2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Parallelization of the Trinity pipeline for *de novo* transcriptome assembly

V. Sachdeva
Computational Science Center
IBM T. J. Watson Research Center
Cambridge, MA
Email: vsachde@us.ibm.com[1]

C. S. Kim
The Hartree Centre
STFC Daresbury Laboratory
Warrington, WA4 4AD
Email: chang-sik.kim@stfc.ac.uk[1]

K. E. Jordan
Computational Science Center
IBM T. J. Watson Research Center
Cambridge, MA
Email: kjordan@us.ibm.com

M. D. Winn
The Hartree Centre
STFC Daresbury Laboratory
Warrington, WA4 4AD
Email: martyn.winn@stfc.ac.uk

***Abstract*—This paper details a distributed-memory implementation of Chrysalis, part of the popular Trinity workflow used for *de novo* transcriptome assembly. We have implemented changes to *Chrysalis*, which was previously multi-threaded for shared-memory architectures, to change it to a hybrid implementation which uses both MPI and OpenMP. With the new hybrid implementation, we report speedups of about a factor of twenty for both *GraphFromFasta* and *ReadsToTranscripts* on an iDataPlex cluster for a sugarbeet dataset containing around 130 million reads. Along with the hybrid implementation, we also use PyFasta to speed up Bowtie execution by a factor of three which is also part of the Trinity workflow. Overall, we reduce the runtime of the Chrysalis step of the Trinity workflow from over 50 hours to less than 5 hours for the sugarbeet dataset. By enabling the use of multi-node clusters, this implementation is a significant step towards making *de novo* transcriptome assembly feasible for ever bigger transcriptome datasets.**

I. Introduction

High throughput sequencing technologies are making a big impact in many areas of life sciences. The most well-known technique is genome sequencing, whereby an organism's whole complement of DNA is sequenced. Reference genomes are being generated for an increasing number of organisms, including some extinct species [1]. Variation within a species is now being considered [2], including hypervariation in cancers [3]. Nevertheless, sequencing is applied in many other areas of life sciences including transcriptomics, ChIP-Seq, metagenomics, etc.

In this work, we have focussed on transcriptomics in which a library of cDNA derived from a sample of RNA is sequenced. The aim is to sequence the mRNA corresponding to transcribed genes, and various techniques exist to select for mRNA or remove other kinds of RNA (such as ribosomal). Transcriptomic sequencing differs from genome sequencing in two crucial ways. Firstly, the population of mRNA depends on the expression levels of genes in the chosen sample, and there can be a very large dynamic range. Secondly, in higher organisms genes are post-processed by alternative splicing to generate multiple isoforms, which need to be distinguished.

Transcriptomics gives information on which genes are expressed in a given cell type, under given conditions, at a given time point. The number of reads which map to a given gene or isoform is a direct measure of the expression level. Thus, transcriptomics is a major route to the study of gene expression, and is rapidly replacing microarrays as the method of choice.

As in genome sequencing, the cDNA libraries are chopped up into millions of short reads which are then sequenced. The computational task is then to re-assemble these sequenced reads into a set of transcripts corresponding to gene products. As the sequencing technology improves, this computational step is becoming the principal bottleneck. A dataset consists of a large set of sequenced reads (provided as a FASTA or FASTQ file) which can have a size on the same order as for genome sequencing. Unlike genome sequencing though, an organism can have multiple transcriptomes corresponding to different cell types or conditions. The recommended practice is to sequence these together into a consensus transcriptome, and thus the size of the dataset is multiplied by the number of experiments considered.

Transcriptome assembly thus works on very large datasets and can require considerable compute resources. It is a high-performance computing (HPC) problem. Nevertheless, many of the commonly-used bioinformatics tools were developed for desktop applications, and most assume a shared memory architecture. Such software struggles with today's large datasets, taking days to run routine jobs, and often exceeding the available memory. At the same time, the trend for HPC is towards more parallelism, with larger numbers of lower power nodes [4]. Adapting bioinformatics tools for multi-node distributed memory architectures is thus essential. With a few

[1]joint lead author

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.67

notable exceptions [5][6][7], distributed memory architectures are not well supported by existing bioinformatics tools.

We have chosen to look at the Trinity pipeline [8] which is one of the most popular packages for transcriptomic assembly [9]. It was originally created for *de novo* assembly, although there is now a protocol for assembly against a reference genome. *De novo* assembly constructs the transcriptome solely from the available reads, and is useful when there is no reference genome available, or if there are likely to be large structural variations (e.g. in cancer cell lines) [10].

Trinity is a pipeline implemented in Perl which wraps a number of underlying programs implementing different stages of the assembly [11]. In it's evaluation, Trinity has been found to be accurate in transcriptome assembly under a variety of conditions, but with high runtimes [12]. The pipeline is very heterogenous in its computational requirements, with early stages requiring large amounts of memory, and later stages being more CPU-intensive. Trinity's assembly pipeline consists of four consecutive modules: Jellyfish, Inchworm, Chrysalis and Butterfly. The modules are separate executables, written in different languages. The Chrysalis step itself is composed of separate submodules including Bowtie [13], GraphFromFasta and ReadsToTranscripts. Previous attempts to speed up Trinity have focused on using OpenMP threads in a shared-memory architecture and reducing I/O operations [14]. We present results here for the Chrysalis module, aimed at speeding up this section of the pipeline by spreading the load across multiple distributed nodes, working seamlessly with the already existing OpenMP implementation. The initial Bowtie step which maps reads to Inchworm contigs has been parallelised by splitting the contigs across MPI processes. Parts of GraphFromFasta and ReadsToTranscripts which were written in OpenMP for shared memory have been changed to a hybrid implementation using MPI across distributed nodes, and OpenMP threads within a node.

We have tested the MPI-enabled hybrid version across a number of datasets, comparing the quality of the resulting transcript as well as the time taken. Repeated runs of the shared-memory version of Chrysalis show a distribution of metrics of the transcriptome, due to the stochastic nature of some of the assembly steps. The results from the MPI-enabled version also show a distribution, which overlaps the shared-memory distribution and is not significantly different.

In Section II, we summarise the algorithms underlying the different components of the Trinity pipeline, along with benchmarking of Trinity to illustrate the computational requirements. Next, in Section III we give a detailed description of our MPI parallelisation scheme and its implementation, and Section IV gives details of our validation method. Results are given in Section V, and we conclude in Section VI with an outlook on future improvements.

