View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by IUScholarWorks

Performance Optimization for the Trinity RNA-Seq Assembler

Michael Wagner, Ben Fulton, and Robert Henschel

Abstract Utilizing the enormous computing resources of high performance computing systems is anything but a trivial task. Performance analysis tools are designed to assist developers in this challenging task by helping to understand the application behavior and identify critical performance issues. In this paper we share our efforts and experiences in analyzing and optimizing Trinity, a well-established framework for the de novo reconstruction of transcriptomes from RNA-seq reads. Thereby, we try to reflect all aspects of the ongoing performance engineering: the identification of optimization targets, the code improvements resulting in 20 % overall runtime reduction, as well as the challenges we encountered getting there.

1 Introduction

High performance computing (HPC) systems promise to provide enormous computational resources. But effectively utilizing the computational power of these systems requires increasing knowledge and effort. Along with efficient single thread performance and resource usage, developers must consider various parallel programming models such as message passing, threading and tasking, and architecture specific models like interfaces to incorporate GPUs. Appropriate development devices such as performance analysis tools are becoming increasingly important in utilizing the computational resources of today's HPC systems. They assist devel-

Michael Wagner
Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden, 01062 Dresden, Germany
e-mail: michael.wagner@zih.tu-dresden.de

Ben Fulton · Robert Henschel
Scientific Applications and Performance Tuning
Indiana University, 2709 E. Tenth Street, Bloomington, IN, USA
e-mail: {befulton, henschel}@iu.edu

opers in two key aspects of program development: first, they help to analyze and understand the behavior of the applications on the HPC system and, second, they help in identifying critical performance issues.

In this paper we present our efforts to analyze and optimize the RNA-Seq assembler Trinity [4]. Trinity is a software tool that was developed for accurate de novo reconstruction of transcriptomes from RNA-Seq data. Early versions of the tool required a great deal of memory and performant hardware. As part of an ongoing process of performance improvement, we used Collectl, Score-P, and Vampir to identify bottlenecks in the pipeline, with diverse causes including memory contention, suboptimal I/O, and streaming inefficiencies. Armed with this knowledge, we were able to introduce modifications resulting in a 20% improvement in overall wall time.

In the following section we present the tool infrastructure that we used to gain insight into the application behavior and performance characteristics. In Section 3 we focus on the methods we used to understand Trinity's overall behavior and the behavior of the individual components. Furthermore, we will demonstrate the resulting optimizations in the Trinity pipeline. In section 4 we discuss certain challenges and restrictions we encountered while using various tools. Finally, we summarize the presented work and draw conclusions.

2 Tool Infrastructure

To better understand the runtime behavior of Trinity, to identify targets for performance optimization, and also to analyze the performance we used state-of-the-art performance tools: the system performance monitor Collectl, the event-based trace collected Score-P and the visual performance analyzer Vampir.

2.1 Collectl

Collectl is a popular performance monitoring tool that is able to track a wide variety of subsystems, including CPU, disk accesses, inodes, memory usage, network bandwidth, nfs, processes, quadrics, slabs, sockets and tcp [2]. It is additionally popular with HPC administrators for its ability to monitor clusters and to track systems such as InfiniBand and Lustre. Collectl works at a high level by sampling the system at intervals to determine the usage of each resource and logs the information to a file.

Collectl has long been incorporated into the Trinity pipeline to monitor various statistics at a coarse-grained level of detail. To minimize the effect on performance, Trinity runs collectl at a sampling rate of 5 seconds, rather than the default 1 second, and only monitors applications launched by the current user.

We extracted statistics from the collectl log generated by Trinity on RAM usage, CPU utilization, and I/O throughput, and created charts summarizing the use of

each individual application in the Trinity pipeline. (Figure 1) In order to determine the performance of each pipeline component, the totals were summed regardless of whether the component consisted of an single, multithreaded application, or multiple copies of an application running simultaneously. From these charts, we were able to crudely assess the relative amount of time each component used, as well as how effectively it made use of available resources.

2.2 Score-P and Vampir

For a more detailed analysis we chose the state-of-the-art event trace monitor Score-P and the visual analyzer Vampir. Score-P is a joint measurement infrastructure for the analysis tools Vampir, Scalasca, Periscope, and TAU [7]. It incorporates the measurement functionality of these tools into a single infrastructure, which provides a maximum of convenience for users. The Score-P measurement infrastructure allows event tracing as well as profiling. It contains the code instrumentation functionality and performs the runtime data collection. For event tracing, Score-P uses the Open Trace Format 2 (OTF2) to store the event tracing data for a successive analysis [3]. The Open Trace Format 2 is a highly scalable, memory efficient event trace data format plus support library.

