Trinity RNA-Seq Assembler Performance Optimization

R. Henschel, M. Lieber, L. Wu, P.M. Nista, B.J. Haas, R.D. LeDuc

R. Henschel, M. Lieber, L. Wu, P.M. Nista, B.J. Haas, R.D. LeDuc,
Trinity RNA-Seq Assembler Performance Optimization,
in Proc. of XSEDE '12, "Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond", ACM, pp. 45:1-45:8, 2012

which has been published in final form (same content) at
http://dx.doi.org/10.1145/2335755.2335842

Trinity RNA-Seq Assembler Performance Optimization

Robert Henschel
Indiana University
2709 E. Tenth Street
Bloomington, IN 47401
henschel@iu.edu

Matthias Lieber
Technische Universität Dresden
01062 Dresden, Germany
matthias.lieber@tu-dresden.de

Le-Shin Wu
Indiana University
2709 E. Tenth Street
Bloomington, IN 47401
lewu@iu.edu

Phillip M. Nista
Indiana University
2709 E. Tenth Street
Bloomington, IN 47401
pnista@indiana.edu

Brian J. Haas
The Broad Institute
7 Cambridge Center
Cambridge, MA 02142
bhaas@broadinstitute.org

Richard D. LeDuc
Indiana University
2709 E. Tenth Street
Bloomington, IN 47401
rleduc@iu.edu

ABSTRACT

RNA-sequencing is a technique to study RNA expression in biological material. It is quickly gaining popularity in the field of transcriptomics. Trinity is a software tool that was developed for efficient *de novo* reconstruction of transcriptomes from RNA-Seq data. In this paper we first conduct a performance study of Trinity and compare it to previously published data from 2011. The version from 2011 is much slower than many other *de novo* assemblers and biologists have thus been forced to choose between quality and speed. We examine the runtime behavior of Trinity as a whole as well as its individual components and then optimize the most performance critical parts. We find that standard best practices for HPC applications can also be applied to Trinity, especially on systems with large amounts of memory. When combining best practices for HPC applications along with our specific performance optimization, we can decrease the runtime of Trinity by a factor of 3.9. This brings the runtime of Trinity in line with other *de novo* assemblers while maintaining superior quality. The purpose of this paper is to describe a series of improvements to Trinity, quantify the execution improvements achieved, and document the new version of the software.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*performance analysis*; J.3 [**Computer Applications**]: Life And Medical Sciences—*de novo DNA sequence assembly, RNA-Seq*

Keywords

application performance analysis, de novo DNA sequence assembly, RNA-Seq

XSEDE '12 Chicago, Illinois USA

1. INTRODUCTION

RNA-sequencing, or RNA-Seq, is a rapidly emerging family of laboratory techniques in the field of transcriptomics where expressed RNA is reverse-transcribed to cDNA, and sequenced. This technique offers a large number of advantages over older microarrays, and is rapidly replacing them in many applications [10, 12].

The National Center for Genome Analysis Support (NCGAS [5]) and XSEDE both support the biological research community, and specifically genomic analysis using second generation DNA sequencers. These sequencers produce an extremely large number of reads, often hundreds of millions, of up to 200 bases in length, with currently targeted datasets involving reads much shorter, such as Illumina 76 base paired-end reads from fragments ranging a few hundred bases in length. To be of value, these reads are typically assembled into contiguous reads, or sequence contigs. In the absence of a template genome, this assembly must be done *de novo*, using various implementations of a de Bruijn graph [7]. Since de Bruijn graphs are RAM intensive, many computing centers have deployed large memory computational clusters such as Mason at NCGAS and Blacklight at the Pittsburgh Supercomputing Center.

Over a wide dynamic range, the number of reads in a sequencing run is proportional to the concentration of the RNA species in the biological sample. This property is used in RNA-Seq to detect changes in RNA expression levels between biological treatments, but it complicates the assembly process. Applications developed for genome assembly perform poorly with samples containing a wide range of initial DNA concentrations. They do not deal well with the complexity of alternative splicing nor do they leverage strand specific sequence data. Therefore, custom assembly tools such as Trinity [8] have been developed to tackle the unique challenges posed by the assembly of RNA-Seq data. Seven different *de novo* sequence assemblers were evaluated with regard to both their biological and computational performance by Zhao et al. [14]. In this evaluation, Trinity was found to produce the best results for the single k-mer class of assemblers, but its runtime was so slow that the authors recommended using Trinity only when computational time was not an issue. Trinity possesses an active development and user base who are making frequent performance improve-

ments. We contributed to this development by analyzing performance modifications that specifically tune Trinity to run efficiently on large memory parallel computing clusters. We have added these enhancements to the official Trinity SourceForge repository and they are available for everyone to use in the 2012-06-08 version. The paper serves as a way for biologists to cite use of this newer, faster version while documenting that nothing has been changed in the algorithm to affect the assemblies it produces. The ability to run what is widely regarded as the best existing *de novo* RNA-Seq assembler, at an execution speed comparable or better than other assemblers will be of great benefit to bioinformaticians and practicing biologists.

We will finish this section by outlining the structure of Trinity and runtime performance of relevant versions and datasets. Section 2 describes optimizations that can be applied to the whole Trinity package without hurting performance, while section 3 outlines performance optimizations for specific components. These changes address component-specific performance bottlenecks.

