TELKOMNIKA, Vol.13, No.2, June 2015, pp. 694 ~ 702
ISSN: 1693-6930, accredited A by DIKTI, Decree No: 58/DIKTI/Kep/2013
DOI: 10.12928/TELKOMNIKA.v13i2.1415

Comparison of Data Partitioning Schema of Parallel Pairwise Alignment on Shared Memory System

Auriza Rahmad Akbar, Heru Sukoco*, Wisnu Ananta Kusuma
Department of Computer Science, Institut Pertanian Bogor
Jl Meranti Wing 20 Level 5, Darmaga, Bogor 16680, Indonesia
*Corresponding author, e-mail: hsrkom@ipb.ac.id

Abstract

The pairwise alignment (PA) algorithm is widely used in bioinformatics to analyze biological sequence. With the advance of sequencer technology, a massive amount of DNA fragments are sequenced much quicker and cheaper. Thus, the alignment algorithm needs to be parallelized to be able to align them in a shorter time. Many previous researches have parallelized PA algorithm using various data partitioning schema, but it is unknown which one is the best. The data partitioning schema is important for parallel PA performance, because this algorithm uses dynamic programming technique that needs intense inter-thread communication. In this paper, we compared four partitioning schemas to find the best performing one on shared memory system. Those schemas are: blocked columnwise, rowwise, antidiagonal, and blocked columnwise with manual scheduling and loop unrolling. The testing is done on quad-core processor using DNA sequence of 1000 to 16000 bp as the input. The blocked columnwise with manual scheduling and loop unrolling schema gave the best performance of 89% efficiency. The synchronization time is minimized to get the best performance possible.This result provided high performance parallel PA with fine-grain parallelism that can be used further to develop parallels multiple sequence alignment (MSA).

Keywords*: Data Partition, Pairwise Alignment, Parallel Processing, Shared Memory*

1. Introduction

The pairwise alignment (PA) algorithm is used in bioinformatics to align a pair of DNA or protein sequences of certain organism, in order to determine the similarity between them [1]. It uses dynamic programming technique to get the best alignment result with complexity of $O(n^2)$, where n is the sequences' length [2]. It is the foundation of the multiple sequence alignment (MSA) algorithm to align more than two sequences altogether. Other than that, it is also used for database sequence searching to find the most similar sequence to the one that is given [3].

The next-generation DNA sequencer technology nowadays can produce a lot of sequence data, up to hundreds of billion base pair (bp) in one run [5]. This big data needs faster processing, so the algorithm needs to be parallelized to speed up the alignment process. Many researches have parallelized MSA, such as Praline [6], ClustalW-MPI [7], MT-ClustalW [8], MAFFT [9], and star algorithm [10]. But only a few that have parallelized PA, such as ParAlign [11] and CudaSW [12].

The data dependency of PA is high due to its dynamic programming nature. Because of this, the data partitioning schema on parallel PA algorithm affects the performance greatly. Several data partitioning schemas that have been applied for PA parallelization were: columnwise [13], diagonal [11], rowwise [14], blocked columnwise [15], and blocked anti-diagonal [16]. Unfortunately, the best partitioning schema on shared memory system is not yet known.

In this paper, we parallelized PA algorithm on shared memory system using four different data partitioning schemas: blocked columnwise, rowwise, antidiagonal, and revised blocked columnwise. We tested and revised each schema to obtain the highest performance possible of parallel PA algorithm.

Received January 7, 2015; Revised March 14, 2015; Accepted April 2, 2015

2. Research Method

2.1. Implementation of Sequential Pairwise Alignment Algorithm

The first step is to implement PA algorithm in sequential manner using global alignment approach. The longest common sequence (LCS) algorithm [17] is used as a basis to develop the sequential PA algorithm. The LCS itself is a global alignment algorithm that is more known as Needleman–Wunsch algorithm. The alignment score is set as follow: the score is incremented by one if both DNA residues are match; else the score is substracted by one. The gap penalty is applied by initializing the score of zeroth row and zeroth column multiplied by its distance from starting point (upper-left corner).

An example of alignment table calculation using gap penalty of -3 can be seen on Figure *1*. This table aligns AGTCA and ATGA sequence resulting in alignment score of 1 (the value of right-bottom corner) and alignment result as follows.

