A Preprocessor for Shotgun Assembly of Large Genomes

Michael Roberts, Brian R. Hunt, and James A. Yorke
Institute for Physical Science and Technology
University of Maryland
College Park MD 20742-2431

Randall A. Bolanos and Arthur L. Delcher*
Celera Genomics
45 West Gude Drive
Rockville MD 20850

February 24, 2004

Abstract

The whole-genome shotgun (WGS) assembly technique has been remarkably successful in efforts to determine the sequence of bases that make up a genome. WGS assembly begins with a large collection of short fragments that have been selected at random from a genome. The sequence of bases at each end of the fragment is determined, albeit imprecisely, resulting in a sequence of letters called a "read". Each letter in a read is assigned a quality value, which estimates the probability that a sequencing error occurred in determining that letter. Reads are typically cut off after about 500 letters, where sequencing errors become endemic.

We report on a set of procedures that (1) corrects most of the sequencing errors, (2) changes quality values accordingly, and (3) produces a list of "overlaps", i.e., pairs of reads that plausibly come from overlapping parts of the genome. Our procedures, which we call collectively the "UMD Overlapper", can be run iteratively and as a preprocessor for other assemblers. We tested the UMD Overlapper on Celera's *Drosophila* reads. When we replaced Celera's overlap procedures in the front end of their assembler, it was able to produce a significantly improved genome.

*Current address: The Institute for Genomic Research, 9712 Medical Center Drive, Rockville MD 20850

1 Introduction

"Shotgun assembly" is one of the approaches used to ascertain a genome's sequence, that is, its sequence of **bases**, each represented by one of four letters, `A`, `C`, `G`, or `T`. The use of the shotgun method to assemble overlapping fragments of a genome is described in [2, 13, 4], and will be summarized below.

1.1 Brief History

Weber and Myers [23] proposed that the entire human genome, with a length of 3 billion bases, could be sequenced with the shotgun method. Previously, shotgun assembly had been applied only to small genomes, or pieces of larger genomes, with fewer than one million bases. Many doubted whether whole-genome shotgun (WGS) assembly of large genomes was feasible [5]. As a preliminary test of concept, Celera Genomics and the Berkeley Drosophila Genome Project achieved a shotgun assembly of the bulk of the *Drosophila* genome, with a length of well over 100 million bases [16]. Considerable later effort has gone into finishing the genome's sequence. In 2001, Celera [21] announced the successful whole-genome shotgun assembly of the human genome. Both [16] and [21] report in considerable detail on an actual shotgun sequencing of a large genome. Shotgun sequencing produces a relatively inexpensive but incomplete draft, and it is sometimes difficult to find the rest of the sequence.

1.2 Preliminary Notions

We divide the WGS assembly method into four phases to clarify the role of the topic of our paper, overlap pairing, and introduce terminology.

Phase 1, shotgun sequencing. WGS assembly begins with the generation of a library of fragments drawn at random from the genome. In this phase, many copies of the entire genome are first broken into a large number of **inserts**, which are screened by length. Each insert consists of a sequence of contiguous bases from the underlying genome, and each base of an insert corresponds to the base at a particular location[1] in the genome. Then, using the procedure described in [19], the base sequences of the ends of each insert are determined. Each reported sequence is called an **untrimmed read**. Untrimmed reads are not always faithful representations of the actual genomic sequence; typically there are sequencing errors (substitutions, insertions, and deletions).[2] The software component, Phred [6], that is usually used to create the untrimmed reads also creates for each base a **quality value**, which is an estimate (on a logarithmic scale) of the probability that a sequencing error was made at that base. While the error rate due to sequencing errors is typically low at the end of the insert (the beginning of the untrimmed read), the fidelity of the untrimmed read sequence to that of the underlying DNA gradually deteriorates until, typically after about 500 bases, it is unacceptable. This is addressed by eliminating the portion of the untrimmed read where the expected error rate per base exceeds some predefined threshold. The sequence that results when the unacceptable end of an untrimmed read is eliminated is called a **read**, and the resulting collection of reads we call a **read library**. Enough inserts are created that the total number of bases in the read library is several times as large as the number of bases in the

[1] A location in the genome may be specified as a pair consisting of a chromosome number along with an offset, which is the number of bases preceding the given base in the chromosome, ignoring intra-species variation in the (length of the) DNA sequence.

[2] In addition, the beginning of each untrimmed read usually contains foreign DNA sequence from the "cloning vectors" that are used to create the inserts. The vector sequence is known, and there are many programs designed specifically to remove it from the reads; henceforth we assume that this step has been completed.

genome; the number of times each base is represented on average is called the **coverage** of the read library. While each read **represents** a particular region of the genome in the WGS approach, no information is obtained as to where in the genome the insert that produced the read came from.

The other information available from this phase is approximate knowledge of the length of the inserts. For the pair of reads derived from opposite ends of an insert, called a **mated pair**, we know the approximate number of bases separating the regions of the genome they represent and also their relative orientation. The insert lengths are typically from 2000 bases up to 150,000 bases for BACs.

Phase 2, overlap pairing. The remaining task of putting the pieces (reads) back together becomes a giant (one-dimensional) jigsaw puzzle known as assembly. If the regions represented by two reads overlap, then the reads' sequences for the common region will be nearly identical. However, two reads could come from regions of the genome that are far apart and still have near-identical end sequences as if they overlapped. This phenomenon is quite common and is due to **repeat** regions of the genome, that is, (nearly) identical sequences occurring at different locations in the genome. If the regions of the genome represented by two reads intersect, this pair of reads is called a **valid overlap**. More generally a pair of reads is called an **overlap** if it is plausible that this pair is a valid overlap based on examination of the reads' sequences. A (plausible) overlap that is not a valid overlap is sometimes called a **spurious overlap**. Overlap pairing, then, consists of creating a set of overlaps, including as many valid overlaps and as few spurious overlaps as possible. These are competing goals. The overlap pairing phase includes all methods based on sequence and quality values used to determine and improve the set of overlaps.

Phase 3, genome assembly. Based on overlap pairing information created in Phase 2, Phase 3 creates the overall large-scale structure and places the reads in that structure [7, 3, 9, 10, 15, 20, 8, 12, 17]. (An alternative approach — not discussed here — that does not use overlap pairing is presented by Pevzner et al. [18] as finding an ideal Euler path through a graph.) The relative positions of the reads are determined (if all goes well) using overlap information (Phase 2), mate information (Phase 1), and possibly information about BACs, such as which chromosome they belong to. We describe here Celera's approach [16, 21] to Phase 3, though others use similar methods.

Phase 3_U, creating unitigs. A unitig is a "contig", i.e., a contiguous collection of plausibly overlapping reads, such that there is a unique way in which these reads can be linearly assembled, consistent with the list of overlaps. Frequently a "consensus sequence" of bases is created for this collection that is an estimate of this part of the genome's sequence. A single unitig can represent a repeat region and its reads could come from all the copies of that repeat.

When the reads of a unitig all come from a single region of the genome, the unitig is called a **U-Unitig**, or just **UU** (for a unitig from a unique part of the genome). Celera estimates the probability that a unitig is a UU from the average spacing between the consecutive reads implied by the read library coverage (see the Unitigger section of [16]).

Phase 3_S, creating scaffolds. Celera next determines which pairs of UUs are linked, that is connected by mated pairs. Since a (small) fraction of the pairs that are denoted mates is incorrect, to declare two UUs linked, Celera requires at least two such pairs. Stringing together collections of UUs into chains using this relative position information results in **scaffolds**.

Phase 3_F, gap filling. Finally mated pair information is used to connect UUs to reads that are not in UUs, through a series of steps, beginning with the most easily and reliably placed, and proceeding to less reliable data, such as reads that are connected to a UU by a single mated pair.

Phase 4, creating a consensus sequence. In this phase, the order of the reads determined in Phase 3 is used together with the read sequences and quality values to create a best estimate of the actual genomic sequence, together with annotations indicating gaps and possible errors.

1.3 Purpose of this Paper

This paper outlines an approach to finding a set of overlaps that can significantly improve WGS assembly of large genomes. We have developed software we call the **UMD Overlapper** that implements this approach. The UMD Overlapper can be used either in place of existing overlap procedures (as done in the tests reported here), or as a preprocessor (correcting reads and quality values) for any self-contained assembler. We note also that it requires only modest computer resources.