1.1 Trinity Structure

Trinity is a Perl wrapper that calls separate applications, each of which passes results on to the next. The first application is Inchworm which uses a greedy search on a k-mer graph to assemble sequence contigs. The output FastA file is passed to Chrysalis which bundles the contigs and builds individual de Bruijn graphs. These graphs, one per file, are passed to Butterfly for computing the final assembly.

1.2 Initial Performance

To assess the baseline performance of Trinity, we used the exact *Drosophila melanogaster* data used by Zhao et al. [14]. The 76 bp paired-end Illumina reads were pre-processed and randomly subsampled to create six datasets of differing size. The complete set has 13.08 Gbp and 107 M read pairs, with subsets containing 7 Gbp and 58 M read pairs, 5 Gbp and 41 M read pairs, 3 Gbp and 25 M read pairs, 1 Gbp and 8 M read pairs, and 0.5 Gbp and 4 M read pairs. We obtained the data directly from the authors.

The performance measurements were conducted on a node of the Mason cluster at Indiana University [3]. A node is equipped with four Intel Xeon L7555 eight core processors running at 1.87 GHz for a total of 32 cores and has 512 GB of memory. The cluster is running Red Hat Enterprise Linux version 6. Each node has access to the Indiana University Data Capacitor Lustre filesystem [13]. By default, the memory filesystem `/dev/shm` is available on all nodes, offering up to 256 GB of very fast local scratch storage.

Figure 1 shows our initial performance observations. It compares the runtime of Trinity 2011-05-19 as observed by Zhao et al. (dotted red curve) with the same version on Mason (dashed green curve). In addition, we used the recent Trinity 2012-03-17 to process the same data (solid blue curve). For Trinity 2011-05-19 we used the same parameters as Zhao et al. (`--CPU 20 --run_butterfly --bfly_opts "--edge-thr=0.05 --compatible_path_extension"`). For the recent Trinity version, we used the parameters `--CPU 20 --kmer_method jellyfish --max_memory 4G --bfly_opts "--edge-thr=0.05" --min_contig_length 300`. These changes are a result of substantial changes in Trinity, for example the default minimum contig length has been lowered from 300 to 200 and Butterfly is now run by default.

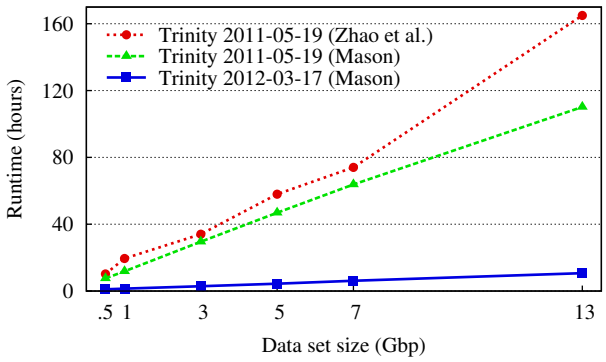

Figure 1: Runtime comparison using different datasets and versions of Trinity.

All tests were done using a default configuration of Trinity installed according to the documentation, using the default compiler on our system, GCC 4.4.6. Input and output data and all temporary files were located on the Data Capacitor. The Trinity 2011-05-19 runtime on Mason closely tracks that reported by Zhao et al. For the largest dataset, Mason performs marginally better, but still requires roughly 110 h. Trinity 2012-03-17 performs significantly better. The 13G dataset finishes in 10.7 h, about the same time as the 0.5G dataset on the old version of Trinity.

Trinity is actively developed and maintained as an open source tool [6]. All major components were updated between the two versions that are compared here. The modular nature of Trinity allows components to be easily changed or replaced. For example, the initial k-mer counting was previously done as first step within Inchworm, but is now implemented using Jellyfish [11] or Meryl [4]. Significant parallelism was also added between the two versions, for example components of Chrysalis now use OpenMP. In the older version, only Inchworm and Butterfly used parallelism, while in the newer version almost all phases are parallel.

2. GENERAL OPTIMIZATION

To characterize the workload of the Trinity workflow, we measured RAM usage, CPU utilization, and I/O throughput using the Collectl tool [2] at a sample rate of 5 s. Figure 2 (a) shows the results for Trinity 2012-03-17 with the 13G dataset and a maximum of 20 CPU cores. The plot shows an I/O intensive preprocessing step, which converts the input FastQ files to FastA format. Additionally, the Chrysalis phase includes four sub-phases, which are started with `system()` from within the Chrysalis program: GraphFromFasta groups contigs generated by Inchworm into clusters (bundles), FastaToDeBruijn creates de Bruijn graphs for each bundle, ReadsToTranscripts assigns the original reads to the bundles, and QuantifyGraph integrates the reads into the structure of the graphs. In Figure 2 the approximately 70 s of FastaToDeBruijn is included in the Chrysalis phase.