AGTCA
A-TGA

This algorithm correctness is verified by aligning two sequences from BAliBASE DNA sequence alignment benchmark [18]. These sequences are *Saccharomyces cereviseae* GlyRS1 (SYG_YEAST) and *Schizosaccharomyces pombe* GlyRS (SYG_SCHPO). Our alignment result is compared with the result of ClustalW 2.1 program using default option.

		A	T	G	A
	0	-3	-6	-9	-12
A	-3	1	-2	-5	-8
G	-6	-2	0	-1	-4
T	-9	-5	-1	-1	-2
C	-12	-8	-4	-2	-2
A	-15	-11	-7	-5	-1

Figure 1. Global alignment for sequence AGTCA and ATGA

2.2. Implementation of Parallel Pairwise Alignment Algorithm

The second step is to develop the parallel version of this algorithm and verify its correctness. Parallelization is implemented using OpenMP, because it is much easier to implement rather than by using processor instruction directly [11] or by using Pthreads [8],[9]. OpenMP is a library to parallellize sequential program into multithreaded on a shared memory system [19]. Four partitioning schemas were tested: blocked columnwise, rowwise, antidiagonal, and blocked columnwise with manual scheduling and loop unrolling. The correctness of parallel PA is verified by comparing its output with the sequential one to satisfy the sequential consistency [20].

2.2.1. Blocked columnwise

The blocked columnwise partitioning schema divides the alignment table per block of columns [15]. The i-th thread (t_i) gets its part of a block from column $(\lfloor \frac{i \cdot n}{t} \rfloor + 1)$ to $(\lfloor \frac{(i+1) \cdot n}{t} \rfloor)$, where n is the number of columns and t is the number of threads. For example, if the number of columns was 9 and the number of threads was 3, then t_0 would get its part of a block of column 1–3. This partitioning schema is illustrated on Figure *2*.

2.2.2. Rowwise

The rowwise partitoning schema divides the alignment table per row [14]. The i-th thread (t_i) gets its part of $(i + xt + 1)$-th row, where $x = 0,1,2,\ldots,(\lceil \frac{m}{t} \rceil - 1)$, m is the number of rows, and t is the number of threads. For example, if the number of rows was 6 and the number of threads was 3, then t_0 would get its part of 1st and 4th row. This partitioning schema is illustrated on Figure 3.

