Parallelization of MIRA Whole Genome and EST Sequence Assembler\textsuperscript{[1]}

Abhishek Biswas*, Desh Ranjan, Mohammad Zubair

Department of Computer Science, Old Dominion University, Norfolk, VA 23529 USA
*abiswas@cs.odu.edu

Abstract

The genome assembly problem is to generate the original DNA sequence of the organism from a large set of short overlapping fragments. MIRA is an open source assembler based on the Overlap Layout Consensus (OLC) graph model which addresses the assembly problem and is widely used by biologists [1,2]. Like other assemblers MIRA takes a long time to compute the assembly for large number of sequences. For example it takes around 24 hours to assemble a dataset with 1.4 million DNA sequence fragments and takes even longer for EST assemblies [3]. In this paper, we report our efforts in parallelizing MIRA assembler. The task of parallelizing MIRA assembler is challenging as it has critical code segments which are inherently sequential but are crucial for a good quality assembly. Initially, we focus on the two time consuming kernels of the MIRA assembler: graph construction, and edge weight calculation algorithm. Our implementation of the parallel code on a machine with 2 Intel Xeon X5550 Nehalem-EP quad core processors achieves linear speedup for these kernels.

1. Introduction

Current DNA sequencing methodologies (with the exception of emerging experimental technologies) cannot sequence DNA fragments of greater than \(-1\) kilobase (kB) in length. We rely on computational methods to assemble a complete DNA sequence from a large number of DNA fragments of smaller size. One popular and cost effective method is based on shotgun sequencing such as 454 pyrosequencing [4]. Shotgun sequencing generates DNA fragments by breaking up multiple copies of the original sequence at random points. Next a software program is used to construct the original DNA from a large set of DNA fragments generated by shogun sequencing. The problem of combining DNA fragments (reads) to reconstruct the source DNA is known as sequence (or genome) assembly problem. The assembly problem is usually modeled as computing the shortest common superstring (SCS), which is a reasonable approximation of the original sequence. The SCS problem can be modeled as a graph problem and is shown to be NP-complete [5] even without presence of errors in the fragments. The assembly problem is hard to solve because some fragments are lost and some have errors and is known to be NP hard [2]. Additional complexity arises when there are repeats in the original sequence. Repeats are multiple identical or nearly identical stretches of DNA which the SCS solution represents only once in the assembled genome. This problem is known as repeat collapse and can lead to serious assembly errors.

A number of heuristics have been proposed that address this problem, and form the basis of many well-known assemblers [1,6,7,8]. Though these heuristic-based assemblers limit the exponential search space, they still take long execution time (in order of days). This is because the graphs generated typically have anywhere from 100,000 to a few million nodes depending on the size of the genome. The average degree of a node is roughly equal to the coverage of the input data which is the number of times a particular area of the genome is copied in the input fragment, typically, 15-20 times.

In this paper we explore the parallelization strategies for open source MIRA assembler [1] in a shared memory multicore environment. MIRA is an open source assembler based on the Overlap Layout Consensus (OLC) graph model which addresses the assembly problem and is widely used by the life sciences community. MIRA is capable of handling next generation shotgun reads from 454, Ion Torrent, Solexa and PacBio machines along with Sanger sequences. MIRA has been used at IMB Jena Genome Sequencing Centre [9] and has been shown to be capable of assembling cosmid sequences in Human genome [10]. MIRA has also been used for \textit{de novo} assembly of 454 pyrosequencing transcriptome projects like [11,12,13,14,15,16].

The major problem with MIRA assembler is that it takes long time to work with large number of sequences, for example it takes around 24 hours to assemble a dataset with 1.4 million reads. This limits the use of MIRA assembler for large sequences and
### 1. Introduction

The Overlap Layout Consensus model represents each read as a vertex in a graph connected by edges, weighted by their pairwise alignment scores [2]. The assembly algorithm seeks to find a Hamiltonian path in this graph. This model has been very popular for genome assembly particularly to handle whole prokaryotic genomes. However, a disadvantage of this method is that repeats can be collapsed and cause rearrangements during assembly and assemblers use error thresholds and repeat tagging to correct such errors.

