Set $S = \{f_1, f_2, \dots f_n\}$ of $n$ sequences
Output: A partition $C = \{C_1, C_2, \dots C_m\}$ of $S$, $1 \le m \le n$
  1. Initialize Clusters:
       $C \leftarrow \{ \{f_i\} \mid 1 \le i \le n\}$
  2. FOR each pair $(f_i, f_j)$ with a maximal match of length $\ge \psi$
     generated in non-increasing order of maximal match length
       $C_p \leftarrow Find(f_i)$
       $C_q \leftarrow Find(f_j)$
       IF $C_p \ne C_q$ THEN
           overlap quality $\leftarrow Align(f_i, f_j)$
           IF overlap quality is significant THEN
               $Union(C_p, C_q)$
  3. Output $C$
```

Fig. 3. Our clustering strategy. Operations on the set of clusters are performed using the Union–Find data structure.

5. Generating promising pairs

In this section, we provide the serial version of our pair generation algorithm, and in Sections 6 and 7 we elaborate on the parallel version. We use the generalized suffix tree (or GST; see Chapter 5 of [2]) data structure for generating pairwise maximal matches of length $\geqslant \psi$ between fragments. The 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 to generate promising pairs uses the GST built on all input fragments and their reverse complementary counterparts.[3] 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 naive approach to generate fragment pairs in decreasing order of maximal match length is to first identify all such fragment pairs and then sort them by their maximal match lengths. This scheme would, however, require $O(n^2)$ space, because sorting entails storing the pairs. Instead, we developed an algorithm with linear space complexity that *directly* generates the promising pairs in the sorted order without storing them. This is achieved through an "on-demand" scheme—i.e., pairs are generated one at a time as needed[4] and are either dispatched for alignment or discarded as dictated by the current clustering. Eliminating the need to store promising pairs and pairwise alignment scores is key to achieving linear space.

Notation: 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. Let $s[i..j]$ denote the substring $s[i]s[i+1]\dots s[j]$, and $|s|$ denote the length of s. Let $N = \sum_{i=1}^{n} |f_i|$. The average length of a fragment is $\approx$ 700–800, making $N \approx 700n$–$800n$. For a node u in the GST, let *path-label*(u) denote the concatenation of edge labels along the path from the root to u. Let the length of u's path-label be denoted by *string-depth*(u). By definition, the path-label of every leaf is a suffix in at least one string. To ensure that each suffix of a string is represented by a leaf, a unique termination symbol '\$' is appended to each input string. For simplicity, this terminal symbol is omitted in measuring the string length $|s|$. Let *leaf*$(s(i))$ denote the leaf corresponding to suffix $s(i)$, and *subtree*(u) denote the set of suffixes corresponding to all leaves in u's subtree.

Definition 1. A string α is a maximal match between fragments f_i and f_j if and only if $\exists$ $f_i(k)$ and $f_j(l)$ such that

- $f_i[k..k'] = f_j[l..l'] = \alpha$, where $k' = k + |\alpha| - 1 \leqslant |f_i|$ and $l' = l + |\alpha| - 1 \leqslant |f_j|$,
- Right maximality: If $k' < |f_i|$ and $l' < |f_j|$, $f_i[k'+1] \ne f_j[l'+1]$,
- Left maximality: If $k > 1$ and $l > 1$, $f_i[k-1] \ne f_j[l-1]$.

Definition 2. For a node u and $c \in \Sigma$, let $\ell_c(u) = \{f_i(j) \mid f_i(j) \in subtree(u);\ j > 1;\ f_i[j-1] = c\}$, and $\ell_\lambda(u) = \{f_i(1) \mid f_i(1) \in subtree(u)\}$. These are collectively called "*lsets*" at u.

The *lsets* at u represent a partition of all suffixes with leaves in u's subtree based on their preceding characters. The algorithm to generate fragment pairs with maximal matches is based on the following key observation.

Lemma 1. *Fragments f_i and f_j share a maximal match α if and only if*

C1. $\exists u$ *such that* $path\text{-}label(u) = \alpha$.
C2. $\exists k$ *and* l *such that* $f_i(k), f_j(l) \in subtree(u)$.
C3. *If u is an internal node, $f_i(k) \in subtree(u')$ and $f_j(l) \in subtree(u'')$, where u' and u'' are two different children of u.*

[3] The reverse complement 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.

[4] In our parallel implementation, we generate pairs in bounded size batches rather than one at a time for maximum utilization of the communication bandwidth, as explained in Section 7.

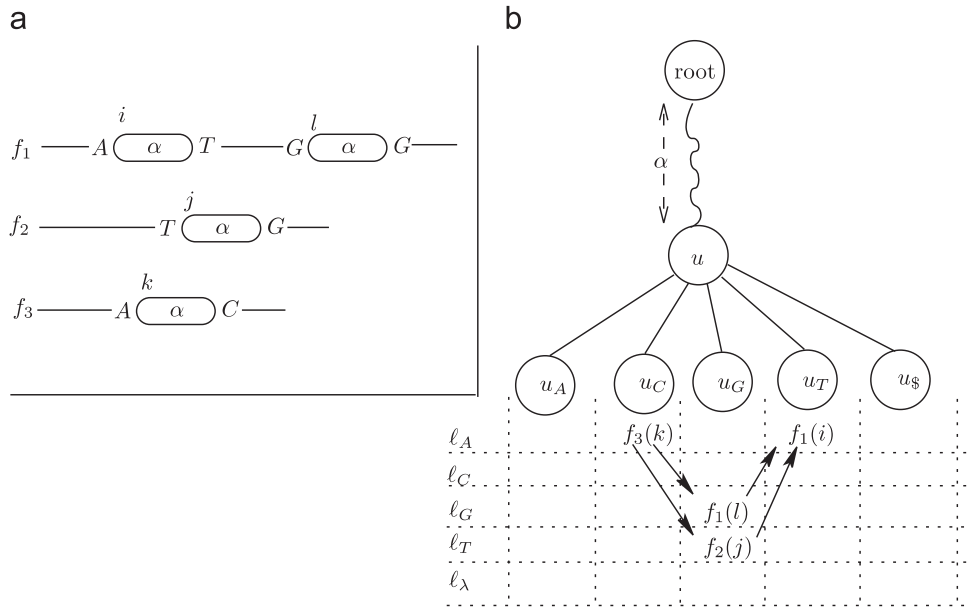

Fig. 4. Detection of maximal match pairs at an internal node u. Part (a) shows three fragments sharing a match α. Part (b) shows the node u in the GST with path-label α with all its children. Each column under a child shows its *lsets* corresponding to characters in $\Sigma \cup \{\lambda\}$. Solid arrows denote the maximal match pairs generated by our algorithm.

