pDindel: Accelerating InDel Detection on a Multicore CPU Architecture with SIMD

Da Zhang, Hao Wang, Kaixi Hou, Jing Zhang, Wu-chun Feng
Department of Computer Science, Virginia Tech
Blacksburg, Virginia, USA
Email: {daz3, hwang121, kaixihou, zjing14, wfeng}@vt.edu

***Abstract*—Small insertions and deletions (indels) of bases in the DNA of an organism can map to functionally important sites in human genes, for example, and in turn, influence human traits and diseases. *Dindel* detects such indels, particularly small indels (< 50 nucleotides), from short-read data by using a Bayesian approach. Due to its high sensitivity to detect small indels, Dindel has been adopted by many bioinformatics projects, e.g., the 1,000 Genomes Project, despite its pedestrian performance.**

In this paper, we first analyze and characterize the current version of Dindel to identify performance bottlenecks. We then design, implement, and optimize a parallelized Dindel (*pDindel*) for a multicore CPU architecture by exploiting thread-level parallelism (TLP) and data-level parallelism (DLP). Our optimized *pDindel* can achieve up to a 37-fold speedup for the computational part of Dindel and a 9-fold speedup for the overall execution time over the current version of Dindel.

***Index Terms*—short-read mapping; indel detection; Dindel; OpenMP; multithreading; vectorization.**

I. Introduction

Sequencing has come a long way since the first-generation sequencing technologies of the 1970s. Technologies from that time period, such as Sanger and shotgun sequencing, successfully contributed to the sequencing of the human genome by 2003 — at a cost of nearly $3,000,000,000 U.S. dollars (USD).

Today, the revolution continues with the short-read, massively parallel sequencing technique of *next-generation sequencing (NGS)*. In contrast to the slow performance and high cost for sequencing a human genome with first-generation sequencing techniques, NGS can make large-scale, whole-genome sequencing accessible and practical. For example, the HiSeq X$^{\text{TM}}$ Ten system, released by Illumina Inc. in 2014, can sequence over 45 human genomes in a single day at an approximate cost of only $1,000 USD per genome. As a consequence, a single NGS sequencer generating on the order of a gigabase (Gb) per run means that we can generate data faster than we can analyze it.

DNA sequencing (DNA-seq), a common NGS application, can discover genomic variations in the form of single nucleotide variants (SNVs), small DNA insertions and/or deletions (***indels***), copy number variations (CNVs), or other structural variants (SVs). During DNA sequencing, artifacts corresponding to different indels introduced by different alignments may affect the quality of the downstream variant calling. Thus, an *indel realignment* step is recommended to minimize those artifacts in post-alignment processing. Two algorithms for the indel realignment are usually adopted: (1) local realignment of gapped reads to the reference genome or alternative candidate haplotype or (2) local de novo assembly of the reads aligned around the target region, followed by construction of a consensus sequence for indel discovery [1]. Tools, such as Dindel [2], Genome Analysis Toolkit (GATK) [3], and SOAPindel [4], implement either of these two algorithms or a mixture of the two.

In this paper, we study Dindel, a bioinformatics application that uses a Bayesian approach for calling small (< 50 nucleotides) indels from short-read data. Dindel uses the former algorithm, i.e., local realignment of gapped reads to the reference genome or alternative candidate haplotype, to do the local realignment and deliver good sensitivity. Rather than improve the sensitivity of Dindel, we focus on studying and improving its performance from the perspective of a "big data" problem on a multicore CPU processor in order to create an optimized *parallel Dindel (pDindel)*.

We first work to improve the performance of the (sequential) Dindel program by restructuring loops to guide the compiler vectorization of *for* loops automatically. However, for Dindel, this auto-vectorization method cannot optimize all of the eight heavy-duty loops because of the indeterminate number of iterations, the divergent control flows, and the noncontiguous memory access patterns. Therefore, we manually vectorize these loops by using the loop reconstruction and the intrinsics from instruction set architecture (ISA).

After evaluating the performance of auto-vectorization and manual vectorization on each loop, we combine these vectorization techniques to achieve the best overall performance for the entire program. In addition to vectorization, i.e., data-level parallelism (DLP), we also exploit thread-level parallelism (TLP) to parallelize the most outer loops, which contain the computation of calculating read-haplotype likelihoods. We then evaluate different granularities of the *thread binding* and use the one that produces the best performance.