The de-Brujin graph model groups the reads into shorter stretches of length k (called k-mers) and representing each read as a path in the graph [17]. The assembly can then be represented as a superpath, a path that includes all of the input paths. Since an edge can be traversed multiple times, repeat sequences are not compressed during assembly. This model was improved by graph reduction to untangle the loops in the graph [18].

The third graph model for of sequence assembly uses string graphs [19]. An overlap graph is built where nodes correspond to reads and edges correspond to overlaps. The assembly is represented by the shortest walk which includes all of the required edges.

### 2. Parallel MIRA Assembly Process

In this section we discuss the MIRA assembly pipeline and propose parallelization strategies for different phases. The basic pipeline of the MIRA assembly process is shown in Figure 1. The input reads are preprocessed based on quality values and ancillary data if provided and presented to the iterative portion of the assembly process.

#### 3.1 Potential Edge Detection Phase

The assembly process proceeds with each read as a node in an graph and the first phase determines the high confidence region (HCR) of each read and scans all the $n^2$ edge possibilities using heuristic match algorithms [20,21]. The match determines if a sequence of length $k$ is present in the matching read with at most $l$ errors. For each sequence the complement is also matched to find all potential edges from it. This SIM algorithm creates two potential edge files named, post-match files, for the forward and complement matches. Each record in these files corresponds to a potential edge, containing the identifiers of the two reads and some offset information for matching. This phase is implemented in parallel in the standard implementation of the software using boost threading library.

### 2. Related Work

Graph-based representation of the genome assembly problem has resulted in three models.

MIRA provides specific routines for handling various read types and leverages mate pair information if available. The assembly process is based on the Overlap Layout Consensus (OLC) graph model [2] with critical code for handling repeats of various lengths. Error correction routines and thresholds based on sequencing technology and quality values are used to detect mis-assemblies and chimeric reads. These routines encode domain knowledge specific rules which are vital for correct assembly and must be preserved in our parallel implementation. It is essential that in the parallel version of MIRA the basic assembly pipeline is not significantly changed and the assembly output is same as that of the sequential implementation. Therefore, we are interested in identifying parallelization opportunities in MIRA and evolve the sequential code to exploit parallelism.

Initially we focus on the two time consuming kernels of the MIRA assembler: graph construction, and edge weight calculation. This phase is responsible for about 50-50% of the assembly time in a typical run of MIRA. For example, during one of our tests these kernels took 30% of the time to assemble a bacterial genome of 5.5 million base pairs from 500,000 reads. In another such test, this phase took about 50% time during an EST assembly of over 1 million reads.

The rest of the paper is organized as follows. Section 2 outlines the graph models popularly used for genome assembly and lists some related combinatorial problems in bioinformatics solved using graph approximations. Section 3 focuses on the core assembly pipeline of MIRA assembler and proposes parallelization strategies. Section 4 describes the parallel implementation of the graph construction and edge weight calculation phase. Section 5 presents the results obtained from parallel refactoring and Section 6 concludes with future work.

### 3. Graph Models

The rest of the paper is organized as follows. Section 2 outlines the graph models popularly used for genome assembly and lists some related combinatorial problems in bioinformatics solved using graph approximations. Section 3 focuses on the core assembly pipeline of MIRA assembler and proposes parallelization strategies. Section 4 describes the parallel implementation of the graph construction and edge weight calculation phase. Section 5 presents the results obtained from parallel refactoring and Section 6 concludes with future work.
3.2 Parallel Graph Construction Phase

The second phase of the assembly algorithm determines the exact edge set and the edge weight by banded Smith Waterman overlap calculation $[22,10]$ for each of the pair of reads generated by the SKIM algorithm. Some edges are rejected based on various conditions and overlap computation is avoided if the overlap length satisfies certain conditions. The edges appear in random order in the potential edge files generated above and weight calculation of one edge is not dependent on the others. We use OpenMP parallel pragmas and TBB containers to refactor this phase and execute it in parallel. We implement a single producer generating multiple tasks, each task computing a certain number of edges. This phase can account for over 30-50% in the serial pipeline and parallelization shows significant improvement in overall time.