Clearly, if one could derive the set of valid overlaps, the task of assembling the reads would be simple. The remaining tasks would be to correct sequencing errors and to sequence gaps in the genome not represented by the reads. But the list of overlaps is never perfect in shotgun sequencing, and in practice the difficulty in assembly depends on the accuracy of the set of overlaps.

The primary goal of our overlapping software is to determine the set of overlaps with greater precision. We use a series of approaches that first creates a set of overlaps and then conservatively discards an overlap only if it is almost certainly spurious. We refined these approaches using a test read library we generated computationally from the assembled *C. elegans* genome (see Section 4.1), and then tested them on actual *Drosophila* reads from Celera. Our techniques provide a number of advantages as described below.

1. Our procedures run quickly. We estimate in Section 6 that if our procedures were used for the human genome, overlap pairing would take about 130 hours (real time using one processor) compared with Celera's 10,000 CPU hours [21]. The utility of this speed is that we can easily introduce additional screening procedures that remove spurious overlaps, thereby making later parts of assembly easier, faster, and more reliable. These iterated error correction routines slow the overlapper by a factor of up to 4. Published procedures require a choice between using very large amounts of RAM and requiring a lot of CPU time to find overlapping pairs for a large genome. (Celera chose the latter, since they had ample distributed computer facilities.) In the former case, the amount of RAM needed grows linearly with genome size, while in the latter case, the computation time grows quadratically with genome size. The UMD Overlapper runs in $N \log N$ time for a genome of size N, with a fixed amount of RAM.

2. We can examine quality values of the reads whenever we examine sequence data without using additional RAM. The use of quality values requires significant additional storage beyond the 2 bits per base required for sequence data. Often the reads are treated as having the same quality value at each base in order to reduce the amount of RAM required. We do not store all of the reads simultaneously in RAM; instead, we sort and process read information sequentially using disk space. Therefore, to use quality values, we require only additional disk space and processing time.

3. By using quality values we can use higher error rates to trim reads, in part because the quality values can help us to better evaluate which reads overlap. Specifically, have found we can truncate reads when the error rate reaches 5% rather than Celera's 2%, thereby allowing the use of almost 50% more of the available sequence data at no additional cost. (This figure is based on the quality values for the Sanger Centre's conservatively trimmed reads from human chromosome 20.) This is potentially a tremendous savings: Celera, for example, has about $100 million worth of sequencing machines, and in addition the machines are quite expensive to run.

4. Those pairs that we judge not to overlap can be used as a reject list. Since we are very conservative when rejecting an overlap, a rejected overlap is almost certainly an invalid overlap. As an illustration, for our *C. elegans* read library this reject list contained 13,100,000 entries, of which only 88 entries were pairs that actually did overlap. Note that there are 5,900,000 correct overlaps, and we found 99.5% of these. In addition, only 6.5% of the pairs we accepted were not valid overlaps. The reject list as described can be used to check a draft assembly to see if rejected

overlaps are implied by the final assembly, which would suggest possible errors.

The methods described here were developed by the authors from the University of Maryland (UMD). The last two authors collaborated in the integration of the UMD algorithms with Celera's assembler and in analyzing the results. In this effort, we used the UMD Overlapper as a preprocessor to the Celera assembler, replacing Celera's overlapper program. As we will describe in detail in Section 4, we applied both Celera's overlap methods and ours to Celera's collection of 3.23 million *Drosophila* reads.[3] We note that Celera's overlapper is tuned to avoid missing overlaps at the cost of adding many spurious overlaps; it generated about 13 times as many overlaps as the UMD Overlapper, and we estimate that it produced about 100 times as many spurious overlaps. When Celera's assembler was applied to each set of overlaps, the UMD overlaps yielded a draft genome with about 6 million more bases than the draft generated with Celera's overlaps. Incidentally, the smaller set of overlaps allowed the Celera assembler to run faster.

2 Our Methods for Overlap Pairing

The most important ideas and programs of our approach are as follows:

Large disks replace RAM. Our procedures, described below, substitute large disk files for large amounts of RAM and yet have computation times that are not dominated by disk access time. To achieve this result, we organize the required information on the disk so that it is always read and used sequentially in large chunks, thus avoiding the time-consuming use of the disk for random access or swap space. Our method also runs in time roughly proportional to the number of reads. See Section 6 for details.

Seed and extend. We call a string of k bases a **k-mer**. Current methods of overlap pairing (including ours) are generally dependent on finding which pairs of reads have in common a short string of bases, sometimes called a **seed**. Our seeds, as is typical, are k-mers, where a fixed value of k is selected in advance. When a pair of reads have a seed in common, the pair is checked in more detail to see if the pair is a (plausible) overlap; that is, an attempt is made to **extend** the match beyond the seed.

Minimizers. Our "method of minimizers" allows us to select from each read a number of special k-mers (to be used as seeds) that we call "minimizers"; we choose them so that about 10% of the possible k-mers in a given read are minimizers, and so that they have the following properties:

(M1) The collection of minimizers cover the entire read, except possibly for a few bases at the ends of the read; and

(M2) If two reads have significant exact overlap, then many of the minimizers chosen from one will also have been chosen for the other.

We ensure property (M1) by choosing at least one k-mer from every possible window of m consecutive (overlapping) k-mers in a read, where $m \leq k$. Furthermore, the k-mer we choose for a given window depends only on the sequence in the window, ensuring that two reads that have sufficient exact overlap will have a minimizer in common. Minimizers are explained in more detail in Section 3.

We use the file sorting procedure described in Section 6 to find all pairs of reads that have a minimizer in common (excluding "repeat" minimizers that occur too frequently in the read library). We call a minimizer **non-repeat** if its total number of occurrences in the read library is

[3]We compare our methods with Celera's because Celera is a leader in this area, has carefully documented their approach, and has allowed us to directly compare our methods with theirs. The reader should keep in mind that Celera's techniques continue to improve over what is in the published record.

below some cutoff r, known as the **minimizer cutoff**. Unless otherwise specified, we ran all test using a minimizer cutoff of 75. Since coverage is typically 15x or less, we exclude only minimizers that almost surely repeat several times in the genome. Table 7 demonstrates the variation in results when the minimizer cutoff is changed.

If two reads X and Y have a minimizer in common, and their offsets or positions (number of bases from beginning of the read) for the minimizer are $o_X > 0$ and $o_Y > 0$, we call $o_R := o_X - o_Y$ the **relative offset of the minimizer** in X and Y. Such a minimizer is evidence that X and Y overlap with relative offset of approximately o_R, e.g. if $o_R > 0$ the first base of Y corresponds roughly to the $(o_R + 1)$st base of X.

1. **MinimizerPairer** is a suite of programs that determines which pairs of reads have a minimizer in common, and prepares the appropriate input data for Extender (see below). Very briefly, MinimizerPairer builds a database of minimizers and their locations in the read library. It sorts this database by minimizer, and then uses the sorted data to build a database of read pairs (with relative offsets) that have a minimizer in common. It then sorts that database by read pair and eliminates redundant entries. This procedure and some benchmark results are described in detail in Section 6.

2. **Extender**. Each pair of reads having at least one non-repeat minimizer in common is checked in detail by a program we call Extender. Thus as with other assemblers, we use a seed and extend approach; the minimizers are the seeds indicating a possible overlap, and Extender tries to extend this match using a relative offset suggested by the positions of the minimizers in the two reads. Extender consists of two routines run in succession, ExtenderA and ExtenderB. While ExtenderB uses the quality values to reject (plausible) overlaps, ExtenderA assumes a fixed error rate, usually the error rate used for trimming the reads as described in Section 1.2. For each prospective overlap, we use a modified Smith-Waterman algorithm [22] to align the two reads, and compute the number of differences D between the two reads in the aligned overlap region. Write $P(E, D)$ for the Poisson probability that D or more events occur when E are expected. Note that $P(E, D)$ is small when we have (many) more differences than expected.

2a. **ExtenderA**. Assume we select 5% for our fixed error rate per base per read for the calculation. Let E_5 denote the number of expected errors in the region of overlap, namely 5% times twice the length of the overlap region. If $P(E_5, D) > 10^{-8}$, the overlap is accepted — unless there is a large cluster of differences near one of the ends of the overlap. In this case, the overlap is rejected if the alignment fails the Poisson calculation above for the end of the overlap region, using only the length of this subregion.

2b. **ExtenderB**. We next use quality values to further reduce the set of overlaps that are accepted by ExtenderA. Based on the quality values in the region of overlap between two reads, we compute the (approximate) expected number of errors E_{qual} as the sum of the probabilities of a mismatch due to sequencing errors for each base in the alignment between the two reads. Then if $P(E_{\text{qual}}, D) < 10^{-8}$ (using the notation above), the overlap fails. Note that throughout the process we only discard overlaps if we are virtually certain they are spurious.