C4. *If* $k > 1$ *and* $l > 1$, $f_i(k) \in \ell_{c'}(u)$ *and* $f_j(l) \in \ell_{c''}(u)$, *for* $c', c'' \in \Sigma$ *and* $c' \neq c''$.

Proof. ($\Rightarrow$ C1,C2,C3,C4) The fact that α is a match between f_i and f_j implies that $\exists f_i(k)$ and $f_j(l)$ that share the prefix α. This implies that the path-labels of *leaf*($f_i(k)$) and *leaf*($f_j(l)$) have α as a common prefix. Moreover, the right-maximality of α implies that there will exist an internal node u with path-label α such that u is the lowest common ancestor between *leaf*($f_i(k)$) and *leaf*($f_j(l)$), unless $\alpha = f_i(k) = f_j(l)$. The left-maximality of α ensures that $f_i(k)$ and $f_j(l)$ are in *lsets* corresponding to two different characters at u, unless $k = 1$ or $l = 1$.

($\Leftarrow$ C1,C2,C3,C4) C1 and C2 imply that $f_i(k)$ and $f_j(l)$ share α as a prefix, and therefore α is a match between f_i and f_j. Condition C3 that *leaf*($f_i(k)$) and *leaf*($f_j(l)$) are in two different children of u implies that the leading characters in the edge-labels to u' and u'' are different, thereby implying the right-maximality of α. Similarly, C4 implies the left-maximality of α. □

Thus, identifying all pairs that satisfy C1 … C4 at a node u is equivalent to identifying all pairs with the maximal match *path-label*(u). For generating pairs in decreasing order of their maximal match lengths, the nodes can be processed in decreasing order of their string-depths.

The pair generation algorithm

S1. Compute the GST of all n fragments. Section 6 provides a parallel algorithm.

S2. Sort the nodes with string-depths $\geqslant \psi$ in decreasing order of string-depths, and "process" the nodes in that order. Use step S3 for a leaf and S4 for an internal node.

S3. Let u be a leaf node. First, the *lsets* at u are computed directly from its labels. As right-maximality is automatically satisfied at a leaf, it suffices to check for C4. Thus, pairs at u are obtained by computing $\bigcup \ell_c(u) \times \ell_{c'}(u)$, where $c < c'$ or $c = c' = \lambda$. An arbitrary ordering of characters in $\Sigma \cup \{\lambda\}$ is assumed in order to limit generating a pair to one of its forms: either $(f_i(k), f_j(l))$ or $(f_j(l), f_i(k))$.

S4. Let u be an internal node. Processing the nodes in sorted order of string-depth ensures that u is processed only after all its children are processed. Therefore, by induction let us assume that the *lsets* at every child of u are already computed. The set of pairs at u is obtained by computing $\bigcup \ell_c(u') \times \ell_{c'}(u'')$, for every pair of different children, u' and u'' of u (to satisfy C3), $u' < u''$, and $c < c'$ or $c = c' = \lambda$ (to satisfy C4). Again, an arbitrary ordering of characters and child nodes is assumed to prevent generating redundant copies of pairs. After generating pairs from u, its *lsets* are computed from the *lsets* of its children as follows: $\ell_c(u) = \bigcup_{u'} \ell_c(u')$, where u' is a child of u.

Fig. 4 shows an example illustrating the pair generation algorithm at an internal node.

Lemma 2. *The run-time complexity of the pair generation algorithm is* $O(N + \#\, pairwise\ maximal\ matches\ generated)$. *The space-complexity is* $O(N)$.

Proof. Step S1 is achieved in $O(N)$ time and space [2, Chapter 5]. Radix sort can be used for S2 because the string-depth values are bounded by the maximum length of a fragment ($\leqslant$1000 bp in practice). The overall cost for computing the *lsets* at all leaves is $2N$ because it requires examining one character for each suffix. Generating pairs by computing a cross-product within the leaf's *lsets* (as shown in S3) takes $O(1)$ time per pair. Similarly, for an internal node u, generating pairs by computing a cross-product across *lsets* of different children (as shown in S4) also takes $O(1)$ time per pair. The *lsets* at each node are maintained as linked lists to allow constant time union operations for creating each $\ell_c(u)$ from its counterparts in u's children. Given that there are at most $|\Sigma|+1$ *lsets* and $|\Sigma|+1$ children for each node, the total cost of generating the *lsets* at an internal node is $O(|\Sigma|^2)$, which is a constant for DNA alphabet. Thus, the run-time for generating each pair at an internal node is also $O(1)$. As the *lsets* at an internal node are created by dissolving the *lsets* at its children, the space complexity is $O(N)$. □

The above scheme generates all maximal matches (of length $\geqslant\psi$) 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. Instead of partitioning the suffixes in a node's subtree, we now partition the corresponding *fragments* into its *lsets*—i.e., the definition of *lsets* is changed. Formally, $f_i \in \ell_c(u)$ only if $\exists f_i(k) \in$ *subtree*(u) such that: (i) $f_i[k-1]=c$, $k>1$, and $c\in\Sigma$, or (ii) $k=1$ and $c=\lambda$. Before generating pairs at an internal node u, the *lsets* at u's children are scanned such that all but one arbitrary occurrence of a fragment are removed. We call this process "duplicate elimination", following which the pair generation algorithm is run as before, except that the generated pairs are of the form (f_i, f_j) instead of $(f_i(k), f_j(l))$. Duplicate elimination is not necessary at leaves because a fragment cannot have more than one suffix at a leaf.

Duplicate elimination at an internal node u is achieved in run-time proportional to the sum of the sizes of the *lsets* over all u's children as follows: Maintain a boolean array of size $2n$ indexed by fragments in their direct and reverse complemented forms, and initialized to 0. Linearly scan the *lsets* and set the entry corresponding to f_i to 1, if f_i is encountered for the first time. Each subsequent occurrence of f_i is then labeled a duplicate by examining its entry in the boolean array, and thereby removed from its $lset$. At completion, the modified *lsets* are scanned again, and this time the entry corresponding to each f_i seen is reset to 0; this initializes the array for the next internal node to be processed.