3.3 Proposed Parallel Contig Building Phase

The next phase is a greedy heuristic to find the partial paths in the graph and build the best contiguous sequence (contigs). Work is in progress to parallelize this phase of the assembly process in a shared memory environment. In this phase the path finder algorithm identifies nodes with high degree and low error and begins assembly process by adding neighboring reads to it and forming contigs. A contig can grow in length, depth or both when a read is added to it and every addition increases the expected error based on the edge weight. Each contig is a consensus sequence of all the overlapping reads that capture a certain region of the genome. Ideally, a contig should have a depth close to the coverage of the input data as each base in the contig should correspond to a base in number of reads stacked up correctly. Also, the length of the contig must be close to the length of the genome. However, due to presence of repeats and errors the contigs cannot be extended beyond a certain length as the total acceptable mismatch error crosses allowed threshold. The backbone build strategy increases the length of the contig and the in-depth strategy adds reads to increase
the coverage. Each has advantages and disadvantages, but, both must be used to successfully build non redundant and correct contigs [10].

A \((n, m)\) look-ahead version of a simple greedy strategy is applied to select the most probable overlap candidate for a given contig. The algorithm extends \(n\) paths from the last \(n\) nodes of a contig upto \(m\) levels and the node generating the best path upto \(m\) levels is selected for assimilation into the contig. The new read selected is checked against the existing contig consensus for errors and if the mismatches are within acceptable threshold, the read is accepted into the contig consensus. This read is then not used by the other contigs in the same pass of the assembly.

The proposed approach to perform the contig build stage in parallel is to simultaneously start assembly from dense nodes with highly divergent sequences separated by certain path length. The independent threads generate a contig with the best possible depth and length. The resulting contigs are analyzed to check for common nodes. In case, contigs contain common nodes, the contig with best assembly for a read is allowed to keep the node and the read is removed from the contig consensus of the other contigs. Therefore, we propose to introduce a contig reduction phase to account for contigs using common nodes. Similar graph reductions in de-Brujin graph model have been applied to resolve repeated regions by duplicating the common paths [18]. Our method in contrast proposes to force contigs to search for alternate paths in the next pass of the graph. Figure 2.A shows a case where two contigs have been extended in parallel through a set of common nodes. The proposed reduction forces one of the contigs to find an alternate path for extension. Figure 2.B shows a similar situation and the reduction forces one of the contigs to extend in the reverse direction using an alternate path. Figure 2.C shows another possible case where one of the contigs is reduced into two contigs. The resulting contigs have to be extended through another path in subsequent passes of the graph.

4. Implementation

In this section we discuss parallel implementation of the graph construction and edge weight computation phase. The SKIM algorithm creates two post-match files with potential forward and reverse direction edges computed from a pair of sequences and their complement respectively. These two files must be processed to generate the actual edge set and the edge weights. The sequential implementation computes Smith Waterman overlap of each edge unless certain conditions are met, in which case, the edge is discarded or accepted even before the calculation. The accepted edges are written to an adjacency file on the disk along with the overlap score and other parameters. This process was refactored to execute in parallel and generate the adjacency list of the graph. Since, the edges need not be processed is any particular order, we used a producer while loop executed by the master thread to create tasks of a certain granularity. Each task performs edge weight computation on its subset of edges independently and writes the valid edges on a concurrent vector. This allows for creation of a large number of concurrent tasks of configurable size and exploits fine grained parallelism. A parallel version of Smith Waterman algorithm is not used as the dynamic programming matrices sizes are small and splitting each overlap calculation from parallel threads would increase overhead.

The three main challenges faced in refactoring the source code were the following. Firstly, MIRA is implemented using C++ and is optimized to reduce the memory utilization. So, many of the results at end of each stage are written onto the disk and a large number of disk writes are performed. This model would seriously impede parallelism as threads would compete for access to the disk. Secondly, MIRA uses Standard Template Library (STL) collections to implement data structures such as the adjacency list, repeat markers and the sequence read pool. Parallel updates on these data structures would have to be synchronized to maintain correctness. Synchronization of the threads by some locking mechanism will also affect parallelism. Finally, the source code also has lot of rule checking and conditional execution of error flagging routines which are often sequential in nature and perform updates on global data structures or write to files on the disk.