We have found a linear approximation to the Poisson rule that is useful for intuition. For an expected number of errors $1 \leq E \leq 20$, we find that $P(E, 2E+12) \approx 10^{-8}$. Thus, if we eliminate an overlap when $P(E, D) < 10^{-8}$ (our usual cutoff), the criteria to reject is approximately $D \geq 2E+12$. If we assume an expected error rate of 1% per base and two reads have an overlap of length L, then $E = 2 \cdot .01 \cdot L$. In this situation our criterion to reject an overlap is approximately $D \geq .04L + 12$.

By comparison, Celera rejects an overlap if $D > .06L$, i.e., $D > 3$ when $50 \leq L \leq 66$. For a hypothetical read library with a uniform 1% error rate, their rule would reject over 2% of the valid overlaps of length between 40 and 66, according to Poisson statistics. But the probability of missing a valid large overlap ($L > 300$) is smaller than with our rule.

3. **MultiCompare**. Next, we eliminate many overlaps by running MultiCompare, which uses multi-read comparisons. We call it the **rule of 4-3** because it compares 4 reads at 3 bases. To apply this rule to read X we consider: (1) the base differences in the overlap regions of the plausible overlaps which contain X; (2) $P(E_{\text{qual}}, D)$ for the plausible overlaps which contain X; (3) the bases and quality values of X.

We begin with an example of the rule. Suppose read X plausibly overlaps reads Y, Z, and W, and that we find the following pattern over some part of X:

read X:	$\cdots$A$\cdots$C$\cdots$G$\cdots$
read Y:	$\cdots$A$\cdots$C$\cdots$G$\cdots$
read Z:	$\cdots$T$\cdots$A$\cdots$C$\cdots$
read W:	$\cdots$T$\cdots$A$\cdots$C$\cdots$

The dots denote additional bases at which reads X and Y agree. The perfect agreement of X and Y over the pictured range validates this part of their sequences; that is, the agreement gives considerable confidence that at least one part of the genome has that exact sequence. The fact that Z and W disagree with X in exactly the same way suggests very strongly that they do not overlap X. We thus eliminate the X-Z and X-W overlaps. Their pattern of base differences with X, however, may still be used to eliminate other overlaps. (**Warning**: this approach is dangerous if duplicate copies of reads, which are often present in online read libraries, have not been eliminated from the database. If X and Y were copies of the same read, then the agreement between X and Y would not validate anything.)

The example above is somewhat simplified from the actual procedure in two aspects. (1) We disallow a validating overlap (the one between X and Y) that has too small a value of $P(E_{\text{qual}}, D)$. For all of the results in this paper, we set a lower limit of $P(E_{\text{qual}}, D) > 10^{-2}$ for a validating overlap. Our purpose here is to use only the most reliable overlaps for validation in our rule of 4-3; overlaps with $10^{-8} < P(E_{\text{qual}}, D) < 10^{-2}$, while not rejected by ExtenderB, are not considered reliable enough at this stage. (2) The method as described above is not effective in a region where X has errors between the outermost bases used in the comparison, since nothing should match it exactly in this region. For this reason we allow X and Y to have one or two differences in the region. In this case the number of consistent differences (for X versus Z and W) must be at least 3 greater than the number of differences between X and Y in the region.

How exactly to implement an overlap rejection rule like this is a nontrivial problem. Kececioglu and Yu [11] propose to form reads into groups that apparently overlap on a given sequence of base pairs and then partition these groups. So in the example above, if reads X, Y, Z and W had previously formed a group, then X and Y would be split into a separate subgroup from reads Z and W. However, we feel that this partitioning approach is difficult to implement, and the results will depend on the order of operations.

Furthermore, we think that requiring a partition of reads into two or more groups before any overlaps can be rejected is an unnecessary strong condition. We remark that for the purpose of correcting errors in read X, it is important to first eliminate as many spurious overlaps as possible. By looking at the reads aligned with different subsequences of X, we can reject different overlaps; thus we do not partition the reads.

Our approach is to consider all the overlaps associated with a given read X and flag all of the overlaps that we can reject using the approach described above. Further, we do not actually delete the flagged overlaps until we have cycled through all of the reads. Sometimes we only require 2 base differences (**rule of 4-2**) in place of the 3 used in the description above. This is usually done in passes that use base-corrected reads (described below) as input.

4. **Corrector**. This method corrects bases and modifies quality values using the same data as MultiCompare. During the multi-comparison stage, when all the reads that plausibly overlap a given read X are considered together, it is easy to identify many sequencing errors and to use this information to make corrections to the reads. This is carried out after the 4-3 rule has eliminated spurious overlaps. For example, if all the reads that plausibly overlap X at a specified base agree on what that base is (and there are at least 2 such additional reads), we call the base **type 1**. If the corresponding base in X agrees with the other reads, we call it **type 1A** (agree); otherwise, we call it **type 1D** (disagree). An example of a type 1D situation is the following:

```
read X:    .........T...
read Y:       ......G......
read Z:          ...G.........
```

For our *C. elegans* read library (see Section 4.1), we find that the probability is 99.8% that a type 1D base is wrong and should be changed to be in agreement with the bases in the other reads. A simple rule for error correction is to change type 1D bases accordingly and to adjust the quality values of both type 1D and type 1A bases in the manner we describe next.

Let S_X denote the set of all type 1 bases of read X. Note that S_X is determined only by examining the other reads. For bases in S_X, we adjust their quality values (which were based only on read X) to take into account their agreement or disagreement with the other reads. Conceptually, we want to assign type 1A bases a higher probability of being correct and assign type 1D bases (before they are changed) a lower probability of being correct. Furthermore, before we correct the type 1D bases, we do not want to substantially change expected number of incorrect bases in S_X, averaged across all reads X. Based on the data from our *C. elegans* read library, we change the probability that a type 1D base is wrong to 99.8%, and for each type 1A base, we reduce its probability of being wrong by a factor of 100. The latter number keeps the expected number of errors across the read database roughly unchanged.

Now we take the step of actually changing the type 1D bases and, since empirically this change should be correct with probability 99.8%, we assign each of them a probability of 0.2% of being wrong. At this point, we have corrected most of the base errors in the read library, and the new quality values reflect our increased confidence in all type 1 bases.

The preceding description illustrates our methodology for adjusting quality values. By taking quality values into account — not changing high-quality type 1D bases while changing certain other low-quality bases – we were able for our *C. elegans* read library to eliminate 96% of the errors where 3 or more reads overlap, while reducing the proportion of changes that were wrong by a factor of 10. For our *Drosophila* assemblies with Celera, we corrected bases somewhat more aggressively, as described in Section 5.

When using Corrector iteratively, as described below, it is critical that we do not increase the quality value of a base twice based on the same information. One Corrector has changed a quality value, the corresponding base is marked to disallow further changes.

5. **Iterated Correction**. Why do we bother to correct errors if we are interested primarily in overlap pairing? This is done because we use the new reads and quality values in lieu of the originals to run another pass ("Pass II") through the above procedures. When we rerun MinimizerPairer with errors corrected, more pairs of reads have a minimizer in common, and we find over half of those true overlaps that had been missed. Many of the new plausible overlaps are in fact spurious. Next we rerun Extender. As we mentioned earlier, we set a cutoff so that we reject only one valid overlap of every 100 million. But the corrections we made decreased the expected number of differences by about 95% (again, for our *C. elegans* read library). Of course the correction process can be very non-uniform, which is why we need corrected quality values. For an overlap of length

Methods used	1 & 2a	1 & 2a & 6	3 passes
T_{ratio}	97.0%	99.8%	99.5%
F/T	26%	58%	7%
Approximate run time	1 hour	2 hours	12 hours

Table 1: The first two columns of data report results obtained on *C. elegans* faux reads without using quality values. For a given set of plausible overlaps, T_{ratio} means the fraction of all true overlaps of at least 40 bases that are in the set, while F/T means the ratio of spurious (false) overlaps in the set to true overlaps in the set. The numbers of the methods refer to the text of Section 2. In particular, Methods 1 and 2a (MinimizerPairer and ExtenderA) find a set of overlaps, and Method 6 (Symmetrizer) finds other overlaps implied by this set. The parameter values for Method 1 were $k = 20$, $m = 20$, $r = 75$ (20-mers, window size 20, minimizer cutoff = 75). The results in the final column are based on three iterations of our methods, and are described further in Table 2. All timings are for a 1GHz Intel/Linux desktop computer.

