International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 12, Number 2, pages 446–466, July 2022

PZLAST: an ultra-fast sequence similarity search tool implemented on a MIMD processor

Hitoshi Ishikawa
PEZY Computing, K.K., Tokyo, Japan

Hiroshi Mori
Department of Informatics, National Institute of Genetics, Shizuoka, Japan

Koichi Higashi
Department of Informatics, National Institute of Genetics, Shizuoka, Japan

Yoshiaki Kato
Computational Astrophysics Laboratory, RIKEN, Saitama, Japan

Tomofumi Sakai
Infinite Curation, Inc, Tokyo, Japan

Toshikazu Ebisuzaki
Computational Astrophysics Laboratory, RIKEN, Saitama, Japan

Ken Kurokawa
Department of Informatics, National Institute of Genetics, Shizuoka, Japan

Received: January 5, 2022
Revised: April 6, 2022
Accepted: May 30, 2022
Communicated by Akihiro Fujiwara

Abstract

We have developed an ultra-fast sequence similarity search tool named PZLAST on a MIMD processor PEZY-SC2. In this paper, we show the merit of MIMD features in reducing the load imbalance among the threads. They are also effective in the implementation of the alignment for long sequences. Additionally, we point out two problems related to the implementation on an ultra-parallel computation accelerator as follows: (1) Deciding the optimal amount of inputs prior to the run is extremely difficult and usually even impossible, and (2) Keeping up the parallelism efficiently throughout the whole computation is not always possible. A feedback strategy and an accumulation strategy are proposed to overcome these problems and their results are shown to be valuable in reducing the accidental memory overflow in runtime and speeding up the processing time.

Keywords: Sequence Similarity Search Tool, MIMD Processor, PEZY-SC, PEZY-SC2, BLAST, CLAST, PZLAST

1 Introduction

By digitizing the genomic information of microbiomes in the environment using a sequencer and analyzing it, various findings including the statistical properties of their communities can be obtained. In the metagenomics field, frequently the genomic information already known is stored as a reference database, and the unknown genomic information is collated with the reference database. Although there are various ways to analyze unknown genomic information depending on the purpose, it is a usual procedure to start with searching the matching points and homologies between the reference sequences in the known database and the unknown query sequences.

Recently, NGS (Next Generation Sequencing) [3] has made it possible to digitize a large amount of genomic information at a higher speed than the conventional sequencers, which extremely enriches both the size of reference database and the newly sequenced queries. The computational cost required for analyzing these large data has also increased enormously.

From the computational point of view, many-core processors such as GPUs have been effectively used for various applications, which have been often used in supercomputer systems as well. According to the execution form of the instructions, they are categorized into SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) types. The SIMD type processors are characterized in that all the threads in the processor execute the same instruction at the same time, whereas the MIMD type processors allow each thread to execute each different instruction. Currently, there exist the other categories between SIMD and MIMD, where synchronization is required in a unit (e.g. warp) but not required between the units. However, the majority of current GPUs are hard to handle a thread level independency. In this paper, we show that a genomic sequence similarity search tool can be efficiently constructed on PEZY-SC2, which is a MIMD type many-core processor. Additionally, we introduce two strategies to utilize the parallelism of the multiple threads efficiently in this application and examine their effectiveness.

2 Related work

2.1 BLAST

```
>sequence1
GTCCACGCAGCCGCTGACAGACACACCATGAGAACC
TTACGGGACTACCCGA
>sequence2
TGGCCGGGCAGATGCAAAGCCTGGTGATGCAGAGTC
GGGCAAAGGCGCAGCCTTCGTGTCCAAGCAGGAGGG
CAGCGAGG
...
```

Figure 1: An example of FASTA.

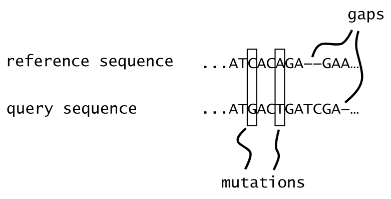

Figure 2: An example result of sequence alignment including the gaps and mutations.