Figure 2 (a) also reveals that not all phases of Trinity make use of multiple cores. Multithreading is implemented using Pthreads in Jellyfish and using OpenMP in Inchworm and small parts of GraphFromFasta, as well as ReadsToTranscripts. QuantifyGraph and Butterfly are executed once for each independent bundle. Parallelism is achieved using an

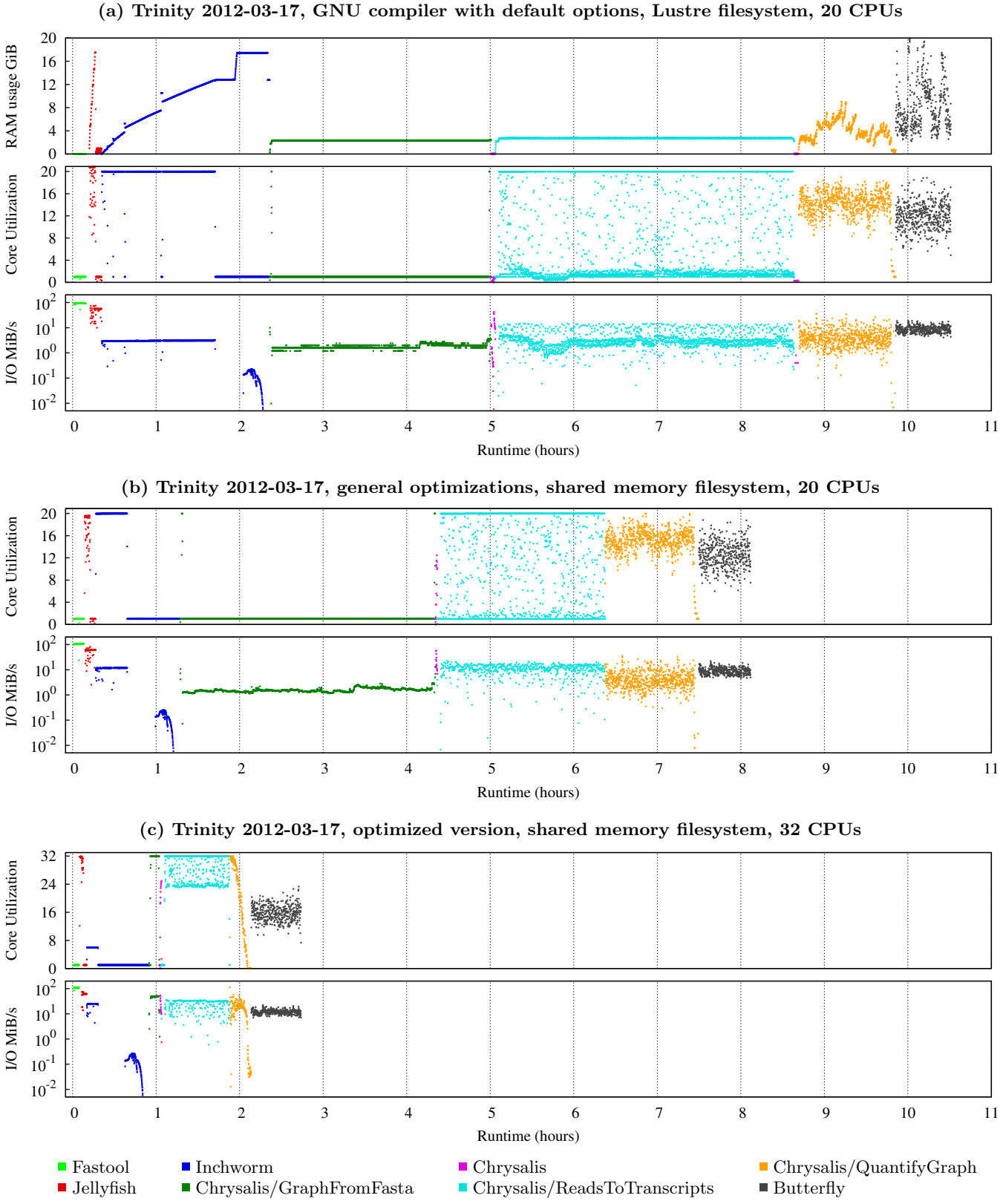

Figure 2: Measurement of RAM usage, CPU core utilization, and I/O throughput over the runtime of Trinity 2012-03-17 components with the 13G dataset. The RAM usage pattern did not change substantially and is shown for the original version only. Note that due to the sample rate of 5 s the shortest running instances of QuantifyGraph and Butterfly are not recorded, which leads to underestimated values.

OpenMP program, which starts QuantifyGraph and Butterfly in a parallel loop using `system()`. Note that Butterfly is written in Java, while all other phases are written in C++.

We started optimizing Trinity by employing general high performance computing best practices. We changed the location of the Trinity working directory to be located in a memory filesystem. This should benefit all I/O operations. Next we built all components using the Intel compiler and allowed the compiler to use more aggressive optimization techniques. Finally, we added thread and process pinning. This is especially important on machines that exhibit non-uniform memory access (NUMA) behavior.

2.1 Memory Filesystem

When possible, placing input and output files on a memory filesystem can greatly improve the performance of I/O intensive applications. Figure 2 (a) shows the I/O characteristics of Trinity. Up to and including GraphFromFasta, read and write operations are performed on a small number of large files. For these phases to run efficiently, the filesystem that holds the Trinity working directory needs to provide sufficient I/O bandwidth, effective file caching, and read ahead capability. Comparing Trinity running on Lustre with Trinity running on `/dev/shm`, a Linux implementation of a memory filesystem, we found no performance difference. For these phases, Lustre performs just as well as a memory filesystem since all actual I/O is hidden from the application by the filesystem cache, especially on systems with large amounts of memory. In contrast, ReadsToTranscripts and subsequent phases operate on a large number of small files. For these phases, the filesystem needs to provide sufficient metadata performance and to be able to deal with a large number of small files. For example, during the processing of the 13G dataset, 254 485 temporary files are created in 287 subdirectories. We found that ReadsToTranscripts benefits the most from running on a memory filesystem. For the 13G dataset, the execution time of ReadsToTranscripts is reduced by 43%, from 3h 33min to 2h 02min, when switching from a Lustre filesystem to `/dev/shm`.