200 (the two reads having 400 bases in the overlap) and a mean error rate of 1%, the expected number of errors E_{qual} is 4. Thus, according to the ExtenderB rule that rejects an overlap with D differences if the Poisson probability $P(E_{\text{qual}}, D)$ is less than 10^{-8}, we reject an overlap if $D \geq 20$. On the second pass, after correction, a likely value for E_{qual} is 0.2, and thus Extender rejects an overlap if $D \geq 6$. Hence, Extender becomes much more selective. Continuing the run, we find that since there are fewer spurious overlaps, many of the bases that were not type 1 are now type 1. When we ran through the whole process a third time (Pass III) on our *C. elegans* read library, we wound up with less than 4% as many sequencing errors as at the start.

By way of comparison, Pevzner et al. [18] describe an error correction technique based on the idea of making base changes in reads to decrease carefully the number of 20-mers that occur only once or twice in their set of reads, eliminating 91% of the errors and introducing a new error for every 160 that are corrected (our rate in Pass I on *C. elegans* was one new error for every 4500 correct changes). They also have a technique in which they eliminate an unspecified fraction of the reads that are still troublesome, thereby eliminating about another 6% of the original errors, for an overall 97% reduction of errors in a typical bacterial project.

6. **Symmetrizer**. This routine, which finds missing overlaps, may be run after some or all of the steps above. If read X plausibly overlaps reads Y and Z, and the relative offsets of X with Y and Z suggest that Y and Z overlap, then Y and Z are sent to Extender if the pair has not been checked already. Symmetrizer yields the great majority of overlaps missed by MinimizerPairer. However, many of the overlaps it detects are from heavily repetitive regions, and are in fact spurious (see Table 1). We use Symmetrizer mainly as a diagnostic tool, and we do not use it iteratively. In particular, we did not use Symmetrizer in either UMD+Celera run, described in Section 4.1.

Results with test data. Tables 1–2 summarize the results of experiments using the methods described above on our *C. elegans* read library (see Section 4.1). Table 3 illustrates the impact of our various methods on the set of overlaps. The measures T_{ratio} and F/T that we use to quantify the quality of an overlap set are defined in the caption of Table 1.

Note that while Pass III yielded an improvement over Pass II (Table 2), the impact of using Symmetrizer after Pass II was not nearly as dramatic as it was after Pass I (Table 1). In particular, Pass II found most of the valid overlaps that were missed in Pass I. Using the rule of 4-3 in Pass III allowed us to improve T_{ratio} further without degrading F/T, while using the rule of 4-2 in Pass III significantly reduced F/T without degrading T_{ratio}.

Pass number	I	II	III	III (w/4-2)
T_{ratio}	96.9%	99.4%	99.5%	99.4%
F/T	7.00%	7.00%	6.95%	5.32%
% of original base errors removed	94.4%	95.9%	96.1%	96.1%
valid overlaps rejected	1 per 4400	1 per 48,000	1 per 67,000	1 per 2395

Table 2: Iterated correction: results for *C. elegans* faux reads after each of three consecutive passes:
• Pass I consisted of Methods 1–4 (MinimizerPairer, Extender, MultiCompare, and Corrector). The output of Pass I was an altered set of read sequences and quality values in which corrections have been made.
• Pass II ran the same routines on the altered data set, yielding sequences and quality values that are further improved, together with a set of overlaps. Pass III used this dataset.
• Pass III began with Method 6 (Symmetrizer), using the overlaps from Pass II, followed by Methods 2–4 (Extender, MultiCompare, and Corrector). Some valid overlaps are never found, but of the nearly 6 million valid overlaps that were, only 88 were rejected at the end of Pass III. Of the original base errors, 2% are in regions of 1x or 2x coverage, and we did not correct any of these. Overall, Corrector changed 1 base incorrectly for every 3300 correct changes.
• The last column shows the results when Pass III used the rule of 4-2 instead of 4-3.

Statistic	MinimizerPairer $k = m = 20, r = 75$	ExtenderA 5% error rate	ExtenderB quality values	MultiCompare (4-3)
F/T	230.1%	50.4%	13.4%	7.0%
# valid overlaps rejected	—	0	2	122

Table 3: Breakdown of the impact of the various methods. The columns show the overlap statistics after each phase of Pass II in the *C. elegans* run described in Table 2. The total number of valid overlaps for our read library is 5.9 million. Thus, T_{ratio} remained essentially constant at the 99.4% reported in the previous table.

3 The "Method of Minimizers"

Here we elaborate on the "method of minimizers", a novel method employed to greatly reduce the amount of time and storage needed for overlap pairing. Section 2 touched on some properties of minimizers (which we called M1 and M2). As a first step, we select an ordering for the set of all k-mers of fixed length k.

More precisely, if S is a string of $m+k-1$ bases, then it contains exactly m consecutive k-mers, where "consecutive" means each k-mer is shifted by one base from the previous one. (The "first" k-mer of S begins with the first base, and the jth k-mer begins with the jth base.) To find a "minimizer", we examine m consecutive k-mers and select the smallest, in the sense of our chosen ordering. We refer to m as the **window size**. If M is the smallest k-mer in S, then we say M is the **simple minimizer** for S. ("Simple" denotes the fact that we do not examine the k-mers of the reverse complement of S.)

For a string S or a read X, we denote its reverse complement by $-S$ or $-X$. An **(absolute) minimizer** for S is a k-mer that is the smallest of all the k-mers in both S and $-S$. By definition, S and $-S$ have the same minimizer.

Finally, we define the (m, k)**-minimizers** of a read X to be those k-mers in X that are absolute

minimizers for some substring S of X with length $m + k - 1$. In nontechnical discussion we will often call a (m, k)-minimizer simply a **minimizer**. The following assertions are true of minimizers:

- If two reads have an error-free overlap of at least $m + k - 1$ bases, then they must have an (m, k)-minimizer in common.
- A read has the same minimizers as its reverse complement.

We say the **offset of a k-mer** M in a read is n if the first base of M is the n-th base of the read. Thus:

- Two consecutive (m, k)-minimizers in a read have offsets differing by at most m.

Hence if $m \leq k$, then Property M1 (from Section 2) holds. That is, the collection of (m, k)-minimizers cover the entire read, except for at most $m - 1$ bases at each end. Property M2 also holds, in that two reads with error-free overlap of many more than $m+k-1$ bases will have several (m, k)-minimizers in common.

Example. Suppose we are using 20-mers and the window size is 20. If X is a read of length 400, it must have at least 19 minimizers. The first minimizer has offset at most 20, the second at most 40, etc. Similarly, if two reads have 400 bases of error-free overlap, then they must have at least 19 minimizers in common.

There are a number of orderings that can be used for defining the minimizers of a read; below is one example. What is important is that the same ordering should be used for all of the reads.

Example. For each position in a k-mer, assign a different digit (0, 1, 2, 3) to each of the 4 bases `A`, `C`, `G`, `T`. Then, thinking of k-mers as k-digit numbers, order the k-mers numerically.

For the results reported in this paper, we assign the values 0, 1, 2, 3 to `C`, `A`, `T`, `G` respectively for the odd numbered bases of k-mers and assign 0, 1, 2, 3 to `G`, `T`, `A`, `C` respectively for the even numbered bases. The purpose of varying the base values by position is to avoid having minimizers that start with strings like `AAAAAAAA` or `CCCCCCCC` that are relatively prone to sequencing errors.

We call the number of times a minimizer occurs in the read library the **minimizer count**.[4] As described in Section 2, when computing which pairs of reads have a minimizer in common, we ignore minimizers with a count of more than r. The effect of the minimizer cutoff along with the window size is explored in Section 6.

4 Results

4.1 Read Libraries Used

Drosophila data set. The results in [16] are based on a set of 3.23 million reads (with mated pair information) for *Drosophila melanogaster* generated by the Berkeley Drosophila Genome Project (BDGP) and Celera. These reads yield a 13x coverage of the genome when the reads are trimmed to limit the average predicted error rate to about 2% per base near the end, where the error rate is typically highest. About 3/4 of the reads have mates. We discuss four assemblies of these reads.

- "Celera2000" – denotes the assembly of these reads called "WGS" in [16].
- "Celera2002" – denotes a more recent reassembly using improved techniques, including base-error correction of the reads. We discuss Celera's results with the permission of Celera.