There is a key difference between the new $lset$ partitioning scheme and the previous one. Because the fragment occurrences retained at a node u are arbitrary, *lsets* at u may no longer be unique. As it is only the retained occurrences that participate in pair generation, it is possible that a fragment pair, f_i and f_j, containing *path-label*(u) as a maximal match is no longer generated at u. This happens only if the maximal match *path-label*(u) is embedded in a longer maximal matching pair of substrings between f_i and f_j. However, this is not a problem because it can still be guaranteed that such a pair is generated at least once under some other node. The proof is as follows: among the maximal matches between (f_i, f_j), consider a longest maximal match β, and let v be the node with that path-label. For the case of leaf, the proof is trivial. If v is an internal node, then the fact β is a maximal match will ensure that f_i and f_j occur in *lsets* corresponding to two different characters or λ, under two different children. Moreover, the fact that β is also a *longest* maximal match between f_i and f_j ensures that the duplicate elimination step has no effect on the occurrences of f_i and f_j, leaving both intact for the algorithm to successfully generate the pair at v.

The run-time cost for eliminating duplicates at an internal node u is proportional to the sum of the sizes of *lsets* of all its children, which is bounded by size of *lsets* at u times $(|\Sigma|+1)$—a term bounded by the number of pairs generated at u, as $|\Sigma|=4$ for DNA alphabet. Thus, a pair can still be generated in $O(1)$ time. The above scheme only guarantees that a fragment pair is generated at most once under a given node u. Note that there could be multiple distinct maximal matches between the same fragment pair, each corresponding to a different node in the GST. Therefore, a pair of the form (f_i, f_j) can be generated at most as many times as the number of distinct maximal matches between the fragments.

6. 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 p denote the number of processors.

The first step is to sort all suffixes based on their w-length prefixes, where $w\leqslant\psi$. Partition the fragments such that each processor has approximately $\frac{N}{p}$ bp. 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 approximately $\frac{N}{p}$. While adversarial input such as one bucket containing all N suffixes can be easily constructed, this poses no difficulty in practice because of the following expectation: if input sequence data is random, all of the $|\Sigma|^w$ distinct substrings are expected to occur with equal probability in the set of input fragments, implying an even distribution of suffixes across the $|\Sigma|^w$ buckets. While this may not entirely hold in practice, the fragments are still diverse enough because of sequence cleaning and repeat masking. Therefore, a value between 10 and 12 for w can be expected to generate millions of buckets sufficient to be distributed in a load balanced manner even for thousands of processors. Empirically, a value of $w=11$ was found appropriate for the data and the range of processors we tested (up to 1024 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 $\geqslant w$. 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

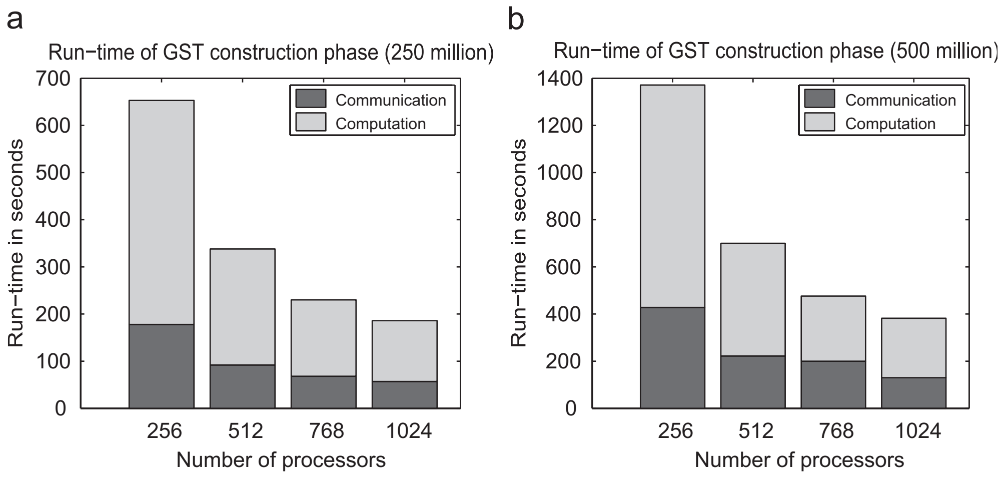

Fig. 5. Parallel run-times for constructing GST on inputs of sizes: (a) 250 million, and (b) 500 million bp.

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 $\geqslant w$. 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 $O\left(\min\left\{\frac{N\times l}{p}, N\right\}\right)$ 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 typical of large scale parallel computers such as the 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.

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

6.1. Experimental results

We studied the performance of our GST construction algorithm by varying the number of processors from 256 to 1024. 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 bp that comprised 322,009 and 649,957 fragments, respectively. Fig. 5 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.

7. Detecting overlaps and managing clusters in parallel

Once a distributed representation of the GST for all input fragments is constructed in parallel, each processor generates promising pairs from its local collection of subtrees using the algorithm described in Section 5 as follows: sort all the nodes in the local subtrees in decreasing order of string-depth and process them in that order. This mechanism of generating promising pairs in the decreasing order of their maximal match lengths within each processor roughly approximates the global sorted order in practice. As the results presented later in this section demonstrate, the alignment work reduction achieved through this approximate generation scheme is significant in practice. Explicitly computing the global sorted order is likely to introduce high communication and synchronization overheads into the parallel run-time enough to negate the benefits realized in terms of the potential reduction in the number of computed alignments.

As pairs are generated, they need to be checked against the current clustering, allocated for alignment if necessary, and the

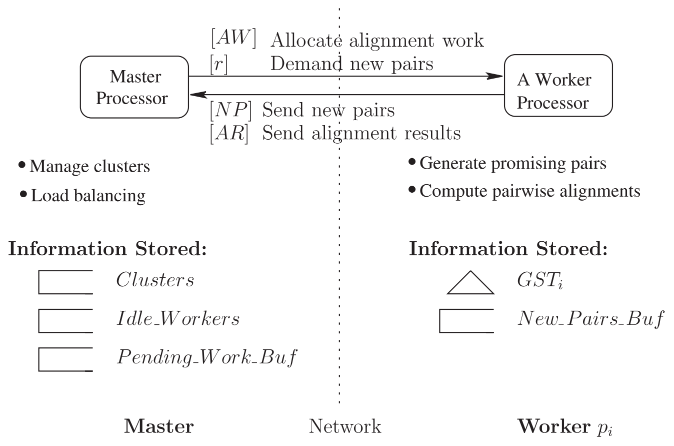

Fig. 6. A "single master–multiple workers" design for detecting overlaps and clustering in parallel, with responsibilities designated as shown.