However, even 512 GiB of memory are not enough to hold the Trinity working directory while providing enough heap space for Butterfly. For the 13G dataset, we reduced the default heap size from 20 GiB to 8 GiB and the number of Java garbage collection threads to 4. By default, the Java Virtual Machine (JVM) creates as many garbage collection threads as it finds cores in the machine, in our case 32. We noticed no performance impact by changing those settings.

In general, using a node local filesystem, like `/dev/shm` or `/tmp`, will reduce the dependency of a Trinity run on the performance of the cluster-wide filesystem. When running on a shared resource, the I/O bandwidth available to Trinity depends heavily on the I/O characteristics of other concurrent jobs. When the working directory is located on a node local filesystem, this dependency is removed and there will be less fluctuation in the execution time of a Trinity job.

2.2 Compilers and Thread Placement

By default Trinity builds all components with the GNU C and C++ compilers. The GNU compiler is a good default choice, since it is available on every platform. However, when users have access to other compilers, like the ones from Intel or PGI, they can offer performance advantages. Since our test system was based on an Intel Nehalem processor, we have tested the Intel compilers. We found that some components benefit greatly from using the Intel compilers while the runtime of others is unchanged.

We built all components of Trinity, except Jellyfish, with the Intel Compiler. This was implemented by either modifying the `configure` line of the component or changing the makefile. Whenever possible, we changed the compiler flags to `-fast`, which covers broadly applicable optimization flags, including interprocedural optimization and optimization specific for the microarchitecture, such as enabling SSE and AVX when available. Because Jellyfish called GCC specific functions, it was the only component that we did not build with the Intel compiler.

Thread placement and pinning can have a large effect on performance of parallel applications. In general, there are two strategies for placing processes and threads on a system with multiple sockets. The default placement strategy on Linux is load balancing, that is, processes and threads are placed on different sockets, so that the available memory bandwidth of the system is shared optimally. Alternatively with compact placement, a socket is filled first, before processes and threads are placed on the next socket. Depending on the needs of the application, either strategy can be optimal. During our analysis of Trinity, we found that thread placement has a noticeable effect on only Inchworm and Chrysalis. In Inchworm, forcing a compact thread placement is beneficial up to 8 threads, as outlined in section 3.1. On the contrary, forcing a compact thread placement for Chrysalis, which calls ReadsToTranscripts via the `system()` function, has a negative impact on performance. This is probably due to how thread placement is handled for child processes and requires further investigation. When running ReadsToTranscripts stand-alone, we observed no noticeable impact of different thread placement strategies.

Figure 2 (b) shows the runtime and I/O throughput of the individual components of Trinity with the general optimizations. Inchworm benefits from using the Intel compiler with the proper thread placement. Its runtime is cut in half compared to Figure 2 (a). ReadsToTranscripts benefits from running on `/dev/shm`. The I/O pattern looks smoother and the average throughput increases.

We checked the correctness of the results by comparing the output of the optimized version of Trinity with the output of the default version. We observed only very minor differences in the final result due to different compiler optimizations that do not affect the overall correctness of the assembly.

3. OPTIMIZING COMPONENTS

Based on Figure 2 (a), we analyzed and optimized the performance of the four most time-consuming components. From here on, we refer to the GCC compiled version of Trinity 2012-03-17 as the original version.

3.1 Inchworm

Inchworm works in four phases. First it reads a FastA file and produces a hashmap data structure representing each k-mer and its occurrence count. Second it prunes the hashmap to remove likely error-containing k-mers. Third it sorts the map and fourth it assembles and writes the contigs. The first phase is parallelized using OpenMP. Every OpenMP thread reads a part of the input data, extracts a k-mer and then inserts this k-mer into the hashmap. Figure 3 shows the runtime behavior of the 2012-03-17 version of Inchworm

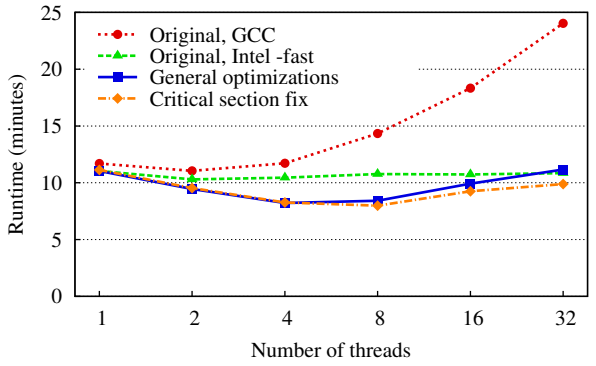

Figure 3: Scalability of original and optimized versions of Inchworm with the 1G dataset.

with the 1G dataset. Using the GCC default configuration, Inchworm runtime drops only slightly when increasing the number of threads from one to two, and increasing the number of threads further actually increase the runtime. At 32 threads, the runtime of inchworm has roughly doubled relative to the single-threaded case.