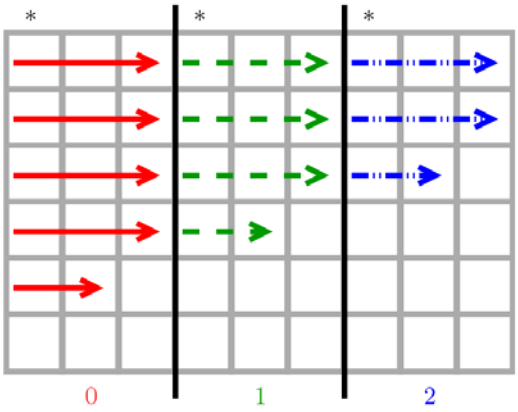

Figure 2. Blocked columnwise

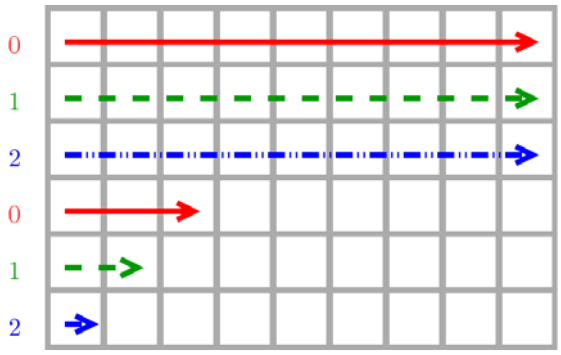

Figure 3. Rowwise

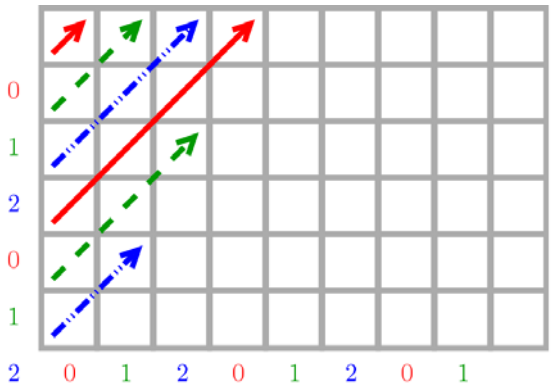

Figure 4. Antidiagonal

2.2.3. Antidiagonal

The antidiagonal partitioning schema divides the alignment table per antidiagonal, crossing from bottom-left to top-right [16]. The i-th thread (t_i) gets its part of $(i + xt + 1)$-th antidiagonal, where $x = 0,1,2,\ldots,(\lceil \frac{m+n-1}{t} \rceil - 1)$, m is the number of rows, n is the number of columns, and t is the number of threads. For example, if the number of rows was 6, the number of columns was 9, and the number of threads was 3, then t_0 would get its part of 1st, 4th, 7th, 10th, and 13th antidiagonal. This partitioning schema is illustrated on

Figure *4*.

2.2.4. Blocked columnwise with manual scheduling and loop unrolling

The first blocked columnwise schema is not optimal because the usage of OpenMP parallel for construct, that is simple but less flexible. To overcome this shortcoming, the data

partitioning mechanism needs to be done manually. The next revision is by applying loop unrolling technique, *i.e.* merging several loop iterations into one. This technique will reduce processor instruction and increase cache locality [21],[22].

2.3. Performance Comparison

The final step is to test the performance of each data partitioning schemas. Several DNA sequence data is generated randomly with varying length of 1000, 2000, 4000, 8000, and 16000 bp as the input. The alignment table calculation is timed using omp_get_wtime function. This process is repeated ten times for each schemas and sequence lengths, and then take the average as the final execution time. The final execution time is compared to get the best performing data partitioning schema.

The test is conducted using Intel Core i5-3470 processor that has 4 cores, Debian GNU/Linux 64-bit operating system, and GCC 4.8.1 compiler that supports OpenMP 3.1 specification.

3. Results and Analysis

3.1. Implementation of Sequential Pairwise Alignment Algorithm

The pseudocode for PA algorithm is presented as a procedure called PAIRWISE-ALIGN, which takes two sequences of *X* and *Y* as the parameters. The score of each cell is calculated by choosing the maximum score between three possible alignment directions: diagonal, up, and left. The diagonal score is calculated by the help of SIMILARITY procedure: if the DNA residue is equal, the score is added by *MATCH* score, else substracted by *MISMATCH* score. The up and left score are calculated by substracting it by linear *GAP* penalty score.

```
MATCH    = +1
MISMATCH = -1
GAP      = -3

SIMILARITY(a, b)
1  if a == b
2      return MATCH
3  else
4      return MISMATCH

PAIRWISE-ALIGN(X, Y)
  m = X.length
  n = Y.length
  let C[0..m, 0..n] be a new table
  for i = 0 to m
          C_{i,0} = i.GAP
  for j = 0 to n
          C_{0,j} = j.GAP

  for i = 1 to m
    for j = 1 to n
              ┌ diag = C_{i-1,j-1} + SIMILARITY(X_i, Y_j)
      C_{i,j} = MAX ├  up = C_{i-1,j} + GAP
              └ left = C_{i,j-1} + GAP
  return C
```

In general, our PA algorithm has been able to align two sequences correctly, whereas ClustalW produces alignment with more contiguous gap. This is caused by the lack of gap extension feature in our PA algorithm. The comparison of alignment result for SYG_YEAST and SYG_SCHPO between ClustalW and our PA is as follows. Based on this result, our simple PA algorithm has been implemented correctly and will become our basis to develop the parallel version of PA algorithm.

```
> ClustalW
SYG_YEAST
ATGAGTGTAGAAGATATCAAGAAGGCTAGAGCCGCTGTTCCATTTAACAGAGAACAGCTA.
..
SYG_SCHPO   ATGA---CAGAAGTT-TCA--AAGGC---AGCAGCT------
TTTGATCGAACTCAGTTC...
        ****   ***** * ***  *****  *** ***     *** *  **  *** *
> PairwiseAlign
SYG_YEAST
ATGAGTGTAGAAGATATCAAGAAGGCTAGAGCCGCTGTTCCATTTAACAGAGAACAGCTA
...
SYG_SCHPO   ATGA---CAGAAG-TTTC-A-AAGGC---AGCAGCT-TTTGATCGAACTCAGTTC-
G--A...
        ****   ***** * ** * *****  *** *** **  **  ***  **  * *  *
```

3.2. Implementation of Parallel Pairwise Alignment Algorithm

3.2.1. Blocked columnwise

The synchronization time that is caused by inter-thread data dependency for blocked columnwise schema is low. Only the first cell of each row (asterisk sign on

Figure *2*) that must be checked whether its left cell has been filled by another thread or not yet?. If a thread runs slower, then the next thread must wait until the former thread finished one row of its part.