Vampir is a well-proven and widely used tool for event-based performance analysis in the high performance computing community [6]. The Vampir trace visualizer includes a scalable, distributed analysis architecture called VampirServer, which enables the scalable processing of both large amounts of trace data and large numbers of processing elements. It presents the tracing data in the form of timelines, displaying the active code region over time for each process along with summarized profile information, such as the amount of time spent in individual functions.

3 Analysis and Optimization

The starting point for the optimization was Trinity 2.0.6 [5] which already contains a number of previous optimization cycles [8]. Trinity 2.0.6 is a pipeline of up to 27 individual components in different programming and script languages, including C++, Java, Perl, and system binaries, which are invoked by the main Trinity perl script. The pipeline consists of three stages: first, Inchworm assembles RNA-seq data into sequence contigs, second, Chrysalis bundles the Inchworm contigs and constructs complete de Bruijn graphs for each cluster, and, third, Butterfly processes the individual graphs in parallel and computes the final assembly.

3.1 Identification of Optimization Targets

Due to the multicomponent structure of Trinity, many performance analysis tools which focus on a single binary were unsuitable to gain a general overview on the Trinity runtime behavior. To better understand the runtime behavior and to identify targets for optimization, we conducted a series of of reference runs using Collectl to measure timings and resource utilization. Figure 1 depicts the initial performance of nine main components in Trinity 2.0.6, processing the 16.4 GiB reference data set of *Schizosaccharomyces Pombe*, a yeast, with 50 million base pairs on a 16-core node on the Karst cluster at Indiana University.

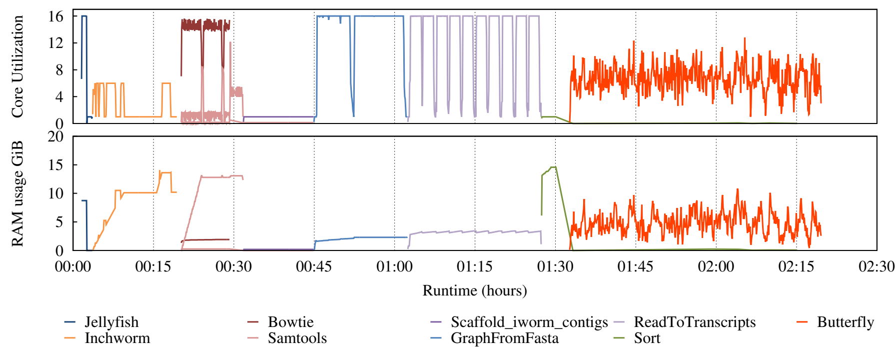

Fig. 1 Resource utilization of original Trinity 2.0.6. version.

Based on the CPU utilization of the individual components we identified Inchworm, scaffold_iworm_contigs, sort, and Butterfly to run in serial or with insufficient parallel efficiency. Inchworm has already been targeted for a complete reimplementation using MPI in a different group and, therefore, was not selected as optimization target again [1]. The optimization of scaffold_iworm_contigs is discussed in Section 3.2 and the optimization of sort is highlighted in Section 3.3. The second stage of Trinity processing primarily involves Butterfly. An optimization of Butterfly would have implied a complete restructuring of the Trinity code, which was infeasible due to Trinity's modular and constantly evolving pipeline. Nevertheless, the second stage recursively calls the main Trinity script, and therefore this stage benefits from our other optimization efforts as each individual de Bruijn graph is processed.

In addition to the obvious optimization targets, we discovered an overhead of frequent forking and joining of parallel regions in ReadsToTranscipts marked by the sharp drops of parallel CPU utilization in the collectl chart (Figure 1). The resulting optimizations are discussed in Section 3.4.

While collectl's CPU utilization displays insufficient multi-core usage it does not expose unbalanced parallel behavior, for instance, busy-waiting cores. Therefore, we analyzed the parallel scaling of the individual components to detect poor scaling components. Table 1 lists the parallel speedup of each component together

with its runtime share[1]. The runtime share was additionally helpful to prioritize our optimization efforts. Next to the above discussed components, Table 1 reveals poor scaling in GraphFromFasta hidden in the collectl graph, which results in over one third of runtime share. The in-depth analysis of the parallel behavior with Score-P and Vampir is detailed in Section 3.5.