[4]The count is the number of times that the k-mer occurs *as a minimizer*. However, typically a k-mer that is a minimizer for one read will be a minimizer for the other reads in which it occurs. Thus, the minimizer count is generally close to the total number of times the k-mer occurs in the read library.

Assembly	Scaffolds ≥ 100 Kbp (#gaps)	All Scaffolds [#scaffolds]	#overlaps
Celera2000	26 with 114.0 Mbp (1887)	119.7 Mbp [2483]	212 million
Celera2002	45 with 122.7 Mbp (1425)	133.6 Mbp [3293]	502 million
UMD+Celera#1	52 with 125.6 Mbp (1481)	139.9 Mbp [4439]	38 million
UMD+Celera#2	53 with 125.7 Mbp (1646)	140.9 Mbp [4926]	36 million

Table 4: The four *Drosophila* assemblies described in Sections 4 and 5, using the same set of trimmed reads. The lengths given exclude gaps in scaffolds.

- "UMD+Celera#1 and #2" – denote two reassemblies using our overlaps and base-error corrections in place of Celera's along with their other assembly techniques from Celera2002. The second used the results of the first assembly.

C. elegans faux reads. The *C. elegans* results in this paper were obtained using a library of faux reads we created giving 6x coverage of the genome. To make their error content as much like actual reads as possible, we used quality value data for actual reads of human chromosome 20 from the Sanger Centre [14]. We created *C. elegans* reads having these quality values, trimmed so that the expected error rate at each end of a read does not exceed 5%. We call these **real quality faux reads**. A very similar approach has been reported in [1]. The resulting database contained 1,065,846 reads with an average length of 537 bases. The overall error rate for bases was 0.86%. We report results using faux reads because with actual reads it is impossible to be certain where the errors are and which reads overlap. (Faux reads also have some problems, which are described in some detail in [1].) Because there are regions of low quality values, clusters of errors occur as in true reads. However, actual reads can be worse, as when they are "chimeric", including bases from two different parts of the genome or including irrelevant vector sequence data due to inadequate trimming.

4.2 Results of Drosophila Assemblies

The numbers in Tables 4 and 5 are Celera's best estimates of how much *Drosophila* genome sequence was found by each assembly. Celera's more recent assembly Celera2002 yielded almost 14 million more base pairs than the Celera2000 assembly. Tables 4–6 show that the UMD+Celera assemblies yielded significantly more base pairs in scaffolds than Celera2002, at the expense of a slight increase in the rate of "unhappy" mated pairs. In particular, Table 4 shows that UMD+Celera#1 assembly yielded an extra 6.3 Mbp (million base pairs) compared to Celera2002 while using 1/13 as many overlaps. UMD+Celera#2 yielded an extra 1.0 Mbp compared to UMD+Celera#1. The UMD+Celera assemblies have more large scaffolds (an increase from 45 to 52 or 53), while adding more than 1000 smaller scaffolds as well.

Table 5 shows that most of the increase in the size of the draft genome seen in Table 4 came about as a result of identifying more of the "simple" parts of the genomic sequence, namely, the U-Unitigs. Whether a region is a U-Unitigs depends on the set of overlaps, and the UMD overlapper reduces the spurious overlaps by a factor of roughly 100. If two reads from different parts of the genome have similar sequences, then the reads might falsely appear to overlap. Such an error can prevent both regions from being contained in U-Unitigs.

For UMD+Celera#1, we used three passes of our software. We made about 1.9 base changes per read. In particular, 5,858,000 changes were made by Pass I, 113,000 by Pass II, and 69,000 by Pass III. Note that the corrected reads given (along with the overlaps from Pass III) to the

Assembly	U-Unitigs	% of all reads
Celera2000	110.6 Mbp	N/A
Celera2002	121.4 Mbp	77%
UMD+Celera#1	127.0 Mbp	79%
UMD+Celera#2	130.3 Mbp	80%

Table 5: Recall that a U-Unitig is a collection of reads that can be uniquely assembled, and almost certainly come from a unique region in the genome (based on having an "A-statistic" of at least 5; see the Unitigger section of [16]). The number listed under U-Unitigs" is approximately the number of bases in the U-Unitigs, but is a slight underestimate. More precisely, it is the sum of the "overhangs"; Celera calls the number of bases between consecutive read starts the "overhang". The right-most column is the ratio of the number of reads placed in U-Unitigs to the number of reads in the database, expressed as a percentage. For UMD+Celera#1 there were about 16,000 U-Unitigs, and there were 135 million bases in all the overhangs in all the unitigs.

	Both mates in same scaffold			Both mates in same contig		
Assembly	Happy	Unhappy	U/H	Happy	Unhappy	U/H
Celera2002	887,758	3406	.0038	855,297	2136	.0025
UMD+Celera#1	910,558	4099	.0045	878,211	2464	.0028
UMD+Celera#2	924,214	4325	.0047	888,101	2397	.0027

Table 6: Numbers of happy and unhappy mated pairs, and their ratio, in each assembly (happiness data for Celera2000 was not available). "Happy" mated pairs are those whose read positions and orientations in the final assembly are consistent with the estimated distance (based on insert length) between the two mates. "Unhappy" mated pairs are those for which the distance or orientation in the assembly is inconsistent with their insert. The mated pairs that are in the same contig are a subset of the mated pairs that are in the same scaffold.

Celera assembler in UMD+Celera#1 only used the base changes made in the first 2 passes. For UMD+Celera#2, we modified the set of overlaps from UMD+Celera#1 using the scaffold information from the UMD+Celera#1 assembly together with mated pair information.

The methods we used in preparing overlaps and base-corrected reads for the Celera assembler are described further in Sections 5 and 6. In particular, Section 5 discusses the specific procedures and parameter choices we used for the runs at Celera, and the additional techniques we used for UMD+Celera#2.

4.3 Results with Faux Reads

Since it is difficult to compare the "correctness" of the assemblies described in the previous section, we now describe results on test data for which we know which reads actually overlap. Recall that we evaluate a list of plausible overlaps using two numbers:

T_{ratio} = fraction of all true overlaps that are in the list
F/T = false/true = ratio of spurious overlaps in the list to true overlaps in the list

Unless otherwise specified, these definitions concern only overlaps of at least 40 bases.

Our methods applied to the *C. elegans* faux read library described above yield an F/T of less than 7% spurious overlaps with $T_{\text{ratio}} = 99.5\%$. (See Tables 1–3 in Section 2 for more details.) This

result is obtained even though we have used reads with a higher error rate than Celera allowed. We now describe some of the highlights of our results with *C. elegans* faux reads.

1. **Our methods found virtually all of the overlaps relatively quickly on a personal computer.** In fact, we obtained our initial results, given in the first two columns of Table 1 of Section 2, using an 866MHz Intel/Linux laptop (with run times about 50% longer than the desktop computer timings reported in the table). We remark also that our parameter values were not chosen to maximize T_{ratio}, and that we can obtain still higher values (over 99.99% for *C. elegans*) but with a greater run time and higher F/T (see Table 7 in Section 6).

2. **We can trim reads at a high error rate.** We truncated reads when the error rate reached 5%, rather than Celera's cutoff of 2%, thereby allowing the use of almost 50% more of the available sequence data at no additional cost. (This figure is based on the quality values for the Sanger Centre's conservatively trimmed reads from human chromosome 20.) This is a tremendous potential savings: Celera for example has about $100 million worth of sequencing machines, and in addition the machines are also quite expensive to run.

3. **We create a reject list, a list of read pairs that have a significant amount in common but we judge not to overlap.** For our *C. elegans* read library, this reject list contains 13,100,000 entries, of which only 88 entries are pairs that actually do overlap. Note that there are 5,900,000 correct overlaps. This list may be used when assembling the genome to indicate that some part of the assembly is incorrect. This would occur when the assembly implies an overlap that has been found to be invalid.

4. **We correct most of the errors in bases to improve overlap determination.** Since we know the *C. elegans* genome sequence and we created the faux reads as described above, we know that there are a total of 4,846,000 errors in the read library. After the first pass of our software, the base-corrected reads had only 272,000 errors. After the second pass, there remained 200,000 errors, which is only 4.1% as many errors as at the start. Many of the errors (80,000) are in regions of 1x and 2x coverage, where we do not do error correction. (See "Iterated correction" in Section 2.)

