Accelerating InDel Detection on Modern Multi-Core SIMD CPU Architecture

Da Zhang

Collaborators: Hao Wang, Kaixi Hou, Jing Zhang

Advisor: Wu-chun Feng

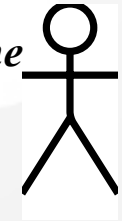
VirginiaTech
Invent the Future

SyNeRG
synergy.cs.vt.edu

Evolution of Genome Sequencing[1]

In 2003[2]:

1 human genome



3.4 billion

13 years

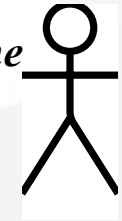
Hundreds of international universities and research centers

1. E.g. DNA sequencing: determining the precise order nucleotides within a DNA molecule.
2. Human genome project (HGP) used First Generation Sequencing technology and was completed in 2003.

Evolution of Genome Sequencing[1]

In 2003[2]:

1 human genome



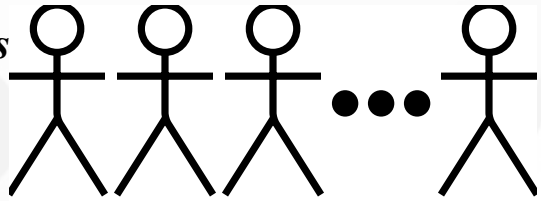
3.4 billion

13 years

Hundreds of international universities and research centers

In 2014[3]:

45 human genomes

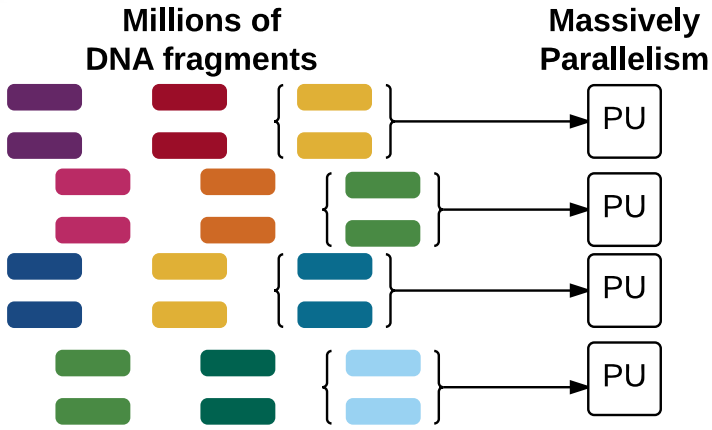


- *$1,000 per genome*
- *A single day*

1. E.g. DNA sequencing: determining the precise order nucleotides within a DNA molecule.
2. Human genome project (HGP) used First Generation Sequencing technology and was completed in 2003.
3. The *HiSeq X™ Ten*[3] released by Illumina Inc. in 2014.

Next Generation Sequencing (NGS)

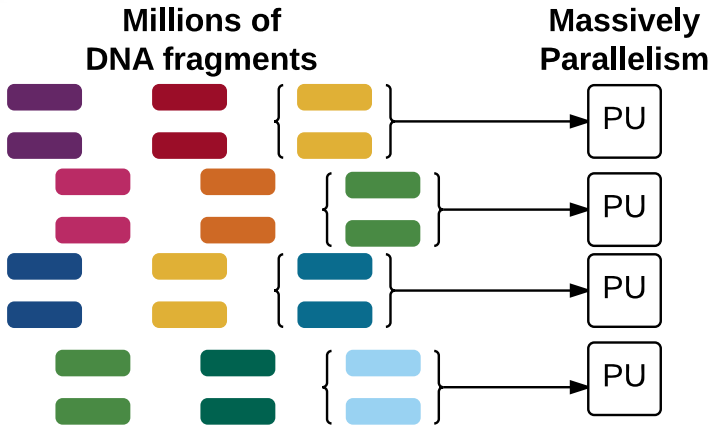
- Next Generation Sequencing (NGS): Millions of short read sequences are processed by tens/hundreds/thousands of processing units in a massively parallelism fashion.



Next Generation Sequencing (NGS)

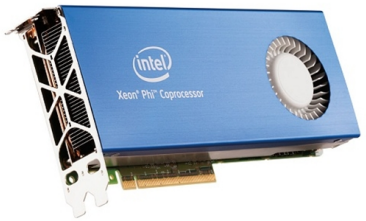
Next Generation Sequencing (NGS)

- Next Generation Sequencing (NGS): Millions of short read sequences are processed by tens/hundreds/thousands of processing units in a massively parallelism fashion.



Next Generation Sequencing (NGS)

- Many software with NGS techniques don't fully explore the computation power of the modern hardware (GPUs, APUs, MIC, FPGA …), due to the lack of expertise on their architectures!

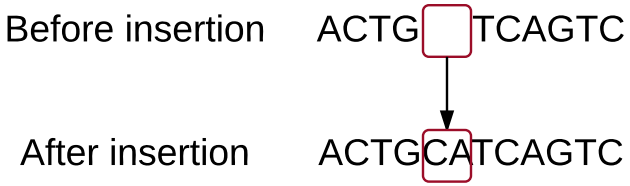


Indel and its Importance

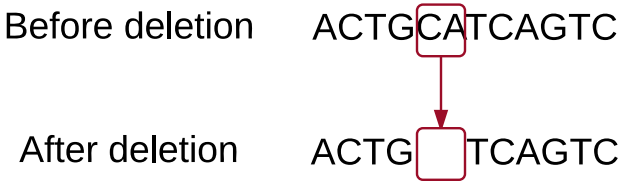
- What is ***indel*** ?
  - Biology term of ***In***sertion and ***Del***etion.

- Why are they so important?
  - Second most common type of polymorphism[1] variant in human genome.
  - Affecting ***human traits*** and causing ***severe human diseases***.
  - In NGS, ***Indels detection*** is recommended in a post-alignment step in order ***to reduce the artifacts*** generated during the sequence alignment.

-Insertion:



-Deletetion:



1. Polymorphism: a DNA variation that is common in population, cut-off point is 1%.

Indels Detection And Dindel

- Tools:
  - ***Dindel***, VarScan, SAMtools mpileup, the Genome Analysis Toolkit (GATK), SOAPindel and so forth.

Indels Detection And Dindel

- Tools:
  - ***Dindel***, VarScan, SAMtools mpileup, the Genome Analysis Toolkit (GATK), SOAPindel and so forth.

- ***Dindel*** [1]
  - Commonly used software that using a Bayesian based realignment model for calling small indels (<50 nucleotides) from short read data.
  - ***Advantage***: Highest sensitivity for calling small low coverage indels from short reads.
  - ***Disadvantage***: Time consumption.

