White Paper
Intel Health & Life Sciences

Removing Load Imbalance in Burrows-Wheeler Sequence Alignment

Abstract

One of the initial goals of the ExaScience Life Lab is to examine how supercomputers can accelerate the processing of whole-genome sequences. Currently, the processing time of a single whole genome is measured in days rather than hours. Sequencing costs have decreased dramatically over the last years, and with the new generation of machines the mythical $1000 human genome has become reality in 2014. As this will have an immediate effect on the sample sizes used in sequencing studies, it is crucial to improve the efficiency of the computing process.

This article focuses on recent advances of the ExaScience Life Lab in optimizing the alignment phase of whole-genome processing. We show how the use of Intel® tools such as Pintools, VTune™, and Intel® Cilk™ allow analyzing and optimizing the performance of the widely-used BWA aln program for alignment. With minimal programming effort, we achieve up to factor two speedup compared to the original code, making the improvements ready to use in the software pipeline of Janssen Pharmaceutica today.

Charlotte Herzeel
ExaScience Life Lab, Leuven
imec

Joke Reumers
ExaScience Life Lab, Leuven
Janssen Pharmaceutica

Pascal Costanza
ExaScience Life Lab, Leuven
Intel, Belgium

Introduction

The whole genome of a single person can be represented by 3 billions of bases (A, T, G & C). The process of sequencing consists of chopping up these long strings into smaller strings, typically in the range of a few hundred bases (fragments), of which one or both ends are read by a machine. This results in millions of short strings called reads, which by themselves cannot be interpreted.

In order to get useful information from a DNA sample, its reads are matched against a known reference genome (see Figure 1). For example, this allows scientists to detect genetic differences between individuals with a disease and healthy controls.

The process of matching reads against a reference —also called *read alignment* — is a computationally challenging effort. The matching operation itself is costly as it is an inexact matching problem that is implemented by heuristic search algorithms.

Additionally, the volume of data that needs to be processed is large — roughly 300GB per sample today— and the demand for sequencing is only

Table of Contents

expected to increase as the price of sequencing technology continues to drop. Projects such as Genomics England[1] and the Saudi Genome Project[2] aim to sequence 100,000 individual genomes within the next few years.

Next, we discuss a performance analysis of BWA aln, one of the most widely used programs for sequence alignment. We observe that the BWA aln program exhibits poor scaling behavior when running in multithreaded mode. Our analysis of the parallelization strategy used in the code reveals that the program suffers from load imbalance, which is also confirmed by profiling experiments using Intel Pintools and VTune. We show that rewriting the parallelization using Cilk removes the *load imbalance*, improving the scaling behavior by a factor two [References 5,6].

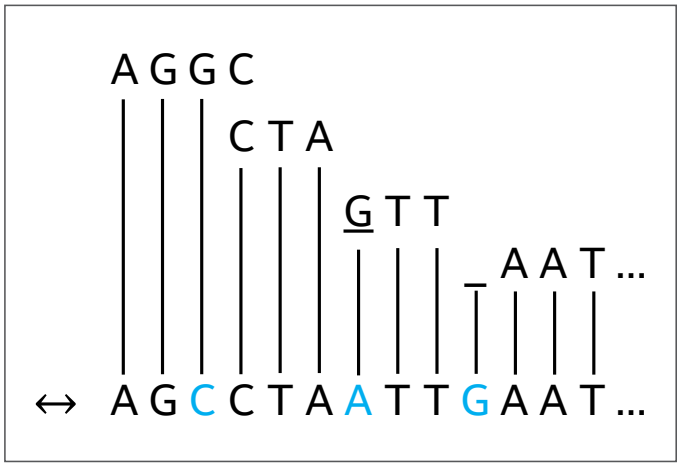


Figure 1: Matching reads against a reference

Burrows-Wheeler Alignment Program

The most prevalently used read aligner is the Burrows-Wheeler Alignment (BWA) algorithm developed by Li and Durbin [References 1,2]. Large genome centers around the world such as the Wellcome Trust Sanger Center and the Broad Institute recommend BWA as the aligner of choice for DNA sequence analysis, which together with its free and open source nature has led to a broad adaptation of the tool in sequence analysis pipelines at academic and commercial sites throughout the world.