A digitized nucleic acid sequence is typically represented by a succession of the four letters of ACGT (DNA) or ACGU(RNA). In the case of protein, it needs more letters to express the amino acids. Figure 1 is an example of FASTA file format commonly used in the field of bioinformatics. A sequence name is described just after '>' followed by its body represented as a string. Sequence similarity searching is a method of searching sequence databases to a query sequence by using alignment, which finds the matching points, homologies and the other related characteristics. Figure 2 illustrates an example of the alignment result which includes the gaps and mutations between the sequences. A sequence similarity search tool calculates the homologies not only for exact matching cases but also for such ambiguous cases. Typically, a sequence similarity search tool refers two sequence groups (reference and query sequences) as its input, and calculates the matching points and homologies between the sequence selected from the reference sequences and the one selected from the query sequences. Since every combination of the reference and query sequences are searched, in the case that the number of sequences becomes large, the computational cost increases enormously. BLAST (Basic Local Alignment Search Tool) [6] is a basic, still popular alignment tool in the bioinformatics field. It performs high-speed and sensitive sequence similarity searching using a method called seed-and-extend alignment. Recently, BLAST+ [1] has been developed, which is a rewritten and improved version of BLAST. It achieves substantial speed improvements compared to the conventional BLAST.

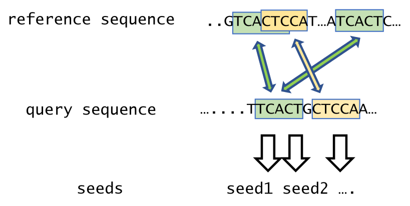

Figure 3: An example of seed creation, where the k-length is 5.

The seed-and-extend alignment method consists of the following two steps.

In the first step, BLAST tries complete matches of k-length string between the reference and query sequences, and the seeds are created at the matching positions (Figure 3).

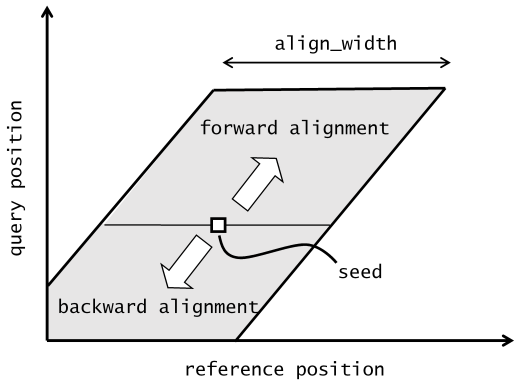

Figure 4: Forward and backward alignment.

In the second step, BLAST performs alignment using dynamic programing for each seed. An alignment consists of a forward and a backward alignment (Figure 4). In the forward alignment, alignment is performed from the seed position toward the end of the sequence, while in the backward alignment, toward the beginning.

2.2 CLAST

2.2.1 CLAST overview

While the target processors of BLAST and BLAST+ are CPU, CLAST (CUDA implemented large-scale alignment search tool) [4] targets at GPU processors. CLAST is a CUDA implementation of BLAST improved in search speed utilizing a huge number of GPU threads in parallel.

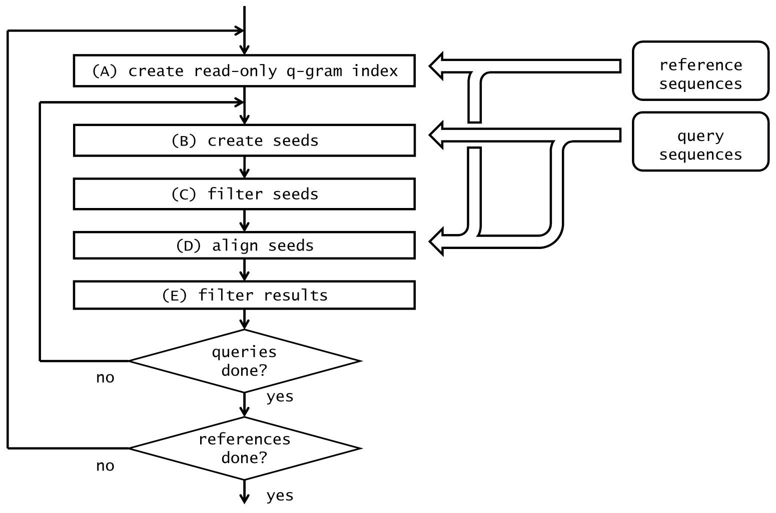

Figure 5: CLAST whole computation flow.