In all, this paper makes the following contributions:

1) An analysis of the Dindel program to identify the performance bottlenecks.
2) Acceleration of the compute-intensive functions within Dindel via thread-level parallelism (TLP) and data-level parallelism (DLP)
3) An evaluation of our optimized *pDindel* on a multicore processor with SIMD processing, which delivers up to a

978-1-4673-9963-9/15/$31.00 ©2015 IEEE

37-fold speedup for the computational part and a 9-fold speedup for entire Dindel program.

II. Related Work

VarScan [5], [6] is an open-source tool that uses heuristic and statistical methods to detect variants in sequences. The old version of VarScan [5] is implemented in Perl; the latest version VarScan2 [6] is implemented in Java. Compared with VarScan, Dindel can provide better accuracy and can be better optimized on modern multicore processors due to its C++ implementation. SAMtools [7] is a set of open-source utilities for manipulating alignments in the SAM/BAM files. The mpileup in SAMtools is based on a Bayesian approach to calculate the likelihoods of possible genotypes from the aligned reads. The Genome Analysis Toolkit (GATK) [3] is a collection of tools primarily for calling variants and genotyping. It also uses a Bayesian genotype likelihood model for calling indels. By adopting the MapReduce framework, it can provide a robust and powerful high-performance computing (HPC) solution to accelerate the processing of a large amount of sequence alignments in parallel. SOAPindel [4] from the Short Oligonucleotide Analysis Package (SOAP) package provides an efficient implementation to identify indels from short-paired reads. Compared to these tools, Dindel achieves the best sensitivity but with the longest execution time for calling small indels because Dindel examines all potential indels rather than a filtered list of indels [2], [4], [8].

III. background

Albers et al. proposed Dindel, short for *d*etection of *in*-*del*s [2]. It is a widely used bioinformatics software for calling small (< 50 nucleotides) indels from short-read data via a Bayesian approach. In order to detect small indels from short-read data, Dindel first realigns all reads to a set of candidate haplotypes generated from the targeting region and then calculates the posterior possibility for each candidate haplotype. As a result, an indel can be estimated. The main workflow of Dindel's realignment algorithm is described as follows. First, identify the set of reads $\{R_i\}$ to be realigned. Second, generate the set of candidate haplotypes $\{H_j\}$. Third, for each read R_i, compute the likelihood $P(R_i|H_j)$ on each haplotype H_j using the probabilistic realignment model and find the maximum $P_{max}(R_i|H_j)$. Fourth, estimate haplotype frequencies from the read-haplotype likelihoods $P_{max}(R_i|H_j)$ and the prior probability of the candidate haplotype. Fifth, estimate quality scores for the candidate indels and other sequence variants.

The sequential Dindel program walks through the read-haplotype pairs one by one independently and calculates the read-haplotype likelihood with the probabilistic realignment model in the third step of the Dindel algorithm. This is the most time-consuming part of Dindel and consumes more than 90% of the total execution time in our experiments. Within the likelihood computation of a read-haplotype pair, the alignment of *every base* in the read *with respect to the haplotype* can be different types: a read base is aligned to a haplotype base; a read base is a part of an insertion; or a read base is aligned to the left or the right of the haplotype.

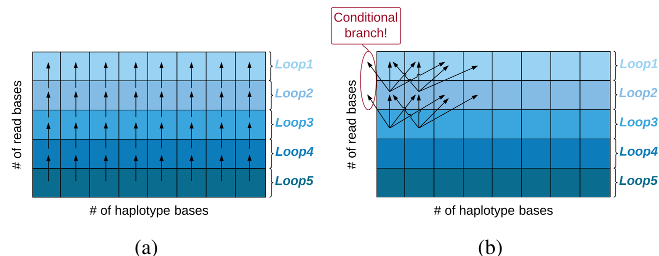

Fig. 1: Memory access pattern and dependency for updating the probabilities of the observed reads bases on generated haplotypes: (1a) regular memory access pattern; (1b) irregular memory access pattern. $A \rightarrow B$ means updating A depends on B.