1. Proposed by y C. A. Albers and et al. in 2010.

Indels Detection And Dindel

- Tools:
  - ***Dindel***, VarScan, SAMtools mpileup, the Genome Analysis Toolkit (GATK), SOAPindel and so forth.

- ***Dindel*** [1]
  - Commonly used software that using a Bayesian based realignment model for calling small indels (<50 nucleotides) from short read data.
  - ***Advantage***: Highest sensitivity for calling small low coverage indels from short reads.
  - ***Disadvantage***: Time consumption.

✧ ***Motivation & Opportunity: Dindel is a unoptimized sequential C++ program that is hindered from fully exploiting the computation power of today's high performance architecture!***

1. Proposed by y C. A. Albers and et al. in 2010.

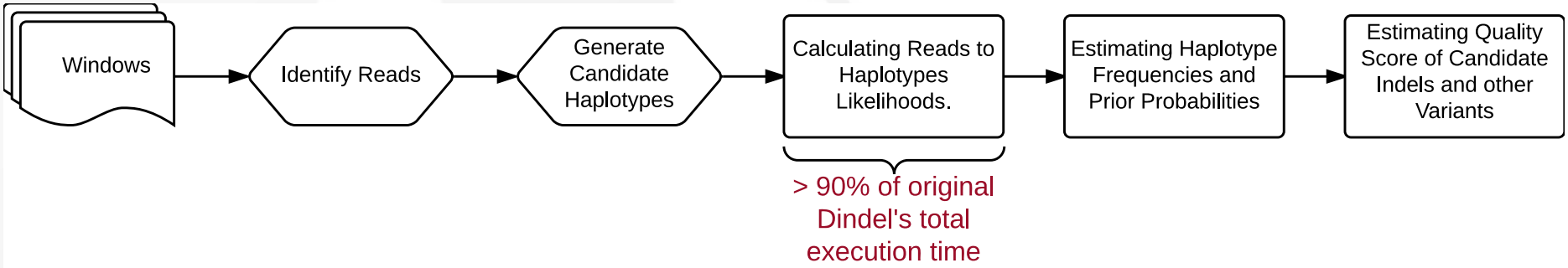
Contribution

1. Analyze Dindel and identify the ***performance bottlenecks***.

2. Accelerate Dindel with ***Thread-Level Parallelism (TLP)*** and ***Data-Level Parallelism (DLP)***.

3. Achieve up to ***37-fold speedup*** for the parallelized code region and ***9-fold speedup*** for the total execution time over the original Dindel on a modern multi-core processor.

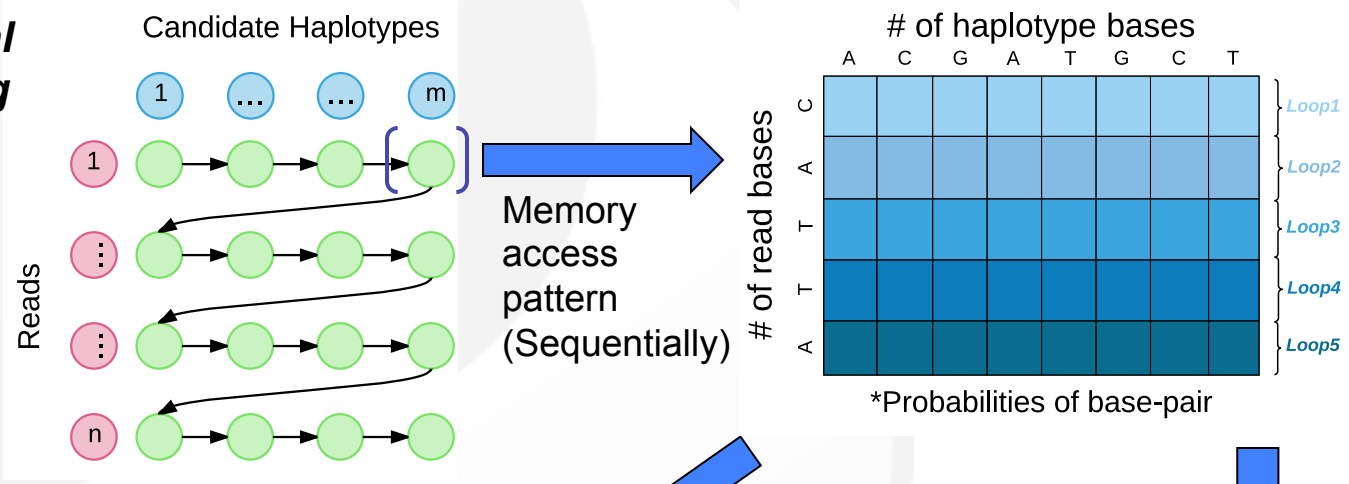
Dindel: Performance Bottlenecks



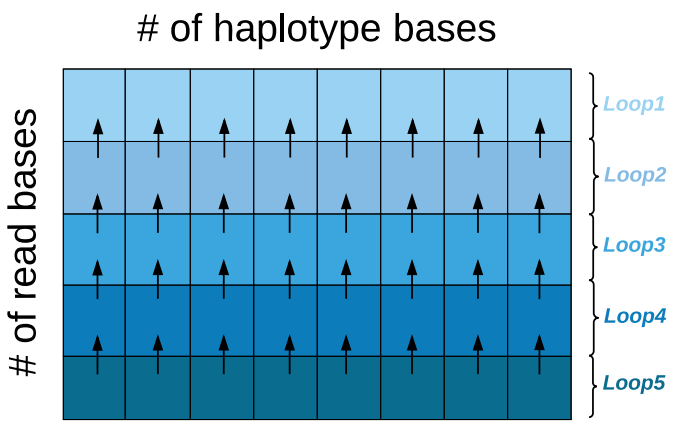
- More than 90% of the computation is spent on processing ***independent*** task sequentially!
- Most of these computation (>66%) presents lots of ***regular*** memory access patterns with some ***irregular*** patterns

Dindel: Sequential Processing & Memory Access Patterns

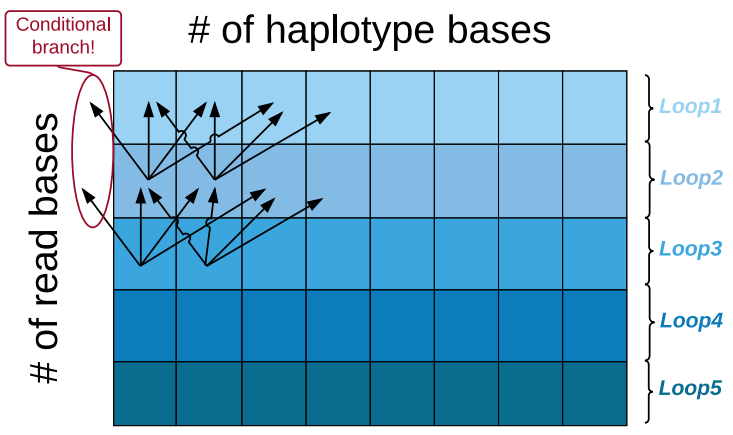
[1]Sequential processing



[2]Regular:



[2]Irregular:



1. In processing figure, green balls mean independent tasks, arrows mean work flow.
2. In memory access figures, arrows mean dependency, e.g. “A -> B” means “updating value of A depending on the value of B”.