Figure 5 illustrates the CLAST whole computation flow which consists of (A) to (E) phases.

In (A) phase, a search table suitable for k-length complete string matching is created from the reference sequences. Next in (B) phase, seeds are made applying the query sequences against the search table using binary searching. Since the seeds created in (B) phase usually include redundant ones, they are reduced by filtering in (C) phase. Then, each seed is aligned strictly using dynamic programming in (D) phase and the final results are filtered in (E) phase according to their alignment length and score. Each phase in Figure 5 is designed suitable for ultra-parallel computation on a huge number of threads, which contributes to the speedup of computation. Since the size of GPU device memory is limited, in the case the reference size or query size are too large to be stored in it, they are divided into separate fields and computed in multiple iterations. Each loop in Figure 5 corresponds to the division of reference and query sequences respectively.

2.2.2 Filtering the seeds

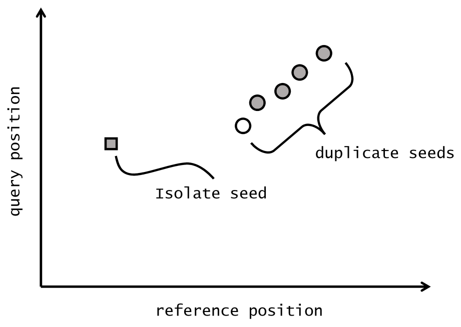

Figure 6: An example of redundant seeds (an isolate seed and duplicate seeds).

Figure 6 illustrates an example of redundant seeds created in (B) phase. The rectangle marker represents an isolate seed around which no other seed is generated. In such case, it is probable that the seed is incorrect. The circle markers are duplicate seeds, which are likely to give the same results by the following (D) phase, and better to be unified into one. Such redundant seeds are filtered out in (C) phase. In most cases, the number of seeds after (C) phase is dramatically reduced, which contributes to a speedup of the alignment in (D) phase.

Despite the above advantages of the filtering in (C) phase, finding the redundant seeds needs sorting process whose computational cost cannot be ignored when the number of seeds becomes huge.

2.2.3 Alignment in a diagonal direction

In (D) phase, alignments are conducted using dynamic programming.

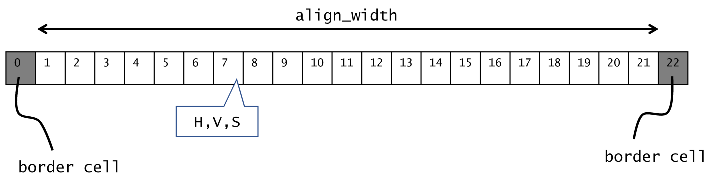

Figure 7: An example of working buffer for alignment.

Figure 7 shows an example of working buffer used in the alignment. It is an array of (`align_width` + 2 border) cells. Each cell includes `H`, `V` and `S`, which represent the horizontal, vertical and maximum scores respectively. These variables in the two border cells are initialized with a very low score prior to an alignment.

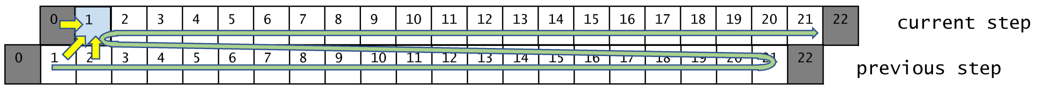

Figure 8: Alignment at diagonal movement.

In an alignment, the working buffer moves diagonally, updating each cell in turn as shown in Figure 8.

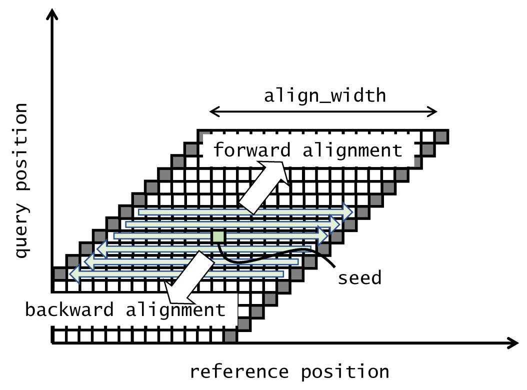

Figure 9: Forward and backward alignment with a working buffer.