The realignment model calculates the probability of the observed read base for a given alignment. Updating these probabilities for each read base with different alignments is the time-consuming component of the calculation of the read-haplotype likelihood. There are data dependencies between probabilities of two adjacent read bases. For a given read-haplotype pair, the updating process in the Dindel program is conducted by *eight loops* on a two-dimensional (2D) matrix, of which one dimension represents the read bases and the other represents the haplotype bases. Different loops introduce different memory-access patterns and dependencies. Figure 1 shows two examples. Specifically, Figure 1a shows a simple one-to-one pattern, and Figure 1b shows a "conditional branch-to-check the corner" case.

IV. Design and Optimization of Parallel Dindel (pDindel)

To improve the performance of Dindel, we seek to parallelize and optimize it for a multicore CPU architecture with SIMD vectorization. We first describe how to vectorize Dindel to exploit data-level parallelism (DLP) and then combine it with thread-level parallelism (TLP) to maximize performance benefits.

A. Data-Level Parallelism (DLP)

In a single instruction, multiple data (SIMD) multiprocessor system, each processing unit executes the same task on different pieces of data. The process of mapping a program onto a SIMD processor is called *vectorization* or *SIMDization* and can either be done automatically by the compiler or manually by the programmer. For example, the GCC/G++ compilers from the GNU Compiler Collection (GCC) can vectorize different kinds of loops, even with simple conditional branch or of nested form [9], by using the flag *-ftree-vectorize* or the flag *-O3*. When auto-vectorization fails, it is the programmer's duty to explicitly vectorize the code for better performance by using low-level intrinsics or even lower-level assembly instructions. The programmer also needs to carefully

handle data dependency, control flow, and corner cases. The *intrinsics* [10] provided by Intel support low-level DLP. Even though the program can be automatically optimized by the compiler, there are still a plenty of opportunities to boost the performance of Dindel by manually applying intrinsics on a multicore CPU with SIMD support because the compiler may miss some vectorization opportunities due to its conservative optimization strategies.

In Dindel, there are eight heavy-duty loops that update the probabilities of the observed read bases with respect to the given alignments on a two-dimensional (2D) matrix in Section III. The regular memory-access pattern and the independence between adjacent cells within one row point to the promise of vectorization. However, after checking the report of the G++ compiler (with flag *-O3* to enable the vectorization), we found that the compiler did *not* vectorize any of the eight loops for various reasons. We summarize these reasons as below:

- #1 ***Indeterminate number of loop iterations:*** The original Dindel code uses a *class data member* as its loop bound, which is considered to be indeterminate by the compiler. GCC/G++ cannot vectorize a *for* loop if its bound is indeterminate.
- #2 ***Divergent control flow:*** Some conditional branches in the loops prevent themselves from being vectorized due to the conservative optimization strategy of the compiler.
- #3 ***Noncontiguous memory access:*** There are two-level nested loops that result in noncontiguous memory access. These complex forms hinder the compiler to vectorize the loops.

We summarize the issues with the eight loops in Table I. While the compiler was unable to vectorize any of the loops, we were able to vectorize these loops by simply refactoring them or manually vectorizing them.

TABLE I: #number corresponds to a reason in the list above.

	reasons for being not vectorizable
loop1	#1, #2, #3
loop2	#1, #2
loop3	#1
loop4	#1, #2
loop5	#1, #2, #3
loop6	#1, #2
loop7	#1
loop8	#1, #2

1) Auto-vectorization: As discussed above, the compiler is not "smart" enough in some cases and needs hints from the programmer with respect to some constraints. We summarize our loop restructuring methods in Dindel below:

- #1 ***Eliminate the indeterminate number of iterations:*** We replace the class data member with a local variable for each loop. See Figures 2 and 3.
- #2 ***Eliminate divergent control flow:*** We apply different approaches to remove the divergent control flow: (1) Use the *ternary operator* (i.e. $e_1?e_2 : e_3$), which can be vectorized with the *mask* instructions by the compiler, in place of the "$if...else...$" conditional statement. (2) Split the loop into two branch-free loops, based on the condition to avoid the "$if...else...$" statement. (3) For the combinations of multiple conditions, use *bitwise operations*, which can be automatically vectorized by the compiler. See Figures 4 and 5 (Note: "&" and "|" are bitwise operations).
- #3 ***Eliminate noncontiguous memory access:*** We interchange the inner and outer loops to promote contiguous memory access and split the new outer loop to avoid divergent control flow, if possible. See Figures 6 and 7.