Design & Optimization Overview

- Data-Level Parallelism (DLP)
  - Auto- and Manual- Vectorization
  - Mixture of auto- and manual- Vectorization
- Thread-Level Parallelism (TLP)
  - OpenMP
- Mixture of DLP and TLP

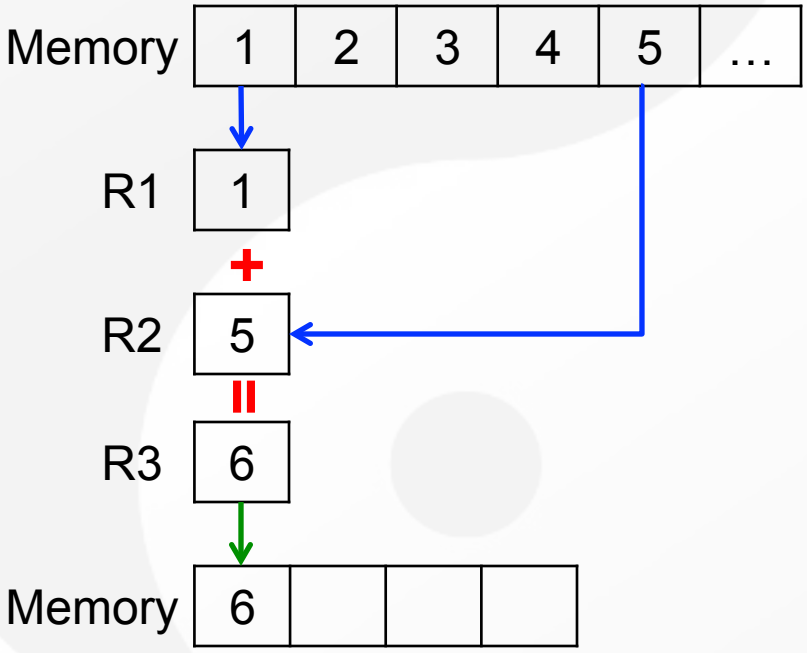
Data-Level Parallelism: Vectorization

- ***Vectorization (SIMDization)***: Preparing a program for using on a SIMD processor.
  - Auto- by compiler, e.g. GCC/G++, ICC/ICPC …
  - Manual- by the programmer, using low level *instrinsics*.

Data-Level Parallelism: Vectorization

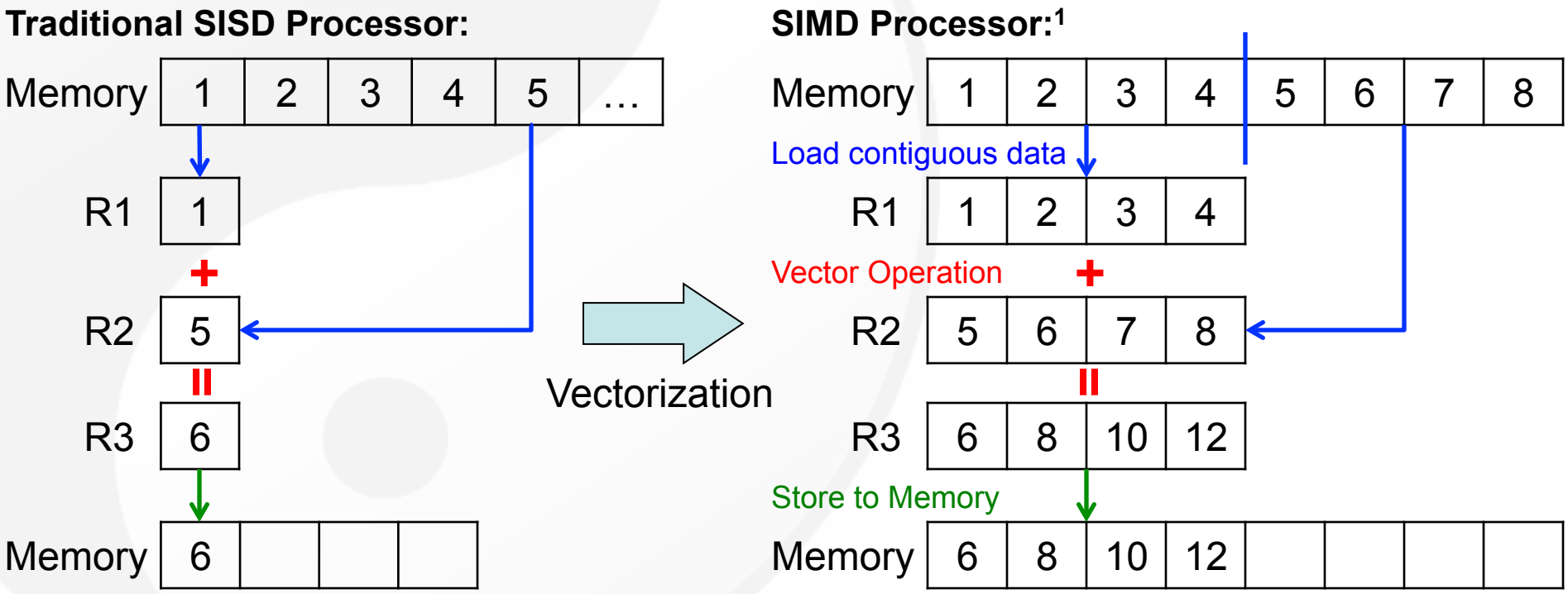
- ***Vectorization (SIMDization)***: Preparing a program for using on a SIMD processor.
  - Auto- by compiler, e.g. GCC/G++, ICC/ICPC …
  - Manual- by the programmer, using low level *instrinsics*.