BWA comes with different algorithms for read alignment, with BWA mem being the most recently introduced algorithm with improved performance and scalability for long reads. However, BWA aln, the original algorithm introduced in 2009, is still widely used for short reads, which is the focus of this paper[3]. The process of reference mappings lends itself easily to parallelization, as the mappings of individual reads are independent. Currently, BWA supports multithreaded execution when appropriate hardware resources are available.

As a litmus test, we set up a scaling experiment using different public workloads (see [References 5,6] for more details). The workload we use in this article is a read set from the 1000 Genomes Project[4] representing a low coverage whole genome sequence of a female individual of the International Hapmap project[5] .

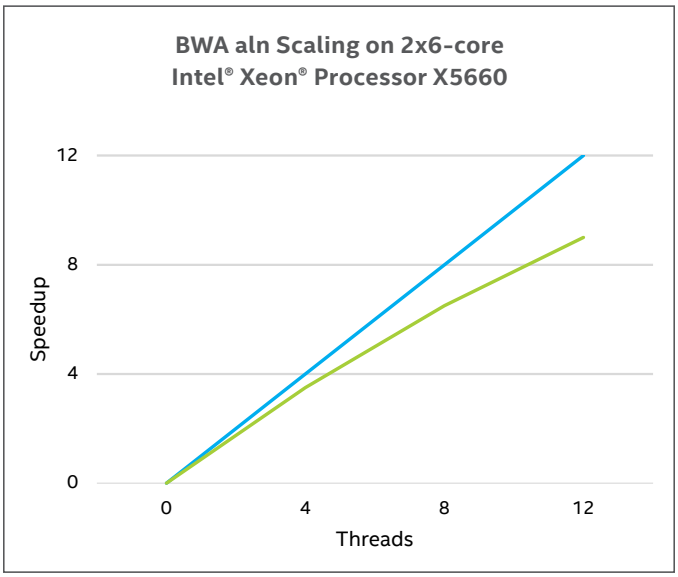


Figure 2: BWA aln scaling on 12 cores

Figure 2 shows the scaling of our workload[6] on a 12-core server equipped with two 6-core Intel® Xeon® X5660 processors. BWA aln does not achieve linear speedup. At 12 threads, we measure that BWA aln achieves 9x speedup, 73% of the potential (linear) speedup on this machine.

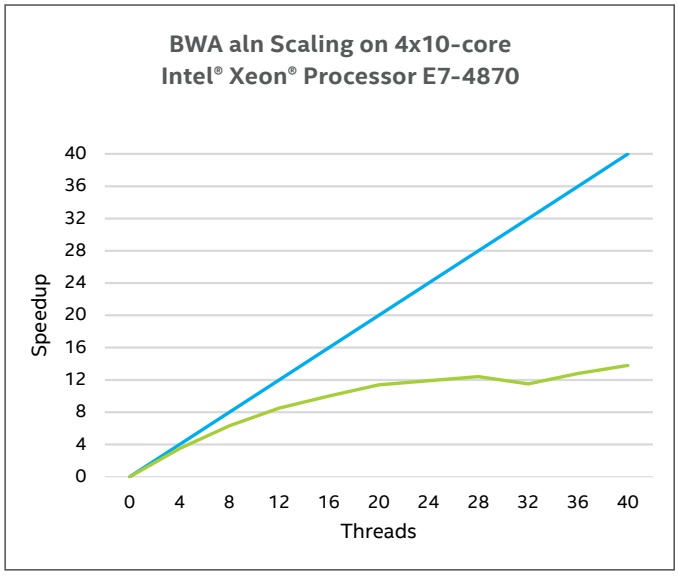


Figure 3: BWA aln scaling on 40 cores

The scaling behavior of the program worsens as the number of sockets and cores increases. Figure 3 shows the scaling behavior of our workload on a 40-core server equipped with four 10-core Intel® Xeon® processor E7-4870 product family-based platforms. The speedup measured for 40 cores is a factor 13.9x, only 35% of the machine's potential.

The reason the BWA aln program scales poorly is because its pthread-based implementation suffers from load imbalance.