After applying these simple optimizations, we are enable to make G++ compiler vectorize all eight loops. The vectorization reports show that the simple loops can be completely vectorized, while some complex loops can only be partially vectorized.

2) Manual vectorization: To achieve better performance, we manually vectorize these loops by directly using SSE2 or AVX intrinsics — 128-bit SSE registers for 32-bit integer variables and 256-bit AVX registers for 64-bit double variables because AVX has weak support for integers. In order to handle

```
1: integer hapSize                              ▷ Class member
2: function UPDATE_PROBABILITY(...)   ▷ Member function
3:     for i = 1 to hapSize do
4:         // carry out some calculation
```

Fig. 2: Pseudocode of #1 before eliminating the indeterminate number of iterations.

```
1: integer hapSize                              ▷ Class member
2: function UPDATE_PROBABILITY(...)   ▷ Member function
3:     integer hSize = hapSize          ▷ Using local variable
4:     for i = 1 to hSize do
5:         // carry out some calculation
```

Fig. 3: Pseudocode of #1 after eliminating the indeterminate number of iterations.

```
1: for i = 1 to hapSize do
2:     if condition1 and condition2 or condition3 then  ▷ Multiple conditions
3:         a[i] ← b[i]
```

Fig. 4: Pseudocode of #2 before eliminating divergent control flow.

```
1: for i = 1 to hapSize do
2:     c ← condition1 & condition2 | condition3 ▷ Using bitwise operation
3:     a[i] ← c ? b[i] : a[i]        ▷ Using conditional operator
```

Fig. 5: Pseudocode of #2 after eliminating divergent control flow.

```
1: for i = 1 to hapSize do
2:     for j = 1 to MaxLengthDel do
3:         if j ≤ anchor then
4:             a[j][i] ← b[i] + d
5:         else
6:             a[j][i] ← c[i] + d
```

Fig. 6: Pseudocode of #3 before eliminating noncontiguous memory access.

```
1: for j = 1 to anchor do
2:     for i = 1 to hapSize do
3:         a[j][i] ← b[i] + d
4: for j = anchor + 1 to MaxLengthDel do
5:     for i = 1 to hapSize do
6:         a[j][i] ← c[i] + d
```

Fig. 7: Pseudocode of #3 after eliminating noncontiguous memory access.

```
1: const anchor
2: for i = 1; i < hapSize; i = i + 1 do
3:     if i < anchor then
4:         a[i] ← a[i] + b[i]                    ▷ Branch A
5:     else
6:         a[i] ← a[i] + e                       ▷ Branch B
7:     if a[i] < c[i] then
8:         a[i] ← c[i]                           ▷ Branch C
```

Fig. 8: Pseudocode before applying manual vectorization

```
1: const anchor
2: for i = 1; i < anchor; i = i + 4 do          ▷ Branch A+C
3:     vector1 ← a[i : i + 3]
4:     vector2 ← b[i : i + 3]
5:     vector1 ← vector1 + vector2
6:     vector3 ← c[i : i + 3]
7:     mask ← vector1 < vector3
8:     vector4 ← (mask & vector3) | (reversed_mask & vector1)
9:     a[i : i + 3] ← vector4
10: for i = anchor; i < hapSize; i = i + 4 do  ▷ Branch B+C
11:     vector1 ← a[i : i + 3]
12:     vector2 ← (e, e, e, e)
13:     vector1 ← vector1 + vector2
14:     vector3 ← c[i : i + 3]
15:     mask ← vector1 < vector3
16:     vector4 ← (mask & vector3) | (reversed_mask & vector1)
17:     a[i : i + 3] ← vector4
```

Fig. 9: Pseudocode after applying manual vectorization

divergent control flows, we use the mask intrinsics, which is a general solution to manually vectorize branches. Alternatively, some conditionals can be handled by splitting the original loop into several small loops if the resulting loops have contiguous memory access and regular access patterns.