The synchronization could be done only for $m \times t$ times, but because of OpenMP limitation the synchronization still be done for $m \times n$ times. The usage of OpenMP parallel for directive although simpler but less flexible, so the checking is still done at each cell. Thus, the parallel algorithm performance becomes less efficient. Below is the pseudocode to parallelize PA using this schema.

```
for i = 1 to m
   parallel for j = 1 to n                          // schedule(static)
      while C_{i,j-1} == null
         wait
            ┌ diag = C_{i-1,j-1} + SIMILARITY(X_i, Y_j)
      C_{i,j} = MAX ├  up = C_{i-1,j} + GAP
            └ left = C_{i,j-1} + GAP
```

The result yields up to 3.00 times faster execution time using blocked columnwise schema on 4 threads. Thus, the efficiency—speedup per number of threads—of this schema is 75%.

3.2.2. Rowwise

The synchronization time for rowwise schema is high, it is done for $m \times n$ times. Every single cell must check whether its upper cell has been filled by another thread or not yet. Despite of that, the parallel algorithm performance of this schema does not become worse. This is caused by the nature of shared memory system, *i.e.* no data movement involved between threads. Below is the pseudocode to parallelize PA using this schema.

```
parallel for i = 1 to m                             // schedule(static,1)
   for j = 1 to n
      while C_{i-1,j} == null
         wait
            ┌ diag = C_{i-1,j-1} + SIMILARITY(X_i, Y_j)
      C_{i,j} = MAX ├  up = C_{i-1,j} + GAP
            └ left = C_{i,j-1} + GAP
```

Unexpectedly, the result of rowwise schema is better than blocked columnwise schema. It yields up to 3.18 times faster execution time using this schema on 4 threads. Thus, the efficiency of this schema is 80%.

3.2.3. Antidiagonal

The synchronization time for antidiagonal schema is very high, two times of the rowwise schema, that is 2 × m × n times. Every single cell must check whether its left and upper cells have been filled by another thread or not yet. Moreover, the non-linear memory access pattern of this schema makes the parallel algorithm performance much worse. Below is the pseudocode to parallelize PA using this schema.

```
parallel for k = 1 to m+n                          // schedule(static,1)
  // upper diagonal
  if k < m
    for i = k downto 1, j = 1 to n
      while C[i,j-1] == null or C[i-1,j] == null
        wait
          ┌ diag = C[i-1,j-1] + SIMILARITY(X_i, Y_j)
      C[i,j] = MAX ├  up = C[i-1,j] + GAP
          └ left = C[i,j-1] + GAP
  // lower diagonal
  elseif k > m
    for i = m downto 1, j = k-m to n
      while C[i,j-1] == null or C[i-1,j] == null
        wait
          ┌ diag = C[i-1,j-1] + SIMILARITY(X_i, Y_j)
      C[i,j] = MAX ├  up = C[i-1,j] + GAP
          └ left = C[i,j-1] + GAP
```

As expected, the result of antidiagonal schema is the worst among the others. It yields up to 1.44 times faster execution time using this schema on 4 threads. Thus, the efficiency of this schema is only 36%.

3.2.4. Blocked columnwise with manual scheduling and loop unrolling

3.2.4.1. Manual scheduling

The loop scheduling is done manually using block decomposition method [23] as a substitute for OpenMP parallel for construct. Therefore, the synchronization can be done for only m × t times, that is only once at first cell of each row (asterisk sign on

Figure *2*). This manual scheduling makes blocked columnwise schema more efficient. Below is the pseudocode of blocked columnwise schema that has been revised using manual scheduling.

```
parallel
  id = Thread.id
  t  = Thread.size
  jfirst = ⌊id*n/t⌋ + 1
  jlast  = ⌊(id+1)*n/t⌋
```

3.2.4.2. Loop unrolling

The loop for each row will be unrolled by factor of two, *i.e.* iteration for two rows will be merged as one. This is done by making two loop copy for $C_{i,j}$ and $C_{i+1,j}$ and also using increment of two for each iteration. A checking mechanism is added in the middle of the loop to check whether the last row has been reached or not. This is necessary to anticipate when the number of rows (m) is not divisible by the unroll factor.