II. Existing Algorithms and Implementation

De-novo transcriptome assembly does not depend on a reference genome, instead depending on the redundancy of short reads to find sufficient overlaps between the reads. It subsequently uses these overlaps to assemble a set of transcripts corresponding to expressed genes. For a comprehensive overview of the transcript reconstruction methods for RNA-seq, please refer to [15].

A. Trinity modules

The Trinity assembler is a heterogenous workflow comprised of modules (or software programs), which when run one after the other through a single Perl script (*Trinity.pl*), produces the reconstructed transcriptomes as the final output. In recent years, Trinity has been converted to a modular platform, using third-party tools that can be swapped in and out in future releases. The software modules exchange data through files; the files being output from one software module are then consumed by the following module. It is to be noted that Trinity also includes tools such as RSEM [16], edgeR [17] etc. that take the output of the Trinity workflow and estimate levels of gene expression, in particular for differential expression analysis. We do not include the description of those tools in this paper. For information on these tools, please refer to [18]. As mentioned earlier, Trinity's assembly pipeline consists of four consecutive modules: Jellyfish, Inchworm, Chrysalis and Butterfly. We provide a description and the function of each component below: for more details, please refer to [8].

- Jellyfish: The first step in the Trinity workflow is Jellyfish, which is a tool for fast, memory-efficient counting of k-mers (substrings of length k) in DNA [19]. Jellyfish can read FASTA and multi-FASTA files, outputting it's k-mer counts in a binary format. In the Trinity workflow, *jellyfish count* which outputs the counts of k-mers is followed by *jellyfish dump*, which converts the binary format into text format. The output of the two Jellyfish commands are thus files containing information on all k-mers extracted from the short reads with their counts. Jellyfish can output a single or multiple files depending on the available memory of the system. Jellyfish's output can be extremely voluminous - for example for the sugarbeet dataset for which we provide benchmarking results, the RNA-seq fasta file size is 15 GB, while the Jellyfish output is greater than 100 GB. Another application for k-mer counting that uses less memory than Jellyfish is DSK [20]; however this is not part of the Trinity pipeline yet. Jellyfish's output is consumed by Inchworm, the next step in the Trinity pipeline workflow.
- Inchworm: Jellyfish's output of k-mers and k-mer counts is read by Inchworm as a first step. Since Jellyfish's output can be voluminous, the reading of the k-mers into Inchworm can also take a substantial amount of time. Inchworm constructs a hash table object consisting of pairs or duals - the duals are comprised of the k-mers along with the read abundance of the kmer. Constructing the hash table object from the k-mer file using multiple OpenMP threads can be both time and memory intensive - since Inchworm keeps this entire hash table object in memory, Inchworm's memory footprint can be extremely high. This hash table object is subsequently

sorted in order of decreasing k-mer abundance. Inchworm examines each unique k-mer starting from the most abundant, and generates Inchworm contigs using a greedy extension based on (k-1)-mer overlaps. These contigs are subsequently written to disk. Inchworm thus goes through the following steps:

  - Constructs a k-mer dictionary from all sequence reads removing likely error-containing k-mers, and sorts them in decreasing order of abundance.
  - Selects the most frequent k-mer in the dictionary to seed a contig assembly.
  - Extends the seed in each direction by finding the highest occuring k-mer with a k-1 overlap as shown in Figure 1.
  - Extends the sequence in either direction until it is not extended further, reporting the linear contig.
  - Repeats these steps with the next abundant k-mer until the entire dictionary is exhausted.

  In summary, Inchworm reads in the massive k-mer file written by Jellyfish, does a greedy extension of the kmers in decreasing order of abundance (consisting of the steps above), and then writes a comparatively much smaller file of contigs.
- Chrysalis: Chrysalis clusters minimally overlapping contigs obtained from Inchworm into separate sets of connected components, followed by construction of de Bruijn graphs for each component. Chrysalis itself is composed of two separate modules - *GraphFromFasta* and *ReadsToTranscripts* - which specifically do the following:
  - *GraphFromFasta* clusters related Inchworm contigs into so-called components. It does this by welding pairs of contigs together if read support exists, and subsequently clustering Inchworm contigs using these welds and building de Bruijn graphs for each component. Both of these steps are already parallelized with OpenMP threads, but since each possible pair of Inchworm contigs has to be compared, this process can still be extremely compute-intensive.
  - *ReadsToTranscripts* assigns each read to the component with which it shares the largest number of k-mers, as well as determining the regions within each read that contribute k-mers to the component.

  Apart from the components mentioned above, Trinity also uses *Bowtie* (a third-party tool) to align input reads to Inchworm contigs.
- Butterfly: Butterfly reconstructs feasible full-length linear transcripts by reconciling the individual de Bruijn graphs generated by Chrysalis with the original reads and paired end data. Each Chrysalis component or graph can produce several linear transcripts, which in most cases will correspond to alternative splicing of the gene product.

B. Benchmarking of Original Trinity

To understand the basic characteristics of Trinity performance, we measured memory usage and runtime of each step in Trinity using the Collectl tool [21] distributed with Trinity. To perform this run, Trinity was compiled using GNU compilers and the performance evaluated using a sugarbeet RNA-seq dataset kindly provided by Rothamsted Research, UK. The dataset is 15 GB in size on disk and contains 129.8 M reads, with two subsets of 9 GB (79.2 M single end and left reads) and 6 GB (50.6 M right reads). Our sugarbeet dataset is larger than a typical test dataset in order to illustrate the computational challenges. Nevertheless, it is only representative of a routine RNA-Seq experiment, and much larger datasets are now being generated by sequencing facilities. Since Trinity already came with support for multiple OpenMP threads, this initial run was done using 16 threads on a single iDataPlex node at the Hartree Centre, UK. A single iDataPlex node at the Hartree Centre comprises 2x 8 core 2.6 Ghz Intel SandyBridge processors, with 256 GB of memory. Figure 2 shows the results of this original Trinity run, showing the RAM usage on the Y-axis with the runtime (in hours) along the X-axis.