Inchworm scalability benefits greatly from the Intel compiler, especially when coupled with proper placement and pinning of the OpenMP threads. With the OpenMP runtime library of the Intel compiler, the execution time remains stable with increasing thread count. The best performance, when using `KMP_AFFINITY` and `numactl` to force a compact placement of threads, is achieved with four threads. The default thread placement policy on Linux is to maximize memory throughput and evenly spread threads. With four threads running on a four socket node, each thread is running on a separate socket. A side effect of this placement is that memory is also distributed across the system. For memory access that is not performed by the thread that has allocated the memory, this causes a 'remote memory access'. Since only the first phase of Inchworm is parallel, it is beneficial to force a memory placement that will stack threads on a socket until it is full before moving to the next socket. On our test system, this means that running with 8 or less threads, memory is optimally placed, both during the first phase as well as during the subsequent three phases. We found that a lot of time in the first phase of Inchworm is spent in OpenMP critical sections. Since the FastA file is read by all threads, each file access needs to be protected by a critical section. We changed the code slightly to fuse two critical sections, eliminating half of the critical sections in this phase. This allowed Inchworm to scale up to 8 threads. Table 1 shows runtime of just the parsing phase. With 8 Threads, the version with the critical section fix performs roughly 20% better than the general optimized version.

3.2 GraphFromFasta of Chrysalis

GraphFromFasta, the first subcomponent of Chrysalis, groups minimally overlapping contigs generated by Inchworm into bundles. The bundled contigs represent those that share subsequences as part of alternatively spliced variants or resulting from gene duplications. In relation to runtime, only very small parts of the code are parallelized with OpenMP. The longest serial task is counting the number of reads spanning the junction across two Inchworm contigs.

	Threads			
Version	**1**	**2**	**4**	**8**
GCC default	259	214	236	406
Intel -fast	282	198	210	224
General optimizations	280	171	106	118
Critical section fix	286	176	105	94

Table 1: Scalability of Inchworm parsing phase, runtime in seconds.

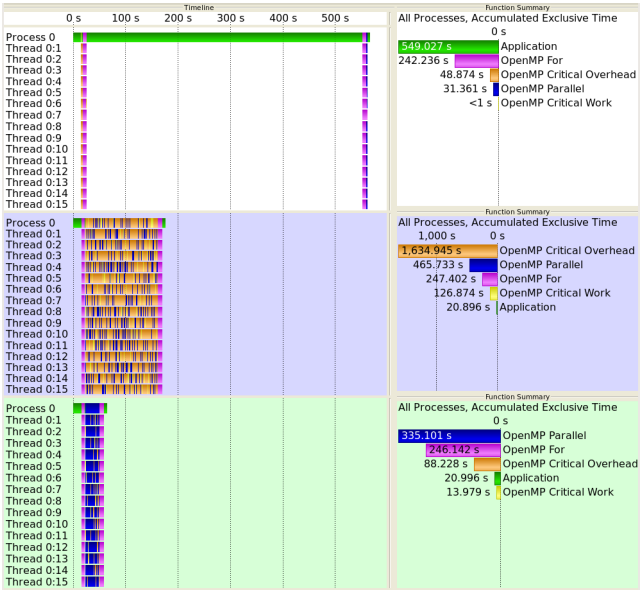

Figure 4: Vampir compare view of original GraphFromFasta (top white), an intermediate version (middle blue), and the final optimized version (bottom green) with the 1G dataset on 16 threads.

This counting phase is basically a while-loop over all input reads, which we parallelized by adding the `#pragma omp parallel` statement and protecting the actual file input with a critical section. The impact on performance is shown in Figure 4 by means of a compare view of the Vampir performance analysis tool [9]. The display comprises a timeline for all 16 threads on the left and a profile of serial application vs. OpenMP runtime on the right. The parallelization of the counting phase (second/blue chart) leads to a considerable speed-up over the original version (first/white chart), however, the critical section is clearly the bottleneck and strongly limits scalability. To increase speed-up, the time spent in the critical section had to be reduced. Therefore, we implemented a lightweight FastA file reader which stores the reads in a vector of `std::string` without the additional parsing performed in the original implementation. As shown in Figure 4 (third/green chart), the time spent in the active part of the critical section decreases approximately 9-fold, which leads to greatly reduced waiting times for the critical section and, thus, to a much more efficient parallelization of the whole phase.

GraphFromFasta was further optimized by the automatic proper selection of the chunk size for the dynamic schedule of both existing OpenMP loops, which eliminated the

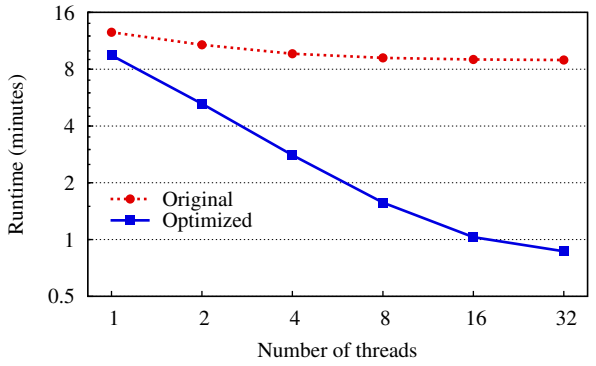

Figure 5: Scalability of original and optimized GraphFromFasta with the 1G dataset.

load imbalance, and improvements in adding entries to the dynamically growing vector of identified junctions between Inchworm contigs.