Repeating this process multiple times realizes the whole alignment as shown in Figure 9.

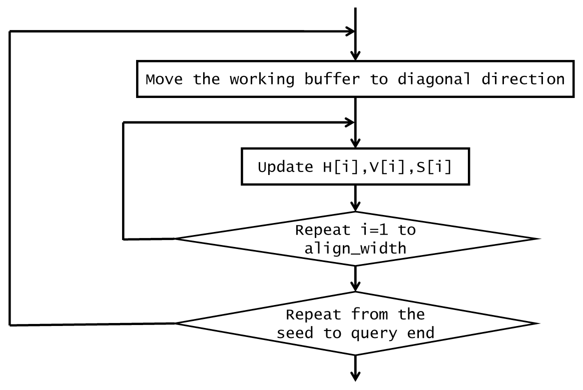
Move the working buffer to diagonal direction
Update H[i],V[i],S[i]
Repeat i=1 to align_width
Repeat from the seed to query end

Figure 10: CLAST alignment flow.

List 1: Cell update rules.

```
H[i] = max(H[i-1]+GAP_PENALTY, S[i-1]+GAP_OPEN_PENALTY+GAP_PENALTY)
V[i] = max(V[i+1]+GAP_PENALTY, S[i+1]+GAP_OPEN_PENALTY+GAP_PENALTY)
S[i] = max(H[i], V[i], S[i]+score(BaseRef, BaseQuery))
```

Figure 10 shows the alignment flow and List 1 shows the cell update rules. Here, `score(BaseRef, BaseQuery)` represents the alignment score between the corresponding reference and query base, which is defined in a score table. The variables needed in calculating `H[i]` are of current step, so there is no risk of breaking coherency. Those in calculating `V[i]` and `S[i]` are of previous step, however the array indices do not include `i` $-$ `1` and the coherency still remains.

2.2.4 Parallelization of each CLAST phase

(A) to (E) phases described above have been parallelized utilizing GPU threads. In (A) phase, k-length strings are sampled out of reference sequences at equal intervals and used to calculate hash values. They are conducted by each GPU thread in parallel. The created hash values are structured into a q-gram index table. Then in (B) phase, query hash values are calculated from query sequences, searched against the q-gram index table, and used in creating the seeds which include the reference and query position pairs. They are possible to be computed in parallel. In (C) phase, the seeds are sorted according to their positions, then each GPU thread compares each seed to the following one and removes it if it is judged as an isolate or duplicate one. In (D) phase, an alignment starting from each seed is conducted by each GPU thread in parallel. In (E) phase, each result is removed if its score is too low or its hit length is too short. This is also done by each GPU thread in parallel.

2.3 PEZY-SC2

In this section, we introduce PEZY-SC2, which is a second generation many-core MIMD processor of PEZY-SC series [5, 8],

Table 1: PEZY-SC2 specification

Frequency(GHz)	0.733
Number of cores	2,048
Number of threads	16,384
Rpeak(TFlops/DP)	2.8
Memory BW(GB/s)	76.8
Memory size(GB)	64

Table 1 shows the specification of PEZY-SC2. A PEZY-SC2 processor has 2,048 MIMD cores and 64GB device memory. Since each core has 8 threads, 16,384 threads are available in total. In the system we used, a PEZY-SC2 processor is connected to a Xeon-D 1572 processor through a PLX PCIe switch chip. Similar to the GPU programming, the PEZY-SC2 program is launched after preparing the input data in the device memory. The computation results are stored into the device memory, which is read by the CPU program when necessary. An OpenCL-like programming environment named PZCL is provided, on which users write both the CPU program and the PEZY-SC2 kernel program. The CPU program sets up the kernel program and buffers, write the initial value to the buffers allocated in the PEZY-SC2 device memory, and then launch the kernel program on the PEZY-SC2 as multiple threads. After detecting the end of all threads in the PEZY-SC2, the CPU program reads the results from PEZY-SC2 buffers to the CPU buffers.