To refactor such a sequential code OpenMP [23] was found to be the best choice as it provides a host of synchronization pragmas for parallel flow control. However, the use of STL objects would cause performance bottlenecks in many cases. Therefore, we used concurrent collections provided by Intel’s Thread Building Blocks (TBB) library [24].

4.1. Refactoring MIRA Implementation

Parallel refactoring process of the graph construction and edge calculation was performed in the following stages to ensure thread safety and avoid untraceable data race errors.

4.1.1. Ensure Thread Safety. The first phase was to ensure the parallel computations of Smith Waterman overlap was independent and the objects access shared resources safely. For example the initialization of the
shared static substitution matrix is performed only once by one of the threads and the other threads are forced to wait and continue once the initialization flag is set for the first time.

4.1.2. Identify Critical Sections. Critical sections in the routines to be executed in parallel have to be identified before parallel code is written. These regions have to be protected by synchronization primitives. E.g. write operations on shared resources like the read pool and the adjacency list file are protected by named critical sections to prevent inconsistent writes.

4.1.3. Introduce Parallel Execution Code. Parallel task generation code is written using OpenMP. The master thread executes a loop reading the file and intermittently spawns tasks of a fixed granularity. These orphaned tasks compute the Smith Waterman overlap for the potential edges passed to it and write the accepted edges to the adjacency file.

4.1.4. Controlled Reduction of Critical Sections. Performance tuning must be done to reduce critical sections and allow as much as parallel execution as possible. MIRA has a large number of STL vector containers containing the read pool, read tags list and the list of discarded reads. Converting all of them into TBB concurrent vectors would require considerable implementation changes due to difference in interfaces and may cause deterioration of serial performance. Therefore, only vectors with large number of parallel updates are converted to concurrent vectors e.g. the discarded reads are updated into a concurrent vector and the accepted edges too are stored in a concurrent vector instead of a file on the disk. Also, optimizations like replacing critical section with atomic operations are done if possible.

The parallel refractory of graph construction and edge weight computation phase is complete. The next section presents the speedup obtained.

5. Results

In this section we report on some numerical experiments showing the performance of our implementation of the parallel graph construction and edge weight computation algorithm. The experiments were performed on a machine with 2 Intel Xenon X5550 Nehalem-EP quad core processors, 8MB cache and 144GB main memory.

The first set of experiments shows the speedup of our algorithm on incrementally larger graphs. The graphs are built from subsets of 454 pyrosequencing reads of *Mycobacterium Shottsii*. The time to calculate the Smith Waterman edge weights for the first pass is shown in Table 1. In this experiment the condition checking modules to bypass Smith Waterman overlap computation are disabled and the overlap is computed for all the edges in the graph. The primary producer thread spawns a task after reading 10,000 potential edge records from the post-match files. Therefore, the granularity of each task is 10,000 Smith Waterman calculations with average overlap length of 237bp. Table 1 shows close to linear speedup and Figure 3 plots in each case vs. number of processors.

<table>
<thead>
<tr>
<th>Smith Waterman Comp.</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>8.48</td>
<td>4.65</td>
<td>2.34</td>
<td>1.75</td>
<td>1.53</td>
</tr>
<tr>
<td>500,000</td>
<td>38.84</td>
<td>20.92</td>
<td>10.42</td>
<td>7.73</td>
<td>5.59</td>
</tr>
<tr>
<td>1,000,000</td>
<td>80</td>
<td>43</td>
<td>21</td>
<td>14</td>
<td>10.6</td>
</tr>
<tr>
<td>5,000,000</td>
<td>470</td>
<td>236</td>
<td>120</td>
<td>82</td>
<td>62</td>
</tr>
<tr>
<td>10,000,000</td>
<td>956</td>
<td>468</td>
<td>249</td>
<td>173</td>
<td>136</td>
</tr>
</tbody>
</table>

