Performance Analysis of BWA Alignment

November 5, 2013

Charlotte HERZEEEL
Pascal COSTANZA
Thomas J. ASHBY
Roel WUYTS
Performance Analysis of BWA alignment

Charlotte Herzeel$^{1,2}$*, Pascal Costanza$^{1,3}$, Thomas J. Ashby$^{1,2}$, Roel Wuyts$^{1,2}$

$^1$Intel Exascale Lab, Kapeldreef 75, B-3001 Leuven, Belgium
$^2$Imec, Kapeldreef 75, B-3001 Leuven, Belgium
$^3$Intel, Veldkant 31, B-2550 Kontich, Belgium

November 5, 2013

1 Objective

The primary objective of our experiments is to understand the performance characteristics of the BWA program for sequence alignment. We set up benchmark experiments with different data sets, reflecting the varying data properties of the outputs produced by the sequencing machines.

1.1 Document Overview

We first give an overview of the BWA program in Section 2. Our focus is on analysing the performance of the alignment component of BWA, namely BWA \texttt{aln}. In Section 3 we discuss our methodology for characterising the performance of the BWA program, which consists of analysing the implementation itself, as well as setting up benchmark experiments with realistic data sets. Subsequently, in Section 4 we describe the workloads we choose for our benchmark experiments in detail. We discuss a workload from the 1000 Genomes project, a workload downloaded from the GCAT project, and a workload obtained from Illumina. In Section 5 we explain the types of measurements we do for each workload. Then in Section 6 we present the results for BWA \texttt{aln}. We observe a large load imbalance and use performance counters to explain this. We also tie back these results to an analysis of the algorithm underlying BWA \texttt{aln}. Additionally, we explain how the load imbalance can be removed using Cilk, resulting in a factor two performance improvement. A companion technical report discusses a similar analysis for BWA \texttt{mem}.

2 Burrows-Wheeler Aligner (BWA)

Burrows-Wheeler Aligner (BWA) [1] by Li and Durbin is a widely used read alignment tool. It uses the Burrows-Wheeler transformation of the reference genome, which not only minimises the memory needed to store the reference, but also allows for a strategy for matching the reads that operates in the order

*Corresponding author: charlotte.herzeel@imec.be
of the read length. The technique was originally proposed in the context of text compression [8] and the matching process needed to be adapted for read alignment to handle mismatches due to mutations (such as SNPs) and indels [1]. There are different options to handling mismatches, and BWA presents one solution. Other read aligners based on Burrows-Wheeler transformation, such as Bowtie and SOAP2, use different strategies for mismatches, which are considered to produce faster albeit less accurate results than BWA [1, 9, 10].

2.1 BWA Components

The BWA program consists of different components that need to be used in a certain order. Fig.1 gives an overview of the different components in the form of a flow chart:

1. The user first needs to build the BWT transformation of the reference genome using the BWA index tool. BWA index takes as input a .fa file and produces several .fa.* files to store the BWT transformation.

2. The BWA program uses the BWT transformation from step 1 for the actual alignment process. The user has the choice between two different alignment tools: aln and mem. BWA aln is the original alignment algorithm from the BWA program’s release in 2009. It is designed for aligning reads up to 100bp long. The BWA mem tool was released in February 2013 and is designed for aligning reads between 70bp and 1 Mbp long, which matches how the read length is expected to evolve for future sequencing machines.

3. If alignment is performed with the BWA aln tool, a third step is required that transforms the output from the aln tool to the .sam format which is a more standard format that can be used by other tools for interpreting the alignment. In contrast, the BWA mem tool directly outputs to the .sam format.

To give the reader an idea of the processing requirements of each step: To build the BWT transformation of the human reference genome (step 1), the BWA program takes ±78 minutes using a single thread on an Intel Xeon® X5660 processor@2.8Ghz with 96GB RAM. For aligning a sample read set of 16GB (step 2), the BWA aln tool requires approximately 25 hours using a single thread on the same processor.

In order to reduce the overall execution time, BWA supports multithreaded execution of the alignment step when appropriate hardware resources are available. In this mode, the reads are evenly distributed over the available cores of a multicore processor so that they can be aligned in parallel. In theory, this should give a linear speedup compared to sequential or single-core execution. In the remainder of the report we investigate why this speedup is not achieved and how it can be improved substantially by replacing the pthread-based parallelisation in BWA by a Cilk solution.

3 Performance Analysis Methodology

Our approach to investigate the performance of the BWA program is two-fold. Firstly, we analyse how the program is implemented, and reverse-engineer the
underlying algorithms to get an understanding of what the potential performance bottlenecks are. Secondly, we set up benchmark experiments with realistic data sets to confirm our theories, as well as to measure what is actually happening when the BWA program is running.

### 3.1 Benchmarking Experiments

Our approach is to run different types of sequencing workloads on different types of servers. We measure:

1. Execution time, scaling behaviour, and efficiency
2. Load imbalance amongst the worker threads
3. The impact of NUMA in multi-socket servers
4. Hardware counters (with Intel® VTune™)

### 3.2 Specification Details of the Benchmark Machines

We show results for running our benchmarks on different types of machines:

1. A 2-socket server equipped with two 6-core Intel® Xeon® X5660 processors @2.8GHz with 96GB RAM.
2. A 4-socket server equipped with four 10-core Intel® Xeon® E7-4870 processors @2.4GHz with 512GB RAM.

All of the above machines run Ubuntu 10.04.3 LTS Lucid Lynx. All benchmarks are compiled using icc version 13.0.1 with -O2 enabled.