Comparing the performance advantages between using mask intrinsics and splitting the loop, we find that splitting the loop delivers better performance than using mask intrinsics. Loop splitting can reduce the number of issued instructions by removing testing operations for the conditional while using mask intrinsics typically adds extra operations to handle the intermediate data (i.e., the masks) and to choose the correct results. Therefore, even though we use mask intrinsics to handle divergent control flow, we switch to the loop splitting whenever possible. Similar to the auto-vectorization, we also restructure complex nested loops to gain better memory-access patterns. In Figure 8, the first "$if...else...$" conditional is avoided by splitting the *for* loop into two sub-loops; and the second "if" conditional is handled by using the mask intrinsics. The vectorized example is shown in Figure 9. We also illustrate the mapping relationships of branches to the resulting loops in the figure.

We summarize the auto-vectorization and manual vectorization as follows. On the one hand, auto-vectorization can provide better programmability and flexibility, but there are a lot of constraints that impede performance. On the other hand, manual vectorization may be more efficient, but it reduces the programmability and portability of the program. In our experimental study, we profile both implementations, present our experimental results, and finally choose the approach, i.e., auto-vectorization vs. manual vectorization, that delivers better performance for each one of the eight heavy-duty loops in the code.

B. Thread-Level Parallelism

Thread-level parallelism (TLP) seeks to exploit the power of tightly-coupled, shared-memory multiprocessors via a multiple instruction, multiple data (MIMD) programming model. In C/C++, the general way to implement TLP is to use the POSIX threads (Pthreads) API. However, because Pthreads delivers low programmability, and moreover, because most of the computations in Dindel come from the *for* loops, we choose to use *OpenMP* — the state-of-the-art technique that provides portability, performance, and better programmability for practicing loop-level TLP in C/C++.

As mentioned in Section III, the Dindel program walks through read-haplotype pairs one by one *independently* for all identified reads to all generated candidate haplotypes. This is implemented in a two-level nested loop. We optimize these loops by applying the OpenMP parallel clause with the dynamic scheduling. The granularity, i.e., how many pairs of reads to haplotypes will be assigned to a thread each time, may significantly affect the performance. We test three primary granularities to identify the best solution.

1) One read-haplotype pair. (Apply OpenMP clause on the inner loop.)

2) All pairs of one read to all haplotypes. (Exchange two for loops and apply OpenMP clause on the inner loop.)
3) All pairs of all reads to one haplotype. (Apply OpenMP clause on the outer loop.)

Figure 10 shows each of these task granularities. The pink circles represent identified reads, and the blue circles represent generated candidate haplotypes. The green circles represent the smallest granularity that is the computation task for a read-haplotype pair. The purple and yellow containers represent packaging multiple smallest tasks into one large granularity task. The details of the performance of different granularities is discussed in Section V.

V. EVALUATION

Experimental Setup

Our experimental platform consists of two Intel(R) Xeon(R) CPU E5-2697 v2 running at 2.70 GHz. Each CPU has 12 physical cores with support for hyper-threading, i.e., two logical cores on each physical core. We use G++ 4.8.2 with flag *-O3 -mavx* for all the tests. Our experiments are carried out on a human genome sample HG01140 from the 1000 Genome Project with the hg19 reference sequence. We present the results of three input files: file1 contains 1004 windows; files2 contains 2008 windows; and file3 contains 10030 windows. For all input files, in the original Dindel, the calculation of the read-haplotype likelihood consumes more than 93% of total execution time, of which the eight loops consume 70% of the execution time.

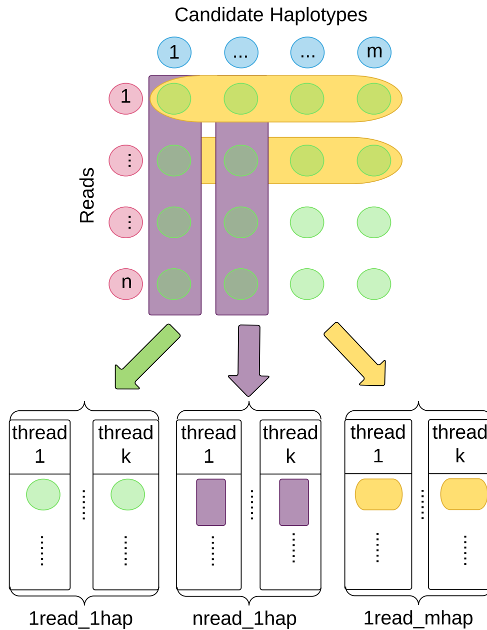

Fig. 10: Different Task Granularities: assuming there are n identified reads and m generated haplotypes in the targeting region, 1read_1hap means that a thread calculates the likelihood for a read-haplotype pair each time; nread_1hap means that a thread calculates the likelihood for all pairs of n reads to one haplotype each time; 1read_mhap means that a thread calculates the likelihood for all pairs of one read to all haplotypes each time.