Original pthread-based Parallelization

Read alignment is an embarrassingly parallel problem. In a typical workload, millions of reads, up to 100 base pairs (= 100 characters) long need to be aligned. Reads are aligned independently from one another, and hence read alignment can be parallelized as a data parallel loop over the read set.

Concretely, the BWA aln code [Reference 2] sets up pthreads equal to the number of cores available on the target machine. The reads are accessed from file and processed in chunks of a fixed size. Each of these chunks is distributed across the threads using a round-robin pattern. Once the reads are distributed, each thread privately aligns its own set of reads. When all threads are finished aligning, the results are written to file, and the next chunk of reads is fetched for processing.

Linear speedup for an implementation such as this is only guaranteed if the work to be done is roughly equal for each pthread, in order to avoid load imbalance.

Measuring Load Imbalance

Load imbalance occurs when certain threads are assigned a heavier workload than others. In such a situation certain threads become idle after some time while others are still processing, causing suboptimal use of the available hardware resources.

To detect whether the BWA aln program suffers from load imbalance, we measure the average execution time spent per thread as follows. Per parallel phase, we measure the time each thread executes. We sort these timings for each phase and combine the phases by pairwise adding the timings according to their length, i.e. add up the smallest timings, then add up the second smallest timings, and so on. This way we measure the load imbalance accumulated over the different parallel phases.

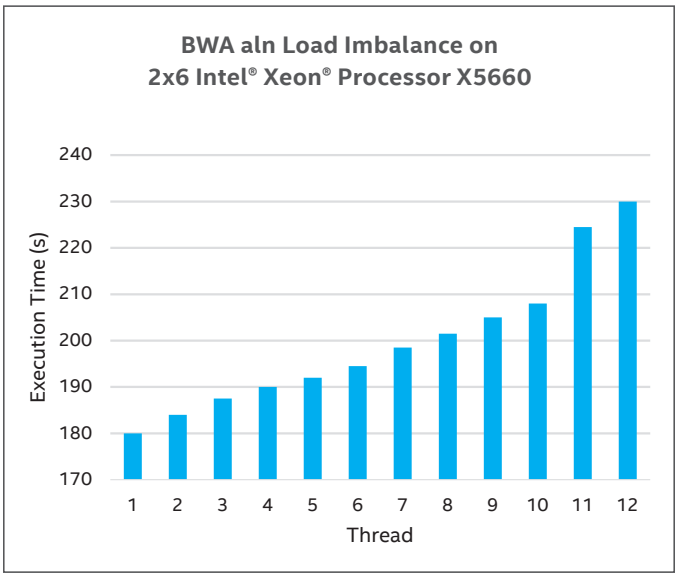


Figure 4: BWA aln load imbalance on 12 cores

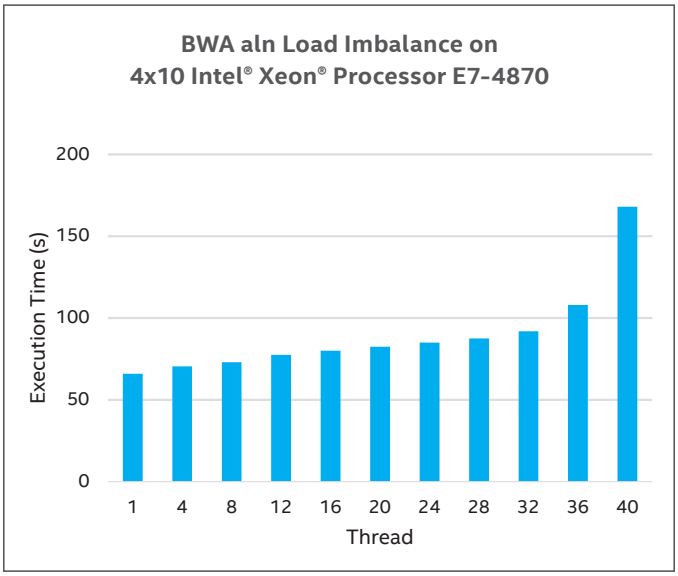


Figure 5: BWA aln load imbalance on 40 cores

Figures 4 and 5 show the average time spent per thread for our 12-core and 40-core server respectively. Both graphs indicate load imbalance. On our 12-core server, the slowest thread takes 1.3 times longer to execute than the fastest thread. For the 40-core server, the difference between the fastest and slowest thread is a factor 2.5.