Traditional SISD Processor:



Data-Level Parallelism: Vectorization

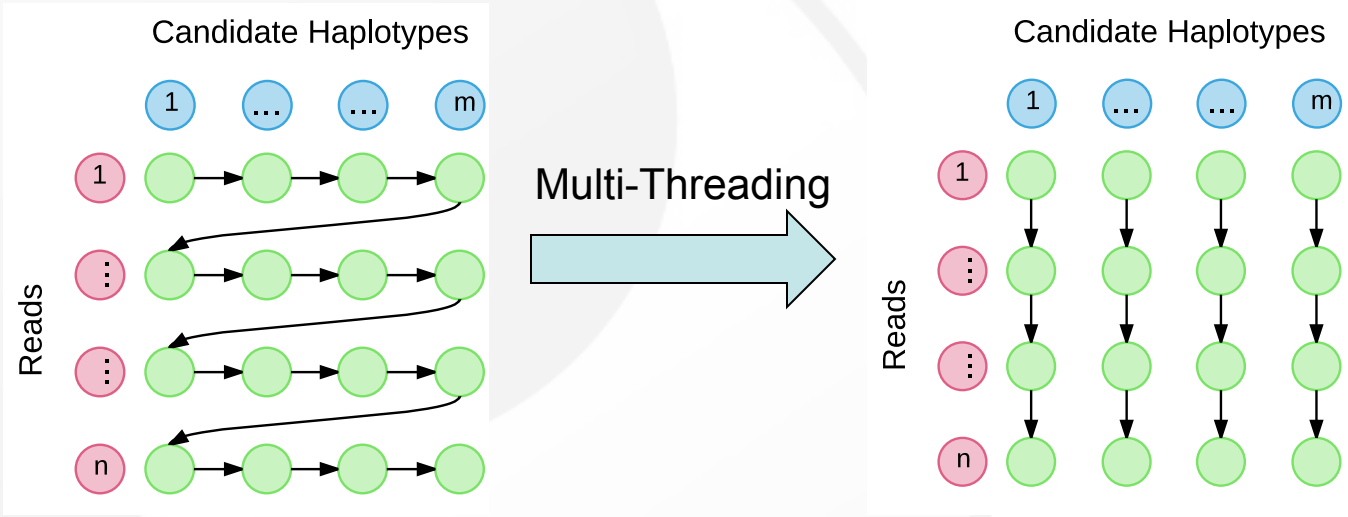
- ***Vectorization (SIMDization)***: Preparing a program for using on a SIMD processor.
  - Auto- by compiler, e.g. GCC/G++, ICC/ICPC …
  - Manual- by the programmer, using low level *instrinsics*.



✧ We adopt a mix solution (auto- & manual-) for best practice performance!

1. Assume processing 32-bit integers with 128-bit SSE register.
2. Now Intel's AVX512 support 512-bit register that can handle ***16*** 32-bit integers, ***8*** 64-bit doubles.

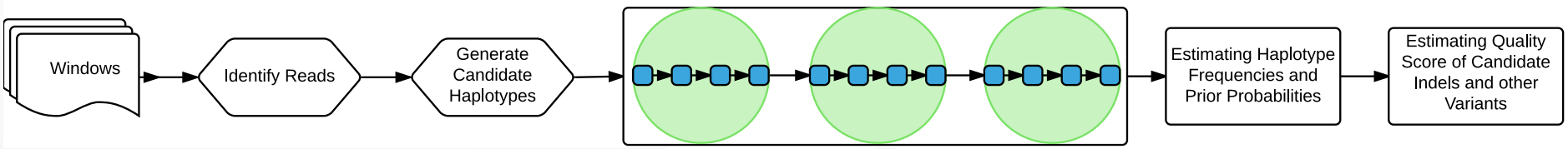
Thread-Level Parallelism



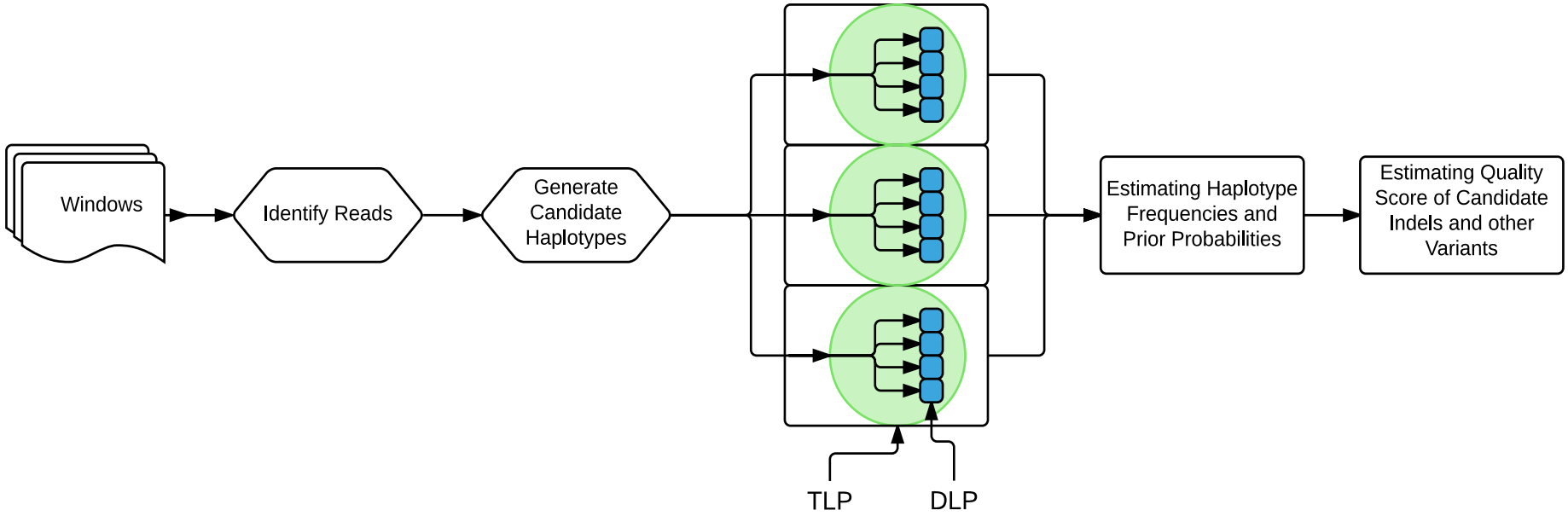
- ***Using OpenMP instead of traditional Pthread solution***
  - ***OpenMP***: A framework for developing parallel application in C,C++ and Fortran.
  - *Advantages*: Progrmmability, portability, same level performance as Pthreads.
- **Making each thread process one "green" task for load balance reason.**

[1] Most time consumption region within Dindel. Cover more than 90% sequential Dindel total execution time.

Dindel: Before & After Optimization



✧ Parallelization with TLP and DLP.