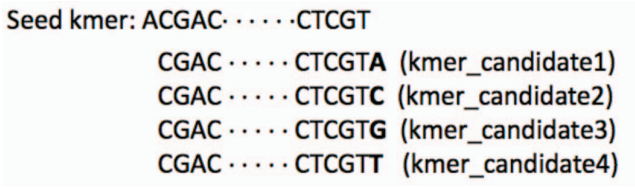

Fig. 1. Seed extension by k-mer with (k-1) overlaps.

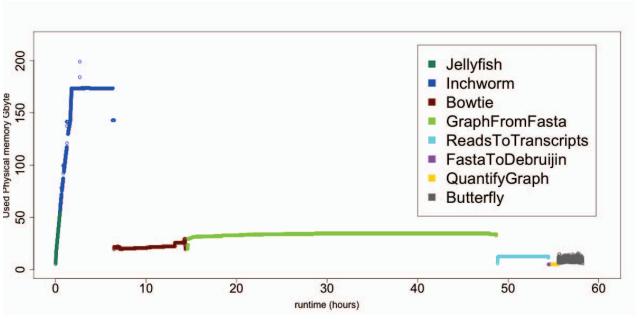

Fig. 2. Measurement of RAM usage (Y-axis) and the runtime (X-axis) of Trinity workflow run using single node of 16 cores and 256 GB of memory for the sugarbeet dataset.

As can be seen from Figure 2, it is clear that even for sugarbeet dataset, the runtime of the entire Trinity pipeline is close to 60 hours. Chrysalis is seen as the most time-intensive phase of the Trinity pipeline. The Chrysalis phase itself is composed of several sub-steps: Bowtie, GraphFromFasta, ReadsToTranscripts, FastaToDebruijn and QuantityGraph. Most of the runtime in Chrysalis is in three steps: Bowtie, GraphFromFasta and ReadsToTranscript, which appear in the same order in the Chrysalis step. For this reason, we decided to focus our parallelization efforts mainly on these three components. GraphFromFasta and ReadsToTranscripts are already parallelized for a shared-memory architecture with OpenMP threads. Thus, our

efforts were focused on a distributed memory implementation of these components, that works with the existing OpenMP implementation. For Bowtie, we leverage PyFasta [22], that can be used to split the target sequences amongst the MPI processes, thus not requiring any source code changes. In the next Section III, we include details on our parallelization, with the subsequent computational speedups reported in Section V.

III. Parallelization of Trinity workflow

As can be seen from the previous description of Trinity modules and benchmarking results, Trinity software is meant for usage only on a shared-memory machine. To improve the performance of the Trinity workflow, we focused on the most compute-intensive parts of Trinity which includes *Bowtie*, *GraphFromFasta* and *ReadsToTranscripts*.

We also had to be careful that our additional MPI code works well with the existing OpenMP code that is already being used for most of the time-intensive loops in Chrysalis; thus the OpenMP sections had to be changed to a hybrid model using MPI across multiple nodes, and OpenMP within a node. In the subsections below, we describe the changes made for the MPI implementation of Bowtie, GraphFromFasta and ReadsToTranscripts. We detail our parallel implementations below in the order that they are run in the Chrysalis workflow.

A. MPI implementation of Bowtie

The main objective of Chrysalis is building Inchworm bundles where each bundle is a cluster of Inchworm contigs. The Inchworm contigs in the same bundle are used for full reconstruction of transcripts. To build Inchworm bundles, Chrysalis first aligns input reads to Inchworm contigs using Bowtie. Based on the output from Bowtie alignment, the subsequent step searches pairs of Inchworm contigs of which both ends are to be combined for the construction of scaffold, provided that some of input reads are aligned onto single end of each contigs. This output is later combined with "welding" pairs of Inchworm contigs from GraphFromFasta for full construction of Inchworm bundles. More details of "welding" pairs of Inchworm contigs are described in the following section.

Bowtie already has an option for using multiple threads simultaneously on a single node to achieve a faster alignment speed. However, with millions of input reads, it can require several hours of runtime as shown in Figure 3. To speed up the alignment process, we ran Bowtie on multiple nodes by splitting the target sequences of Bowtie, i.e. the Fasta file of Inchworm contigs. The Fasta file was partitioned using the PyFasta python module, which evenly splits the target sequences amongst the rank nodes for parallel alignment processing. Each node then produces an alignment output file in SAM format, and the files from all nodes are merged into a single file at the end of the job. Our approach is different from [23], which did not use MPI, but looked at different partitioning of reads and genomes over nodes. Our partitioning of the Inchworm contigs over nodes is a special case of their more general study.

B. MPI implementation of GraphFromFasta

GraphFromFasta contains two compute-intensive loops. The first loop goes through each Inchworm contig; first, it finds all possible k-mers from the current contig, and subsequently it harvests "welding" subsequences which match sub-regions of other contigs. The size of the welding subsequence is $2k$ consisting of the seed k-mer and left- and right-flanking $\frac{k}{2}$-mers. That is, the first loop decides if common subsequence exists to "weld" two Inchworm contigs into the same Inchworm bundle at the end of GraphFromFasta. The loop is already multi-threaded with OpenMP threads; since the work done per Inchworm contig is not uniform (depending on the contig, either it is welded with other nchworm contigs or not), the OpenMP scheduling policy is *dynamic*. Each thread gets multiple Inchworm contigs, working on them until they run out, at which point they again get multiple contigs. The "chunksize" or the number of Inchworm contigs processed by each OpenMP thread is proportional to the number of Inchworm contigs divided by the number of threads.

Our focus was to change this loop into a hybrid loop, with additional speedup coming from the use of multiple nodes, each running multiple OpenMP threads. In the beginning, we pre-allocated chunks of Inchworm contigs to each MPI process. However, this did not give us a good speedup, especially when using the multiple threads. Our current implementation uses a "chunked round robin" strategy with each MPI process getting a chunk, distributing to its multiple threads, and then working on the next chunk. Mathematically, in the outer loop, chunk i consisting of n Inchworm contigs is allocated to MPI rank p if $i(modulo)p = 0$. The chunk consisting of n contigs is subsequently divided amongst the OpenMP threads in an inner loop. A contig's data is thus accessed by the sum of the index of the chunk with the index of the contig within the chunk. The OpenMP scheduling strategy is kept as dynamic as in the original loop. We had to be careful with such a strategy however, as there might be the case that some MPI processes might still try to get a full chunk, even though the number of Inchworm contigs left is less than the chunk size. Thus, the end index of the inner thread loop might have to be changed depending on how many Inchworm contigs are left for the MPI process. Figure 3 shows our "chunked round robin" distribution strategy.