Table 1 Parallel speedup of the main Trinity components together with their runtime share.

	Parallel Speedup					Component Runtime Share (in %)				
Number of Cores	1	2	4	8	16	1	2	4	8	16
Trinity (total)	1.00	2.07	3.80	6.22	8.78	100.0	100.0	100.0	100.0	100.0
Jellyfish	1.00	1.84	3.23	5.20	7.34	1.4	1.6	1.7	1.7	1.7
Inchworm	1.00	1.07	1.21	1.38	1.48	5.0	9.7	15.7	22.7	29.7
Bowtie	1.00	1.63	3.04	5.77	5.35	3.9	4.9	4.9	4.2	6.4
Samtools sort	1.00	1.64	2.95	5.09	5.01	4.7	6.0	6.1	5.8	8.3
Samtools view	1.00	1.44	2.11	2.86	2.78	4.9	7.1	8.9	10.7	15.6
Scaffold_iworm_contigs	1.00	0.99	0.99	0.99	1.00	1.0	2.2	4.0	6.5	9.2
GraphFromFasta	1.00	1.36	1.67	2.01	2.27	8.8	13.4	20.1	27.3	34.2
ReadsToTranscripts	1.00	2.31	4.08	5.97	9.42	39.6	35.6	36.9	41.3	36.9
Sort	1.00	1.13	1.05	0.81	1.00	0.4	0.7	1.5	3.1	3.6
Butterfly	1.00	1.40	2.33	4.52	7.00	28.0	41.3	45.7	38.5	35.1

3.2 Optimization of Samtools and Scaffold_iworm_contigs

Figure 1 shows a summation of the total CPU time used by each Trinity subcomponent. Thus, areas where only a very small number of CPU's were in use were of particular concern. One such area is evident from the 30 minute mark to the 45 minute mark. For approximately 10% of the total runtime, two processes ran: scaffold_iworm_contigs utilizing a single CPU and "samtools view" which used almost no CPU, and no other processes are running at that time. The minimal CPU usage seemed to indicate a prime location for performance improvement.

Investigating this area further, we found the following: A tool that had run earlier in the process (Bowtie) had produced a Binary Sequence Alignment/Map (BAM) file. The information in the file was extracted to text using the standard bioinformatics tool suite Samtools, and the text was processed by scaffold_iworm_contigs, a Perl script. Feeling that it would be more effective to parallelize the processing in C++, we ported the Perl script to a new code in that language. This, in turn, allowed us to take advantage of a Samtools C library which read the BAM file directly, and thus removed the need to extract the BAM file to text. After testing these changes,

[1] Since all components are called again during the Butterfly stage runtimes may deviate from the collectl chart in Figure 1.

we determined that the processing time required for this area of the application had fallen to 1% or less of the total time, equalling a runtime reduction of about 10 minutes. Almost the entire time spent in this area of the application had been spent in converting the binary file to text and processing the text. It was then felt that further parallelization of the code was unnecessary.

3.3 Optimization of Sort

The CPU chart in Figure 1 shows that sort runs for about 6 minutes to sort a large text file containing the distribution of reads to components. While Trinity already tries to utilizes the parallel version of sort, the collectl chart reveals that sort runs only in serial mode. On further investigation we discovered that all the reviewed systems that run Trinity in production mode use a sort from the GNU coreutils 8.4 version. However, a parallel version of sort was not introduced until GNU coreutils in the version 8.6 from October 2010. Consequently, we installed a current version of the GNU coreutils on the productive Trinity systems.

In addition, we discovered that the memory requirements of sort increase logarithmically with the number of parallel threads. As a result, the memory limitation that can be passed to Trinity with the option `--max_memory` can critically limit the performance of the parallel sort. For some data sets we recorded memory requirements larger than 200 GiB, which were available on the productive systems since Trinity requires large memory allocations in other stages, as well. Hence, we advise adapting Trinity's `--max_memory` option to the actual available memory to retrieve optimal performance in sort.

Applying both of these sorting optimizations, the parallel version of sort and a maximal memory limit, the runtime of sort is decreased from over 6 minutes to 40 seconds.