Evaluation: Set Up

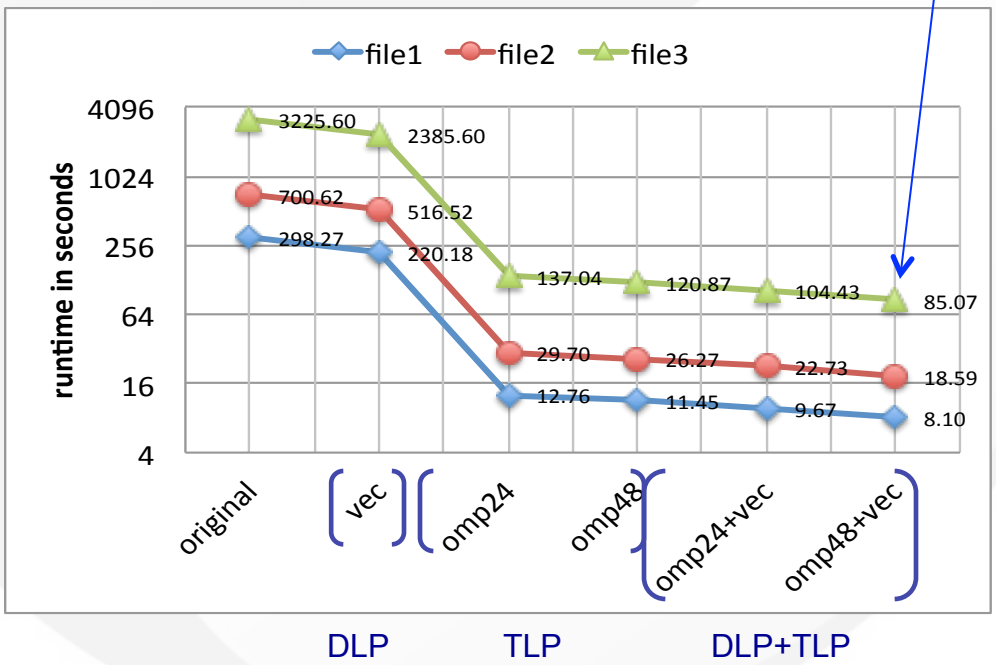
- CPUs: two Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz with ***AVX*** and ***Hyper-Threading***. There are ***24 physical cores*** and 48 logical cores.
- Compiler: G++ 4.8.2 with -O3 and -mavx.
- Three data sets:

Data sets	Original Dindel runtime in seconds	Likelihood calculation runtime coverage	Eight heavy-duty loops runtime coverage
file1	321	92.79%	69.56%
file2	757	92.52%	69.43%
file3	3480	92.68%	69.58%

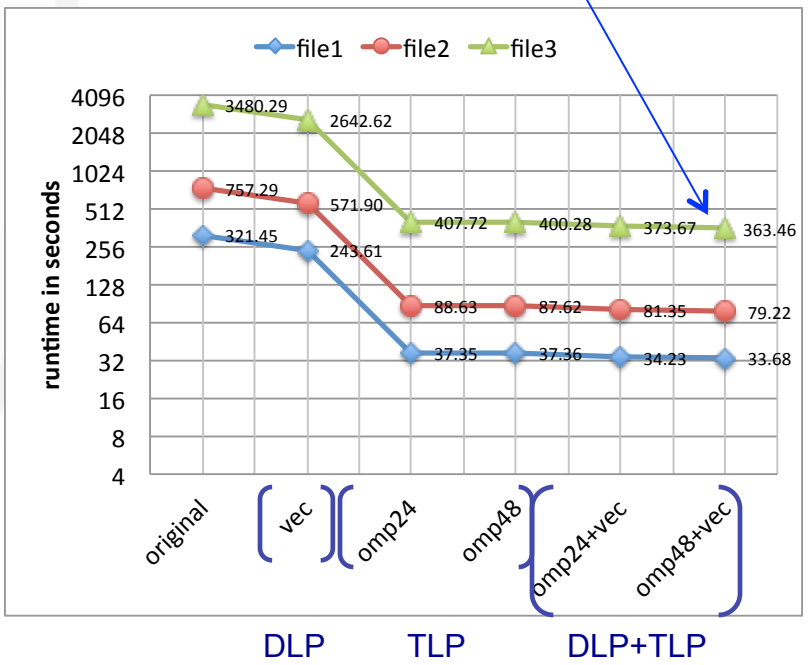
Performance Evaluation

- With 48 threads and vectorization, we achieve up to ***9-fold speedup*** for total execution time and ***37-fold speedup*** for calculation of read-haplotype likelihood.
  - Hyper-threading, which hide high memory access latency, further improve performance when increase threads number from 24 to 48.

Performance of parallelized region of the program:



Overall Performance:



- vec: vectorization.
- omp#: optimization with # threads.

Recall of Contribution

1. Analyze Dindel and identify the ***performance bottlenecks***.

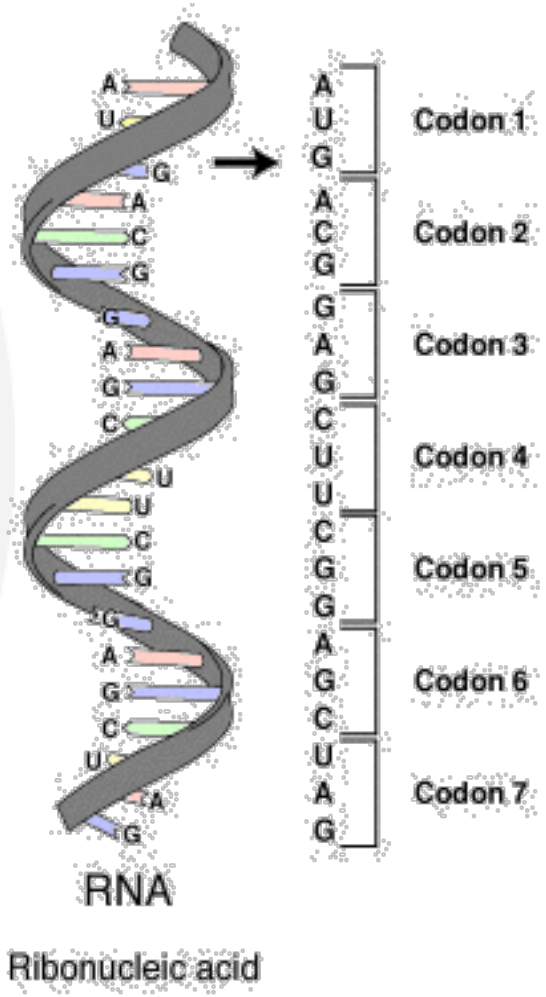
2. Accelerate Dindel with ***Thread-Level Parallelism (TLP)*** and ***Data-Level Parallelism (DLP)***.