Once all the MPI processes are done, they have a vector of the "welding" subsequences, which have to be pooled together on each rank from every rank before the second loop. As a first step, the vector of the subsequences are packed into a single sequence for MPI communication. Consequently, each MPI process then exchanges the size of this packed sequence to every other rank for subsequent communication using *MPI_Allgatherv* which pools together the sequences on every rank. At the end of the first loop, every rank, thus has a pool of sequences combined from every other rank. This pool of sequences is subsequently used for the second loop.

The second compute-intensive loop finds pairs of Inchworm contigs sharing any "welding" subsequence harvested from

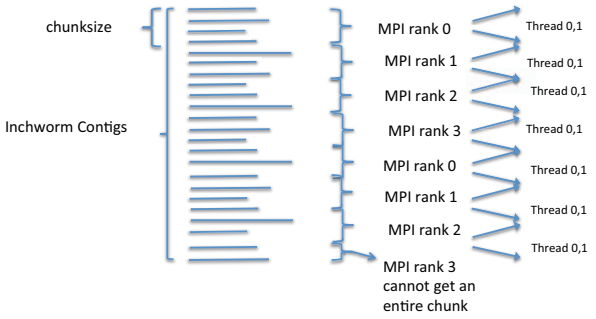

Fig. 3. Chunked round robin strategy for hybrid MPI + OpenMP code with 4 MPI processes and 2 OpenMP threads as an example.

the first loop. Our "chunked round-robin" strategy for the distribution of Inchworm contigs is also used in this loop. The output of the second loop is the list of indices for pairing Inchworm contigs for "welding", which is pooled on every rank in a similar way as in the first loop. First, the integer values for pairing indices are packed into single integer array for MPI communication. Subsequently, each MPI process exchanges the size of this integer array to other ranks for communication which is then used for pooling the pairing indices together on every rank. Since only integer arrays are exchanged, the second loop uses substantially less communication compared to the first loop where vectors of strings are exchanged between rank nodes.

C. MPI implementation of ReadsToTranscripts

ReadsToTranscripts assigns each input read to the Inchworm bundle against which the largest number of its constituent k-mers align. Since ReadsToTranscripts works with the input reads file which can be extremely large, ReadsToTranscripts does not try to load the entire input reads file into memory, but instead relies on a streaming reads model. This is opposite to GraphFromFasta; since GraphFromFasta only works with the Inchworm contig file which can be much smaller, it reads the entire file into memory. ReadsToTranscripts uploads chunk of input reads from the file into memory depending on a command-line parameter *max_mem_reads* which decides the number of input reads uploaded into memory at a time. These reads are inserted into a vector of strings, which is then used in the compute-intensive part of the loop. This part links each input read to the Inchworm bundle for which the largest number of its possible k-mers are aligned. This compute intensive loop is also OpenMP enabled with the set of uploaded input reads distributed over the OpenMP threads.

For a hybrid implementation, it was obvious that we needed to parallelize the uploading of the short reads across the multiple MPI processes. One of our strategies was to let only a master node or rank read the sequences and distribute to the other "slave" nodes. However, this strategy involves relatively heavy communications between master and slave nodes which leads to a bottleneck particularly as the number of slave nodes increases. Our second updated implementation allowed every rank to read the *max_mem_reads* by counting the number of chunks of the *max_mem_reads* uploaded into memory by a MPI process. If this count value is not a multiple of the rank, then the MPI process simple discards the uploaded input reads, and then reads another chunk of the input reads. This process continues until the count value is a multiple of the process's rank, at which point the reads are distributed amongst the OpenMP threads of this MPI process. This approach does make every process read redundant data (each process in fact reads the entire file), but excludes the necessity of MPI communication.

At the end of ReadsToTranscripts, a file with information on the reads aligned to the Inchworm contigs is written by each process. There is a final command at the end by the master node which combines the multiple files into a single file with a simple *cat* command. We have found the overhead of this concatenation step to be fairly low; another option is merging the data at the root process from all the processes and only let the root process write the final output.

Our current software methodology works as follows: *Trinity.pl* which is the Perl script that calls all the Trinity components has been extended with an argument for the number of processes (*nprocs*). This argument is then used in the command-line for the Chrysalis executable, which calls GraphFromFasta and ReadsToTranscripts separately from within it's source code. If the source code is compiled with MPI support primitive enabled, the command line for GraphFromFasta and ReadsToTranscripts is prepended with a suitable MPI runtime mechanism (such as *mpirun -np nprocs*) that allows both of these software modules to be run with multiple processes. In Section IV, we show the validation methodology of our parallel implementation. In Section V, we show the performance results obtained by the MPI enabled GraphFromFasta and ReadsToTranscripts, as well as the distributed version of Bowtie.

IV. Validation of Parallel Trinity

To show that the hybrid parallelized Trinity produces equivalent results in the reconstruction of transcripts to the original version of Trinity, we performed two sets of validation tests. These tests indicate that there is no significant difference in the output between both versions of Trinity. It should be noted that Trinity produces slightly indeterministic [18] output, in which the outputs from multiple runs using the same input data set can be slightly different, and therefore we do not expect identical results between both versions of the code.

The first test is an all-to-all sequence alignment approach, in which all reconstructed transcripts from the hybrid parallelized Trinity were aligned to those from the original Trinity using the Smith-Waterman algorithm, as implemented in the FASTA program [24]. Due to the indeterministic nature of Trinity, multiple results from ten repeated runs for each version of Trinity (OpenMP-only and MPI+OpenMP) were obtained. In addition to aligning transcripts between the different versions of Trinity, we also aligned transcripts from the different runs

of the original Trinity, in order to understand the expected level of variation in the output. The alignment results using a whitefly data set downloaded from the internet [25] comprised of a total of more than 420,000 reads with left and right reads of approximately 210,000 each are shown in Figure 4. They show no significant difference between the two versions of the code according to a two sample t-test, thus indicating that the output from the hybrid-parallelized Trinity has equal quality to the one from original version of Trinity.