The final scalability comparison of GraphFromFasta with the 1G dataset is shown in Figure 5. Both versions have been compiled with the Intel compiler and `-O3 -xHost` optimization flags. While the original version only scales a little, the optimized parallel version shows a speed-up of 11 when increasing the core count from 1 to 32. Note, that speed-up is likely to increase with larger datasets, since the runtime per read within the counting phase increases with a larger number of junctions and, consequently, the scalability bottleneck of the file input is more and more hidden. The improved scalability leads to a 10-fold speed-up of the optimized version compared to the original for the 1G dataset.

3.3 ReadsToTranscripts of Chrysalis

ReadsToTranscripts is a subcomponent of Chrysalis which assigns individual reads to the bundles created by GraphFromFasta. It first builds a k-mer index of all bundles. Then it assigns each read to the bundle with the most k-mer hits and appends the reads to individual FastA files for each bundle. This procedure is carried out in several iterations, each processing 1 M successive reads from the input FastA file. The assignment as well as the output are parallelized using OpenMP parallel loops over the reads and the bundles, respectively. As shown in Figure 6, the serial input of the reads is apparently the main scalability bottleneck of ReadsToTranscripts. We applied the same optimizations as for GraphFromFasta by using a lightweight FastA file reader, resulting in an 8 times speed-up of the reading phase.

A second performance problem indicated in Figure 6 is the abnormally high runtime of file open and close calls beginning with the second output stage. This is probably due to an interference of I/O buffering within the C library and OpenMP threads. We reimplemented the output using the basic systems calls `open`, `write`, and `close`, which do not buffer I/O. To achieve high throughput, we buffered the output explicitly to call `write` only once per bundle and iteration. This optimization led to an approximate 30 times speed-up of the output phase, which is now hardly noticeable in the Vampir Timeline of the optimized version in Figure 6.

Figure 7 shows the final performance comparison of the original and optimized version of ReadsToTranscripts with the 1G dataset running on the memory filesystem. Both ver-

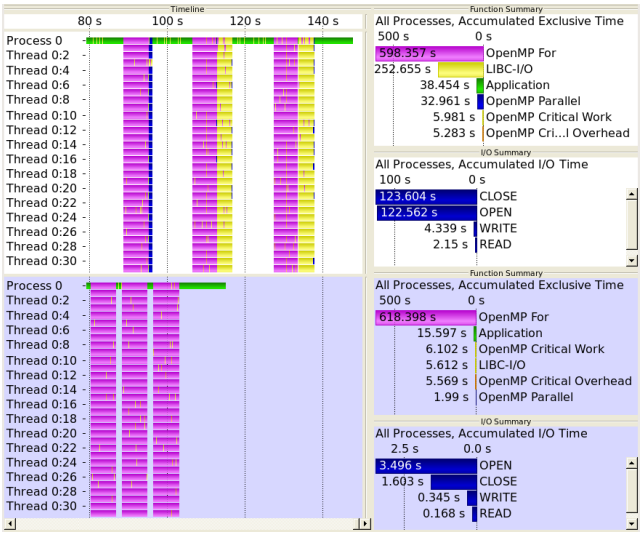

Figure 6: Vampir compare view of orignal ReadsToTranscripts and the optimized version with the 1G dataset on 32 threads for the first three iterations.

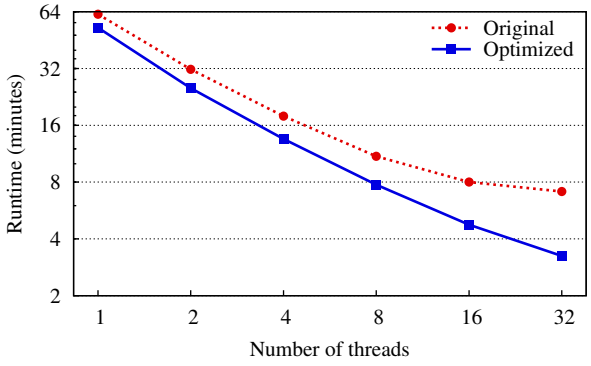

Figure 7: Scalability of original and optimized ReadsToTranscripts with the 1G dataset.

sions were compiled with the Intel compiler and the `-fast` optimization flag. While the optimizations have little impact at serial execution, the scalability of ReadsToTranscripts has improved clearly, resulting in less than 50 % runtime compared to the original version with 32 threads. The speed-up from 1 to 32 threads improved from 8.7 to 16.2.

3.4 QuantifyGraph of Chrysalis

QuantifyGraph is the last step of Chrysalis and integrates the original reads into the individual de Bruijn graphs of the bundles. It is started for each bundle independently by the ParaFly program, which uses `system()` within an OpenMP parallel loop to execute QuantifyGraph in parallel. For the 13G dataset, 23 107 bundles were created, which offers a large potential for parallelization. The runtime per instance depends on the number of reads mapped to the bundle. We obeserved runtimes between 160 ms (less than 10 reads) up to 25 min (3 M reads). QuantifyGraph consumes most of its time sorting the k-mers and then integrating the reads into the graph. In both phases, the relational operator '`<`' for

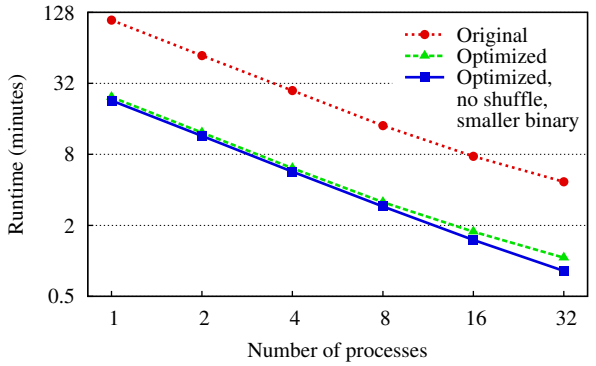

Figure 8: Scalability of original and optimized versions of QuantifyGraph with the 1G dataset.