3. Achieve up to ***37-fold speedup*** for the parallelized code region and ***9-fold speedup*** for the total execution time over the original Dindel on a modern multi-core processor.

BACKUP

Nucleic acid sequence

- Three most important molecules for all form of life:
  - DNA, RNA and Protein
- Nucleotide:
  - Biological molecules
  - Building blocks of DNA and RNA
- Nucleic acid sequence:
  - Succession of letters
  - Indicating the order of Nucleotides within a DNA or RNA molecule
- Short-read:
  - Reads that were shorter (e.g. 50-150bp) than the mainstream technologies(1000bp), but produce high-quality deep coverage of genomes.



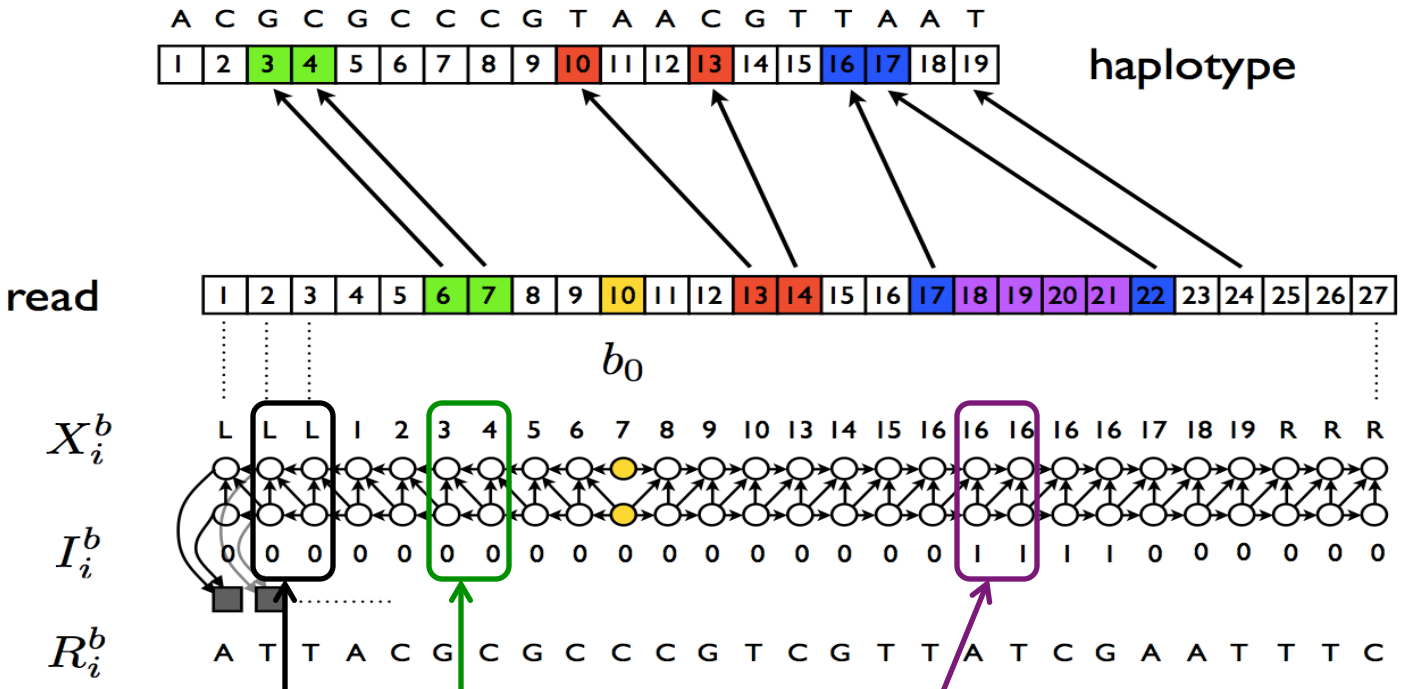
Indels Detection And Dindel

- Indels Detection Algorithms:
  - Local realignment of gapped reads to the reference genome or alternative candidate *haplotype*[1].
  - Local de novo assembly of the reads aligned around the target region followed by construction of a consensus sequence for indel discovery.
- Tools:
  - ***Dindel***, VarScan, SAMtools mpileup, the Genome Analysis Toolkit (GATK), SOAPindel and so forth.
- ***Dindel*** [2]
  - Commonly used software that using a Bayesian based realignment model for calling small indels (<50 nucleotides) from short read data.
  - Advantage: Highest sensitivity for calling small low coverage indels from short reads.
  - Disadvantage: Time consumption.

✧ ***Motivation & Opportunity: Dindel is a unoptimized sequential C++ program, that hindered from fully exploiting the computation power of today's high performance architecture!***

1. A haplotype is a set of DNA variations, or polymorphisms, that tend to be inherited together on the same chromosome.
2. Proposed by y C. A. Albers and et al. in 2010.

Dindel: Probabilistic Realignment Model



- b_0: Anchor point.
- X_i^b: position in haplotype that read base b aligns to.
- I_i^b: {0,1}, indicates whether read case b is part of insertion w.r.t. haplotype.

- The alignment types of read bases w.r.t. haplotype:
  - A read base aligns to a haplotype base.
  - A read base aligns to the left or right of the haplotype.
  - A read base is part of an insertion.

Vectorization Opportunities in Dindel

- *Eight heavy-duty loops* are not vectorized by compiler (G++ 4.8.2)!
  - ① Number of iterations can not be computed in advance.
    - Dindel uses a *class data member* as the loop bound. However, the vectorizer cannot recognize the class data member.
  - ② Divergent control flow.
    - Conditional branches prevent vectorization.
  - ③ Noncontiguous memory access.
    - Two-level nested loops result in noncontiguous memory access.