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.

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 tasks (by aligning pairs), also generate tasks (by generating 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. Henceforth, we call such processors *passive* while those that still have pairs to generate are called *active* processors. In the interest of maintaining parallel efficiency, it is necessary to keep passive workers busy computing alignments. Also, allocating pending alignment computations to passive workers ahead of any active worker can help balance pair generation and pairwise alignment computation dynamically.

With the above goals in mind, we designed an iterative solution with responsibilities as shown in Fig. 6. The master and worker processors interact iteratively until all promising pairs are generated and all alignments identified as necessary have been computed. The master processor is responsible for maintaining the clusters, selecting and allocating pairs for alignment computation, and load balancing. Each worker processor is responsible for generating promising pairs from its local GST portion in decreasing order of maximal match length, computing alignments for pairs allocated by the master processor, and reporting the alignment results to the master processor. To reduce communication setup costs, the worker processors send pairs in batches instead of one pair at a time. Similarly, the master processor also allocates pairs for alignment computation and collects their results in batches.

The master and workers store and maintain the following information.

Information at the master processor:

- *Clusters*: The current set of clusters maintained using the Union–Find data structure. This allows the operations of finding the cluster containing a given fragment, and the merging of two clusters, to run in amortized time given by the inverse of Ackermann's function, a constant for all practical purposes.
- *Pending_Work_Buf*: A fixed size buffer to temporarily store the pairs selected but not yet dispatched for alignment computation. This is implemented as a queue.
- *Idle_Workers*: A list of all passive workers who do not have any alignment work allocated to them. This is implemented as a queue.

Information at a worker processor P_i:

- GST_i: The local portion of GST.
- *New_Pairs_Buf*: A fixed size buffer to temporarily store newly generated promising pairs that have not yet been sent to the master processor. This is implemented as a queue.

The following messages are exchanged between the master and an arbitrary worker processor P_i during one iteration:

- *AW*: A new batch of alignment work allocated by master to P_i. The number of pairs sent in each batch, called *batch size*, is a fixed, user-specified parameter with value b. *AW* is implemented as an array.
- r: The number of promising pairs to be sent by P_i during its next communication with the master.
- *NP*: A batch of new promising pairs sent by P_i to the master processor. This is implemented as an array.
- *AR*: A list of alignment results sent by P_i to the master processor. The results are for the alignments computed over the most recent batch of pairs allocated by the master to P_i. *AR* is also implemented as an array.

Figs. 7 and 8 detail the algorithms for the master and a worker processor, respectively. In each iteration, the master processor polls for messages from any of the workers. When a message arrives from a worker P_i, the master updates $Clusters$ using the alignment results that are satisfactory, scans the batch of newly generated pairs from P_i, and adds only those pairs for which alignments are necessary to *Pending_Work_Buf*. It then repeatedly extracts batches of size b from *Pending_Work_Buf*, dispatching each batch to an idle worker. If all workers become idle, then it signals the end of clustering. If no more idle workers remain and if there is more work left in the *Pending_Work_Buf*, then the next batch of b pairs are allocated to P_i. In the same message, the master also piggybacks the number of new pairs, r, that it expects to receive from p_i in its next communication; r is given by: $\min\left\{\frac{|NP|}{|NP'|} \times \frac{p}{p_{active}} \times b, \frac{|Pending\_Work\_Buf|}{p_{active}}\right\}$, where p_{active} denotes the number of active processors. The main idea is to request as many pairs as necessary to expect that b of them would be selected for alignment computation,

```
Algorithm 2 Algorithm for Master Processor

1. $Clusters \leftarrow$ Initialize such that each fragment is in a cluster of its own
   $p_{active} \leftarrow p$
   $Idle\_Workers \leftarrow \emptyset$
2. REPEAT
     Blocking Receive until message from an arbitrary processor $P_i$
     $NP \leftarrow$ new promising pairs
     $AR \leftarrow$ alignment results
     IF $NP = \emptyset$ AND $P_i$ is active THEN
         Mark $P_i$ as passive
         Decrement $p_{active}$
     Update $Clusters$ based on $AR$
     $NP' \leftarrow$ Identify pairs in $NP$ that need alignment computation
     Add $NP'$ into $Pending\_Work\_Buf$
     FOR EACH $P_j \in Idle\_Workers$ DO
         $AW \leftarrow$ Dequeue $min\{b, |Pending\_Work\_Buf|\}$ pairs
         IF $AW \neq \emptyset$ THEN
             Send $AW$ to $P_j$
             Remove $P_j$ from $Idle\_Workers$
     $AW \leftarrow$ Dequeue $min\{b, |Pending\_Work\_Buf|\}$ pairs
     $r \leftarrow min\{\frac{|NP|}{|NP'|} \times \frac{p}{p_{active}} \times b, \frac{|Pending\_Work\_Buf|}{p_{active}}\}$
     IF $AW \neq \emptyset$ OR $r > 0$ THEN
         Send $(AW, r)$ to $P_i$
     ELSE
         Insert $P_i$ into $Idle\_Workers$
   UNTIL all workers become idle
3. Send termination signal to all workers
4. Output $Clusters$
```

Fig. 7. Algorithm for the master processor. Bold font indicates a communication step.

while not overflowing *Pending_Work_Buf*. In other words, this load balancing strategy aims at regulating the inflow of work so as to keep the outflow roughly constant.

In each iteration, a 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, the worker computes alignments on the batch of pairs allocated by the master during the previous iteration. This is effective in masking the communication wait time with computation. If alignment computation is completed before the master replies, then the worker processor resumes from its earlier state of pair generation and generates fresh batches of promising pairs from its local GST portion until either a message from the master arrives or its temporary store *New_Pairs_Buf* is full. If a worker becomes passive, it keeps itself busy by computing alignments that the master allocated.