3.4 Optimization of ReadsToTranscripts

Next to the more obvious optimization targets, ReadsToTranscripts revealed an overhead of frequent forking and joining of parallel regions. Within the main loop a predefined number of reads is loaded and than processed in a parallel region. The repeatedly forking and joining of the parallel region can be seen by the sharp drops of parallel CPU utilization in the collectl chart (Figure 1) and causes unnecessary overhead due to thread creation.

By increasing the default number of reads to be loaded from 10 to 50 million the number of thread spawns was drastically reduced and the total runtime of ReadsToTranscripts was reduced by about 6 minutes. In addition, we identified an unnecessary entering of the parallel region if no additional reads were loaded, i.e., after all reads had been processes a final loop iteration work on zero input data. By leaving

the loop iteration directly when no additional reads are available, we additionally reduced the runtime of ReadsToTranscripts. This change is particularly effective in the Butterfly stage where only a few reads are processed and the main loop is only processed one instead of twice. This resulted in a minor decrease of the runtime of Butterfly of about one minute.

3.5 Optimization of GraphFromFasta

Along with the above discussed components, Table 1 reveals poor scaling in GraphFromFasta. Since collectl's CPU utilization (Figure 1) did show that all 16 CPUs are utilized most of the time, it could be inferred that the poor scaling is the result of ineffective parallel patterns. To further investigate the issue we recorded the parallel behavior with the event-based trace monitor Score-P and analyzed it with Vampir.

Due to the massive data volumes and application slow down involved with tracing we recorded and analyzed the GraphFromFasta component using only a small test data set. Our initial assumption that the poor scaling originates from a load imbalance between the OpenMP threads proved to be wrong, since only a little time was spent in OpenMP critical sections or synchronization operations[2].

To get further detail and reduce application slowdown, we recorded the application again without automatic compiler instrumentation but with manual instrumentation of the statements in the main loop. In addition, we recorded all OpenMP operations. Figure 2 shows the runtime behavior in comparison for one, two, four, eight and 16 threads in Vampir from top to bottom with white, red, yellow, green, and blue background, respectively. The left side depicts the active function over time one the horizontal axis and the different threads on the vertical axis. The accumulated exclusive runtime over all threads for each code region is represented in the function summary on the right side.

The analysis, represented in Figure 2, revealed that the work load in the first part of GraphFromFasta increased nearly linearly with the number of OpenMP threads resulting in practically no parallel speed up with more than two threads. The cause for this was the frequent creation and destruction of string stream objects within an inner loop of the frequently called function *is_simple*. The string stream creation was internally locked by a mutex which resulted in excessive wait time, since all threads simultaneously created the string stream objects with a very high frequency. This can be seen by the increasing amount of time spent in the code region creating the string stream object from about 25 seconds to 260 seconds.

By moving the string stream creation out of the inner loop and only clearing the string streams in the inner loop, we were able to avoid the serialization in this critical section. This resulted in a drastically increased parallel scaling and, therefore, a remarkable reduced runtime for the first part of GraphFromFasta. In addition to the better scaling, the serial runtime was reduced, as well; for the test data set, the serial

[2] Note: the load imbalance shown in Figure 2 only occurs for the small test data set. For regular data sets the load is almost balanced.

runtime was reduced from 72 to 45 seconds. Figure 3 shows the improved scaling of the optimized version in a Vampir timeline for a 1 GiB test data. In this case the parallel speed up was increased to 8.9 instead of 2.3 with the unoptimized version. For the *S. Pombe* data set with 50 million base pairs the runtime of GraphFromFasta was reduced from 18:26 to 4:40 minutes.