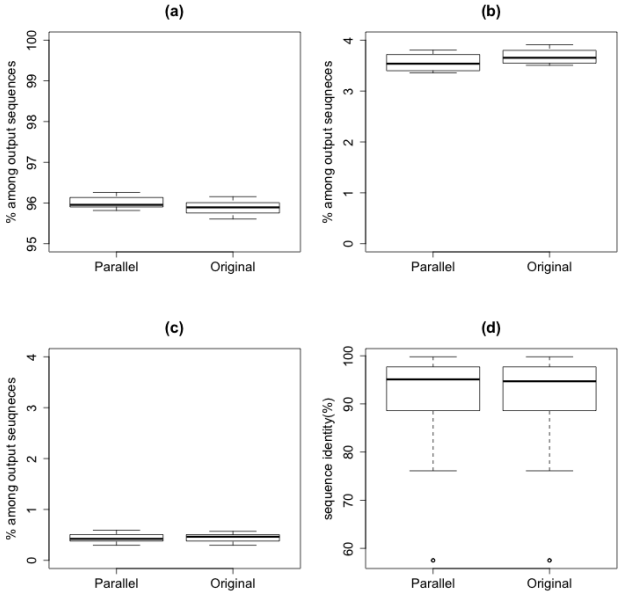

Fig. 4. Alignment of the reconstructed transcripts from parallelized Trinity to the ones from original Trinity using Smith-Waterman algorithm in FASTA program using whitefly dataset. The results are categorized into three groups; (a) 100% identical match for full length, (b) less than 100% identical match for full length and (c) less than 100% identical match for partial length. The distribution of identities/similarities of aligned sequence pairs in (c) is described in (d). "Parallel" represents the sequence alignment of two sets of reconstructed transcripts from parallelized Trinity and original Trinity, respectively. "Original" represents the alignment of two sets of reconstructed transcripts from original and original Trinity.

The second test involves measuring the number of reconstructed transcripts identified as known transcripts. The number was simply measured by aligning the reconstructed transcripts obtained from runs of Trinity against a set of reference transcripts, and this number was compared between the two versions of Trinity. The reference transcripts are comprehensive and well-annotated sets of transcript sequences, obtained from the Trinity FTP site for the Schizophrenia and Drosophila datasets. The Schizophrenia dataset consists of 9.2 million left reads and 6.15 million right reads, for a total of 15.35 million reads with a size of about 8 GB on disk. The Drosophila dataset consists of 50 million left and right reads, with a total size of about 10 GB on disk. Four related numbers were counted in this test:

- The number of genes of which at least one reconstructed isoform was aligned in full length onto one of the reference transcripts (see graphs (a) and (c) of Figure 5).
- The number of reconstructed isoforms aligned in full length onto one of the reference transcripts (see graphs (b) and (d) of Figure 5).

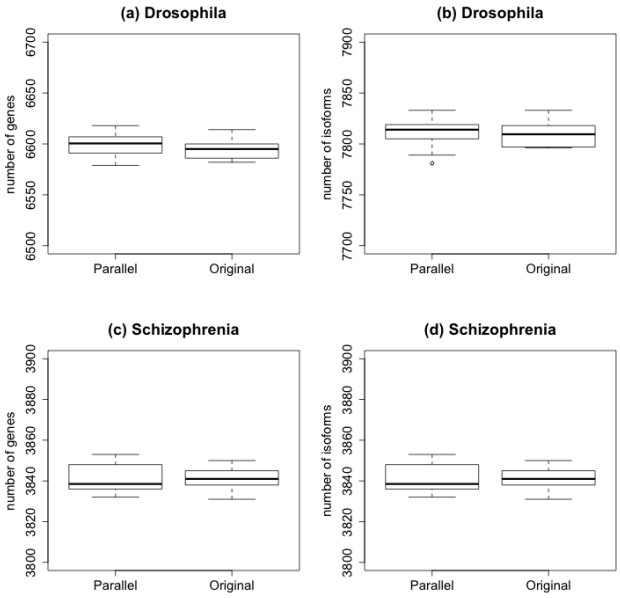

Fig. 5. Alignment of reconstructed transcripts from both versions of Trinity to the reference transcripts; number of fully reconstructed genes/isoforms in full-length for Schizophrenia (a, c) and Drosophila (b, d) datasets among the reference transcripts. Note the number of reconstructed genes in full-length represents the case that at least one isoform is reconstructed in full-length.

- The number of genes of which at least one reconstructed isoform corresponds to a fusion of multiple full-length reference transcripts (see graphs (a) and (c) of Figure 6).
- The number of reconstructed isoforms which correspond to a fusion of multiple full-length reference transcripts (see graphs (b) and (d) of Figure 6).

The "fused" transcripts considered in the last two numbers are single reconstructed transcripts including multiple full-length transcripts from the reference set. These transcripts are reconstructed as end-to-end fusions in some cases due to overlapping UTRs or other factors. These are likely false-positive reconstructed transcripts; however, these are still counted separately as being reconstructed transcripts due to their full length. Comparing these four numbers indicates that there is no significant difference in the outputs from both versions of Trinity.

V. Results

In this Section, we report the results obtained by running the distributed memory versions of GraphFromFasta, ReadsToTranscripts and Bowtie. The MPI enabled source code was compiled with OpenMPI 1.6 which is an open-source MPI implementation, using the GNU compiler version 4.4.6. Our test hardware is an iDataPlex cluster, known as "Blue Wonder", comprising 512 nodes each with 2x 8 core 2.6 GHz

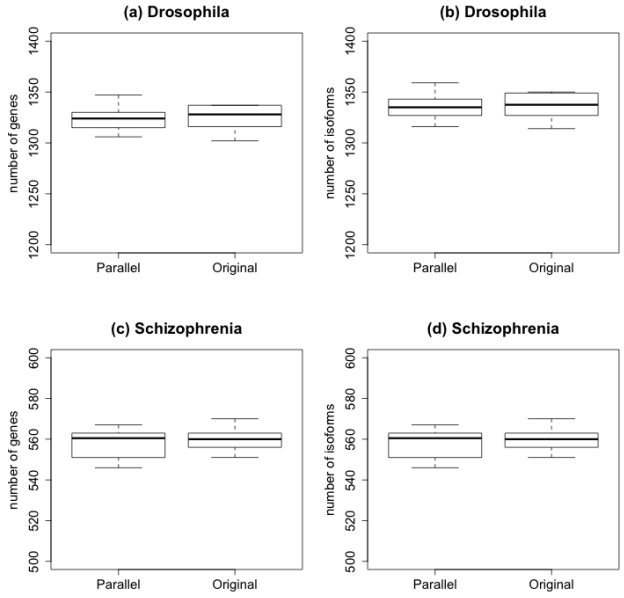

Fig. 6. Alignment of reconstructed transcripts from both versions of Trinity to the reference transcripts; number of reconstructed genes/isoforms in full-length as "fused" transcript for Schizophrenia (a, c) and Drosophila (b, d) datasets. Note "fused" transcript is defined as single reconstructed transcript including transcripts from multiple genes/isoforms.