3.3 Sequencing Software Versions

To make this report, we use the following software:

1. BWA version 0.7.5a-r405
2. samtools version 0.1.19
3. vcftools version 0.1.11

4 Workloads

The data properties of sequencing workloads vary. For example, the length of the reads depends on the sequencing machine used to produce a data set. Different data sets also have different coverage, a property that determines the reliability of the read data. By default, reads within the same data set have the same length, but quality clipping may be applied to keep only the parts of which the quality is above a certain threshold, resulting in data sets where the reads have different lengths.

We therefore choose to experiment with different types of sequencing workloads. We define four workloads with different data properties: workshop, 1000Genomes, Bioplanet, and Illumina. These workloads are all defined in terms of existing (and realistic) data sets. Alternatively, one could generate artificial data sets using the dwgsim tool [5].

4.1 Workshop

This is a small example from a tutorial on how to use BWA. The data set itself is a small subset of a data set from the 1000 Genomes project. This is not a realistic workload in terms of size, but the data itself is real. We use this data set for small tests only.

Properties:

- **type**: single-end, quality clipped
- **#reads**: 2,621,440
- **read length**: 37-108bp
- **coverage**: 4x (?)
- **size**: 181MB (zipped)
- **file**: NA20589.fastq.gz
- **reference**: human_g1k_v37_chr20.fa
url: http://genome.sph.umich.edu/wiki/Tutorial:_Low_Pass_Sequence_Analysis

trimming: We set the threshold quality for read trimming to 15 (cf. -q option of BWA aln).

cmd: bwa aln -t 12 -q 15 human_g1k_v37_chr20.fa NA20589.fastq.gz
     1> output.sai 2> log

4.2 1000Genomes

This is a data set downloaded from the 1000 Genomes project[3]. Coverage of 1000 Genomes data is low (4-60x).

Properties:

type: paired-end. We only use the first read set. Hence all benchmark numbers are for the **1st read set only**.

#reads: 14,875,357
read length: 108bp
coverage: 4-60x (?)
size: 1.1GB (zipped)
file: ERR013029_1.filt.fastq.gz
reference: human_g1k_v37_chr20.fa
url: ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/NA20589/sequence_read/
trimming: We set the threshold quality for read trimming to 15 (cf. -q option of BWA aln).

cmd: bwa aln -t 12 -q 15 human_g1k_v37_chr20.fa ERR013029_1.filt.fastq.gz
     1> output.sai 2> log

4.3 Bioplanet

This is a data set downloaded from the GCAT project [4]. GCAT stands for Genome Comparison & Analytics Testing. It is the name of a tool suite for comparing and benchmarking the accuracy of alignment tools. The data sets they offer are either artificially generated with the **dwgsim** tool [5] or subsets from Illumina data. The data set we use is one that is generated with **dwgsim**.

Properties:

type: paired-end

#reads: 11,945,250
read length: 100bp
coverage: artificial
4.4 Illumina

This data set is downloaded from the Illumina website [6]. This is real data produced by an Illumina HiSeq system with high coverage (200x).

Properties:

- **type:** paired-end. We only use the first read set. Hence all numbers are for the 1st read set only.
- **#reads:** 205,185,641
- **read length:** 100bp
- **coverage:** 200x
- **size:** 16GB (zipped)
- **file:** ERR091575_1.fastq.gz
- **reference:** hg19.fa
- **url:** Study PRJEB2891 from [http://www.ebi.ac.uk/ena/data/view/ERP001230](http://www.ebi.ac.uk/ena/data/view/ERP001230)

trimming: We set the threshold quality for read trimming to 15 (cf. -q option of BWA aln).

**cmd:**

```
bwa aln -t 12 -q 15 hg19.fa ERR091575_1.fastq.gz 1> output.sai 2> log
```

4.5 Quality control of workloads

We can use the quality checks that bio-engineers perform on aligned data to verify the correctness of the output across different benchmark runs.

4.5.1 Pre-alignment QC

Before alignment, a number of quality checks are performed by the bio-engineer which give her an indication of what the alignment may look like, and may be used to change parameters of the alignment tool. We do not use these quality checks ourselves, but provide them for completeness. Tab. 1. gives an overview of pre-alignment quality checks for each of our workloads. They are generated using FastQC version 0.10.1.
Table 1: Quality control of raw reads

<table>
<thead>
<tr>
<th>Workload</th>
<th>Number of reads</th>
<th>Average base quality</th>
<th>%GC content</th>
<th>Duplication rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workshop</td>
<td>3,620,845</td>
<td>27.30</td>
<td>43</td>
<td>7.46</td>
</tr>
<tr>
<td>1000genomes</td>
<td>14,875,357</td>
<td>31.76</td>
<td>38</td>
<td>21.07</td>
</tr>
<tr>
<td>Bioplanet</td>
<td>11,945,250</td>
<td>17.00</td>
<td>48</td>
<td>1.01</td>
</tr>
<tr>
<td>Illumina</td>
<td>205,185,641</td>
<td>36.48</td>
<td>40</td>
<td>10.26</td>
</tr>
</tbody>
</table>

Figure 2: SNV calling flow graph. Step 1: Converting BWA .sam output to .bam and .bcf using samtools (a). Step 2: Variant calling using bcftools (b).

4.5.2 Single nucleotide variants QC

We can use the quality checks that bio-engineers perform on the results of variant calling on the aligned data to verify the outputs of our benchmark runs. Fig. 2 gives an overview of the steps are applied using samtools and bcftools to perform variant calling and generate a .vcf file.

We then perform quality checks on the .vcf file using vcftools. Tab. 2, Tab. 3, Tab. 4 shows the information generated with vcf-tools version 0.1.11 using vcf-stats and vcf-tstv for alignment output generated with BWA aln. We check that these values are the same across different benchmark runs.