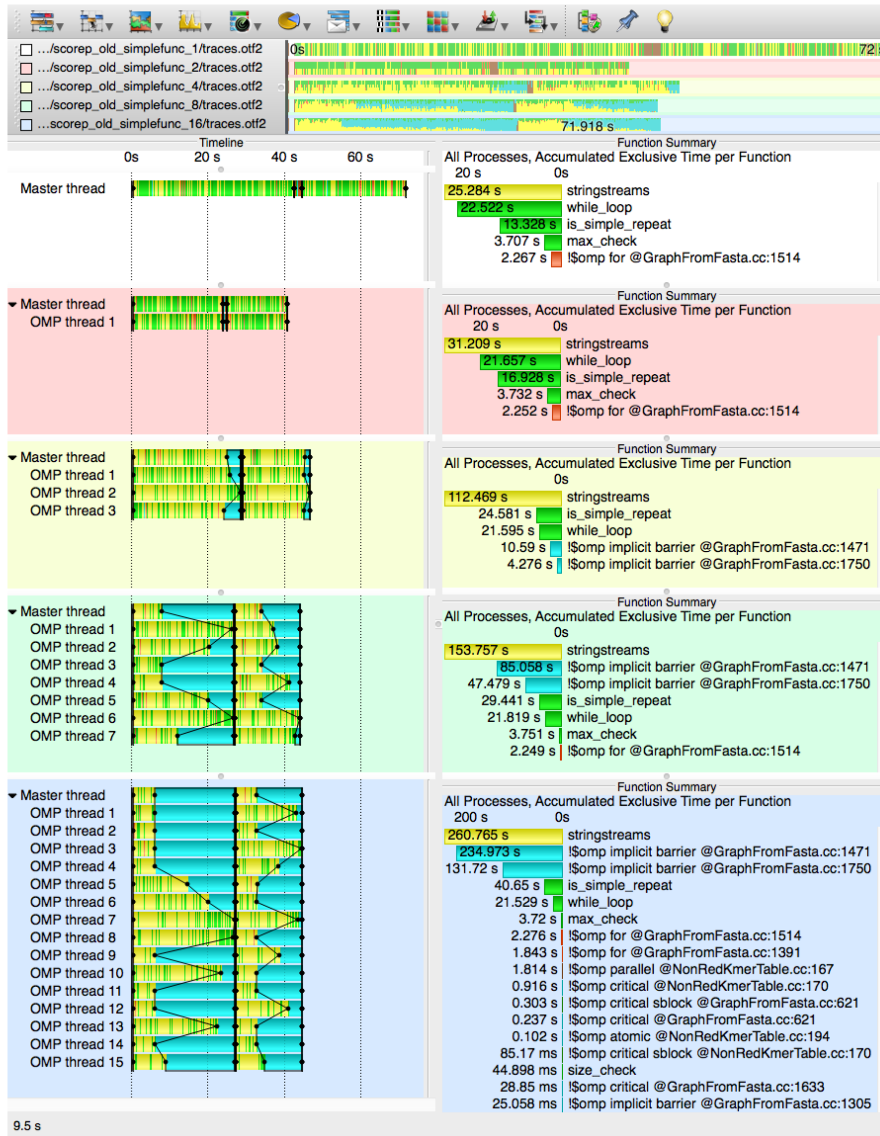

Fig. 2 Resource utilization of original Trinity 2.0.6 version for a small test data set.

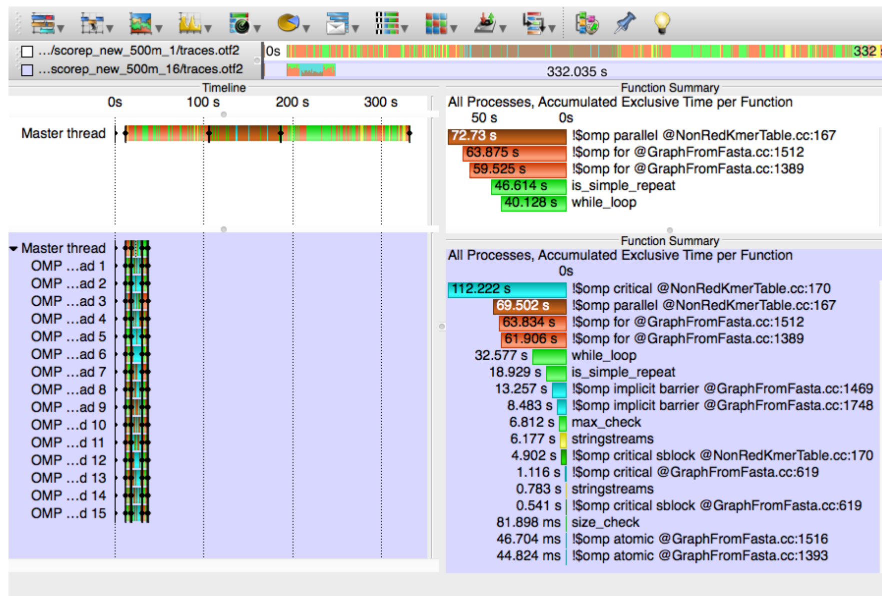

Fig. 3 Resource utilization of optimized Trinity version for medium size data set.