Intel SandyBridge processors making 8,192 cores in total. Out of these 512 nodes, 256 nodes have 128 GB of memory which are the nodes we used for the MPI benchmarking.

The input dataset we used for the benchmarking is the sugarbeet dataset, which is 15 GB in size on disk and contains 129.8 M reads, with two subsets of 9 GB (79.2 M single end and left reads) and 6 GB (50.6 M right reads). The same dataset was used in the original benchmarking of Trinity (see Figure 2).

A. Hybrid (MPI+OpenMP) GraphFromFasta

Figure 7 shows the results of the MPI enabled hybrid GraphFromFasta code using the sugarbeet dataset as the input. The number of processes was varied from 16 to 192; each node runs a single MPI process, with 16 OpenMP threads. We started the runs with 16 nodes as the runtimes below 16 processes exceeded the maximum queue time of the parallel jobs. This graph shows the time taken separately in loops one and two, both of which were converted to a hybrid implementation, along with the total time taken in GraphFromFasta. Along with the loops one and two, GraphFromFasta also consumes time in other tasks setting up the k-mers before the second loop and generation of the final output after the second loop. As explained in Section III, loop one decides if a common subsequence exists to "weld" two Inchworm contigs into the same Inchworm bundle, while the second loop finds pairs of Inchworm contigs sharing any "welding" subsequence harvested from the first loop. This figure shows the lowest and the highest time taken in the loops, amongst all the MPI ranks, as a measure of load imbalance.

For all performance analysis, we consider the representative time as the processes with the highest times. For loop one, at 128 and 192 nodes, using data from the nodes with the highest time, we get a speedup of 8.31 and 11.93 compared to time of the loop from 16 nodes. For loop two, the speedups are 7.62 and 5.64 respectively using the loop timings at 16 nodes. At 192 nodes, the speedup of loop two is primarily lower due to load imbalance with the highest time of a process more than three times the process with the lowest time. For loop one as well, the highest MPI rank time is 50% higher than the lowest MPI rank time for the same number of nodes. Some of this load imbalance is due to the nature of the problem: there is a very wide variation in the lengths of reconstructed transcripts with some lengths being in tens of thousands, while others only a few hundred characters. This leads to an imbalance in the amount of work each node has to carry out. Currently, we have a static partitioning strategy amongst the nodes; in the future, we might experiment with a dynamic partitioning strategy to reduce this load imbalance.

For the entire GraphFromFasta, the baseline performance is the performance measured with the OpenMP only version run with 16 threads on one node (122610 seconds). The time taken by 16 nodes, each running 16 threads, is 27133 seconds, while with 192 nodes, each employing 16 threads, the total runtime decreases to 5930 seconds. These runtimes correspond to speedups of 4.5 and 20.7 respectively for the GraphFromFasta overall. This low speedup is primarily due to the share of the non-MPI regions accounting for a increased percentage of the total GraphFromFasta time with increasing nodes, for example at 16 nodes, the time taken by both the loops comprises 92.44% of the total time of GraphFromFasta, which falls to 57.4% at 192 nodes. Figure 8 shows the breakup of the GraphFromFasta times into separate timings of loop 1, 2 and the non-parallel regions. As can be seen, non-parallel regions account for an increasing percentage of the total time of GraphFromFasta; at 128 processes, the total percentage of time taken by the non-parallel regions accounts for 63.3% of the total time of GraphFromFasta. At 192 nodes, the load imbalance especially in loop 2 leads to the share of the non-parallel regions decreasing. Our future work will also involve parallelizing other parts of GraphFromFasta, as well as reducing the load imbalance which will further help speed up the overall time of execution.

B. Hybrid (MPI+OpenMP) ReadsToTranscripts

Figure 9 shows the results of the hybrid ReadsToTranscripts, the second part of the Chrysalis module. We again use the sugarbeet dataset as the input. We also continue the mode of execution of running 16 threads per node, with a single MPI rank, which has been shown to give the best performance. We show the time taken in the main loop which was MPI-enabled, together with the total time taken in ReadsToTranscripts. Besides the MPI-enabled loop, ReadsToTranscripts also spends time in assigning k-mers to Inchworm bundles

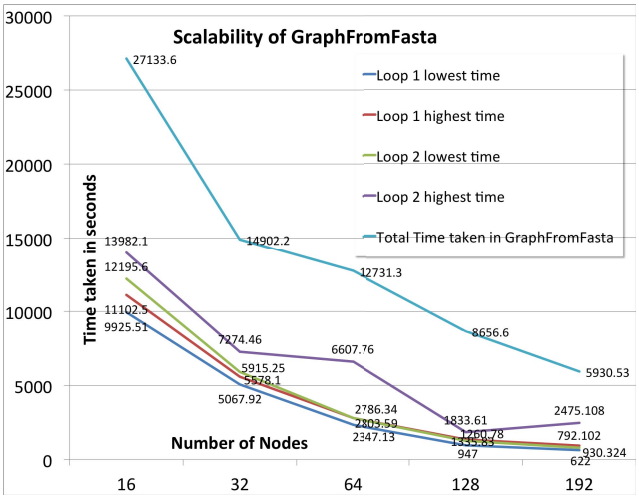

Fig. 7. Results of parallel (MPI+OpenMP) GraphFromFasta implementation showing the time taken in the loops and the total time taken in GraphFromFasta with increasing number of nodes.