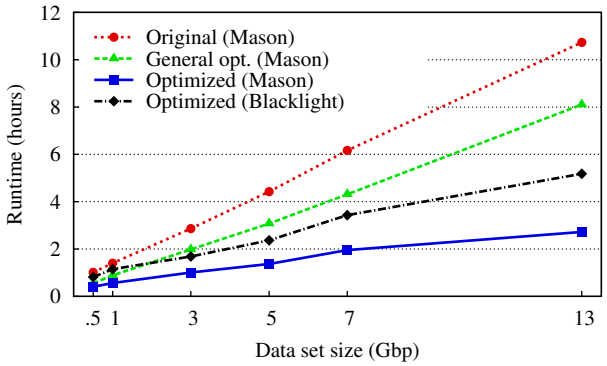

Figure 9: Trinity runtime comparison of original, generally optimized and fully optimized version.

the k-mer data structure is called frequently. Optimization of this operator led to a 2.8-fold speed-up of QuantifyGraph, except for very small bundles with less than 100 reads. The runtime for small bundles is dominated by parsing the reads from the FastA file and creating files indicating the start and successful completion of the QuantifyGraph instance. We optimized both by reducing the buffer size for reading a single sequence from the FastA file from 200 MB to 1 kB and by replacing the file operations based on `system()` with standard C library calls. With these optimizations, the runtimes of QuantifyGraph instances for the 13G dataset now vary from 5 ms (less than 10 reads) up to 9 min (3 M reads).

The parallel execution of QuantifyGraph can be optimized by switching off the shuffling of the individual instances. This yields better load balance, since the largest bundles are usually the first – a result of the greedy algorithm of Inchworm. Additionally, we reduced the startup overhead by creating smaller binaries. The size of the original static binary, compiled with the Intel compiler and the `-fast` optimization flag, was 4.1 MiB. We removed the `-openmp` flag, which is not required for QuantifyGraph, and linked the binary directly from the object files instead of linking them as static library. This reduced the size to 1.7 MiB.

Figure 8 compares the scalability of the original version of QuantifyGraph, the optimized version with only the source code changes, and the version with the additional optimizations. All versions have been built with the Intel compiler and the `-fast` optimization flag. QuantifyGraph shows a good scalability on all 32 cores of the cluster node. At 32 cores, the optimized version runs 5.7 times faster than the original. When only considering the source code modifications, the improvement factor is 4.4.

3.5 Other components

When Trinity is called with two FastQ input files for paired reads, they are converted to FastA files prior to the first phase. We modified `Trinity.pl` to perform this work in parallel, rather than sequential. If the underlying filesystem is capable of sustaining the required I/O bandwidth, the conversion will be sped up by roughly a factor of two.

Trinity allows for specifying the parallelism it uses via the command line parameter `--CPU`. However, internally it is capped at 22 and requires changes to `Trinity.pl` to override this. For all components except Inchworm, we have set the new maximum parallelism to 128. For Inchworm, we have capped it at 6, since we found that to be the optimal.

When using Jellyfish for k-mer counting, which is the fastest of all available choices, the parameter `--max_memory` is required. We found out that a value of `20GB` is optimal for the 13G dataset, which gives a runtime benefit of about 2 min in comparison to our initial setting of `4GB`.

3.6 Final performance comparison

Figure 9 shows the runtime of three different configurations of Trinity 2012-03-17 for all datasets. The dotted red curve shows the runtime of the original version, compiled using the default settings and running from a Lustre filesystem, as discussed in section 1.2. The dashed green curve shows the runtime of a generally optimized version. This includes all the optimizations discussed in section 2, such as running from a memory filesystem and building all components using the Intel compiler with a more aggressive optimization level. This basically covers all the optimization that can be performed without touching the source code or making in-depth modifications to `Trinity.pl`. The solid blue curve shows the runtime of our fully optimized version which builds on the generally optimized version and adds all the modifications discussed in this section. While the general optimizations show the highest impact with small datasets (1.8 times speed-up with 0.5G vs. 1.3 with 13G), the code optimizations greatly reduce the runtime especially for large datasets. For the 13G dataset, the speed-up over the general optimized and the default version is 3.0 and 3.9, respectively. For comparison, we conducted measurements on the XSEDE system Blacklight [1], an SGI UV 1000 shared memory system with 256 blades, each equipped with two eight core Intel X7560 processors. The runtimes for the fully optimized version, shown with a dash-dotted black curve, are 1.2 to 1.9 times slower compared to Mason. The slowdown is mostly due to the OpenMP parallel components of Trinity, which leads us to the assumption, that the NUMA transfer between the compute blades is probably the bottleneck. However, this requires further analysis.