3 PZLAST

3.1 Porting CLAST to a MIMD processor

PZLAST (PEZY implemented large-scale alignment search tool) is a transplant of CLAST to PEZY-SC2. In this porting, the functionalities of (A) to (E) phases shown in Figure 5 have been maintained. While CLAST is implemented on CUDA Thrust library, PZLAST is implemented on PZCL. The behavior of each thread is written freely in C language (partly C ++) which is suitable for MIMD architecture. The MIMD features of PEZY-SC2 cause some differences between the PZLAST and CLAST software implementations.

3.1.1 Reducing the load imbalance among the threads

As for (D) phase, the forward and the backward alignment are executed in the different computational stages in CLAST. This is because the GPU architecture is categorized to SIMD and every threads have to process the same instruction synchronously. On the other hand, in PEZY-SC2, the behavior of each thread can be freely described as long as there is no data dependency between the threads. Therefore, a forward and a backward alignment are concatenated into one, and the processing speed has been improved. Furthermore, the processing time of alignment frequently changes depending on the sequence length.

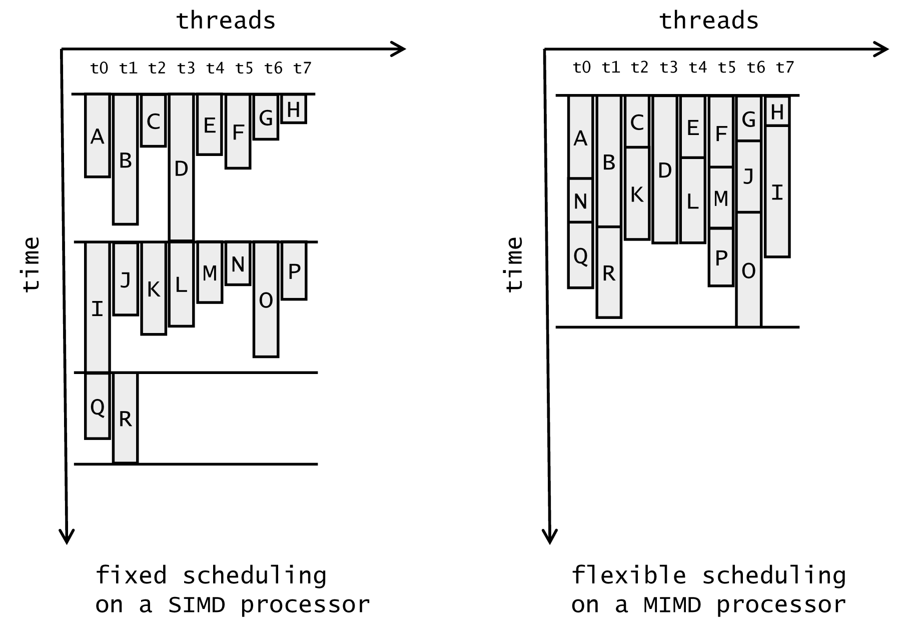
threads
t0 t1 t2 t3 t4 t5 t6 t7
time
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
fixed scheduling on a SIMD processor
flexible scheduling on a MIMD processor

Figure 11: Task execution in each thread.

Thus, rather than assigning the tasks to each thread in a fixed manner, assigning them in a flexible scheduling is suitable to reduce the load imbalance among the threads (Figure 11). Left of this figure shows the case that every threads need to synchronize on a SIMD processor. Whereas in the right of this figure, each thread can run independently on a MIMD processor and can be assigned any tasks freely.

List 2: Flexible scheduling with PZCL atomic operations.

```
// Initialize the atomic counter.
if (thread_id == 0) {
    pz_atomic_xchg(atomic_counter, 0L);
}

// All threads wait until the initialization of atomic counter has been done.
sync();

while (true) {
    // Get the job id dynamically.
    long i = pz_atomic_inc(atomic_counter);

    // Break the loop if the job id is bigger than or equal to total.
    if (total <= i) break;

    // Execute i-th job
    ...
}
```

This is easily realized with atomic operations in PZCL as List 2. Here, `pz_atomic_xchg(p, val)` swaps the old value stored at location `p` with the new value given by `val`. `pz_atomic_inc(p)` reads the value (referred to as `old`) stored at location pointed by `p`, stores (`old` + `1`) at location pointed by `p`, and then returns `old`.