	Reasons of 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

Auto-Vectorization: Loop restructure1

- ***Loop restructure1 "number of iterations can not be computed in advance"***:
  - Replace the class data members with local variables for each loop boundary.

```
1: 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 restructure.

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

Fig. 3: Pseudocode of #1, after restructure.

Auto-Vectorization: Loop restructure2

- ***Loop restructure2 "divergent control flow"***:
  - Using ternary operator (also called conditional operator, i.e. $e_1 ? e_2 : e_3$) instead of "if...else..." condition.
  - Splitting the loop to two parts according to the condition and then avoid the "if...else..." condition.
  - For *combinations of multiple conditions*, use ***bitwise operations***.

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

Fig. 4: Pseudocode of #2,before restructure

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

Fig. 5: Pseudocode of #2,after restructure

Auto-Vectorization: Loop restructure3

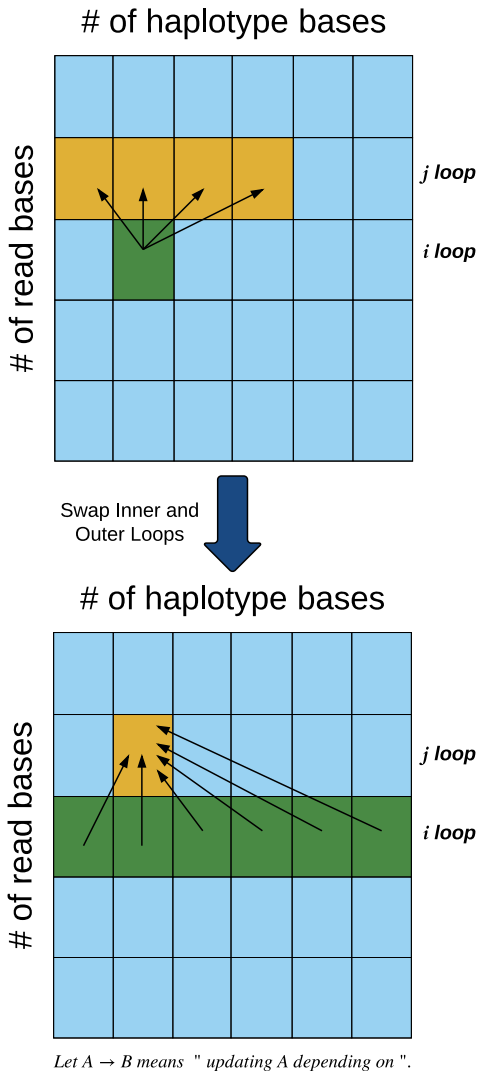
- ***Loop restructure3 "noncontiguous memory access"***:
  - Swapping the inner and the outer loops to make the memory access contiguous and separate the new outer loop to avoid control flow.

```
1: for i = 1 to hapSize do
2:     for j = 1 to MaxLengthDel do
3:         carry out some operations
```

Fig. 6: Pseudocode of #3,before restructure

```
1: for j = 1 to anchor do
2:     for i = 1 to hapSize do
3:         carry out some operations
4: for j = anchor + 1 to MaxLengthDel do
5:     for i = 1 to hapSize do
6:         carry out some operations
```

Fig. 7: Pseudocode of #3,after restructure



Manual-Vectorization

- ***Why?***
  - Compiler is conservative in optimization with respect to many restrictions.
  - With the knowledge of the program, the programmer is more likely to find vectorization opportunity.
- ***Intrinsics:***
  - ***Assembly-coded functions***, allow using C++ function calls and variable in place of assembly instructions and provide access to instructions that cannot be generated using standard constructs of the C and C++ language
  - Expended ***inline*** to eliminate function call overhead, same ***benefit as using inline assembly with readability***.
- ***Choosing intrinsics for Dindel:***
  - Using 128-bit SSE registers for 32-bit integer variables and 256-bit AVX registers for 64-bit double variables.
- ***Handling conditional branch:***
  1. *Using Mask*: General approach to handle branches in manual-vectorization.
  2. *Avoiding condition by splitting loop*: each resulting loop is condition-free, but only applicable to some conditions.

Better performance!

- ***Choosing 2 to handle conditional branch whenever it is applicable.***

Manual-Vectorization: Pseudocode

Splitting loop. Using mask.

```
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]
5:     else
6:         a[i] ← a[i] + e
7:     if a[i] < c[i] then
8:         a[i] ← c[i]
```

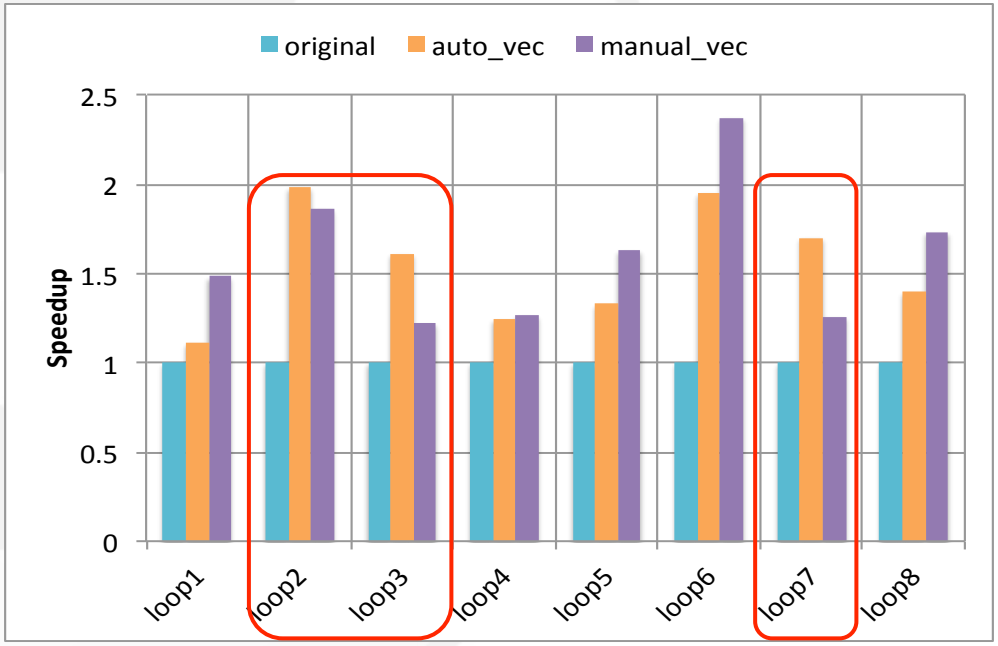
Fig. 8: Pseudocode before applying manual vectorization

```
1: const anchor
2: for i = 1; i < anchor; i = i + 4 do
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
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

Auto- vs. Manual- Vectorization

- ***Auto- win***: Compiler further improve performance by apply other optimizations, software prefetch, loop unrolling, etc. But compiler cannot apply these optimizations on intrinsics codes.
- ***Manual- win***: Computation-intensive operations. And the programmer is more confident and aggressive to do vectorization with the knowledge of the program.