Experimental Results

1) Vectorization: We compare the performance using auto-vectorization and manual vectorization for the eight loops to find the best optimization method for each one. As shown in Figure 11 with refactoring codes for all eight loops, auto-vectorization and manual vectorization deliver noticeably better performance than the original Dindel program, especially for loop 6. The compute-intensive operations of loop 6 deliver more benefits from the vectorization. (This loop with multiple conditions was shown in Figures 4 and 5.) In addition, we observe that in most cases, manual vectorization achieves greater speedup than auto-vectorization. This is because the compiler is too conservative in selecting the best optimizations even when the compiler flag *-O3* is used. For the loops with complex logical and control flows, which prevent the compiler from comprehensively and correctly analyzing the code, manual vectorization with the knowledge of the programmer can exploit more parallelism and achieve better performance.

However, we can observe that for loops 2, 3, and 7, auto-vectorization outperforms our manual vectorization. A possible reason is that along with the auto-vectorization, the compiler could also automatically analyze source code and apply other optimizations we did not apply in the manual optimization, e.g., software data prefetching, loop unrolling, loop tiling, etc., which can further improve performance.

2) OpenMP with different granularities: We test the performance with different granularities by applying TLP-only optimizations. The normalized speedup over the sequential

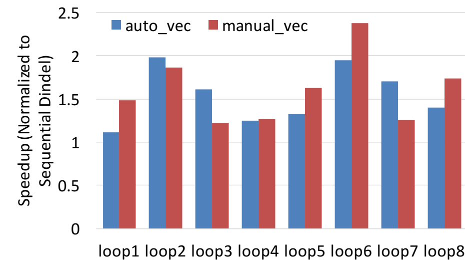

Fig. 11: Auto-vectorization vs. manual vectorization on eight *for* loops

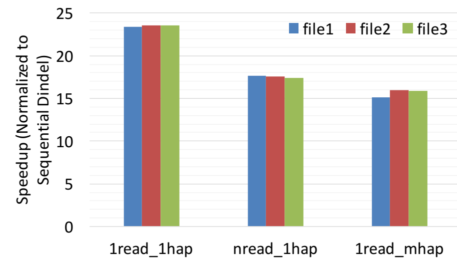

Fig. 12: Performance of using TLP with different granularities

Dindel program using 24 OpenMP threads for calculating the read-haplotype likelihoods is presented in Figure 12. As shown in the figure, the finest granularity provides the best performance and achieves up to a 23-fold speedup over the sequential Dindel code. This performance improvement comes from the workload balance across different threads. With the smaller granularity and dynamic scheduling of OpenMP, the threads will wait less time after finishing current workload. Based on this observation, we use the finest granularity in our final solution.

3) OpenMP+Vectorization: Finally, we test all of our optimizations by combining DLP and TLP optimizations (DLP+TLP). We also change the number of threads (24 threads or 48 threads) to check whether hyper-threading benefits overall performance. Figure 13a presents the normalized speedup for calculating the read-haplotype likelihoods, and Figure 13b presents the normalized speedup for the whole Dindel program. Although there are only a total of 24 physical cores across our dual 12-core CPUs, the optimized Dindel code performs best when using 48 threads, as the hyperthreading can hide the high memory-access latency. After applying vectorization (vec) in addition to the OpenMP multithreading (omp24 or omp48 in the figures), we achieve up to a 37-fold speedup for calculating the read-haplotype likelihoods and up to a 9-fold speedup for the total execution time of pDindel (normalized to the sequential Dindel code). Moreover, for the different input files, our implementation exhibits stable and scalable results.