3.6 *Optimization Results*

In total, the optimizations reduced the initial runtime of the 50 million base pair *S. Pombe* data set from 02:20 hours by 31 minutes to 01:49 hours, which equals a reduction of 22 %. Figure 4 shows the resource utilization of the original Trinity 2.0.6 version in comparison to the version including the above described optimizations. It highlights the runtime reduction in scaffold_iworm_contigs due to the optimization in samtools, in GraphFromFasta, ReadsToTranscripts, and sort.

4 Tool Challenges and Restrictions

Although, performance analysis tools have been incredibly helpful in the above described optimization, their use was not without certain pitfalls and limitations. In following section highlight some of the major issues we encountered.

Collectl's five second interval biases timing of components. Running many of the tools required us to calculate a tradeoff: Was the amount of data that was collected sufficient to accurately determine performance bottlenecks, while at the same time being limited enough to find useful results? Collectl, for example, defaults to logging performance counters once per second. In order to minimize the impact on actual performance, the collectl monitor that is built into Trinity records only once

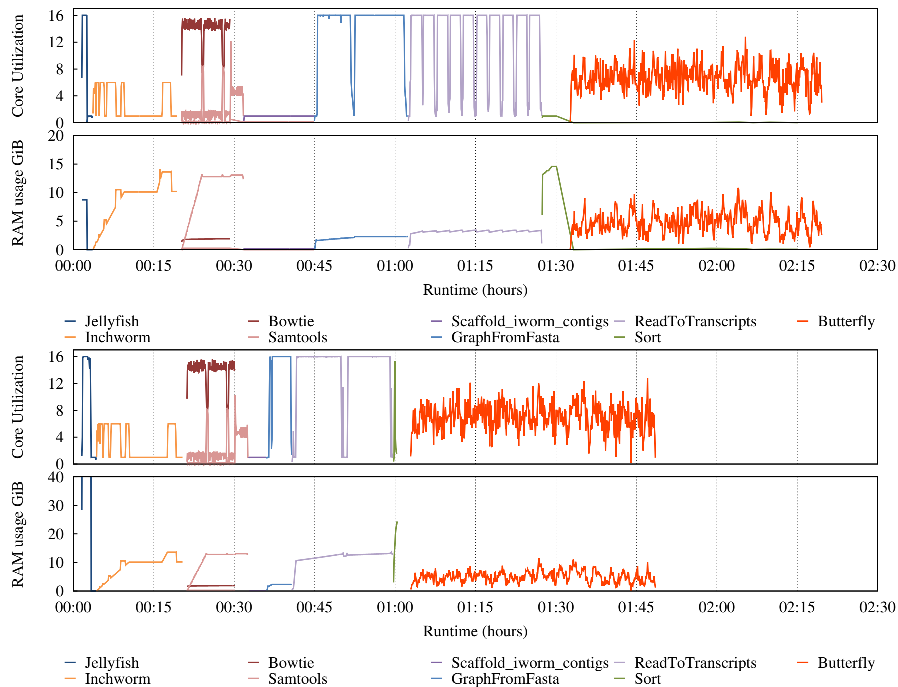

Fig. 4 Resource utilization of original Trinity 2.0.6 version (top) and optimized version (bottom).

every five seconds. We chose to keep this default. However, certain embarrassingly parallel sections of Trinity may launch codes that complete in less than five seconds, and the time spent in these codes may not register with Collectl. As a result, our analysis of the total CPU usage of, for example, Butterfly, may underestimate the actual CPU usage.

Busy-waiting is not detectable in collectl's CPU chart. A casual glance at the performance graph of Collectl is quite encouraging: several of the codes are displaying near-perfect usage of CPU. For example, the ReadsToTranscripts component, which runs from 40:00 to 60:00, shows very consistent usage of all 16 of the CPU's available. An analysis with a different tool shows different information, however: ReadsToTranscripts is spending a lot of time waiting in OpenMP critical sections. Many implementations of OpenMP use busy-waiting in critical sections, and Collectl is unable to differentiate between this busy-waiting and actual CPU time.