Verifying Load Imbalance using Pintools

To figure out whether the observed load imbalance is inherent to the program or related to OS jitter or hardware effects, we set up an experiment where we measure the number of instructions and time spent per read using Intel Pintools [Reference 7].

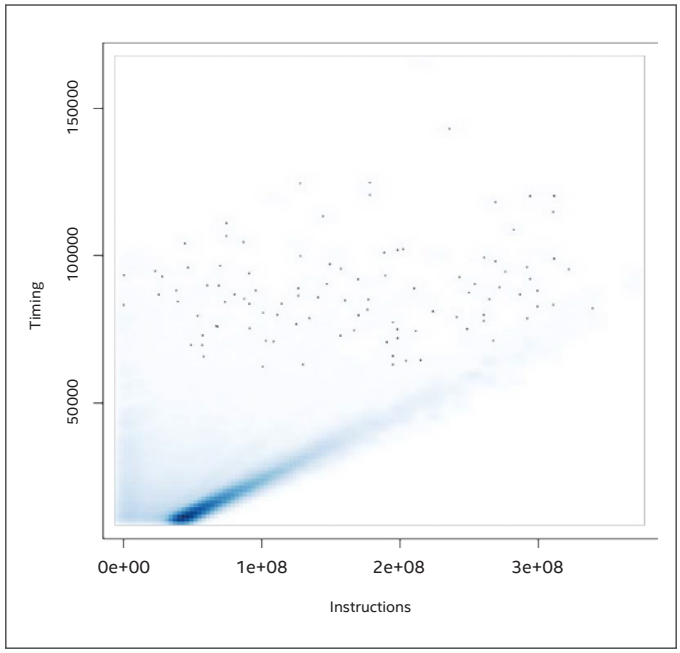


Figure 6: BWA aln Load Instruction Counts

Figure 6 shows a scatterplot for the 1000 Genomes workload that maps the number of instructions per read onto the time measured to execute those instructions. The graph shows the measurements for 10 runs. It is a smooth scatterplot: The color intensity scales with the number of data points.

We observe the following:

1. The number of instructions spent for aligning a particular read is deterministic: It is always the same across different runs.
2. The time spent for aligning a particular read varies across runs but is mostly stable. Roughly 81% of all reads have a runtime that varies less than 3ms across 10 runs. The larger variations are likely due to (unpredictable) hardware effects such as cache misses, branch mispredictions, NUMA effects, OS jitter, etc.
3. The number of instructions needed to align a particular read highly depends on the read. For example, in case of the 1000 Genomes workload, the *shortest* read requires 23,268 instructions to be aligned whereas the longest read needs 370,734,262 instructions to be processed. This is a factor 16,000x difference. This is not an outlier: Figure 6 shows that the reads are widely spread across these bounds.
4. Unsurprisingly, the more instructions a read requires to be aligned, the more time it takes to process. Figure 6 shows a clear uphill pattern as we go from left to right. The correlation factor is 0.72, indicating a strong linear relationship between the number of instructions per read and the time spent per read.
5. Figure 6 also shows that the time spent on a particular read is not always entirely determined by the number of instructions it requires. The cloud of dots above the linear curve in Figure 6 indicates certain reads need additional time to process compared to reads that require similar amounts of instructions. These outliers are likely due to hardware effects such as cache misses, branch mispredictions, etc. This is actually the same point as observation number two.

It is clear from this that different reads need different numbers of instructions/execution time to process, and these differences contribute to the overall load imbalance we observe when running the BWA aln program.

Simulating and Predicting Load Imbalance

We can predict the load imbalance caused by the reads having different processing requirements by simulating how the BWA aln program distributes the reads across threads and counting the number of instructions each thread receives this way. If we then compare the thread with the smallest number of instructions to the thread with the largest number of instructions, we get a factor that tells us how many more instructions the *slowest* thread needs to execute compared to the *fastest* thread.