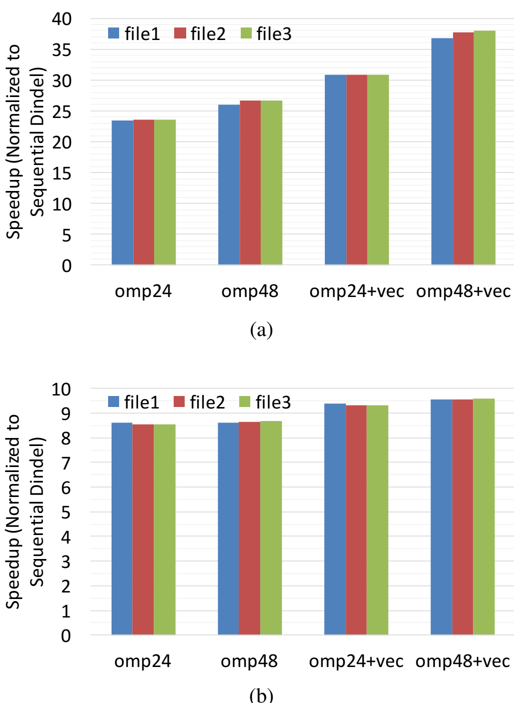

Fig. 13: Performance comparison with various optimizations: (a) Performance of calculating read-haplotype likelihoods with different optimization strategies; (b) Performance of Dindel with different optimization strategies. Note: vec: vectorized; omp#: OpenMP with # threads.

VI. Conclusion and Future Work

Dindel is an important bioinformatics application that calls small indels from short-read data. In this paper, we propose the design, implementation, and optimization of our parallel Dindel (pDindel) implementation on multicore processors with SIMD processing. In our evaluation, we illustrate that our optimized pDindel is scalable, stable, and can achieve up to a 37-fold speedup for the computational part and up to a 9-fold speedup for the overall execution time over the original Dindel implementation. In the future, we will extend our parallelism of Dindel to multiple nodes with the distributed memory programming models, e.g., MPI and MapReduce.

Acknowledgment

This work was supported in part by NSF-BIGDATA program via IIS-1247693 and NSF-XPS program via CCF-1337131.

The authors would like to thank to Dr. Nataliya Timoshevskaya for serving as the initial sounding board for parallelizing Dindel.

References

[1] R. Bao, L. Huang, J. Andrade, W. Tan, W. A. Kibbe, H. Jiang, and G. Feng, "Review of current methods, applications, and data management for the bioinformatics analysis of whole exome sequencing," *Cancer Informatics*, vol. 13, no. Suppl 2, pp. 67–82, 2014.

[2] C. A. Albers, G. Lunter, D. G. MacArthur, G. McVean, W. H. Ouwehand, and R. Durbin, "Dindel: Accurate indel calls from short-read data," *Genome Research*, vol. 21, no. 6, pp. 961–973, 06 2011.

[3] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, and M. A. DePristo, "The genome analysis toolkit: A mapreduce framework for analyzing next-generation dna sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, 09 2010.

[4] S. Li, R. Li, H. Li, J. Lu, Y. Li, L. Bolund, M. H. Schierup, and J. Wang, "Soapindel: Efficient identification of indels from short paired reads," *Genome Research*, vol. 23, no. 1, pp. 195–200, 01 2013.

[5] D. C. Koboldt, K. Chen, T. Wylie, D. E. Larson, M. D. McLellan, E. R. Mardis, G. M. Weinstock, R. K. Wilson, and L. Ding, "Varscan: variant detection in massively parallel sequencing of individual and pooled samples," *Bioinformatics*, vol. 25, no. 17, pp. 2283–2285, 09 2009.

[6] D. C. Koboldt, Q. Zhang, D. E. Larson, D. Shen, M. D. McLellan, L. Lin, C. A. Miller, E. R. Mardis, L. Ding, and R. K. Wilson, "Varscan 2: Somatic mutation and copy number alteration discovery in cancer by exome sequencing," *Genome Research*, vol. 22, no. 3, pp. 568–576, 03 2012.

[7] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 08 2009.

[8] J. A. Neuman, O. Isakov, and N. Shomron, "Analysis of insertion-deletion from deep-sequencing data: software evaluation for optimal detection," *Briefings in Bioinformatics*, 2012.

[9] Auto-vectorization in gcc. [Online]. Available: https://gcc.gnu.org/projects/tree-ssa/vectorization.html

[10] The intel intrinsics guide. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[11] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.