Instrumentation is difficult for modules written in nonstandard languages. Trinity is a multi-language application: many sections are written in C++, while others are written in Java. A few are also written in Python, and the "glue" which holds the pipeline together is written in Perl. However, most HPC performance analyzers focus on standard languages such as C/C++ or Fortran. Other than with tools rely-

ing on system information like Collectl, it was not possible to record nonstandard language modules or even the entire pipeline. Therefore, an in-depth performance analysis with Score-P and Vampir could only be applied to a subset of components. In addition, it was necessary to break down the Trinity pipeline in a way allowing to monitor the components individually.

Tracing results in massive data collection and application slowdown. An in-depth analysis of individual components was additionally constrained by the massive data collection and application slow down due to the monitoring with Score-P. These effects occurred especially during the monitoring of the C++ components; a well-known issue for programming concepts frequently using small helper functions or get and set class methods [9]. As result, we had to turn of automatic compiler instrumentation and use smaller test data sets to limit trace sizes and application slow down.

Only manual instrumentation helped identifying the problem. Event tracing tools such as Score-P rely on functions as smallest recorded entities. While this is sufficient for many applications, in codes using a flat call hierarchy this can lead to too little detail. During the analysis of GraphFromFasta, with function instrumentation we were able to narrow the issue down to function with 70 lines of code including two nested for loops, with only small clues as to where to look further. Only a manual instrumentation of individual statements revealed the thread contention during the creation of string streams highlighted in Figure 2. However, manual instrumentation requires detailed knowledge of both the monitored application and the trace monitor.

5 Conclusion

This paper highlights our efforts in analyzing and optimizing Trinity, a well-established framework for the de novo reconstruction of transcriptomes from RNA-seq reads. With the help of the performance tools Collectl, Score-P, and Vampir we identified bottlenecks in the pipeline, with diverse causes including thread contention, suboptimal I/O, and streaming inefficiencies. We optimized the runtime behavior of the components samtools, scaffold_iworm_contigs, in GraphFromFasta, ReadsToTranscripts, and sort. In total, the optimizations reduced the initial runtime of a 50 million base pair *S. Pombe* reference data set from 02:20 hours by 31 minutes to 01:49 hours, which equals a reduction of 22 %.

References

1. Pierre Carrier, Bill Long, Richard Walsh, Jef Dawson, Carlos P. Sosa, Brian Haas, Timothy Tickle, Thomas William: The Impact of High-Performance Computing Best Practice Applied to Next-Generation Sequencing Workflows. In: bioRxiv, http://dx.doi.org/10.1101/017665 (2015)
2. Collectl, http://collectl.sourceforge.net
3. Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf: Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In: Applications, Tools and Techniques on the Road to Exascale Computing, Advances in Parallel Computing 22, pp. 481–490 (2012)
4. Manfred G. Grabherr, Brian J. Haas, Moran Yassour, Joshua Z. Levin, Dawn A. Thompson, Ido Amit, Xian Adiconis, Lin Fan, Raktima Raychowdhury, Qiandong Zeng, Zehua Chen, Evan Mauceli, Nir Hacohen, Andreas Gnirke, Nicholas Rhind, Federica di Palma, Bruce W. Birren, Chad Nusbaum, Kerstin Lindblad-Toh, Nir Friedman, and Aviv Regev: Full-length Transcriptome Assembly from RNA-Seq Data Without a Reference Genome. In: Nature Biotechnology, 29(7):644–652 (2011)
5. Brian J. Haas: Trinity Release v2.0.6. http://github.com/trinityrnaseq/trinityrnaseq/releases/tag/v2.0.6
6. Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel: The Vampir Performance Analysis Tool Set. In: Tools for High Performance Computing, pp. 139–155 (2008)
7. Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Tools for High Performance Computing 2011, pp. 79–91 (2012)
8. Robert Henschel, Matthias Lieber, Le-Shin Wu, Phillip M. Nista, Brian J. Haas, Richard D. LeDuc: Trinity RNA-Seq Assembler Performance Optimization. In: Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond (XSEDE'12), pp. 1–8 (2012)
9. Michael Wagner, Jens Doleschal, Andreas Knüpfer, and Wolfgang E. Nagel: Selective Runtime Monitoring: Non-intrusive Elimination of High-frequency Functions. In: Proceedings of the International Conference on High Performance Computing & Simulation (HPCS), pp. 295–302, 2014.