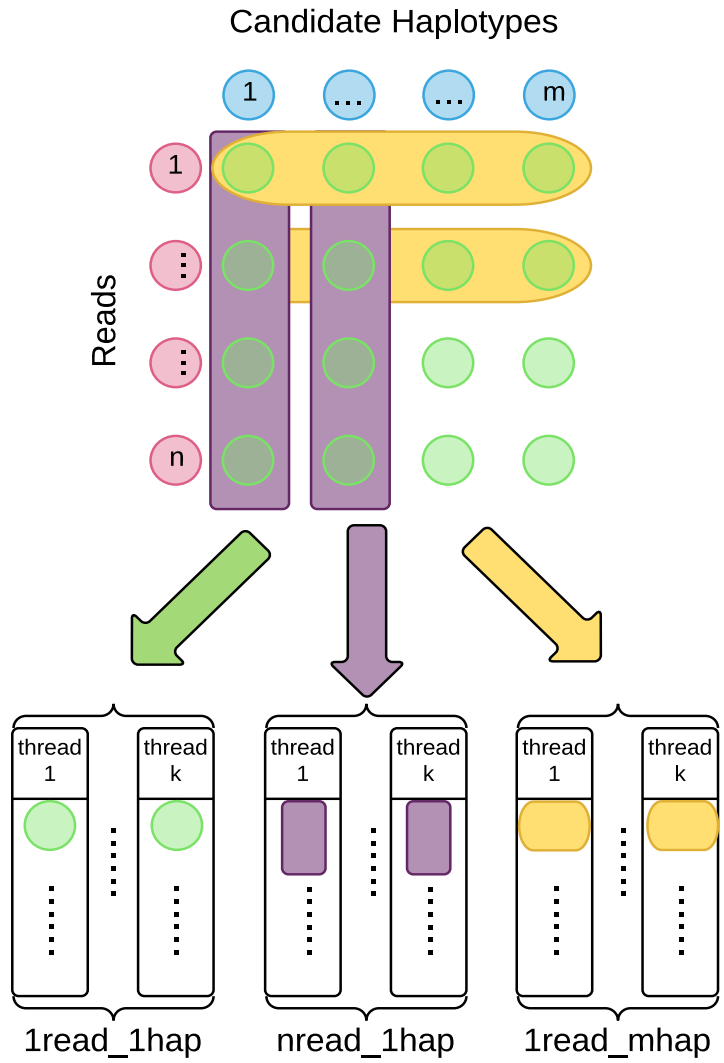


- **We choose either auto- or manual- vectorization if it beats the other!**

Thread granularities

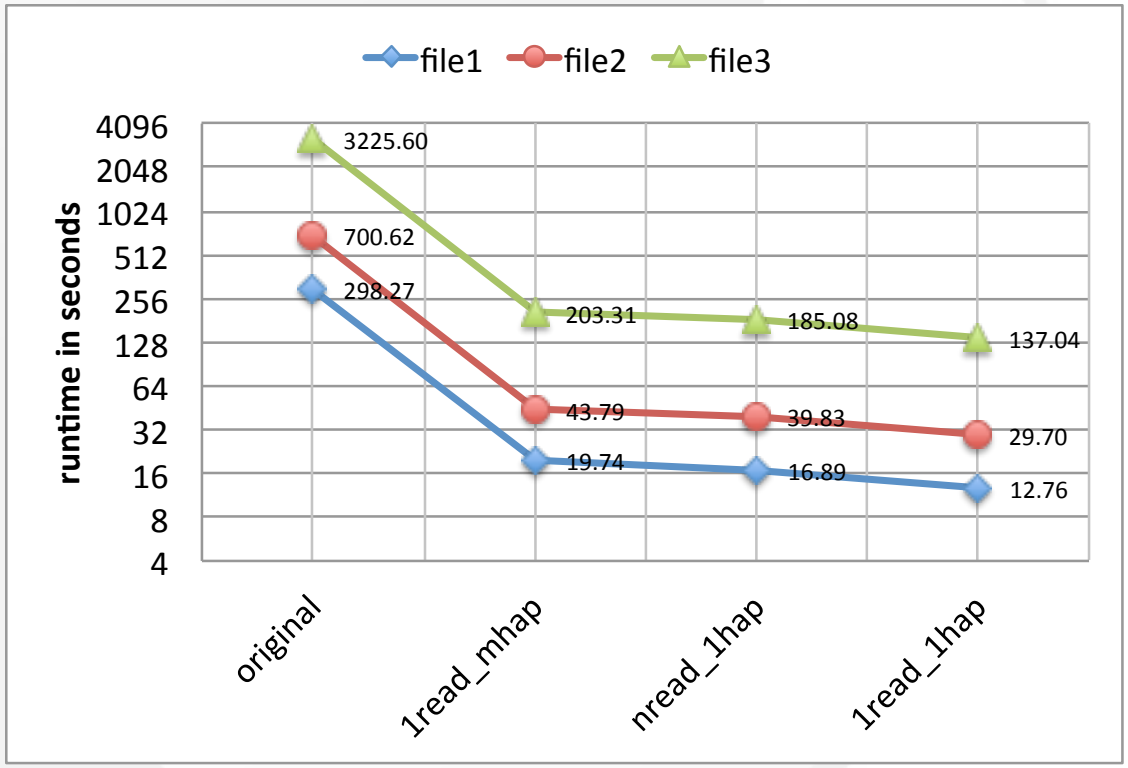
For every incoming task:

- ***1read_1hap***
  - Each thread process one pair of read-haplotype.
- ***Nread_1hap***
  - Each thread process pairs of all reads to one haplotype.
- ***1read_mhap***
  - Each thread process pairs of one read to all haplotypes.

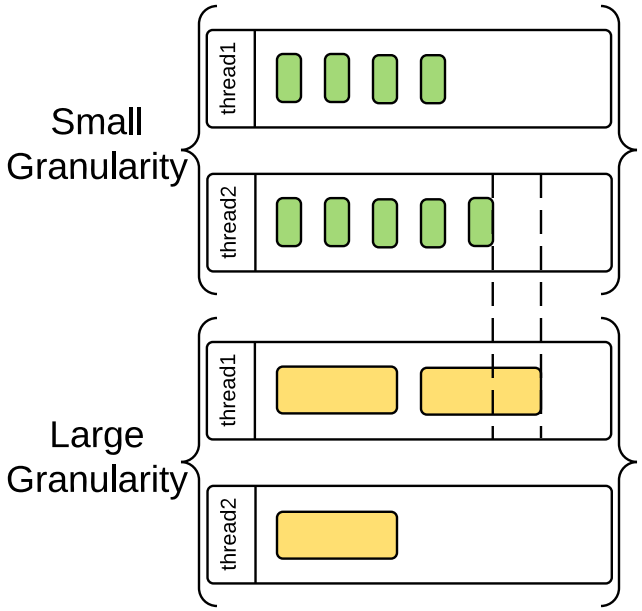


Performance of Different Granularities

- Up to 23-fold speedup for using 1read_1hap.
  - Primarily because workload imbalance.



Performance of using TLP with different granularities.[1]



Workload Imbalance

1. 24 threads with different granularities.