7.1. Run-time and space complexity

The parallel work in the master–worker phase for performing pair generation, alignment and clustering is $O(K + K' \times l^2 + K \times A(n))$, where K is the number of promising pairs generated, $K'(\leqslant K)$ is the number of pairs selected for alignment, l is the maximum length of a fragment aligned, and A is the inverse of the Ackermann function required to access the union-find data structure for clustering. Although K, and therefore K', are $\Theta(n^2)$ in the worst-case, the maximal match based definition of promising pairs and the algorithmic heuristics in our approach provide a significant reduction in practice. As an example, for $n \approx 1.6$ million maize fragments, $K \approx 48$ million pairs and $K' \approx 22$ million pairs.

The overall space complexity for each worker processor is $O\left(\frac{N}{p}\right)$. The constant of proportionality in our current implementation is $\approx$ 80 bytes per input character. Moreover, our implementation supports the version of pair generation algorithm that performs duplicate elimination, requiring an additional $2n$ bits per processor. For the master processor, the union-find data structure is implemented as an array of n integers and, therefore the space complexity is $O(n)$. Even for systems with a low memory footprint per processor such as our IBM BlueGene/L that has only 512 MB per node, this implementation allows values over 100 million for n, assuming 4 bytes per integer.

Algorithm 3 *Algorithm for a Worker Processor P_i*

```
1. $AW \leftarrow$ Generate next $b$ promising pairs from $GST_i$
2. $AR \leftarrow$ Compute alignments on $AW$
3. $AW \leftarrow$ Generate next $b$ promising pairs from $GST_i$
4. $NP \leftarrow$ Generate next $b$ promising pairs from $GST_i$
5. $r \leftarrow b$
6. REPEAT
     **Send** $NP$ and $AR$ to master
     $AR \leftarrow$ Compute alignments on $AW$
     $(AW,r) \leftarrow$ **Non-blocking Receive** from master
     REPEAT
          Generate $r$ pairs from $GST_i$ and add to $New\_Pairs\_Buf$
     UNTIL message arrives from master OR $New\_Pairs\_Buf$ is full
     IF no message from master THEN
          $(AW,r) \leftarrow$ **Blocking Receive** until master sends a message
     $NP \leftarrow$ Extract $r$ pairs, first from $New\_Pairs\_Buf$ and then from $GST_i$ if
     necessary
   UNTIL no more promising pairs to generate from $GST_i$
7. REPEAT
     $AW \leftarrow$ **Blocking Receive** from master
     $AR \leftarrow$ Compute alignments on $AW$
     **Send** $AR$ to master
   UNTIL master sends termination signal
```

Fig. 8. Algorithm for each worker processor. Bold font indicates a communication step.

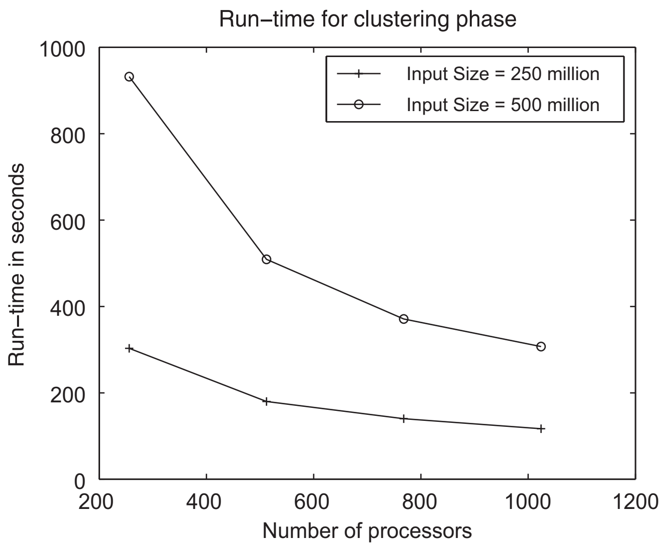

Fig. 9. Total parallel run-time for the entire clustering algorithm excluding that of GST construction.

7.2. Experimental results

We studied the performance of our master worker implementation on the BlueGene/L (see Fig. 9). 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 1024, 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 1024 processors on the 250 million input, and from 9% to 16% for the 500 million input—indicating that the processor size can be quadrupled when doubling the problem size to maintain parallel efficiency. This is expected because of the near quadratic growth in the alignment workload with input size. Note that a full sequencing project will generate over 22 billion bp (30 million fragments each about 750 bp long), on which tens of thousands of processors can be utilized with our scheme. In fact, we tested an 8192-processor run on an input containing as small as 1.15 billion bp, and observed a total run-time of 75 min.

Table 1 shows the number of promising pairs generated as a function of the input size. This table also shows the effectiveness of our clustering heuristic in significantly reducing the number of alignments computed. For the entire maize data, which has 1,607,364 fragments of total size 1.252 billion bp, only $\sim$ 44% of the pairs generated are aligned. However, less than 4% of the pairs aligned contributed to merging of clusters, indicating the presence of numerous medium-sized ($\sim$ 100 bp) repeat 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 [16].

Currently, 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

Table 1
The number of promising pairs generated, aligned, and accepted by our algorithm as a function of input size on the maize data

Input size (N)	250 Mbp	500 Mbp	1000 Mbp	1252 Mbp
Input number of fragments (n)	322,009	649,957	1,338,106	1,607,364
No. promising pairs generated	2×10^6	6×10^6	34×10^6	47×10^6
No. promising pairs aligned	0.8×10^6	2.8×10^6	16×10^6	20.3×10^6
No. promising pairs accepted	0.1×10^6	0.4×10^6	0.7×10^6	0.8×10^6
Fraction of generated pairs not aligned (i.e., %savings)	60	53.3	52.9	56.8

All entries are rounded to the first decimal place. The last row shows the fraction of pairs for which alignment computations were deemed unnecessary by our approach.

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 a Myrinet Pentium cluster. An alternative is to change the underlying design to allow scaling the number of master processors with processor size. There are a few key challenges: if the set of clusters is replicated on all master processors, then explicit synchronization steps are required to dynamically monitor and update the individual copies of the local clustering to keep them consistent across master processors. Otherwise, it will not be possible to take full advantage of our heuristic scheme—a lack of most recent clustering information could result in an undesirable increase in the redundant alignment workload.

