Supplementary Materials
Holistic optimization of an RNA-seq workflow for multi-threaded environments

February 1, 2019

Appendix 1: Details of optimized implementation

Original implementation structure

The original implementation consisted of a master bash script to handle the setup of the directories and passing of parameters to the two Python scripts that performed the UMI based quantitation of the RNA-seq data. The output files were text files that had the counts for each gene. For the purposes of benchmarking, we moved the system call to BWA from within the Python script to a shell script. This allowed us to time and measure the memory consumption of that component separately.

Optimized implementation

The optimized workflow duplicated the bash script and replaced the Python scripts with compiled C++ executables that took the same arguments (and have some additional configuration options such as the number of threads).

The details of the implementation for the three components of the workflow, split, align, and merge follow:

- **Split:** The split executable reads and decompresses if necessary, a list of paired-end fastq files and splits them into separate fastq files with the sequence tag as part of the title line. Files are organized by the well as specified by the barcode in the sequence tag.

  The software asks the user for a file which lists the barcodes and wells that they map to. It also asks the user for the maximum allowable number of base pair variations due to sequence errors that can be tolerated when matching the wells. Since the barcodes are small (6 bases), the split software maps the sequence to an integer and generates a lookup table for all possible tags and maps them to a well when the identity of the well is unambiguous and the variation in base pairs is within tolerance.

All other sequences are mapped an "unmatched" well value. This is a constant time operation and is very fast due to the small size of the table.

The first read sequence is then read and the barcode matched. This is appended to the title line for the read. The actual sequence from the cDNA is read from the second read. The well is determined from the barcode and the newly constructed read is written to the corresponding fastq file. Unmatched barcodes are written to an unmatched fastq file. This task is parallelized using OpenMP. Each thread operates on a separate pair of input paired-end fastq files, and then generates a separate set of output well-specific files.

The major optimizations are in the parallelization at the level of input files, the fast lookup and resolution of barcodes and the separation of the reads by wells. The last step is a key optimization for later steps.

- **Align:** The align step is performed using a shell script. The script asks the user how many threads to use (parameter nThreads). The master thread assembles a list of all the fastq files from the split step (typically the number of pairs of input files times the number of wells). It then spawns nThreads processes that call BWA. Each thread goes through the list of fastq files, and checks whether another thread has created a lock file indicating it is working on that file. When finding a fastq file to work on, the worker thread writes a new lock file to prevent other threads from operating over the same file. The added parallelism increases the speed of the align step by 70% when using 16 threads.

- **Merge:** The original merge step did not have reads organized by file and compiled the counts using a single thread and a single large hash table. The new merge uses OpenMP to generate and manage threads that work simultaneously on different files.

  The division of files by wells also reduces the size of the hash (since the barcode part of the sequence tag is the same in each file) and the hash table (the number of reads is also reduced). The code further increases the efficiency of hashing by first mapping the sequence tag to an integer and using an unordered_set to check for uniqueness. The original implementation used the sequence concatenated with metadata to form a string that is hashed using a default Python text hashing function. Hashing of a long string requires several passes of the hash function and is slower than hashing a 64 bit integer. However, the major savings in speed and memory are due to the division of the data into files sorted by wells. This is especially important when 384 well plates are used instead of 96 well plates and when using many threads, which will consume more memory.

  A major implementation difference in the merge procedure is the filtering by UMI. The original UMI paper assumes that any additional reads with the same UMI that map to the same position are amplification artifacts and should be discarded. The merge module follows this recipe by default but also allows the user flexibility to exclude reads with the same

UMI that map nearby, which could happen, for example, due to minor sequencing errors or ambiguities. However, the original Python merge implementation excluded duplicate UMI's that mapped to the same gene even if they mapped to different positions. Our merge procedure does have a that flag that allows this alternative approach. Figure S1 in the supplemental materials compares the lists of differentially expressed genes that are obtained when using gene level and position level filtering of reads with identical UMI's.

Testing of optimized implementation

Testing of the optimized implementation is complicated by 3 factors. The first is the difference in the manner that UMI filtering is performed. The second is the fact that BWA will produce slightly different alignments even when the same reads are organized in different files. This is due to the way that it handles equally scoring alignments by choosing one at random. Finally, the implementations of the sorting by Python and C++ differ on how ties are handled.

During initial development, we split the files and filtered duplicate UMI's in the same manner as the original. This produced the identical counts as the original. Some results were sorted slightly differently due to the differences in sorting between Python and C++. However, the splitting and the filtering are key to the optimization and correctness of the new implementation and will produce slightly different results from the original implementation. Consequently, after these features were added, we only tested whether the optimized software produced the same results as previous implementations. For all benchmarks, the output files were compared to the single thread runs and were identical except for the order of items which had not been sorted and were written to the file in a different order by the different threads.

Appendix 2: Benchmark data

See Table S1 and Table S2 in the Appendix show the raw median data in Figure 1 in the main manuscript.

Table S1: Execution times in hours:minutes:seconds. The median execution time across three trials is shown.