This loop unrolling technique by factor of two reduces the synchronization time to half, that is ½ × *m* × *t.* Below is the continued pseudocode of blocked columnwise schema that has been revised using loop unrolling by factor of two.

```
for i = 1 to m by 2
   while C[i,jfirst-1] == null
      wait
   for j = jfirst to jlast
                ┌ diag = C[i-1,j-1] + SIMILARITY(X_i, Y_j)
      C[i,j] = MAX ├  up = C[i-1,j] + GAP
                └ left = C[i,j-1] + GAP
      if i == m
         break
                ┌ diag = C[i,j-1] + SIMILARITY(X_i+1, Y_j)
      C[i+1,j] = MAX ├  up = C[i,j] + GAP
                └ left = C[i+1,j-1] + GAP
```

The result of the blocked columnwise schema with manual scheduling and loop unrolling by factor of two yields up to 3.54 times faster execution time on 4 threads. Thus, the efficiency of this schema is 89%.

3.2.5. Verification of Parallel PA Algorithm Correctness

All version of the parallel PA above have been tested for sequential consistency by comparing their outputs with the sequential PA algorithm. All of them produced the same output, therefore the parallel algorithm implementation are 100% consistent with sequential algorithm. This verification is important because incorrect implementation of multithreaded program leads to race condition that result in inconsistent output.

3.3. Performance Comparison

The antidiagonal schema yields the worst performance with efficiency of 36%. It is understandable because of its non-linear memory access pattern that causing a lot of cache misses.

The rowwise schema yields better performance than blocked columnwise schema with efficiency of 80% and 75% respectively. This result is beyond our expectation, because rowwise schema has more inter-thread data dependency. The main reason of this is because the implemention is on shared memory system, where data movement between threads is non-existent, *i.e.* each thread is accessing the same shared memory space. The result would be different if it is implemented on distributed memory system. Another reason is that the first blocked columnwise schema above is not optimal yet.

The second blocked columnwise schema that has been revised yields the best performance with efficiency of 89% on 4 threads. The loop unrolling technique (merging two row iterations into one) on this schema has been proven to reduce synchronization time, hence increasing the computation portion of the parallel algorithm. The comparison of those data partitioning schemas that have been tested is shown on Figure 5.

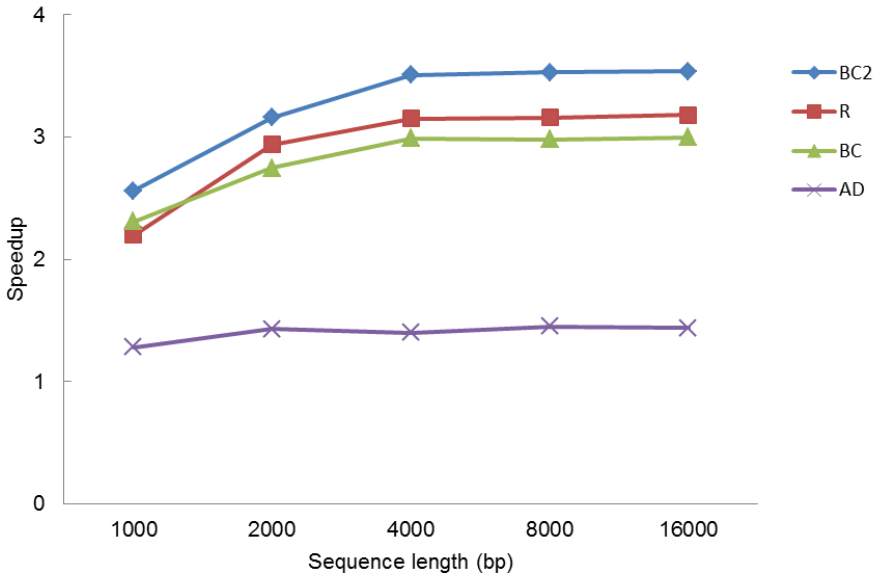

Figure 5. Performance comparison of antidiagonal (AD), blocked columnwise (BC), rowwise (R), and revised blocked columnwise (BC2) on 4 threads

4. Conclusion

The revised blocked columnwise schema using manual scheduling and loop unrolling yields the highest performance of 89% efficiency. The manual scheduling makes the blocked columnwise schema more efficient and the loop unrolling technique halves the synchronization time. In shared memory system, the performance is defined by the portion of synchronization time. The synchronization time must be minimized to maximize the performance.