With the current single master implementation, we observed a gradual decrease in its availability (i.e., idle time) from 90% to 70% when the processor size was increased from 256 to 1024, keeping the input size fixed. This is because more worker processors are added for the same computational workload, resulting only in an increase of the frequency at which the master processor is consulted. While scaling the number of master processors gradually with the worker processor size is a plausible strategy, the other challenges inherent in such an approach, as mentioned above, need to be addressed in designing an effective solution. An easier and perhaps more effective alternative is to modify the current single master approach such that it empirically strikes a balance between processor size and the granularity of the computational workload dispatched to each worker processor, such that the frequency at which messages arrive at the master does not increase with increase in processor size.

8. Maize genome assembly

Our initial parallel assembly framework was designed to carry out assemblies of gene-enriched fragments derived from maize to make them available to the maize genetics community as quickly as possible [8]. Newer versions of the assembly were generated as additional sequences became available. The analysis results presented in this paper are based on maize genomic data composed of 3,124,130 fragments. This includes 852,838 methyl-filtrated (MF) [27] and High-C_0t (HC) [35] 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 bp) 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 2.

As with any 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* [7]. In addition, we designed a database of known and statistically defined repeats [8] 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. The last two columns in Table 2 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.

The results of applying our parallel genome assembly framework are as follows: preprocessing the 3,124,130 fragments downloaded from GenBank took 1 h by trivially parallelizing on 40 processors of a Myrinet Pentium cluster with 1.1 GHz Pentium III processors and 1 GB RAM per processor. Our clustering method partitioned the resulting 1,607,364 fragments (over 1.25 billion bp) in 102 min on 1024 nodes of the BlueGene/L. Construction of the GST, containing over 2.5 billion leaf nodes, took only the first 13 min. We used CAP3 [14] for assembling the fragments in each resulting cluster. This assembly step

Table 2
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)

Fragment type	Before preprocessing		After preprocessing	
	No. fragments	Total length (in millions)	No. 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	2526	1,607,364	1252

finished in 8.5 h on 40 processors of the Myrinet Pentium cluster through trivial parallelization.

Our clustering resulted in a total of 149,548 clusters containing two or more input fragments, and 244,727 singletons. Singletons are fragments that do not cluster with any other fragment because of sharing no overlap and/or having a high repetitive content that was masked during preprocessing. The average number of input fragments per non-singleton cluster is 9.00, while the maximum is 86,369, or 5.37% of the input size. It should be emphasized that targeted and effective masking of repeats has significant influence on the largest cluster size [8]. On an average, each cluster assembled into 1.1 contigs. Given that the CAP3 assembly was 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 for conventional assembly. The results of this assembly are available at http://www.plantgenomics.iastate.edu/maize under "Download".

While the above results demonstrate the effectiveness of the proposed massively parallel framework, it is important to ensure the correctness of the assembly. Validation was carried out using a number of computational and experimental methods. Alignments to ten highly accurate maize genes (74 kb of highly finished sequence), as described previously [9], indicated a less than 1 nucleotide in 10,000 was incorrect relative to the benchmark. These few inconsistencies are residual errors from the sequencing process. Even so, this assembly satisfies the quality standard agreed upon for the human genome [20]. For extensive validation of the assembly parameters used during this study and large-scale experimental verification of predicted novel maize genes on an earlier version of maize data with less than a million fragments, the reader is referred to [9]. Perhaps the most comprehensive validation is the successful use of our assemblies by hundreds of researchers over the past three years. Misassemblies can cause a variety of experimental failures, which have been seldom observed when our assemblies are used in a variety of projects. In addition, our own experiments showed that the rate of experimental failure was no higher in assembled regions than well-defined controls in a previous assembly [9]. Given that the best benchmarks are the DNA molecules in a maize cell, each successful experiment is an empirical ratification of our assembly as an accurate reconstruction of the maize genome.

9. Whole genome shotgun and environmental assemblies

To demonstrate the effectiveness of the proposed approach for other types of sequencing projects, we consider whole genome shotgun (WGS) sequencing, and the more recent approach of simultaneously sequencing fragments from thousands of bacterial genomes in environmental samples.

9.1. Assembly of the drosophila genome

To demonstrate the effectiveness of our cluster-then-assemble approach to conventional whole genome shotgun sequencing, we reassembled the recently sequenced genome of the fruit fly *Drosophila pseudoobscura*. This fruit fly species diverged from the model organism *Drosophila melanogaster* approximately 25–55 million years ago [28], and its genome contains approximately 205 million bp. It was sequenced using a WGS approach by the Baylor College of Medicine [28].

The initial data set consisted of 2,686,355 sequences downloaded from GenBank's trace archive. After trimming using Lucy under default parameters, a total of 2,666,207 high quality fragments were obtained totaling 1.81 billion bp. The corresponding coverage provided by these trimmed reads (8.8X) is consistent with the observed coverage in the *D. pseudoobscura* assembly (9.1X; [28]), suggesting that trimming will not affect the resulting number of contigs. Estimation of repeats and masking of repeat sequences is key to producing a manageable largest cluster size. Repeats can be identified through their statistical over-representation in a random sample. Because WGS fragments themselves comprise a random sample, we used 32,462 randomly chosen fragments (0.1X coverage) to predict 5407 high-copy sequences in this fruit fly genome. Repeat masking resulting in 2,074,483 fragments that comprised 1.37 billion unmasked bases. Clustering of this masked data took 3.1 h on 1024 nodes of BlueGene/L. A total of 32,893 non-singleton clusters and 174,277 singleton clusters were generated. The average cluster size is 57.77 and the largest cluster is composed of 140,307 fragments (about 6.76% of the total).

To demonstrate the importance of repeat masking to the proposed cluster-then-assemble approach, we performed clustering without repeat masking. Not only did clustering take 24 h on 1024 BlueGene/L nodes due to the large number of pairwise alignments forced by the repeats, almost 50% of the fragments were combined into one large cluster.

Table 3
Performance results on clustering whole genome shotgun fragments of *Drosophila pseudoobscura* and environmental sample from Sargasso Sea using 1024 node BlueGene/L

Input data	No. fragments	Total length (in billion bp)	Clustering time (in min)		Promising pairs (in millions)		
					Aligned		Not aligned
			GST	Total	Accepted	Rejected	
Drosophila	2,074,483	1.36	15	187	1.87	26.6	52.78
Sargasso Sea	1,660,141	1.47	28	199	0.85	26.3	35.90

Biological validation of our assembly was carried out by using the published genome as benchmark. We aligned over 1.3 million random WGS fragments to the draft *D. pseudoobscura* assembly using BLASTN [1] (95% identity over 80% of the fragment's length). The lower identity was used to compensate for sequencing errors (95% identity) and lower coverage to accommodate residual vector sequence (80% of the fragment's length) that may be present in low quality sequences. Only the best match was stored for further analysis. Significantly, 27,830 out of 28,185 clusters post-masking (98.7%) map to a single benchmark sequence. This result suggests that even prior to assembly our clustering methodology in tandem with masking achieves high specificity.