Table 1 displays the load imbalance factors we compute this way for the 1000 Genomes workload for simulations with different numbers of threads. One observation is that the expected load imbalance goes up as the number of threads increases. This is logical since the size of the workload always stays the same.

In our experiments, we observe that the measured load imbalance deviates somewhat from the predicted load imbalance. This is due to the fact that our simulation does not take into account hardware effects such as cache misses, branch mispredictions, and so on. This is worst in case of the 40 threads predictions/measurements because our 40-core server is a 4-socket machine that suffers more from NUMA effects than our 12-core server with 2 sockets.

#THREADS	EXPECTED
1	1.0
4	1.05
8	1.09
12	1.14
16	1.18
32	1.30
40	1.36
64	1.53
128	1.95

Table 1: Expected load imbalance based on simulating read distributions.

The important conclusion to take away from Table 1 is that, purely based on instruction counts, there is an uneven distribution of the workload amongst threads, which leads to load imbalance in the BWA aln program because its pthread-based implementation does not anticipate this effect.

Removing Load Imbalance using Cilk

Intel® Cilk™ Plus [References 3, 4] is an extension for C/C++ for task-based parallel programming. It provides constructs for expressing fork/join patterns and parallel for loops. These constructs are mapped onto tasks that are executed by a dynamic work-stealing scheduler. With work stealing, a worker thread is created for each core. Every worker thread has its own task pool, but when a *worker thread* runs out of tasks, it *steals* tasks from worker threads that are still busy. This way, faster threads take over work from slower threads, automatically balancing the overall workload.

Cilk-based Parallelization

By using Cilk, we are able to drastically improve the scaling behavior of the BWA aln program. The code changes necessary to switch to using Cilk are minimal.

Concretely, we replace the for loop that spawns pthreads by a Cilk for loop. However, there are some intricacies with making sure that each worker thread has its own private data structures for implementing the matching process, in order to avoid contention. Our solution is to initialize these data structures per worker thread and store them in a globally accessible array before the for loop is executed. Inside the for loop, we then use Cilk's introspective operator to query the running worker thread's ID, which we use for indexing the global array to access the worker's private data structures.

Improved Scaling Results

Figure 7 compares the scaling of our Cilk version of BWA aln (green) with the scaling of the original pthread-based implementation (orange) on our 12-core server equipped with two 6-core Intel Xeon X5660 processors. To allow a direct comparison, our speedup graphs use the same baseline: 1-threaded execution of unmodified BWA aln. With the Cilk version, we achieve a factor 10x speedup or 82% of the potential linear speedup (blue) compared to 9x speedup or 73% efficiency for the unmodified BWA aln code.

The results for our 40-core server consisting of four 10-core Intel Xeon processor E7-4870 product family-based platforms are even better, as shown in Figure 8. Our Cilk version achieves 30x speedup or 75% of the machine's potential (green). The original pthread-based code achieves only 13.86x speedup or 35% efficiency (orange).

Summary

The multithreaded mode of BWA aln, a widely used program for sequence alignment, scales poorly on modern multicore servers. When profiling the code, we observe the program suffers from load imbalance. Using Intel Pintools, we set up an experiment that counts the numbers of instructions that are executed per thread. This analysis reveals the program distributes the workload unevenly across the threads, which inherently leads to load imbalance.

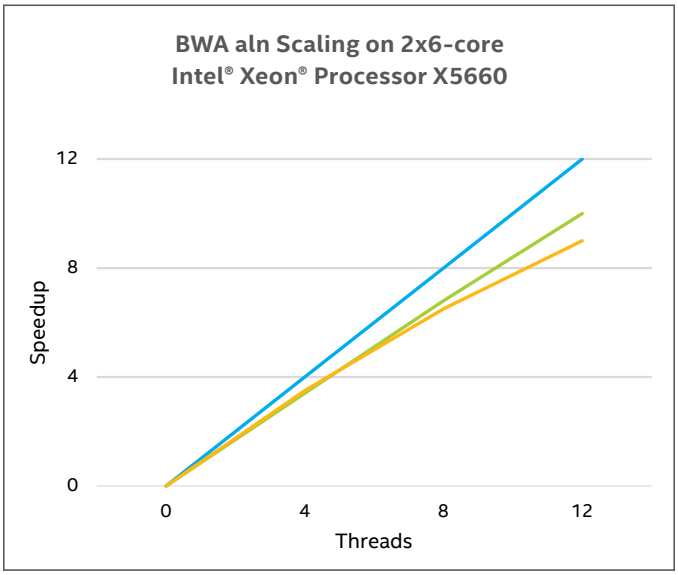


Figure 7: BWA aln Improved Scaling on 12 cores with Cilk (green) compared to the original code (orange)