5 Experimental Data

The benchmarks are all done for BWA version 0.7.5a-r405, compiled with icc version 13.0.1 with -O2 enabled. Each benchmark was run at least 10 times.

We run each workload on four different versions of BWA:

1. The original unmodified BWA code where paralellisation is done via pthreads. This configuration is referred to as icc-pthreads.

2. The original unmodified BWA code with the option numactl --interleave=all, referred to as icc-pthreads-numactl. The option numactl --interleave=all

\[\text{Vcf stands for variant calling format}\]
<table>
<thead>
<tr>
<th>Workload</th>
<th>Number of SNPs</th>
<th>Number of indels</th>
<th>Ts/Tv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workshop</td>
<td>37,968</td>
<td>1700</td>
<td>2.09</td>
</tr>
<tr>
<td>1000genomes</td>
<td>26,167</td>
<td>604</td>
<td>2.03</td>
</tr>
<tr>
<td>Bioplanet</td>
<td>82,253</td>
<td>12,163</td>
<td>0.50</td>
</tr>
<tr>
<td>Illumina</td>
<td>3,136,295</td>
<td>270,368</td>
<td>2.06</td>
</tr>
</tbody>
</table>

Table 2: Quality control of SNV calling (1/3)

<table>
<thead>
<tr>
<th>Workload</th>
<th>C &gt; A</th>
<th>A &gt; T</th>
<th>G &gt; T</th>
<th>C &gt; G</th>
<th>G &gt; C</th>
<th>G &gt; A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workshop</td>
<td>1693</td>
<td>1728</td>
<td>1283</td>
<td>1618</td>
<td>1615</td>
<td>6584</td>
</tr>
<tr>
<td>1000genomes</td>
<td>1058</td>
<td>1019</td>
<td>1083</td>
<td>1073</td>
<td>1031</td>
<td>4101</td>
</tr>
<tr>
<td>Bioplanet</td>
<td>6569</td>
<td>6983</td>
<td>6659</td>
<td>6567</td>
<td>6598</td>
<td>6603</td>
</tr>
<tr>
<td>Illumina</td>
<td>133412</td>
<td>111959</td>
<td>134504</td>
<td>135178</td>
<td>135969</td>
<td>538254</td>
</tr>
</tbody>
</table>

Table 3: Quality control of SNV calling (2/3)

<table>
<thead>
<tr>
<th>Workload</th>
<th>A &gt; C</th>
<th>A &gt; G</th>
<th>T &gt; G</th>
<th>T &gt; A</th>
<th>T &gt; C</th>
<th>C &gt; T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workshop</td>
<td>1554</td>
<td>6359</td>
<td>1554</td>
<td>1256</td>
<td>6442</td>
<td>6282</td>
</tr>
<tr>
<td>1000genomes</td>
<td>1153</td>
<td>4593</td>
<td>1181</td>
<td>1046</td>
<td>4613</td>
<td>4216</td>
</tr>
<tr>
<td>Bioplanet</td>
<td>7086</td>
<td>7024</td>
<td>7161</td>
<td>7137</td>
<td>7274</td>
<td>6592</td>
</tr>
<tr>
<td>Illumina</td>
<td>130598</td>
<td>517858</td>
<td>129998</td>
<td>112127</td>
<td>519162</td>
<td>537186</td>
</tr>
</tbody>
</table>

Table 4: Quality control of SNV calling (3/3)

makes sure memory allocation is spread across the different sockets. With this option, we can see whether NUMA effects have an impact on the performance of BWA.

3. A modified version of BWA where we replace the pthreads parallelisation with Cilk code. Cilk removes load imbalance, no matter the source.

4. Our Cilk version of BWA with the option `numactl --interleave=all`, referred to as `icc-cilk-numactl`.

For each of our workloads we display four graphs:

1. A histogram displaying the execution time.

2. A histogram displaying the load imbalance between worker threads. The graph displays the average execution time per thread per run. It measures the accumulated load imbalance of the different parallel phases as follows. Per parallel phase we measure the time each thread executes. We sort these timings and then combine the timings of the different parallel phases by pairwise adding them according to their length, i.e. add up the smallest timings, then add up the second smallest timings, etc. This way we measure the load imbalance accumulated over the different parallel phases where the slowest thread determines the overall execution time.

3. A speedup graph displaying the scaling behaviour of the different BWA
versions. All speedup graphs use the execution time using the original BWA (cf. icc-pthreads) with 1 thread as a baseline.

4. An efficiency graph for the different BWA versions. Similarly to the speedup graphs, the efficiency graphs use the execution time for 1 thread using the original BWA code (cf. icc-pthreads) as a baseline.

6 BWA aln

BWA aln is the original alignment algorithm in the BWA program. It is designed for aligning short reads up to 100bp long.

6.1 Original pthreads Parallelisation

Read alignment is an embarrassingly parallel problem. In a typical workload millions of reads, up to 100bp (200 characters) long, need to be aligned. Reads are aligned independently from one another. Hence read alignment can be parallelised as a data parallel loop over the read set.

Concretely, the BWA code sets up pthreads equal to the number of cores on the target processor. Each pthread executes a sequential alignment loop for an equal share of the reads.

The read file is not processed in one go, but rather processed in chunks of a fixed size. Each of these chunks is distributed across the threads using a round-robin pattern, as shown in Fig.3. Hence given n threads, the 1st read is assigned to the 1st thread, the 2nd read to the 2nd thread, the nth read to the nth thread, the n+1th read to the 1st thread, the n+2th read to the 2nd thread, and so on.