Fig 3. Plot of speedup vs. no. of processors

The second experiment shows the performance of the algorithm during the first pass of graph construction and edge weight calculation for various graphs. The graphs in this experiment are generated from reads of various bacteria. The rows in Table 2 correspond to graphs built for assembly of the following bacteria in top down order: *Mycobacterium vanbaalenii* PYR-1, *Mycobacterium marinum* M, *Escherichia coli* K12, 3 datasets of *Mycobacterium Shottsii* and *Mycobacterium pseudoshottsii*. The granularity of each task is still 10,000 potential edge records, but, some of which may bypass overlap calculation.
The speedup is close to linear for all the datasets. The table also shows that the absolute time of graph construction depends on the number of edges and the average overlap length. Absolute time of the run is inversely proportional to the average length of the overlap and can significantly affect the computation time even for relatively small graphs (see Table 2 row 1). Figure 4 is the plot of the speedup vs. number of processors.

### Table 2. Computation Time of Real Graphs

<table>
<thead>
<tr>
<th>Smith Waterman Comp.</th>
<th>Avg. Overlap (bp)</th>
<th>Time on Processors (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4 x 10^6</td>
<td>237</td>
<td>875 455 220 145 112</td>
</tr>
<tr>
<td>7.0 x 10^6</td>
<td>310</td>
<td>140 83 41 21 10</td>
</tr>
<tr>
<td>8.3 x 10^6</td>
<td>484</td>
<td>395 198 99 49 24</td>
</tr>
<tr>
<td>9.9 x 10^6</td>
<td>654</td>
<td>585 288 144 72 36</td>
</tr>
<tr>
<td>1.2 x 10^7</td>
<td>945</td>
<td>450 225 113 56 28</td>
</tr>
<tr>
<td>1.5 x 10^7</td>
<td>1240</td>
<td>1020 510 255 127 63</td>
</tr>
</tbody>
</table>

A further experiment was performed to study the effect of the number of Smith Waterman computations performed per thread (see Table 3). This experiment was performed on a graph with 9.44x10^6 edges. Smith Waterman calculation was performed for each edge and the average overlap size was 220bp. The results show that the execution time does not vary significantly with grain size. However, the thread creation overhead deteriorates performance for very small grain sizes. This is expected in a shared memory environment where threads have very small critical sections and large numbers of threads exploit fine grained parallelism.

### Table 3. Grain size split of a graph with 9.44x10^6 edges vs. Processors

<table>
<thead>
<tr>
<th>Grain Size</th>
<th>Time on Processors (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 4 6 8</td>
</tr>
<tr>
<td>10</td>
<td>911.88 494.4 238.49 161.24 123.1</td>
</tr>
<tr>
<td>100</td>
<td>860.61 437.38 222.52 154.35 112.18</td>
</tr>
<tr>
<td>1000</td>
<td>862.75 458.0 216.54 152.24 113.23</td>
</tr>
<tr>
<td>10000</td>
<td>860.52 459.01 218.6 152.62 113.86</td>
</tr>
<tr>
<td>100000</td>
<td>858.97 462.43 228.28 160.36 122.26</td>
</tr>
</tbody>
</table>

### 6. Conclusion

In this paper we propose parallel strategies for parallelization of MIRA genome and EST sequence assembler. OpenMP task model is used to exploit fine grained parallelism in a multicore environment for graph construction and edge weight computation algorithm. A parallel version of the contig build algorithm is proposed with a contig reduction phase to resolve overlapping contigs constructed in parallel. The parallel implementation of the graph construction phase achieves close to linear speedup for various graph sizes and reduces the overall execution time significantly. The parallel refactoring does not change the graph structure and the assembler completes assembly to produce the same output. In one of our test runs, the overall assembly time for 500,000 454 pyrosequencing mate pair reads was reduced from 12.5 hours to 8.5 hours.

Implementation of the proposed parallel strategy for contig building is expected to further reduce execution time. The algorithm and the effect of parallel construction of contigs and their subsequent reduction on the assembly output is the subject of our current research. Also, refactoring core modules of MIRA for better compatibility with parallel computing models is in progress.
7. References