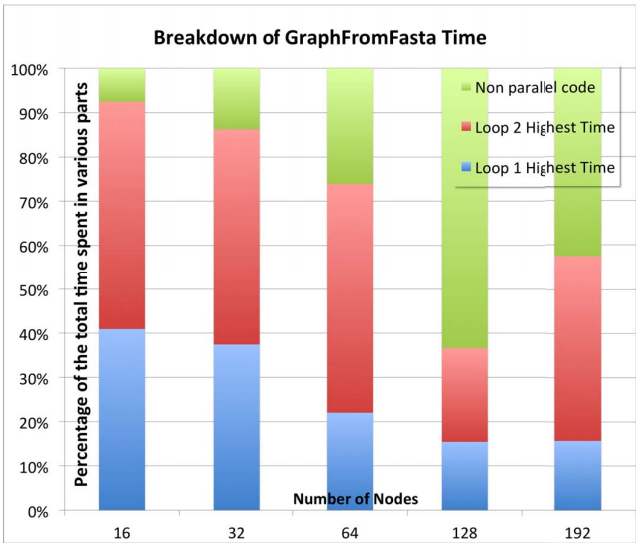

Fig. 8. Breakdown of GraphFromFasta times showing the times taken in loop 1, 2 and non-parallel regions. All times are normalized to 100%.

as well as concatenation of the separate files from every process. The assignment of k-mers to Inchworm bundles is OpenMP-enabled, and we have not converted this to a hybrid implementation yet.

At 32 nodes, the total runtime of ReadsToTranscripts takes less than 20 minutes: thus, this step of Chrysalis does not represent as significant a computational overhead as GraphFromFasta and Bowtie. The scalability of the MPI loop is almost linear, from about 3123 seconds on 4 nodes to less than 373 seconds on 32 nodes, representing a speedup of 8.37. At 32 nodes, the percentage of time spent in the MPI loop represents less than 20% of the total time spent in ReadsToTranscripts with the remaining time primarily taken in the OpenMP-enabled assignment of k-mers to Inchworm bundles. On a single node, the ReadsToTranscripts, using 16 threads took a total runtime of 20190 seconds. At 32 nodes, we thus achieve a overall speedup of 19.75 for the entire ReadsToTranscripts execution.

A very small percentage of the time is taken in the concatenation of the files from the multiple processes: this time stays constant (below 15 seconds) atleast up to 32 processes. As in Figure 7, we have also shown the processes with the highest and lowest times (373 and 310 seconds) spent in the loop: thus, compared to GraphFromFasta, the load imbalance in ReadsToTranscripts is much lower.

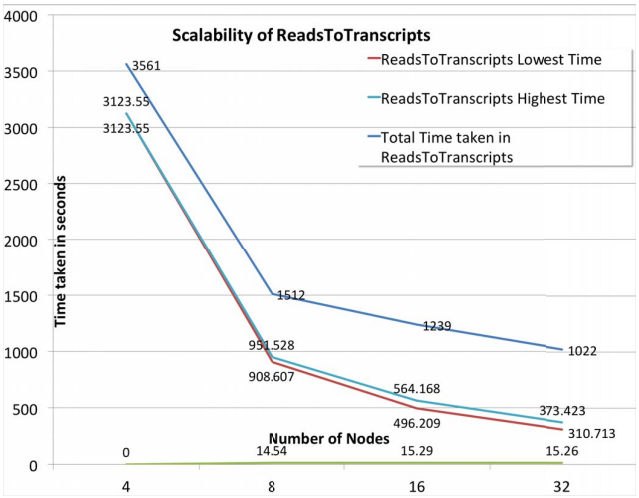

Fig. 9. Results of parallel (MPI+OpenMP) ReadsToTranscripts implementation showing the time taken in the main loop and the total time taken in ReadsToTranscripts with increasing number of nodes.

C. MPI Implementation of Bowtie

Figure 10 shows the results of a scaling experiment for the parallelized Bowtie again using the sugarbeet input dataset. This run was also completed using 16 threads, with one MPI rank per node. Since Bowtie with multiple nodes requires splitting the Fasta file of Inchworm contigs, we include runtimes for Fasta file splitting using PyFasta and the actual runtime for MPI-Bowtie, as well as the total Bowtie runtime. The figure shows that the splitting of the Fasta file using PyFasta took more runtime than the subsequent Bowtie step, partially due to the fact that PyFasta is a single thread process. We consider this step as a possible overhead to be worked on for better performance. In summary, we got a speedup of a factor of three when Bowtie was implemented in parallel using 128 nodes compared to single node implementation which took slightly more than 8 hours.

Overall, the Trinity workflow execution with the parallel Bowtie, GraphFromFasta and ReadsToTranscripts is shown in

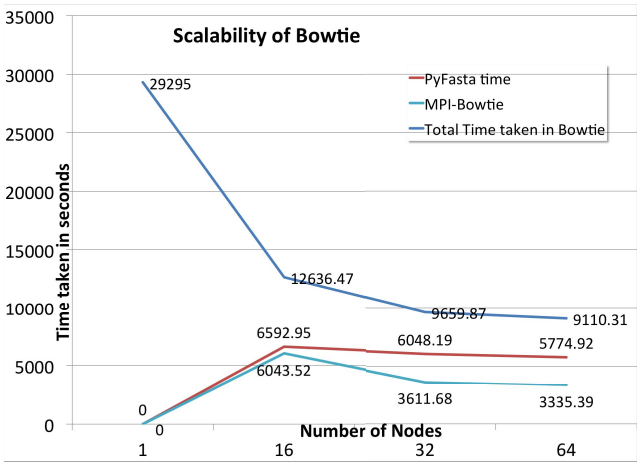

Fig. 10. Results of parallel Bowtie implementation showing the time taken in Bowtie and time taken by PyFasta to partition the Fasta file.

Figure 11. We again used the Collectl tool to collect statistics from the run, using 16 nodes, each running a single MPI process with 16 threads. This figure, compared to Figure 2, shows the substantially lower time taken in Chrysalis workflow using the sugarbeet dataset as the input.