Linear speedup for such an implementation is only guaranteed if the work to be done is roughly equal for each pthread, in order to avoid load imbalance. To detect if there is load imbalance possible, we need to inspect the algorithms underlying the BWA aln tool.

6.2 Burrows-Wheeler Alignment Algorithm

We reverse-engineer the algorithm underlying the BWA aln tool from the papers [1] and the BWA implementation. We discuss it to investigate the challenges it presents for efficient multithreaded execution.

The Burrows-Wheeler alignment algorithm relies on the definition of two auxiliary data structures. These data structures are defined in terms of a compressible version of the reference, which is created via the so-called Burrows-Wheeler transformation. E.g. $BWT(\text{abrakadabra})$ would be $\text{ard$rcaaaabb}$. Given the Burrows-Wheeler transformation of the reference, the table $c_{tab}$ stores for each character $c$ in the (genetic) alphabet how many characters occur in the transformation that are lexicographically smaller than $c$. A second table, $occ_{tab}$ is defined so that a function $\text{occ}(occ_{tab}, c, k)$ returns the number of occurrences of the character $c$ in the prefix $BWT(ref)[1...k]$. In principle, the table $occ_{tab}$ has for each character as many entries as the length of the reference, but BWA

\footnote{We always refer to the latest version of BWA, i.e. the bwa-0.7.5a download on [2].}

\footnote{This is actually configured via the $-t$ parameter of the BWA program.}
Figure 3: BWA aln: Read files are processed in chunks of which the reads are distributed across threads in a round-robin pattern.

only stores the information for every 32 characters. For the human reference, \textit{occ.tab} is around 3GB large [1].

Given the tables \textit{c.tab} and \textit{occ.tab}, finding out where (or whether) a read matches the reference, is a simple calculation. Fig.4 shows pseudo code for matching a read. The code consists of a loop that iterates over the characters of the read (\textit{r}). Each iteration references the \textit{occ.tab} and \textit{c.tab} to compute a new starting point (\textit{sp}) and end point (\textit{ep}), which represent a range from which the indexes—where the read matches the reference—can be calculated.

The code in Fig.4 actually only works for reads that match the reference exactly. For reads with mutations or indels, additional work is needed. The code for inexact matches is shown in Fig.5. For inexact matches, multiple alternative matches are checked and explored using a priority queue to direct the order in which the alternatives are explored. It is not important at this point to understand all the details, the structure of the code remains roughly the same as in Fig.4. What is important to note is that for inexact matches additional work is necessary.

The code in Fig.4 and Fig.5 embodies certain patterns that have consequences for multithreaded code:

1. The ratio of memory operations versus other operations is high: 28% (computed with Intel® VTune™ Amplifier XE 2013). Memory operations may have a high latency and stall processors.

2. In standard multicore servers, cores are clustered in sockets. Cores on
different sockets have different access times to different regions in memory, cf. non-uniform memory access (NUMA) architecture. A core can access memory on its own socket faster than memory on other, remote sockets. By default, each pthread allocates memory on its own socket. In BWA, the data structure that stores the reads, as well as the tables $c_{tab}$ and $occ_{tab}$ are allocated at the beginning of the execution, before the alignment starts. This means that all pthreads which are not on the first socket will have slower access time to these tables.

3. Aligning a read that matches some part of the reference exactly is cheaper than matching a read that has mutations or indels.

4. Reads may have varying lengths (e.g. after quality clipping). Since each character of a read needs to be processed by the loop in Fig.4, longer reads will take longer to match.

The above points are all sources for load imbalance amongst the pthreads: There is load imbalance because certain threads will have slower access to the $c_{tab}$ and $occ_{tab}$ tables, as well as the data structure that stores the reads, and there is load imbalance because certain threads may have to handle longer or more mutated reads than others.

6.3 Cilk Parallelisation

Intel® Cilk™ Plus [12, 13] 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, balancing the overall workload.

We replaced the pthread-based parallel loop in BWA by a Cilk for loop. There are some intricacies with regard to making sure that each worker thread has its own priority queue for intermediate matches, to avoid contention of a shared queue. Our solution is to initialise the priority queues before executing the parallel loop, one for each worker thread. The priority queues are stored in a global array so that they are globally accessible by the worker threads. Inside the for loop, we use Cilk’s introspective operator for querying the running worker thread’s ID, which we then use to identify the priority queue the worker thread accesses.

By using Cilk, we were able to drastically improve the scaling behaviour of BWA aln. A paper on this was accepted at PPAM’13 [11].

6.4 BWA aln on 2×6-core Intel® Xeon® X5660 @ 2.8GHz

This section discusses the runs on the server equipped with 2×6-core Intel® Xeon® X5660 processors. This server thus has 12 cores, but we run the benchmarks with 24 threads, to see the impact of using hyper-threading. We claim that hyper-threading may help for BWA because of the algorithm’s scattered data access pattern and the fact that it is unpredictable which character will
def exact_bwc(r):
    n = len(r)
    i = n - 1
    c = r[i]
    sp = c_tab[c] + 1
    ep = c_tab[next_letter(c, abc)]
    j = i - 1
    while not(j < 0 or sp > ep):
        nc = r[j]
        sp = c_tab[nc] + occ(occ_tab, nc, sp - 1) + 1
        ep = c_tab[nc] + occ(occ_tab, nc, ep)
        j -= 1
    return (ep - sp + 1, sp, ep)