Initially, atomic variable `atomic_counter` is set to 0. Each thread gets the job id from `atomic_counter`, increase it and execute the assigned job. This way, a thread can get a new job as soon as it finishes the previous job.

Also in (B) phase, a table search is performed using binary searching, and the processing time of each thread tends to vary depending on the depth of searching and the latency of memory access. Thus, the flexible scheduling utilizing the MIMD features improve the performance in the seed creation.

3.1.2 Two phase alignment for long sequences

The alignment algorithm described in section 2.2 fails in some cases.

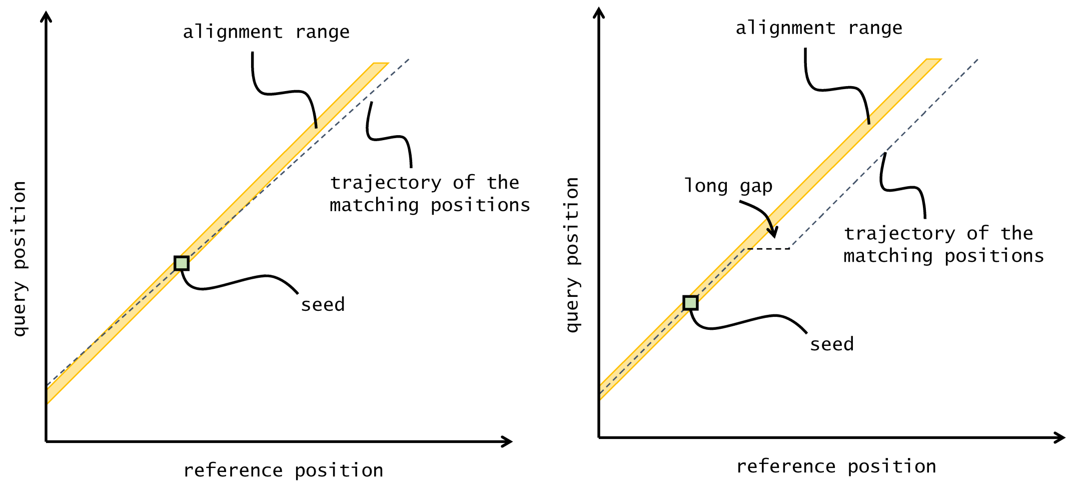

Figure 12: Problems in long sequences.

Left of Figure 12 shows the case that a reference sequence includes many short gaps. Right of this figure shows the case of a long gap. In these cases, the alignment range restricted by `align_width` is not sufficient to cover the whole trajectory of the matching positions and cannot make a good result. This is because the working buffer can move only in a diagonal direction.

To adapt these cases, we added the alignment algorithm the ability to move in a vertical or horizontal direction.

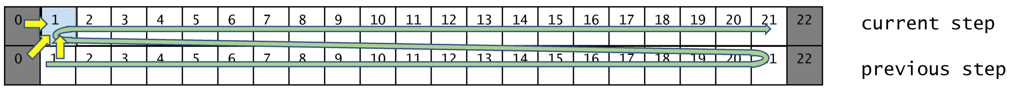

Figure 13: Alignment at vertial movement.

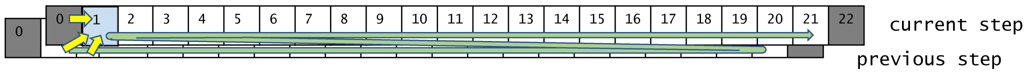

Figure 14: Alignment at horizontal movement.

Figure 13 and 14 illustrate the vertical and horizontal movement of the working buffer. Changing the direction of movement in this way breaks the coherency of updating each cell. To deal with this problem, we added a new variable `D` which represent diagonal score, to each cell in Figure 7, and divided the update stage in Figure 10 into two phases.