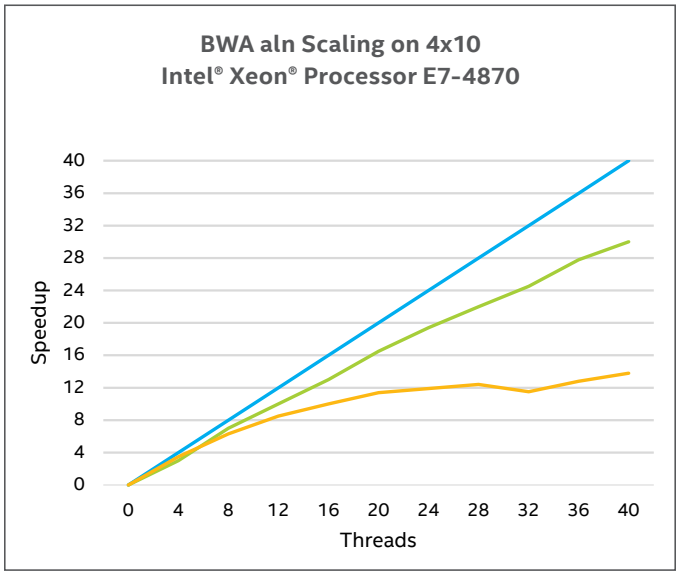


Figure 8: BWA aln Improved Scaling on 40 cores with Cilk (green) compared to the original code (orange)

The pthread-based implementation of BWA aln can be easily changed to use Cilk, a task-based parallel programming extension for C/C++, which achieves automatic load balancing through the use of work-stealing scheduling. Using Cilk, we achieve up to a factor two speedup compared to the original BWA aln mode.

Acknowledgements

This work is funded by Intel, Janssen Pharmaceutica, and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT).

References

[1] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics 2009, 25(14):1754-60 (2009)

[2] Burrows-Wheeler Aligner, http://bio-bwa.sourceforge.net, version bwa-0.7.5a, May 2013

[3] Ch. E. Leiserson. The Cilk++ Concurrency Platform. The Journal of Supercomputing, vol. 1, iss. 3, pp. 244-257, March 2010, Kluwer Academic Publishers (2010)

[4] Intel® Cilk™ Plus, http://software.intel.com/en-us/intel-cilk-plus

[5] Ch. Herzeel, P. Costanza, T. Ashby, R. Wuyts. Performance Analysis of BWA Alignment. Technical Report Exascience Life Lab, November 2013 (2013).

[6] Ch. Herzeel, T. Ashby, P. Costanza, W. De Meuter. Resolving Load Balancing Issues in BWA on NUMA multicore architectures. Proceedings Parallel Bio-Computing workshop, September 2013, Springer Verlag LNCS 8385

[7] Pin – A Dynamic Binary Instrumentation Tool, http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool (2013)

Footnotes

[1] http://www.genomicsengland.co.uk/

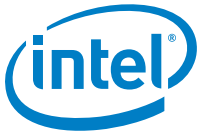
[2] http://rc.kfshrc.edu.sa/sgp/

[3] All benchmarks reported in this paper, and the Cilk modifications discussed here, were done with BWA 0.7.5a, released in May 2013. BWA mem was aded in BWA 0.7.1 in March 2013. A load balancing mechanism similar to the one described in this paper was added to BWA mem, but not for BWA aln, in BWA 0.7.6 in January 2014.

[4] http://www.1000genomes.org/

[5] http://hapmap.ncbi.nlm.nih.gov/

[6] The graphs we show only comprise the parallel sections of the BWA program (see [6]).



0714/DW/MIM/PDF ♻ Please Recycle 330718-001US