Implementation	Threads	Split time	Align time	Merge time	Total time
Original	1	5:31:06	19:38:09	4:16:12	29:25:27
Optimized	1	3:14:51	17:10:16	0:50:37	21:15:44
Original	4	5:48:43	6:33:31	4:19:17	16:41:31
Optimized	4	1:04:18	5:06:22	0:17:59	6:28:39
Original	16	5:22:31	4:02:41	4:17:49	13:43:01
Optimized	16	0:53:02	2:24:58	0:12:43	3:30:43

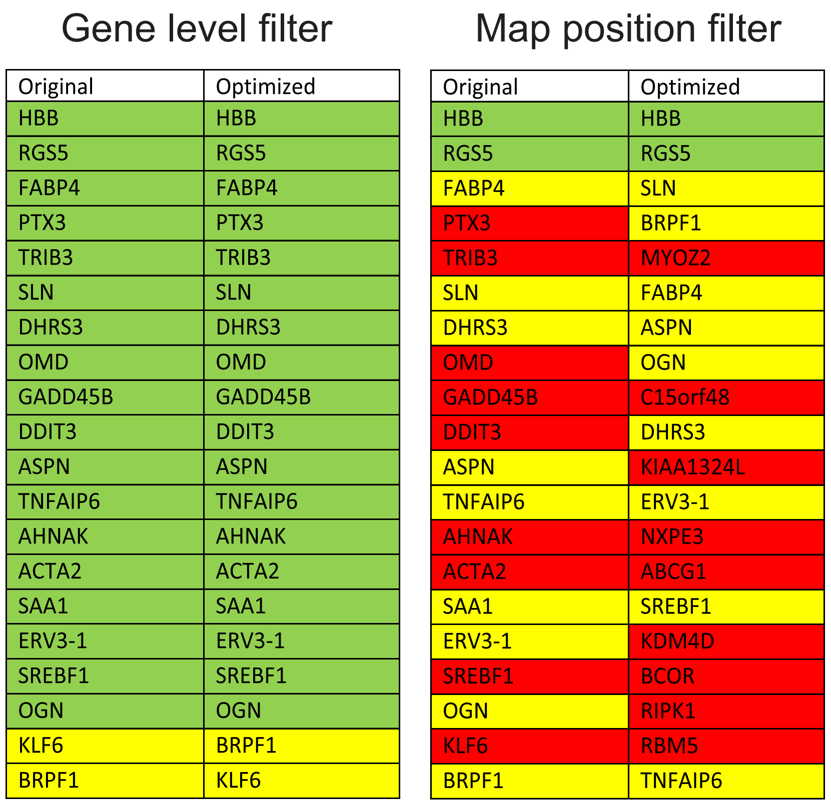
Table S2: Memory usage in Gigabytes. The median memory usage across three trials is shown.

Implementation	Threads	Split RAM	Align RAM	Merge RAM	Max RAM
Original	1	1.710	0.350	13.228	13.228
Optimized	1	0.008	0.351	0.923	0.923
Original	4	1.710	1.398	13.228	13.228
Optimized	4	0.021	1.402	1.477	1.477
Original	16	1.710	5.592	13.228	13.228
Optimized	16	0.069	5.609	3.571	5.609

Table S3: Effect of optimized computational module (combinations of split, align, merge) on total median run time.

Optimization	Total run time (16 Threads)
None	13:43:01
Split	9:13:32
Align	12:05:18
Merge	9:37:55
Split and Align	7:35:49
Split and Merge	5:08:26
Align and Merge	8:00:12
Split, Align and Merge	3:30:43

Appendix 3: Comparison of biological results



Gene level filter

Original	Optimized
HBB	HBB
RGS5	RGS5
FABP4	FABP4
PTX3	PTX3
TRIB3	TRIB3
SLN	SLN
DHRS3	DHRS3
OMD	OMD
GADD45B	GADD45B
DDIT3	DDIT3
ASPN	ASPN
TNFAIP6	TNFAIP6
AHNAK	AHNAK
ACTA2	ACTA2
SAA1	SAA1
ERV3-1	ERV3-1
SREBF1	SREBF1
OGN	OGN
KLF6	BRPF1
BRPF1	KLF6

Map position filter

Original	Optimized
HBB	HBB
RGS5	RGS5
FABP4	SLN
PTX3	BRPF1
TRIB3	MYOZ2
SLN	FABP4
DHRS3	ASPN
OMD	OGN
GADD45B	C15orf48
DDIT3	DHRS3
ASPN	KIAA1324L
TNFAIP6	ERV3-1
AHNAK	NXPE3
ACTA2	ABCG1
SAA1	SREBF1
ERV3-1	KDM4D
SREBF1	BCOR
OGN	RIPK1
KLF6	RBM5
BRPF1	TNFAIP6

Figure S1: Comparison of differentially expressed gene lists. As detailed in the SOP, EdgeR was used to obtain a set of the top differentially expressed genes in the presence and absence of the drug Trastuzumab. We applied the same analysis to transcript counts obtained from the Soumillon Python scripts and the transcript counts from our optimized software. The 20 genes that show the greatest difference in expression between control and treatment groups are listed. The Soumillon Python scripts excluded reads with identical UMIs that map to the same gene. The original method described by Bray *et al.* excludes identical UMIs that map to the same position. By default, the optimized software uses position based filtering but has the option (-g flag) for gene level filtering. We compare gene lists obtained from both types of filtering with the original gene list. Identically ranked genes are in green, genes that appear in both lists but in different order are in yellow and genes that appear in one of the two lists are in red. When gene level filtering is used by the optimized software, the results are almost identical to those obtained using the original Python scripts with only a transposition between the final two genes in the list. The small differences are due to actual differences from BWAs use of random numbers to choose between equally good alignments. This has been confirmed by manually examining the intermediate SAM files. When the default position based filtering is used there are significant changes in the identity and ordering of the top 20 genes. When coverage is large, we expect significant differences, as the gene level filtering results in exclusion of a greater number of reads than position level filtering.