Figure 4: Alignment of a read (exact)

def inexact_bwc(r, z):
    results = []
    pq = Queue.PriorityQueue()
    pq.put((0, len(r) - 1, z, None, None))
    while not(pq.empty()):
        pm = pq.get()
        p = pm[0]
        i = pm[1]
        z = pm[2]
        sp = pm[3]
        ep = pm[4]
        if z < 0: #1 too many substitutions
            None
        elif i < 0: #2 match found
            results.append((sp, ep))
        else:
            pq.put((p + 1, i - 1, z - 1, sp, ep)) #3 assume delete
            c = r[i]
            for nc in abc[1:len(abc)]: #4 generate substitutions
                nsp = c_tab[nc] + occ(occ_tab, nc, abc, sp - 1) + 1
                nep = c_tab[nc] + occ(occ_tab, nc, abc, ep)
                if nsp <= nep:
                    pq.put((p + 1, i, z - 1, sp, ep)) #5 assume mutation
                    if c == nc: #6 assume exact match
                        pq.put((p - 2, i - 1, z, nsp, nep))
                    else: #7 assume insert
                        pq.put((p + 1, i - 1, z - 1, nsp, nep))
        return results

Figure 5: Alignment of a read (inexact)
be matched next (Section 6.2). Cores with hyper-threading share a number of resources between threads: cache, branch prediction, instruction fetch and decoding, and execution units. Hyper-threading is only effective when none of these resources is a limiting factor on the overall performance of a thread. For example, with hyper-threading, one thread can use the execution units while another thread is waiting for a memory fetch (that is not in cache) to resolve. Or when branch mispredictions are frequent, one thread can use the execution units while another thread is waiting for the misprediction to be resolved. Since the data access pattern in BWA in not predictable, we think branch prediction and cache use are suboptimal. Fig. 6 shows a run through Intel® VTune™, which indicates a high number of retire stalls, flagging a high branch misprediction rate of ±10%. Hence hyper-threading may pay off for BWA.

Figure 6: BWA aln: 1000Genomes: Intel® VTune™ report showing high retire stalls

### 6.4.1 1000 Genomes

Fig. 7 shows our measurements for executing the 1000 Genomes workload. The speedup graph shows that the original BWA code does not scale linearly (cf. icc-pthreads): At 12 threads it achieves 9.88× speedup, only 82% of the potential speedup (cf. 0.82 efficiency). The load imbalance graph shows there is a clear load imbalance: The slowest thread is ±1.20× slower than the fastest thread.

We think the poor scaling and load imbalance partially stems from branch mispredictions and the random data memory access pattern in the algorithm. This is confirmed by the speedup graph showing us that hyper-threading improves scaling: We get a factor 12.53× speedup at 24 threads (cf. icc-pthreads).

Another source of load imbalance are likely NUMA effects because the data structures are by default allocated on the same socket by the main program thread. The runs with numactl --interleave=all to distribute the memory allocation across the sockets further improves scaling (cf. icc-pthreads-numactl).

With this option, we see a speedup factor of 10.54× (or 0.88 efficiency) and for 24 threads we see 14.49× speedup. The Cilk versions of BWA (cf. icc-cilk and icc-cilk-numactl) further improve the scaling behaviour. With numactl enabled, we see a 11.07× speedup for 12 threads (or 0.92 efficiency) and 14.86×
speedup for 24 threads. This suggests that the Cilk version fixes an additional source of load imbalance.

Figure 7: BWA aln: 1000Genomes: 2×6-core Intel® Xeon® X5660

6.4.2 Bioplanet

Fig. 8 shows the results for executing the Bioplanet workload. The original BWA implementation (cf. icc-pthreads) does not scale linearly. For 12 threads we see 9.78× speedup or 0.82 efficiency. With hyper-threading, we see 11.09× speedup for 24 threads. There is a clear load imbalance shown in the load imbalance graph: the slowest thread is ±1.22× slower than the fastest thread. By activating numactl --interleave=all we get 9.98× speedup at 12 threads and 11.32× speedup at 24 threads (cf. icc-pthreads-numactl).

The Cilk version of BWA scales best, yielding 10.44× speedup for 12 threads (or 0.87 efficiency) and 15.67 speedup at 24 threads (cf. icc-cilk-illumina-numactl).

6.4.3 Illumina

Fig. 9 shows the results for the Illumina workload. The original BWA code does not scale linearly (cf. icc-pthreads): For 12 threads we see 5.76× speedup, only 48% of the potential speedup (cf. 0.48 efficiency). The load imbalance graphs indicates a factor 1.22× difference between the slowest and fastest thread. For
24 threads we see 8.67× speedup. With numactl we see 5.25× speedup at 12 threads and 8.71× speedup at 24 threads.

The Cilk version of BWA greatly improves scaling. For icc-cilk-numactl we see 11.42× speedup (or 0.95 efficiency) and 16.19× speedup for 24 threads.

### 6.4.4 Conclusions

There are a number of observations to make:

1. For all workloads, BWA aln does not scale linearly. At 12 threads, we see 0.82 efficiency for both the 1000 Genomes and Bioplanet benchmarks. For Illumina we see only 0.48 efficiency.

2. For all workloads, we see a large load imbalance amongst the threads: The slowest thread is roughly 1.20× slower than the fastest thread.

3. Hyper-threading always pays off, indicating that part of the load imbalance is due to instructions being stalled due to branch mispredictions and cache misses. These issues are inherent to the algorithmic structure of the sequencing algorithm underlying BWA aln.

4. Our Cilk version of BWA achieves better scaling for all benchmarks: For 12 threads, we achieve 0.92 efficiency for 1000 Genomes, 0.87 efficiency for Bioplanet, and 0.95 efficiency for Illumina.
5. There is a large difference between the efficiency observed for the Illumina workload (0.48) and the 1000 Genomes and Bioplanet workloads (0.82) for the plain `pthreads` version of BWA (`icc-pthreads`). This suggests that the properties of the reads in the data sets themselves contribute to the load imbalance.