5 Procedures Used in UMD+Celera#1 and UMD+Celera#2

We now discuss in more detail how we used our methods to produce the overlaps and base-corrected reads that, when used with the Celera assembler, produced the results in Tables 4–6 for *Drosophila*. While we are in principle able to use reads that are longer, we used the same set of trimmed *Drosophila* reads in UMD+Celera#1 and UMD+Celera#2 as Celera did in Celera2002, in order to better evaluate the impact that more carefully selected overlaps made in the final assembly.

In both their 2000 and 2002 assemblies, Celera blocked out heavily repetitive regions; 20-mers occurring more than roughly $r = 500$ times were not used to find plausible overlaps. Celera2002 also used error correction methods developed by Celera. The UMD+Celera assemblies blocked repeat regions much more aggressively, using a minimizer cutoff of $r = 75$. UMD+Celera#2 used some scaffold information created in UMD+Celera#1, together with mated pair information, as described below.

UMD+Celera#1 allowed plausible overlaps as short as 20 bases, but some of the short overlaps seemed to cause mis-assemblies. Thus, UMD+Celera#2 required (like Celera2000 and Celera2002) a 40-base minimum overlap, eliminating 525,000 overlaps from UMD+Celera#1. Of the 38,000,000 overlaps used in UMD+Celera#1, about 1,000,000 had lengths at least 40 and did not appear in the Celera2002 list of 512,000,000 overlaps.

The software modules and nomenclature that follow were described in Section 2.

5.1 Parameters Common to Both Runs

MinimizerPairer used 20-mers with a window size (m) of 20 and a minimizer cutoff (r) of 75. ExtenderA assumed a fixed error rate of 5% per base per read with an elimination criterion of $P(E_5, D) < 10^{-8}$. ExtenderB used an elimination criterion of $P(E_{\text{qual}}, D) < 10^{-8}$. MultiCompare used the rule of 4-3 (rather than 4-2) for overlap elimination. Only overlaps that satisfied $P(E_{\text{qual}}, D) > 10^{-2}$ were used for region validation when applying the rule of 4-3.

5.2 Overlap Procedures Used in UMD+Celera#1

We ran three passes, each using Methods 1–4 (MinimizerPairer, Extender, MultiCompare, and Corrector) on Celera's *Drosophila* read library. Parameters for MinimizerPairer, Extender, and MultiCompare were specified above.

Corrector. The plausible overlaps used for error correction were limited to those that passed the Poisson test $P(E_{\text{qual}}, D) > 10^{-2}$. In addition, a different rule was used to change bases than that described in Section 2. We designed the original rule assuming 6x coverage. We made the change due to the high coverage of the *Drosophila* genome (13x to 15x). (In tests we have run since, we have found that the rule from Section 2 is effective for high coverage, but that typically one more pass is necessary.)

Suppose that we are analyzing a base b_X in read X which has quality value q_X to see if the base or quality value should be changed. The rules we used for making base and quality changes break down into the following cases:

- **The overlapping reads all agree with the base**. (In Section 2 we called this a type 1A base.) If all the plausible overlaps agree with b_X then we add 20 to q_X, i.e., change the quality value so that the probability of error is 100 times smaller. (But if the value of q_X has been changed in prior passes, then its value is left unchanged.)

- **Some overlapping reads disagree with the base**. This case includes the type 1D bases defined in Section 2, but there are additional cases in which we make a base change. The rule we used in these runs breaks down as follows:

  - **Low quality case**. This is the case where $q_X \leq 20$ (i.e., the probability of the base b_X being wrong is at least 10^{-2}). In this case, the base is changed (via a substitution, insertion, or deletion) when a majority of the overlapping reads (including X) agree on the change.
  - **High quality case**. This is the case where $q_X > 20$. To change the base in this situation, the change must satisfy two criteria. The first is that a majority of the overlapping reads (including X) agree on the change, as in the low quality case. The second criterion is that at most three overlaps with read X disagree with the change. (This criterion works well with 13x coverage, but not with 6x.)

  When a base is changed, we give the revised base (if it isn't a deletion) a quality value of 21. In the case of an insertion, we leave the quality value of the surrounding bases unchanged.

  When some overlapping reads disagree with base b_X but the number or pattern of disagreements won't allow us to change it, we leave the quality q_X unchanged as well.

Celera's assembler was given the corrected reads from Pass II as well as the overlaps from Pass III. Since the base errors are corrected at the end of a pass, the overlaps constructed in Pass III were

computed for reads incorporating corrections made only in Passes I and II. Base error corrections determined in Pass III were used solely as an indicator as to how many more errors might be left.

Throwing out a few questionable overlaps. With about 13x coverage, we have a generous collection of reads and the quality of our data is more important than maintaining all of the reads and overlaps. Therefore we made the following deletions.

1. All reads that in Pass I had more than 5% changes (4200 out of 3,230,000 reads) were eliminated from further consideration, i.e., they were allowed no overlaps.

2. All overlaps that had more than 6% differences after processing were eliminated. (Altogether 82,000 overlaps were eliminated, including the overlaps that the eliminated reads would have had).

Note that discarding an occasional read and all of its overlaps is unlikely to cause problems, whereas eliminating an overlap can cause problems — it can break what would be a contig into two pieces, creating a "gap" between contigs whose length is actually negative.

5.3 Overlap Procedures Used in UMD+Celera#2

Some reads were not placed by the Celera assembler in the draft genome of UMD+Celera#1. This can happen if, for example, the read X is in a repeat region and there are too many overlaps for it to be placed reliably. But when such a read has a mate and the mate is reliably positioned, that is, it lies in a U-unitig (or UU for short), we can infer an approximate position for X. Almost 3% of the reads met this criterion. For each such read X, we checked to see if X has any overlaps that are consistent with the inferred position, and we discarded any of the overlaps from UMD+Celera#1 that were inconsistent with it, in the manner described below. We submitted the altered set of overlaps to the assembler to obtain UMD+Celera#2.

The set of overlaps that were submitted to the Celera assembler in UMD+Celera#1 will be called Overlaps1, and here we describe how we constructed Overlaps2, the set for UMD+Celera#2. We refer to the draft genome produced by UMD+Celera#1 as Assembly1, which includes information as to where each read is placed in the final assembly, if it is indeed placed (many were not). Our strategy to create Overlaps2 was to both add to and subtract from Overlaps1 based on information from Assembly1, and then to winnow the resulting overlaps in our usual manner.

We classified each of the original 3.23 million reads according to how it was placed in Assembly1 as follows:

- Class 0: the read was not placed in a UU nor did it have a mate placed in a UU (444,200 reads);
- Class 1: The read was placed in a UU (2,720,000 reads);
- Class 2: The read was not placed in a UU but its mate was. (62,500 reads).

Note that Classes 0 and 1 include some reads with no mates. The goal was to try to get many Class 2 reads positioned in UMD+Celera#2 based on the position of their mates, and to that purpose we modified the set of overlaps.

To create Overlaps2, we began by assigning each Class 2 read an inferred position (with a standard deviation) in the same scaffold as its mate, based on the position of the mate. (Note that we did not have available information as to where the scaffolds were located relative to each other. If that information had been available, we would have allowed overlaps of the read with consistently positioned reads in an adjacent scaffold. Had we done that, some of the scaffolds might have been merged. Without that information, we can have two Class 2 reads that actually overlap, but we miss that fact due to their being mated with Class 1 reads in adjacent scaffolds.)

We then ran MinimizerPairer; if the minimizer count was no bigger than 75, then the overlaps would have been checked in the first run. Whenever the minimizer count was over 75, we looked for possible overlaps of Class 2 reads with other reads in the same scaffold with consistent positions. We then ran ExtenderA on the new overlaps, adding those that passed to Overlaps1.

From the combined list of overlaps, we eliminated each overlap of a Class 2 read X with a read Y if Y: (1) was in a different scaffold than X; (2) was in the same scaffold as X but had a position or orientation inconsistent with the X-Y overlap; or (3) belonged to Class 0 (not assigned to a scaffold). Overall, we eliminated about 1.2 million overlaps in this procedure.

Then we ran ExtenderB and MultiCompare with the rule of 4-3, and the overlaps that were still acceptable were collectively called Overlaps2. We then ran Corrector, and found that 46,800 base changes were recommended. However, we did not use these changes in UMD+Celera#2, since by the procedure we used in UMD+Celera#1, that would have required another pass to find the set of overlaps most consistent with the altered reads.

We then ran the Celera assembler to yield Assembly2. In the end, we gained about 1 Mbp of draft genome, or about 16 bases per Class 2 read.