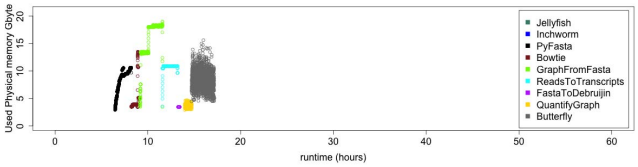

Fig. 11. Parallel Trinity run using 16 nodes, each with 16 cores and 128 GB of memory. Running instances of Inchworm/Jellyfish are not recorded for MPI-parallelized Trinity

VI. Conclusions and Future Work

Our efforts have so far focused on a distributed-memory implementation of Chrysalis components GraphFromFasta and ReadsToTranscripts, with the goal of producing an MPI implementation working seamlessly with the OpenMP threads that is already part of both the software modules. Overall for the sugarbeet dataset, we have reduced runtimes of GraphFromFasta and ReadsToTranscripts by over a factor of 20. The speedup comes essentially from the ability to use multi-node architectures, as widely available in traditional clusters, rather than relying on a single high-performance workstation. This fundamental change will allow Trinity to tackle the ever-increasing datasets that are being collected. We will continue our work by focusing on the non-parallelized regions of Chrysalis, as well as continue to investigate more optimal ways to partition the workload amongst the distributed nodes.

Reduction of the memory footprint of *de novo* transcriptome assembly is another active area of research we are pursuing. This covers the large memory footprint of the Inchworm module, as well as the per-node memory requirements of the MPI version of Chrysalis. Optimizing usage of NVRAM (non-volatile RAM) for the Trinity workflow and exploring MPI-I/O for RNA-Seq data are other areas where we are looking into. Overall, our focus is not just to optimize the individual components of the *de novo* pipeline, but to optimize the *de novo* workflow as a whole for a high-performance computing environment.

Acknowledgment

We would like to thank Brian Haas of the Broad Institute for his advice on Trinity software and source code, as well as providing the datasets for the validation of the hybrid parallelized Chrysalis. We thank Keywan Hassani-Pak of Rothamsted Research, UK for providing the sugarbeet dataset, as well as many useful discussions. We also wish to thank the Hartree Centre at the STFC Daresbury Laboratory, UK for providing the computational resources used for this work.

References

[1] K. Prufer and *et al*, "The complete genome sequence of a neanderthal from the altai mountains," *Nature*, 2013.
[2] Genomics England, "100k genome project," WWW page, http://www.genomicsengland.co.uk/100k-genome-project/.
[3] Sanger Trust Institute, "Cancer genome project," WWW page, http://www.sanger.ac.uk/research/projects/cancergenome/.
[4] S. Kamil, J. Shalf, and E. Strohmaier, "Power efficiency in high performance computing," in *Proceedings of the IEEE International Distributed and Processing Symposium*, Miami, FL, Apr. 2008.
[5] J. Simpson, K. Wong, J. Schein, and S. J. I. Birol, "ABySS: a parallel assembler for short read sequence data," *Genome Research*, vol. 19, no. 6, pp. 1117–1123, 2009.
[6] B. G. Jackson, P. S. Schnable, and S. Aluru, "Parallel short sequence assembly of transcriptomes," *BMC Bioinformatics*, vol. 10, no. Suppl 1, 2009.
[7] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies," *Journal of Computational Biology*, vol. 17, no. 11, pp. 1519–1533, 2010.
[8] M. G. Grabherr and *et al*, "Full-length transcriptome assembly from RNA-Seq data without a reference genome," *Nature BioTechnology*, vol. 29, pp. 644–652, 2011.
[9] Y. Wang and S. A. Smith, "Optimizing de novo assembly of short-read RNA-seq data for phylogenomics," *BMC Genomics*, vol. 14, no. 328, 2013.
[10] J. A. Martin and Z. Wang, "Next-generation transcriptome assembly," *Nature Reviews Genetics*, vol. 12, pp. 671–682, 2011.
[11] B. Inst., "RNA-Seq De novo Assembly using Trinity," http://trinityrnaseq.sourceforge.net.
[12] Q.-Y. Zhao, Y. Wang, Y.-M. Kong, D. Luo, X. Li, and P. Hao, "Optimizing de novo transcriptome assembly from short-read RNA-Seq data: a comparative study," *BMC Bioinformatics*, vol. 12, no. Suppl 14, 2011.
[13] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. R25, 2009.
[14] R. Henschel, M. Lieber, L.-S. Wu, P. M. Nasta, B. J. Haas, and R. D. LeDuc, "Trinity RNA-Seq assembler performance optimization," in *Proc. 1st Conference of the Extreme Science and Engineering Discovery Environment (XSEDE)*, Chicago, IL, 2012.
[15] T. Steijger, J. F. Abril, P. G. Engstrm, F. Kokocinski, T. J. Hubbard, R. Guig, J. Harrow, and P. Bertone, "Assessment of transcript reconstruction methods for rna-seq," *Nature Methods*, vol. 10, pp. 1177–1184, 2013.
[16] B. Li and C. N. Dewey, "RSEM: accurate transcript quantification from RNA-Seq data with or without a reference genome," *BMC Genomics*, vol. 14, no. 328, 2013.
[17] M. D. Robinson, D. J. McCarthy, and G. K. Smyth, "edgeR:a Bioconductor package for differential expression analysis of digital gene expression data," *BMC Genomics*, vol. 26, no. 1, pp. 139–140, 2009.

[18] B. J. Haas and *et al*, "De novo transcript sequence reconstruction from RNA-seq using the trinity platform for reference generation and analysis," *Nature Protocols*, vol. 8, pp. 1494–1512, 2013.
[19] G. Marcais and C. Kingsford, "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers," *Bioinformatics*, vol. 27, no. 6, pp. 764–770, 2011.
[20] G. Rizk, D. Lavenier, and R. Chikhi, "Dsk: k-mer counting with very low memory usage." *BMC Bioinformatics*, vol. 29, no. 5, pp. 652–653, 2013.
[21] Collectl, "Collectl Tutorial - The Basics," http://collectl.sourceforge.net.
[22] "PyFasta: A fast, memory-efficient, pythonic (and command-line) access to fasta sequence files," https://pypi.python.org/pypi/pyfasta/.
[23] D. Bozdag, A. Hatem, and U. V. Catalyurek, "Exploring parallelism in short sequence mapping using Burrows-Wheeler Transform," in *Proc. Int'l Parallel and Distributed Processing Symp. Workshops and Phd forum (IPDPSW 2010)*, Atlanta, GA, Apr. 2010, pp. 1–8.
[24] W. R. Pearson, "Searching protein sequence libraries: comparison of the sensitivity and selectivity of the smith-waterman and fasta algorithms." *Genomics*, vol. 11, no. 3, pp. 635–650, 1991.
[25] Evolution and Genomics, "Whitefly dataset," http://evomics.org/learning/genomics/trinity/.