![Graphs showing execution time, load imbalance, scaling, and efficiency for BWA aln on Illumina data set.](image)

Figure 9: BWA aln: Illumina: 2×6-core Intel® Xeon® X5660

### 6.5 BWA aln on 4×10-core Intel® Xeon® E7-4870 @ 2.4GHz

In this section we discuss the measurements for running the workloads on the server equipped with 4×10-core Intel® Xeon® E7-4870 processors.

#### 6.5.1 1000 Genomes

Fig. 10 shows our measurements for the 1000 Genomes workload. We see that BWA aln scales poorly (cf. `icc-pthreads`): At 40 threads we achieve 12.73× speedup or only 0.32 efficiency. The load imbalance graph indicates that the slowest thread is about 2.83× slower than the fastest thread. We see that NUMA contributes to the load imbalance. When the command `numactl --interleave=all` is used, we see a speedup of 17.40× on 40 threads (cf. `icc-pthreads-numactl`). The load imbalance graph for `icc-pthreads-numactl` (not shown) tells us that the slowest thread is roughly 2.02× slower than the fastest.
Performance Analysis BWA Alignment

fastest thread, whereas without numactl, we see difference of factor 2.83× (cf. Fig.10).

Our Cilk version of BWA aln further improves scaling. For 40 threads we see 29.76× speedup or 0.74 efficiency (cf. icc-cilk). The effect of numactl on the Cilk version is minor (cf. icc-cilk-numactl).

Figure 10: BWA aln: 1000Genomes: 4×10-core Intel® Xeon® E7-4870

6.5.2 Bioplanet

Fig.11 shows the results for the Bioplanet workload. Again, the original BWA aln scales badly (cf. icc-pthreads). For 40 threads we see 13.22× speedup or 0.33 efficiency. The load imbalance graph shows that the slowest thread is about 2.20× slower than the fastest thread.

Using numactl --interleave=all has a positive impact, indicating that NUMA effects are involved in causing the load imbalance: We see 17.85× speedup or 0.37 efficiency on 40 threads (cf. icc-pthreads-numactl). If we would look at the load imbalance graph for icc-pthreads-numactl (not shown), we would see that the difference between the slowest and fastest thread is a factor 1.94× versus a factor 2.20× for the icc-pthreads version (cf. Fig.11).

Our Cilk version of BWA aln scales the best. It achieves 30× speedup on 40 threads or 0.75 efficiency (cf. icc-cilk). Using the numactl command with the Cilk version only has a minor positive impact: 31× speedup or 0.77 efficiency (cf. icc-cilk-numactl).
6.5.3 Illumina

Fig. 12 show the measurements for the Illumina workload. The original BWA aln code scales poorly, only achieving 17.84× speedup or 0.44 efficiency with 40 threads. There is again a clear load imbalance: The slowest thread is ±1.83× slower than the fastest thread.

NUMA effects seem to be part of the load imbalance: When using numactl --interleave=all, we see a speedup of 20.26× or 0.51 efficiency for 40 threads. When we investigate the load imbalance graph for icc-pthreads-numactl (not shown), we see that the slowest thread is about 1.57× slower than the fastest thread, whereas the difference for the icc-pthreads version is a factor ±1.83× (cf. Fig.12).

Our Cilk version again gives the best scaling results. For 40 threads, we see a 28.42× speedup or 0.71 efficiency (cf. icc-cilk). With the numactl option, the Cilk version achieves 29.23× speedup or 0.73 efficiency.

6.5.4 Conclusions

We make the following observations:

1. The original pthreads implementation of BWA aln scales very poorly for all of our workloads. For 40 threads, we see 0.32 efficiency for the 1000
Figure 12: BWA aln: Illumina: 4×10-core Intel® Xeon® E7-4870

Genomes benchmark, 0.33 efficiency for the Bioplanet benchmark, and 0.45 efficiency for the Illumina benchmark.

2. For all workloads we see a severe load imbalance amongst the threads. For 1000 Genomes, there is a factor $2.83\times$ difference between the slowest and fastest thread. For Bioplanet the factor is $2.20\times$. For Illumina, the slowest thread is $1.83\times$ slower than the fastest thread.

3. NUMA effects seem to be causing part of the load imbalance in the pthreads version. If we specify numactl --interleave=all, the original pthreads version of BWA aln has a better efficiency: 0.44 for 1000 Genomes (vs. 0.33), 0.37 for Bioplanet (vs. 0.32), and 0.51 for Illumina (vs. 0.45).

4. Our Cilk version of BWA always achieves better scaling than the original pthreads code. For 40 threads, we see 0.74 efficiency for 1000 Genomes, 0.75 efficiency for Bioplanet, 0.73 efficiency for Illumina. Specifying numactl options seems to have only a minor (positive) effect on the Cilk version of BWA aln.

6.6 Further Analysis of the Load Imbalance in BWA aln

In Section 6.2 we discussed that we think that the load imbalance is partially caused by the fact that different reads take different times to align. To prove
this claim we set up an experiment where we measure the number of instructions and time spent per read.

6.6.1 Counting the number of instructions per read

Fig.13 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. Note that this is a smooth scatterplot, meaning that the colour 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. For example, ±81% of all reads have a runtime that varies less than 3 ms 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 16,000× difference. This is not an outlier: Fig.13 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. Fig.13 shows a clear uphill pattern as we go from left to right. The correlation factor is 0.72, indicating a strong linear relationship between number of instructions per read and the time spent per read.