Remark. For the 62,500 reads in Class 2, we investigated how many overlaps they had in Overlaps1. We categorized these reads as follows (the second and third categories are not disjoint):

- 11,800 had no overlaps (these are often in heavily repetitive regions);
- 32,600 had an overlap with a Class 1 read, and 8400 of these overlap only Class 1 reads;
- 42,300 had an overlap with a Class 2 read, and 18,100 of these overlap only Class 2 reads.

6 Sorting Large Files

In this section we illustrate, using test data, the effect of varying the minimizer parameters m (window size) and r (minimizer cutoff), defined in Section 3. In addition, we describe how our software uses file sorting techniques to process large read libraries with only a modest amount of RAM. In particular, we estimate the time needed for the initial stages of overlap pairing on a human-size genome.

6.1 Parameter Dependence of Results

We have tested the suite of MinimizerPairer, ExtenderA, and Symmetrizer extensively on our *C. elegans* faux read library. These routines are described in Section 2 (methods 1, 2a, and 6) and do not use quality values. We present the results as an example of how overlap pairing time and quality can depend on the parameters of our method of minimizers, namely the window length m and the minimizer cutoff r.

Table 7 shows results using k-mer length $k = 20$ and various values of the window size m and minimizer cutoff r. Results were quite similar for values of k from 20 to 30. Timings are total run time (including disk access) on a dual processor 1 GHz Intel-Linux desktop computer with 1 GB of RAM and an external 70 GB SCSI disk. (We did not parallelize our software for these runs; we used a dual processor machine only because our single processor laptop did not have sufficient disk space to perform all the runs.)

m	$r = 25$	$r = 75$	$r = 250$	$r = 1000$	$r = \infty$	
	99.685%	99.889%	99.974%	99.993%	99.993%	T_{ratio}
1	0.45	0.88	1.37	1.97	2.61	F/T
	4:45	7:39	11:37	20:38	35:58	Run time
	99.666%	99.882%	99.971%	99.991%	99.992%	T_{ratio}
3	0.40	0.83	1.29	1.85	2.41	F/T
	2:59	5:02	9:24	16:17	27:48	Run time
	99.610%	99.860%	99.961%	99.986%	99.987%	T_{ratio}
8	0.34	0.73	1.17	1.65	2.06	F/T
	1:50	3:19	6:34	13:31	18:45	Run time
	99.472%	99.804%	99.934%	99.972%	99.973%	T_{ratio}
20	0.24	0.58	0.99	1.39	1.66	F/T
	1:12	2:09	4:15	7:45	12:26	Run time

Table 7: Results of running MinimizerPairer, ExtenderA, and Symmetrizer (Methods 1, 2a, and 6 of Section 2) on our *C. elegans* read library using 20-mers ($k = 20$). To compute T_{ratio}, we considered overlaps of 20 bases or more. Run time is in hours and minutes. Note that m denotes the window size and r is the minimizer cutoff, and that $m = 1$ and $r = \infty$ corresponds to using all 20-mers.

6.2 Timings and Data Organization

For the procedures described below, our approach substitutes large disk files for large RAM and yet has computation time that is not dominated by disk access time. In addition, our method runs in time roughly proportional to the number of reads.

Many people feel that getting large amounts of data from a disk is prohibitively slow. This depends on whether the disk access is sequential or random. For example, suppose the seek time for a disk is 0.003 seconds, (i.e., extremely fast), and a 1 GB file can be read in one minute (the speed of the standard disk on our laptop). Reading a 200 MB file containing 1 million records would take about 12 seconds, while if the desired 1 million records were scattered across the disk, it would take as much as 50 minutes (i.e., $10^6 * 0.003$ seconds) to read them.

To avoid using random disk access we organize the required information on the disk so that it is always read and used sequentially. Data that is too big to fit into RAM is sorted into large files so that each file is accessed only a small number of times, usually once. When the file is read, its data is in an order in which it can be processed immediately for the next step. Disk access therefore represents a minority of the run time.

We note that ARACHNE [1], the Whitehead Institute's assembler, uses some sorting of 24-mers and their locations (similar to our first sorting step) via large disk files.

To begin our analysis of the time necessary to run a large genome, recall that our *C. elegans* read library has about 1.1 million reads with average length 540, giving 6x coverage of the roughly 100 million known bases. We now describe our algorithm and use our results from the *C. elegans* read library to project the time required to run MinimizerPairer and ExtenderA on a hypothetical genome having 3 billion bases. This example is modeled on the human genome. We assume we have 40 million reads averaging 600 bases in length, providing us with 8x coverage.

The reader can view our construction as examples of how to obtain organized data by sorting files. With each step the information available becomes richer.

Procedure for sorting huge files. We sort a huge file F as follows. First, read F and write each record to one of several "bucket" files, each of which is small enough to fit in the computer's memory. The records are placed so that all records in bucket file 1 are smaller than all records in bucket file 2, etc. Then read each bucket file and sort it in RAM. (We use the standard UNIX `qsort` routine for sorting records in RAM.) At this point the sorted records are generally immediately used or transformed into another form before writing to disk; hence the fully sorted file F need never exist. Nonetheless, speaking of the organized data as a "file" is a useful conceptual device. If F is not going to be used again, the files from which it is made are deleted. Read sequences are stored 4 bases per byte. If we use quality values, we can store each base and its quality value in a byte, using 2 bits for the base and 6 bits for the quality value.

For our *C. elegans* run with parameters $m = 20$, $k = 20$, and $r = 75$, we needed less than 5 GB disk space. For our hypothetical large genome, the largest intermediate "file" created would be around 300 GB. However, our software has the capability to create and process this intermediate data in batches, for instance breaking the 300 GB file into six 50 GB batches. In this way we could do the entire computation on a single disk drive without increasing the computation time significantly.

Time estimates. The estimated timings below are for an 866 MHz Intel/Linux laptop computer with 384 MB of RAM and a standard internal 40 GB disk drive. Sorting a huge file requires time for operations in RAM and time for disk access. Sorting, excluding disk access, requires about 1 minute per 100 MB when the size of the record is about 10 bytes. Reading and writing huge files each require about 1 minute per GB.

Our file names S, ROM, etc. reflect the initials of the contents of their records. If the file is sorted by one of its fields, the field is indicated by the first letter of the name. Hence "ROM" used below is sorted by the R field, that is, read number.

The file S. We begin with file S, which contains the read sequences. The reads are numbered consecutively, 1, 2, ... For our *C. elegans* run, this file was about 150 MB; for our hypothetical large genome:

S file size = Average record size $*$ #reads $\approx (600/4) * 40\text{M} = 6$ GB

The files ROM **and** MRO. Read in the read sequences in S one at a time. Determine which k-mers are minimizers. Create a file ROM whose records have the form (minimizer, offset, read number). We call this file ROM because initially it is sorted primarily by read number and secondarily by offset; the **offset** tells the position of the minimizer in the read. We allocate 5 bytes per minimizer, 4 per read number, and 2 per offset. Recall that a minimizer can be a minimum for a substring of a read or for its reverse complement. We indicate which with an extra bit in the record, resulting in a 12-byte record.

Sort ROM by minimizer yielding MRO; (ROM is no longer needed so it is deleted). Since each minimizer's records are together in MRO, it is simple to count the number of records for each minimizer, and this can be carried out as the file is created. If the number of occurrences of a minimizer in the read library exceeds the minimizer cutoff (r), all the records in MRO with that minimizer are flagged. The minimizer cutoff has an effect on the results and the run time, but the dependence has to be determined by experiment.

For our *C. elegans* run, the ROM file size was 560 MB and took 3 minutes to create from S. Sorting ROM to create MRO took 5 minutes. As mentioned above, these times include disk access as well as CPU time. For our hypothetical large genome:

ROM file size = Record size $*$ genome length $*$ coverage / 10

$\approx$ 12 bytes $*$ 3B $*$ 8 / 10 $\approx$ 29 GB
Time to create and sort $ROM \approx$ 5 minutes $*$ 29 / 0.56 $\approx$ **4.5 hours**

The files ORR **and** RRO. We use the file MRO (which is sorted by minimizer) to create a file of read pairs having the same minimizer. Only unflagged (non-repeat) minimizers are used when creating the pairs. All the records with a given minimizer are ordered by read number.

Read MRO; for each pair of records having the same unflagged minimizer, (minimizer, offset1, readnumber1), (minimizer, offset2, readnumber2), create a record for a new file ORR, with records, (offset2–offset1, read number2, readnumber1) where readnumber2 > readnumber1. We can also create a separate file, $MRO_{repeats}$, consisting of the flagged records in MRO. It can be used for later analysis of repeats.