9.2. Assembly of environmental genome sequences

Most bacterial species cannot be currently studied under laboratory conditions. They can, however, be subjected to modern DNA sequencing approaches and such "metagenomics" approaches have recently been applied to "survey" entire microbial communities [33]. Assembly of heterogeneous samples of DNA is an open problem convoluted by closely related bacterial sequences present in a sample. Even so, such complications do not affect clustering and, although an assembler would have the difficult task of differentiating highly similar sequences, deconvolution would be made easier by reducing individual problem sizes using clustering.

We tested the effectiveness of our clustering approach on the largest environmental WGS data set currently available: 1.66 million fragments obtained from the Sargasso Sea [33]. After removing ubiquitous sequences from the sample by masking, a total of 825,696 clusters were obtained generated including 129,741 non-singleton clusters. Although comprehensive validation of resulting contigs would be difficult because of inherent complexities, our cluster-then-assemble approach could enhance any future environmental assembler. This is especially true given the extensive diversity in such samples; the Sargasso Sea data set contains over 1800 unique species, many of which could generate one or more clusters.

Table 3 shows further experimental results on our clustering of *Drosophila* fragments and environmental sequences. The total run-time is about the same in both cases because the numbers of the pairs aligned are roughly the same ($\sim$ 26–27 million pairs). The table also shows the savings achieved by our clustering strategy in the number of pairs selected for alignment computation—for the environmental data $\sim$ 57% of the pairs generated are not selected for alignment, while for the *D. pseudoobscura* data this fraction is 65%.

10. Conclusions and future directions

We presented the design and development of an efficient clustering-based framework for genome assembly on massively parallel distributed memory computers. We demonstrated the effectiveness of our approach on random shotgun sequencing, gene-enriched sequencing, and sequencing of environmental samples. Run-time results on a 1024 node BlueGene/L show significant reduction in assembly turnaround times, from weeks to less than a day. Faster turnaround times also encourage more experimentation, which is useful as a host of parameters influence the final assembly. Perhaps equally important, the proposed scheme fully automates assembly for large-scale sequencing projects when compared to the previous approaches that often require manual intervention for data partitioning, storing of intermediate results and running multiple programs.

An important contribution of our framework is the assembly of gene-enriched maize fragments, which are frequently being 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 $\approx$ 8 million fragments for every 1024 BlueGene/L nodes (each with 512 MB). This would enable us to cluster 30 million fragments on about 4000 nodes. Moreover, we conducted a few preliminary experiments on 8192 nodes and the scaling results are encouraging. We believe that a continued improvement of our algorithmic techniques on large-scale parallel computers 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.

The effectiveness of our clustering approach can be further enhanced by resolving inconsistent overlaps during cluster formation. By reducing the largest cluster size, this will increase available parallelism during the assembly phase. Even with the limitations of single linkage clustering, the partitioning of original data is sufficient to allow assembly software to run on commonly available desktop machines. Recent advances in high throughput sequencing, such as the 454 GS20 sequencer that can sequence nearly 200,000 fragments in about 4 h, are causing a significant gap between data generation and data processing capabilities. We believe parallel approaches such as the one presented here should become increasingly more important as high throughput sequencing techniques become mainstream.

Acknowledgments

We thank Sam Ellis, Kurt Pinnow and Brian Smith of IBM Rochester for facilitating access to the BlueGene/L supercomputer at the initial stages of this research. We are grateful to the anonymous reviewers for commenting on an earlier version of this manuscript. This research was supported in part by NSF Grants CNS-0521568 and DBI-0527192, and an IBM Ph.D. Fellowship.

References

[1] S.F. Altschul, W. Gish, W. Miller, et al., Basic local alignment search tool, J. Mol. Biol. 215 (1990) 403–410.

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

[3] K. Arumuganathan, E.D. Earle, Nuclear DNA content of some important plant species, Plant Mol. Biol. Reporter 9 (3) (1991) 211–215.

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

[5] J.A. Bedell, M.A. Budiman, A. Nunberg, et al., Sorghum genome sequencing by methylation filtration, Public Lib. Sci. 3 (1) (2005) e13.

[6] J.L. Bennetzen, V.L. Chandler, P.S. Schnable, National Science Foundation-sponsored workshop report. Maize genome sequencing project, Plant Physiol. 127 (2001) 1572–1578.

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

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

[9] 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, Proc. Nat. Acad. Sci. USA 102 (2005) 12282–12287.

[10] O. Gotoh, An improved algorithm for matching biological sequences, J. Mol. Biol. 162 (1982) 705–708.

[11] P. Green, Phrap—the assembler, 1994. ⟨http://www.phrap.org⟩.

[12] D. Gusfield, Algorithms on strings, trees and sequences, Computer Science and Computational Biology, Cambridge University Press, Cambridge, London, 1997.

[13] P. Havlak, R. Chen, K.J. Durbin, et al., The Atlas genome assembly system, Genome Res. 14 (2004) 721–732.

[14] X. Huang, A. Madan, CAP3: a DNA sequence assembly program, Genome Res. 9 (9) (1999) 868–877.

[15] X. Huang, J. Wang, S. Alurum, et al., PCAP: a whole-genome assembly program, Genome Res. 13 (9) (2003) 2164–2170.

[16] A. Kalyanaraman, S. Aluru, V. Brendel, S. Kothari, Space and time efficient parallel algorithms and software for EST clustering, IEEE Trans. Parallel Distrib. Systems 14 (12) (2003) 1209–1221.

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

[18] J.C. Mullikin, Z. Ning, The Phusion assembler, Genome Res. 13 (1) (2003) 81–90.

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

[20] National Human Genome Research Institute, Standard finishing practices and annotation of problem regions for the human genome project, 2001. ⟨http://www.genome.gov/10001812⟩.

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

[22] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol. 48 (1970) 443–453.

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

[24] W.R. Pearson, D.J. Lipman, Improved tools for biological sequence comparison, Proc. Nat. Acad. Sci. USA 85 (1988) 2444–2448.