5. Fig.13 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 Fig.13 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, NUMA effects etc. This is actually the same point as observation nr. 2.

It is clear now that different reads need different numbers of instructions/execution time to process. In the next section we therefore investigate how much these differences contribute to the overall load imbalance we observe when running BWA aln.

6.6.2 Predicting load imbalance based on read distribution

We can predict the load imbalance caused by the fact that different reads have different processing requirements by simulating the distribution of the reads across the 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.

Our simulation of distributing the reads across threads strictly follows the way the BWA code does the distribution. As we discussed in Section 6.1, reads are stored in files that are processed in chunks of a fixed size. For each chunk, the reads are distributed in a round-robin manner across the threads for alignment. Once a chunk is processed, the next chunk of reads is loaded from file into memory, a process that repeats itself until the entire read file has been processed. For \( N \) reads this gives us \( \left\lfloor \frac{N}{\text{chunk}} \right\rfloor \) distributions of reads across threads and consequently \( \left\lfloor \frac{N}{\text{chunk}} \right\rfloor \) parallel processing phases. The overall load imbalance is then determined by accumulating the load imbalance of the different parallel phases.

Hence in our simulation we first generate the different read distributions by assigning the reads per chunk in a round-robin manner to the (virtual) threads. The workload for each thread is represented by the total number of instructions
of the reads assigned to that thread. To combine the multiple distributions we then sort the threads according to their instruction load and combine them following this order. That is to say, we add up the instruction loads of the threads with the smallest load in each distribution, add up the instruction loads of the threads with the second smallest loads in each distribution, and so on. This gives us a combined distribution that reflects the load imbalance accumulated across the different chunks/parallel phases.

The load imbalance is then estimated by dividing the highest instruction load by the lowest instruction load in the distribution as this gives us a factor of how much more work the slowest thread is assigned compared to the fastest thread. Table 5 displays the load imbalance factors we compute this way for the 1000 Genomes benchmark for simulations with different numbers of threads. This table tells us how much load imbalance we can expect purely based on how the instruction counts are distributed across the threads. The first column of Tab.5 displays the number of threads per simulation, which ranges from 1 to 128. The second column displays the difference between the thread with the most instructions and the thread with the fewest instructions as a factor. For example, for 12 threads, we see that the slowest thread has $1.14 \times$ more instructions to execute than the fastest thread. We observe that the load imbalance factor goes up as the number of threads increases in our simulation. This is logical because the more threads there are, the more impact extreme reads—with proportionally large instruction counts—will have on the parallel phases.

<table>
<thead>
<tr>
<th>Load Imbalance</th>
<th>Expected</th>
<th>Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>n/a</td>
</tr>
<tr>
<td>4</td>
<td>1.050357</td>
<td>n/a</td>
</tr>
<tr>
<td>8</td>
<td>1.094122</td>
<td>n/a</td>
</tr>
<tr>
<td>12</td>
<td>1.14385</td>
<td>1.20</td>
</tr>
<tr>
<td>16</td>
<td>1.179302</td>
<td>n/a</td>
</tr>
<tr>
<td>32</td>
<td>1.304211</td>
<td>n/a</td>
</tr>
<tr>
<td>40</td>
<td>1.362940</td>
<td>2.02</td>
</tr>
<tr>
<td>64</td>
<td>1.527168</td>
<td>n/a</td>
</tr>
<tr>
<td>128</td>
<td>1.949450</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 5: BWA aln: 1000 Genomes: Expected load imbalance based on how instructions are distributed across threads.

How do the numbers in Tab.5 compare to the actual load imbalance we measure for the benchmarks in Section 6? For 12 threads, Tab.5 tells us that, purely based on the difference between instructions assigned to the threads, the slowest thread will be at least a factor $1.14 \times$ slower than the fastest thread. The slowdown we see based on the actual timing measurements in Section 6.4 is a factor $1.20 \times$. For 40 threads, Tab.5 says that we should see a load imbalance of at least a factor $1.36 \times$. The load imbalance we identify in Section 6.5 based on the measurements for 40 threads is a factor $2.83 \times$ (prior numactl use). The large difference between the predicted and measured slowdown is due to the fact that the runs on the 40-core server suffer more from hardware effects such

---

4Assuming each instruction takes roughly the same time to execute.
as NUMA. For example, when using `numactl` to neutralise NUMA effects, we observe a slowdown factor of roughly $2.02 \times$ (cf. Section 6.5), which is closer to the predicted load imbalance of factor $1.36 \times$, and our particular use of `numactl` does not even guarantee the optimal memory layout for our 40-core server. Hence although it is clear that there is a load imbalance because threads are assigned different amounts of instructions, hardware factors such as NUMA also contribute to the observed load imbalance.

### 6.6.3 Impact of chunk size

The chunk size has a big impact on the expected load imbalance for a particular data set. In Tab.6 we show the load imbalance that would be expected if the chunk size were one, i.e. if all reads would be read into memory at the same time and processed in one go by a single parallel phase. Although there is still load imbalance expected for a chunk size of one, it is far less than what is predicted (and observed) for the default chunk size of the BWA program (cf. Tab.5). The chunk size determines how many reads are processed in one go and consequently how many parallel phases occur. The default chunk size of the BWA program is 262,144. For the 1000 Genomes data set, which consists of 14,845,357 reads (cf. Section 4.2), this means there are 57 parallel phases required to process the workload. We also note that the number of reads that each phase processes is much smaller than the total number of reads in a workload. We explained in the previous section that different reads may have very different processing requirements in terms of numbers of instructions, seeing extreme differences up to a factor $16000 \times$. Statistically, given the large number of reads in a typical workload, if all reads are processed in one go as one big chunk, the _extreme_ reads will be distributed across the threads, minimising the overall load imbalance, giving us the load imbalance we predict in Tab.6. However, if the chunk sizes are much smaller than the total read set, one _extreme_ read may have a severe impact on the load imbalance of a single parallel phase. Because there is an implicit synchronisation between phases, all threads but one are waiting for the processing of the extreme read to finish. Overall, we can then expect to accumulate a more severe load imbalance across the different parallel phases, which is what we predict (and observe) in Tab.5 when using the default chunk size of the BWA program.