Sort ORR by readnumber2, then readnumber1, producing file RRO, in which all records with the same two read numbers are grouped together. Then replace each group of records for a given read pair with a single record containing the range of relative offsets found; call the resulting file $RROO$. Note that it is not necessary to actually write RRO to disk.

For our *C. elegans* run, the ORR file size was 1.5 GB and the $RROO$ file size was 270 MB. Creating ORR from MRO, sorting ORR to RRO, and creating $RROO$ took a total of 12 minutes. Only about 1/3 of the read pairs in $RROO$ represented actual overlaps in the genome, so to estimate the size of ORR and $RROO$ for our hypothetical large genome we multiply the approximate number of actual overlaps by 3. (Of course for a different genome this ratio would be different, but we can control it by adjusting the cutoff we use for repeat minimizers.) Thus for the large genome:

ORR file size $\approx$ 3 $*$ Record size $*$ genome length $*$ (coverage$*$(coverage – 1)/2) / 10
$\approx$ 3 $*$ 12 bytes $*$ 3B $*$ 28 / 10 $\approx$ 300 GB
Time create and sort $ORR \approx$ 12 minutes $*$ 300 / 1.5 $\approx$ **40 hours**

Where do we keep such large disk files? If this virtual file was conceptually divided into 6 parts, each part could be created, used and disposed of before going on to the next part so that much less disk space would be required. The cost would be that MRO would have to be read 6 times, requiring perhaps an extra **2.5 hours**.

The maximum amount of disk space needed for the entire procedure is determined at this stage; it is the sum of the size of S, the size of MRO, the amount of space allocated for sorting, and the size of $RROO$:

$RROO$ file size $\approx$ 3 $*$ Record size $*$ #reads $*$ coverage
$\approx$ 3 $*$ 16 bytes $*$ 40M $*$ 8 $\approx$ 16 GB

Thus the disk space required is about 51 GB plus the space allocated for sorting; with a 100 GB disk drive we could use nearly 50 GB for sorting, or we could of course use a second disk drive.

Once $RROO$ is created, MRO can be deleted, freeing disk space for future files. $RROO$ is small enough that we can save it for later use.

The files $R_2R_1OS_2$ **and** $R_1R_2OS_2$. Read $RROO$, which is sorted by readnumber2. Simultaneously read the file S of sequences and create the file $R_2R_1OS_2$ with records (readnumber2, readnumber1, offset range, sequence2), where sequence2 is the sequence data for read 2.

Sort $R_2R_1OS_2$ by readnumber1, yielding $R_1R_2OS_2$. Since 4 bases can be stored in one byte, if we limit reads to say 800 bases, then each read's sequence can be stored in 200 bytes. For our *C. elegans* run, the $RROS$ file length was 3.7 GB. For our hypothetical large genome:

$R_1R_2OS_2$ file size $\approx 3 *$ Record size $*$ #reads $*$ coverage
$\approx 3 * 220 *$ 40M $* 8 \approx 210$ GB

If we included quality values, the file would be about 4 times larger. This would increase the run time of this sorting step substantially, but based on our experiments with the *C. elegans* read library, it increases the total run time by only about 50%. Using quality values do not require additional disk space because as before, the data can be created in (say) 50 GB parts, with each part used and disposed of before the next is created. The penalty this time is that *RROO* must be read multiple times; again this does change the total run time much.

Run ExtenderA. Read $R_1R_2OS_2$ and simultaneously read in the file S of sequences so that we obtain the sequence of read 1 as well as the sequence of read 2. ExtenderA checks the sequences and either rejects the read pair, or accepts it and reports offset information and a quality value for the pair. For our *C. elegans* run, the total time taken to create $R_2R_1OS_2$, sort it, and send the results to ExtenderA was 90 minutes. Note that it was not necessary to actually write the sorted file $R_1R_2OS_2$ to disk. Most of the time is spent in ExtenderA, and thus the time spent is essentially proportional to the number N of $R_1R_2OS_2$ records. Though the time spent sorting is proportional to $N \log N$, the factor by which $\log N$ increases in going from one large database to another is close to 1, so even the sorting time scales nearly linearly for large databases. Thus for our hypothetical large genome:

Time from *RROO* through ExtenderA $\approx$ 90 minutes $*$ 210/3.7 $\approx$ **85 hours**
Total time for initial overlap pairing $\approx$ **132 hours**

This estimate represents the time needed to produce a set of overlaps that would contain nearly all of the valid overlaps but would still have a sizeable number of spurious overlaps. Our remaining procedures for winnowing overlaps run in a fraction of the time taken up through ExtenderA, so that the total time needed for one pass of our software on a single CPU would be about 1 week. Furthermore, ExtenderA and all subsequent procedures are easily parallelized. Each pass of our iterated procedure requires essentially the same amount of time and resources.

7 Acknowledgments

The first three authors thank Gene Myers and Granger Sutton for the opportunity to test the UMD overlap software at Celera. We also thank Clark Mobarry, Aaron Halpern, Ian Dew, and Karin Remington for their generous assistance. Finally, we thank Wayne Hayes, Granger Sutton, and the referee for their comments on the paper. This research was supported by the National Science Foundation (Grant # 0104087).

References

[1] S. Batzoglou et al. "ARACHNE: A Whole Genome Shotgun Assembler" Genome Research 12, 177-189 (2002).

[2] F. R. Blattner et al. "The Complete Genome Sequence of E. Coli", Science 277, 1453–1474 (1997).

[3] J. K. Bonfield, K. Smith, R. Staden "A new DNA sequence assembly program", Nucl. Acid Res. 24, 4992–4999 (1995).

[4] The Genome Sequencing Consortium, "Genome Sequence of the Nematode C. elegans: A Platform for Investigating Biology", Science 282, 2012–2021 (1998).

[5] P. Green, "Against a whole-genome shotgun", Genome Res. 7, 410–417 (1997).

[6] P. Green and B. Ewing, `http://www.phrap.org/phrap.docs/phred.html`.

[7] X. Huang, "A Contig Assembly Program Based on Sensitive Detection of Fragment Overlaps", Genomics 14, 18–25 (1992).

[8] X. Huang, "An Improved Sequence Assembly Program", Genomics 33, 21–31 (1996)

[9] R. M. Idury and M. S. Waterman, "A New Algorithm for DNA Sequence Assembly", J. of Comp. Bio. 2(2), 291–306 (1995).

[10] J. D. Kececioglu and E. W. Myers, "Combinatorial Algorithms for DNA Sequence Assembly", Algorithmica 13, 7–51 (1995).

[11] J. Kececioglu and J. Yu, "Separating repeats in DNA sequence assembly", in *Proceedings of the 5th ACM Conference on Computational Molecular Biology*, ACM Press, 176–183 (2001).

[12] S. Kim and A. M. Segre, "AMASS: A Structured Pattern Matching Approach to Shotgun Sequence Assembly" J. Comput. Biol. 6(2), 163–186 (1999).

[13] H. W. Mewes et al., "Overview of the yeast genome", Nature 387, 737 (1997).

[14] J. Mullikin et al., `ftp://ftp.ensembl.org/traces/human/qual/Chr_20/`.

[15] E. W. Myers, "Toward Simplifying and Accurately Formulating Fragment Assembly", J. Comput. Biol. 2, 275–290 (1995).

[16] E. W. Myers et al., "A Whole-Genome Assembly of *Drosophila*", Science 287, 2196–2204 (2000).

[17] Z. Ning, A. J. Cox, J. C. Mullikin "SSAHA: A Fast Search Method for Large DNA Databases" Genome Research 11, 1725-1729 (2001).

[18] P. A. Pevzner, H. Tang, M. S. Waterman, "A new approach to fragment assembly in DNA sequencing" in *Proceedings of the 5th ACM Conference on Computational Molecular Biology*, ACM Press, 256–267 (2001); "An Eulerian path approach to DNA fragment assembly", Proc. Nat. Acad. Sci. 98, 9748–9753 (2001).

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

[20] G. G. Sutton et al., "TIGR Assembler: A New Tool for Assembling Large Shotgun Sequencing Projects", Genome Sci. and Technology 1, 9–19 (1995).

[21] J. C. Venter et al., "The Sequence of the Human Genome", Science 291, 1304–1351 (2001).

[22] M. Waterman, *An Introduction to Computational Biology*, Chapman and Hall (1995).

[23] J. L. Weber and E. W. Myers, "Human Whole-Genome Shotgun Sequencing", Genome Res. 7, 401–409 (1997).