The performance data for the 13G dataset presented in Figure 2 (c) reveals a noticable runtime improvement for all optimized components. The improvement factors are 2.3 for Inchworm, 21 for GraphFromFasta, 4.4 for ReadsToTranscripts, and 4.5 for QuantifyGraph. The great speed-up of GraphFromFasta is due to the parallelization of the initially mostly serial component.

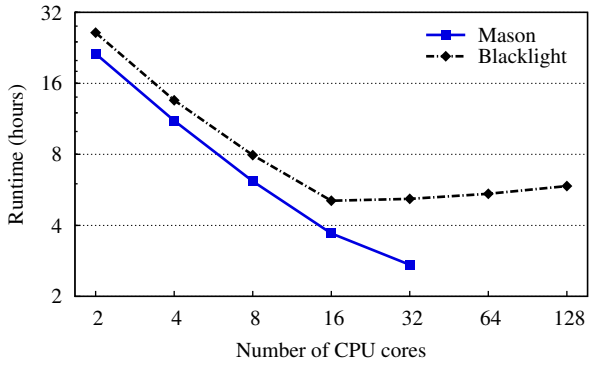

Figure 10: Scalablity of optimized Trinity with the 13G dataset.

Figure 10 shows the scalability of the optimized Trinity version for the 13G dataset on Mason and Blacklight. On Mason, Trinity scales well up to all 32 cores of a single cluster node, even though the scalability of Inchworm is limited to 6 threads. Blacklight shows a good scalability to 16 cores, a full compute blade. However, further scalability is probably inhibited due to heavy NUMA transfer between the blades.

4. DISCUSSION

Beyond the changes that we have already implemented, we see further potential for improving the performance of Trinity. In Inchworm, only the parsing phase is parallelized and the scalability is limited by the critical section protecting the file I/O. This could be improved by rewriting the `FastaReader` class to read and process larger chunks of input data. The pruning phase is currently implemented using an iterator of a hash map. If this could be changed to an iterator that is allowed for workshare directives in the OpenMP 3 specification, this phase could easily be parallelized.

Parts of Chrysalis use a `std::vector` of characters as container for DNA sequences. Replacing it with a `std::string` or a character array would improve the performance, since the number of conversions would be reduced.

For small contigs QuantifyGraph and Butterfly have very short runtimes per instance. Especially for Butterfly, instantiating a new JVM for an operation that only takes a few milliseconds causes a lot of overhead. This could greatly be reduced by parallelizing QuantifyGraph and Butterfly itself, so that only one instance is started.

For a better scalability on large shared memory systems like Blacklight, all OpenMP parallel components should be optimized for local memory allocation to reduce remote memory accesses and cache coherency protocol overhead.

A properly optimized instance of Trinity running on a large-RAM cluster is now a fast alternative for *de novo* sequence assembly of RNA-Seq data. In the past, Zhao et al. had concluded that Trinity was the best single k-mer type of assembler, but that it should be avoided when runtime was a constraining factor. The opposite advice now stands; Trinity should be selected when runtime is a constraining factor - and it remains the best single k-mer assembler. We have checked each optimization reported here to verify that the same sequences are returned by Trinity. Except as noted in section 2.2, all of our optimizations returned the identical sequences as the non-optimized version of Trinity 99.8 or greater percent of the time.

5. ACKNOWLEDGEMENTS

We would very much like to credit the other Trinity developers responsible for many of the major improvements in runtime efficiency realized since the 2011 version, including Michael Ott, David Eccles, Nathan Weeks, and Brian Couger, as well as the other initial Trinity developers including Alexie Papanicolaou, Rick Westerman, Moran Yassour and Manfred Grabherr. We would also like to thank the developers of third-party tools now leveraged as part of Trinity, including Francesco Strozzi for fastool, Brian Walenz for Meryl, and Guillaume Marcais for Jellyfish. We are grateful to Zhao et al. for sharing their filtered *Drosophila melanogaster* dataset. This work was supported in part by NGSgoesHPC funded by BMBF and NCGAS funded by NSF under award No. 1062432. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

6. REFERENCES

[1] Blacklight SGI UV 1000 at PSC. `http://www.psc.edu/machines/sgi/uv/blacklight.php`.

[2] Collectl. `http://collectl.sourceforge.net`.

[3] IU Mason Cluster. `http://pti.iu.edu/hps/mason`.

[4] K-mer Tools. `http://kmer.sourceforge.net`.

[5] National Center for Genome Analysis Support. `http://ncgas.org`.

[6] RNA-Seq De novo Assembly Using Trinity. `http://trinityrnaseq.sourceforge.net`.

[7] C. Geng, Y. KangPing, C. Wang, and S. TieLiu. De novo transcriptome assembly of RNA-Seq reads with different strategies. *Science China Life Sciences*, 54(12):1129–1133, 2011.

[8] M. G. Grabherr, B. J. Haas, M. Yassour, et al. Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nature Biotechnology*, 29(7):644–U130, 2011.

[9] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In M. Resch et al., editors, *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[10] J. Malone and B. Oliver. Microarrays, deep sequencing and the true measure of the transcriptome. *BMC Biology*, 9(1):34+, 2011.

[11] G. Marcais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[12] J. A. Martin and Z. Wang. Next-generation transcriptome assembly. *Nat Rev Genet*, 12(10):671–682, 2011.

[13] C. Stewart et al. MRI: Acquisition of a High-Speed, High Capacity Storage System to Support Scientific Computing: The Data Capacitor. `http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0521433`.

[14] 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*, 12(14), 2011.