It is important to stress that the numbers in Tab.5 and Tab.6 are specific to the 1000 Genomes workload. Other workloads will have different load imbalances. Although the theoretical load imbalance is lowest when there is only one chunk, configuring this may not be possible as it requires storing all reads in memory at once. As a test, we ran the BWA program for one chunk of the 1000 Genomes workload through Valgrind/Massif. We observe that:

1. The support data structures, i.e. \texttt{occ tab} and \texttt{c tab}, use 2.971GB memory.
2. One chunk requires 0.952GB memory.

The 1000 Genomes workload consists of 57 chunks of 262,144 reads. Hence keeping all reads in memory at once would require $57 \times 0.952\text{GB} + 2.971\text{GB} = 57.235\text{GB}$ RAM. The Illumina workload, however, consists of 783 chunks ×
Table 6: BWA aln: 1000 Genomes: Expected load imbalance based on how instructions are distributed across threads if the chunk size is 1.

| $|chunk| = 1$ |
|---|
| 1 | 1.0 |
| 4 | 1.0053355 |
| 8 | 1.0157592 |
| 12 | 1.018045 |
| 16 | 1.0254985 |
| 32 | 1.0301563 |
| 40 | 1.0537376 |
| 64 | 1.0467701 |
| 128 | 1.0862254 |

262,144 reads. To keep all reads in memory for this workload, a rough estimate is that we need at least $783 \times 0.952GB + 2.971GB = 748.387GB$ RAM. The Illumina workload is 1/8 of 1 sample.

6.6.4 Conclusions

To conclude:

1. We observe a large difference in processing requirements per read when counting the number of instructions spent per read.

2. When we simulate the static distribution of the reads across worker threads according to the round-robin strategy in the BWA code, we observe a load imbalance among the threads purely based on the instruction counts. The load imbalance we predict with our simulation matches the load imbalance we measure in the actual benchmark runs to a certain degree. The load imbalance we do not capture with our simulation is to be attributed to hardware effects such as NUMA, cache misses, branch mispredictions, etc.

3. Given a fixed workload size, the load imbalance to expect goes up as the number of threads increases.

4. Independent of the chunk size, there will be load imbalance because different reads have different processing requirements.

5. If the chunk size is one, we expect that a static distribution of the reads across threads is statistically ideal in the sense that it will create the least possible load imbalance for a particular workload. However, this requires having all reads in memory at once, which may not be possible because of memory limitations of the target hardware. Even if the workload fits into memory, a static distribution of the reads across worker threads will always suffer from a certain amount of load imbalance specific to the workload (cf. point 4). Hence a dynamic load balancing approach such as our Cilk version of BWA is always the best solution.

$^5$The reads of the Illumina workload are 100bp, whereas the reads in the 1000 Genomes workload are 108bp.
6.6.5 Further optimisation possibilities

The fact that reads have extremely varying processing requirements—up to a factor $16000\times$ difference in instructions for the 1000 Genomes workload—may also explain why our Cilk version of BWA does not achieve perfect parallel efficiency. For example, for the 1000 Genomes workload we see that our Cilk version achieves only 0.90 efficiency on a 12-core server (cf. Section 6.4.1) and 0.75 efficiency on a 40-core server (cf. Section 6.5.1).

In theory, a work-stealing approach such as Cilk removes load imbalance so that perfect linear scaling is possible (cf. Section 6.3). This however assumes either a) that the parallel tasks that are indivisible require roughly the same time to process or b) that for every indivisible task that requires disproportionally more time to process, there are enough compensating tasks so that all worker threads can be kept busy for roughly the same amount of time.

In our Cilk version of BWA, the alignment of one read is currently an indivisible task. Given the extreme difference between reads in terms of instruction counts and the relatively small number of reads per chunk compared to the total number of reads in the data set, it may be the case that while a certain worker thread is stuck aligning an extreme read, the other worker threads are idle because they cannot help out because the task of aligning a read cannot be divided, and there are no other reads left to align. In such a situation, there could be load imbalance even in a Cilk approach.

There are a number of additional optimisations we could try to assist the Cilk work-stealing scheduler:

1. We could try to parallelise the alignment of individual reads. That way the alignment of a read becomes a divisible task so that different worker threads can help processing the extreme reads.

2. We could reduce the number of chunking steps (cf. Section 6.6.3). This decreases the chance that a worker thread that is stuck with an extreme read prevents the other worker threads from doing any work because the chance would be higher that they could simply fetch other reads to work on. However, as we already discussed in Section 6.6.3, the best possible chunk size for a workload depends on the size of the workload and how much memory is available on the target server. Currently, the chunk size is hard-coded in the BWA program. We would either make this a user-configurable option or adapt the code to automatically detect the best chunk size.

3. We could pipeline the process of reading the reads from file into memory and the process of aligning reads. This way worker threads that cannot assist worker threads with tough reads may already start processing the next chunk of reads. This approach would keep the advantage of the chunking approach w.r.t. reducing the stress on memory use.

References