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

[26] P.A. Pevzner, H. Tang, M.S. Waterman, An Eulerian path approach to DNA fragment assembly, Proc. Nat. Acad. Sci. USA 98 (17) (2001) 9748–9753.

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

[28] S. Richards, Y. Liu, B.R. Bettencourt, et al., Comparative genome sequencing of *Drosophila pseudoobscura*: chromosomal, gene, and cis-element evolution, Genome Res. 15 (1) (2005) 1–18.

[29] F. Sanger, A.R. Coulson, G.F. Hong, et al., Nucleotide sequence of bacteriophage lambda DNA, J. Mol. Biol. 162 (4) (1982) 729–773.

[30] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, J. Mol. Biol. 147 (1981) 195–197.

[31] G. Sutton, O. White, M. Adams, A. Kerlavage, TIGR assembler: a new tool for assembling large shotgun sequencing projects, Genome Sci. Technol. 1 (1) (1995) 9–19.

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

[33] J.C. Venter, K. Remington, J.F. Heidelberg, et al., Environmental genome shotgun sequencing of the Sargasso Sea, Science 304 (5667) (2004) 58–60.

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

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

Ananth Kalyanaraman is an Assistant Professor at the School of Electrical Engineering and Computer Science in Washington State University, Pullman. In 1998, he received a B.E. in Computer Science and Engineering from Visvesvaraya National Institute of Technology in Nagpur, India. Later, he received his M.S. in Computer Science and Ph.D. in Computer Engineering from Iowa State University, Ames, USA, in 2002 and 2006, respectively. His research areas are bioinformatics and computational biology, parallel algorithms and applications, and string algorithms. The primary focus of his work has been on developing high-performance computing solutions for computer and data-intensive problems in the area of computational biology and bioinformatics. Ananth is a recipient of best paper awards at the International Parallel and Distributed Processing Symposium (2006) and the Computational Systems Bioinformatics Conference (2005), and Ph.D. fellowship awards from IBM (2004–2006) and Pioneer Hi-Bred International, Inc. (2003). He serves in the program committees for the International Parallel and Distributed Processing Symposium (2007), the International Conference on Parallel Processing (2007), and the Parallel Bio-Computing Workshop (2007). He is a member of Association for Computing Machinery, Institute of Electrical and Electronics Engineers, Inc., International Society for Computational Biology, Life Sciences Society, and Society for Industrial and Applied Mathematics.

Scott Emrich received the BS in biology and Computer Science from Loyola College in Maryland in 2002 and is a Ph.D. candidate in bioinformatics and computational biology at Iowa State University. He will join the faculty of the University of Notre Dame in 2007 as an Assistant Professor in the Department of Computer Science and Engineering. His research interests include computational biology, bioinformatics and parallel computing. He has received this best paper award at the 2006 IEEE International Parallel and Distributed Processing Symposium and has had his interdisciplinary research featured in journals such as Science. He has also produced the first public large-scale assemblies of both the maize and sorghum genomes, which are in wide use by plant researchers. He is a member of ACM, ISCB and IEEE.

Patrick S. Schnable received his B.S. from Cornell University in 1981. He was awarded a Ph.D. from Iowa State University in 1986 for his studies of the Mu transposon with Peter Peterson. Following a post-doctoral appointment with Heinz Saedler at the Max Planck Institute in Koeln, Germany, he was appointed to the faculty at Iowa State University in 1988. He is currently a professor in the Departments of Agronomy and Genetics, Development & Cell Biology. He also serves as the Associate Director of Iowa State University's Plant Sciences Institute and as the founding director of ISU's Center for Plant Genomics.
Professor Schnable manages a vigorous research program that emphasizes interdisciplinary approaches to understanding plant biology. His own expertise is in the areas of genetics, molecular biology and genomics, but he collaborates with researchers in diverse fields, including biochemistry, plant breeding, plant physiology, bioinformatics, computer science and engineering. His research interests include maize genome structure, heterosis, meiotic recombination, cytoplasmic male sterility, cuticular wax biosynthesis, and the development of new genomic technologies and bioinformatics approaches.
Schnable serves on variety of scientific advisory boards and is an elected member of the American Association for the Advancement of Science Section Committee of the Agriculture, Food and Renewable Resources Section and is chair of the Maize Genetics Executive Committee.
Schnable is an active participant in interdisciplinary graduate training programs, including the Interdepartmental Genetics, Bioinformatics and Computational Biology, Interdepartmental Plant Physiology and the Molecular, Cellular & Developmental Biology graduate programs.

Srinivas Aluru is the Stanley Chair in Interdisciplinary Engineering, and Professor of Electrical and Computer Engineering at Iowa State University. He is a member of the Center for Plant Genomics, L.H. Baker Center for Bioinformatics and Biological Statistics, and the Bioinformatics and Computational Biology graduate program which he formerly chaired. Earlier, he held faculty positions at New Mexico State University and Syracuse University. He received his B.Tech. degree from the Indian Institute of Technology, Chennai, India in 1989, and his M.S. and Ph.D. degrees in Computer Science from Iowa State University in 1991 and 1994, respectively, all in Computer Science. He conducts research in parallel algorithms and applications, bioinformatics and computational biology, and combinatorial scientific computing. His contributions to computational biology are in computational genomics, string algorithms, and parallel methods for solving large-scale problems arising in biology. Aluru is a recipient of the NSF Career award (1997), IBM faculty award (2002), Iowa State University Foundation award for mid-career achievement in research (2006), Warren B. Boast Undergraduate Teaching Award (2005), and best paper awards at the International Parallel and Distributed Processing Symposium (2006) and Computational Systems Bioinformatics Conference (2005). He is an IEEE Computer Society Distinguished Visitor from 2004 to 2006. He served on numerous program committees and has served in leadership roles at several conference and workshops in parallel processing and computational biology, including serving as Program Chair for HiPC 2007, Program Vice Chair for IPDPS 2007, ICPP 2007, HiPC 2006, and SC 2003. He co-chairs an annual workshop in High Performance Computational Biology ⟨http://www.hicomb.org⟩ and co-edited several special issues on this topic. He edited a comprehensive handbook on computational molecular biology, published in 2005. He is a member of ACM, SIAM, ISCB, Life Sciences Society, and a senior member of IEEE and IEEE Computer Society.