We have presented parallel PA algorithm with high performance and fine-grain parallelism that could be used further as a component to develop parallel MSA algorithm on hybrid shared–distributed memory system using OpenMP and message-passing interface (MPI).

Acknowledgement

This research is funded by Cooperation Partnership of National Agricultural Research and Development (KKP3N) in 2013 from Indonesian Agricultural Ministry.

References

[1] Cohen J. Bioinformatics: an introduction for computer scientists. *ACM Computing Surveys (CSUR)*. 2004; 36: 122–158.
[2] Waterman MS, Smith TF, Beyer WA. Some biological sequence metrics. *Advances in Mathematics*. 1976; 20: 367–387.
[3] Edgar RC. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*. 2004; 32: 1792–1797.
[4] Rognes T, Seeberg E. Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*. 2000; 16: 699–706.
[5] Liu L, Li Y, Li S, Hu N, He Y, Pong R, Lin D, Lu L, Law M. Comparison of next-generation sequencing systems. *BioMed Research International*. 2012.
[6] Kleinjung J, Douglas N, Heringa J. Parallelized multiple alignment. *Bioinformatics*. 2002; 18: 1270–1271.
[7] Li KB. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*. 2003; 19: 1585–1586.
[8] Chaichoompu K, Kittitornkun S, Tongsima S. *MT-ClustalW: multithreading multiple sequence alignment*. IEEE International Parallel and Distributed Processing Symposium (IPDPS). Rhodes Island. 2006: 8.

[9] Katoh K, Toh H. Parallelization of the MAFFT multiple sequence alignment program. *Bioinformatics*. 2010; 26: 1899–1900.
[10] Satra R, Kusuma WA, Sukoco H. Accelerating computation of DNA multiple sequence alignment in distributed environment. *Telkomnika Indonesian Journal of Electrical Engineering*. 2014; 12(12): 8278–8285.
[11] Rognes T. ParAlign: a parallel sequence alignment algorithm for rapid and sensitive database searches. *Nucleic Acids Research*. 2001; 29: 1647–1652.
[12] Liu Y, Wirawan A, Schmidt A. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*. 2013; 14: 117.
[13] Hughey R. Parallel hardware for sequence comparison and alignment. *Computer Applications in the Biosciences: CABIOS*. 1996; 12: 473–479.
[14] Martins WS, del Cuvillo J, Cui W, Gao GR. *Whole genome alignment using a multithreaded parallel implementation*. Symposium on Computer Architecture and High Performance Computing. Vitória. 2001: 1–8.
[15] Liu W, Schmidt B. *Parallel design pattern for computational biology and scientific computing applications*. IEEE International Conference on Cluster Computing. Hongkong. 2003: 456–459.
[16] Li J, Ranka S, Sahni S. *Pairwise sequence alignment for very long sequences on GPUs*. IEEE International Conference on Computational Advances in Bio and Medical Sciences. Las Vegas. 2012: 1–6.
[17] Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms*. Third Edition. Cambridge: MIT Press. 2009: 390–396.
[18] Carroll H, Beckstead W, O'Connor T, Ebbert M, Clement M, Snell Q, McClellan D. DNA reference alignment benchmarks based on tertiary structure of encoded proteins. *Bioinformatics*. 2007; 23(19): 2648–2649.
[19] Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*. 1998; 5: 46–55.
[20] Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on.* 1979; 100: 690–691.
[21] Loveman DB. Program improvement by source-to-source transformation. *Journal of the ACM (JACM)*. 1977; 24: 121–145.
[22] Sedgewick R. Implementing quicksort programs. *Communications of the ACM*. 1978; 21: 847–857.
[23] Quinn MJ. *Parallel Programming in C with MPI and OpenMP*. New York: McGraw-Hill. 2003: 118–119.