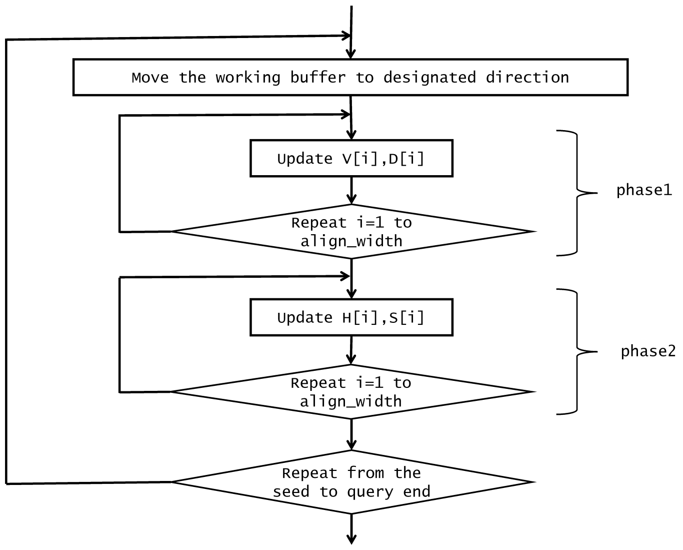
Move the working buffer to designated direction
Update V[i],D[i]
Repeat i=1 to align_width
phase1
Update H[i],S[i]
Repeat i=1 to align_width
phase2
Repeat from the seed to query end

Figure 15: Two phase alignment flow.

List 3: Cell update rules in phase1.

```
if direction == VERTICAL
    V[i] = max(V[i]+GAP_PENALTY, S[i]+GAP_OPEN_PENALTY+GAP_PENALTY)
    D[i] = D[i-1]+score(BaseRef, BaseQuery)
else if direction == HORIZONTAL
    # do nothing
else if direction == DIAGONAL
    V[i] = max(V[i+1]+GAP_PENALTY, S[i+1]+GAP_OPEN_PENALTY+GAP_PENALTY)
    D[i] = D[i]+score(BaseRef, BaseQuery)
```

List 4: Cell update rules in phase2.

```
if direction == VERTICAL
    H[i] = max(H[i-1]+GAP_PENALTY, S[i-1]+GAP_OPEN_PENALTY+GAP_PENALTY)
else if direction == HORIZONTAL
    H[i] = max(H[i]+GAP_PENALTY, S[i]+GAP_OPEN_PENALTY+GAP_PENALTY)
else if direction == DIAGONAL
    H[i] = max(H[i-1]+GAP_PENALTY, S[i-1]+GAP_OPEN_PENALTY+GAP_PENALTY)
S[i] = max(H[i], V[i], D[i])
```

Figure 15 is the revised alignment flow, and List 3 and 4 are the revised update rules. In both phases, the update rules differs depending on the direction of working buffer movement.

To decide the direction of movement, we defined two thresholds in the working buffer.

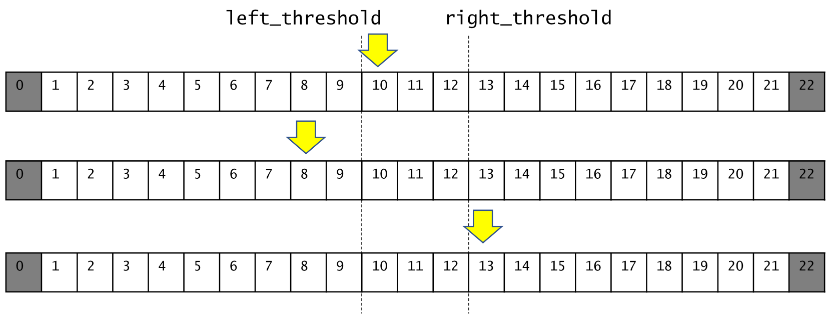

Figure 16: How to decide the direction of movement.

Figure 16 shows how to decide the direction of movement. If the cell of maximum score is to the left than the `left_threshold`, vertical movement is selected. If it is to the right than the `right_threshold`, horizontal movement is selected, otherwise diagonal is selected.

This decision algorithm works well in the case of left of Figure 12, in which no long gaps exist and the change of the trajectory of the matching positions is gradual. With a long gap as shown in the right of this figure, this decision algorithm is not sufficient. In such case, the scores in almost all cells become low simultaneously and the cell of maximum score cannot indicate the suitable moving direction. Since the seeds created in (B) phase in Figure 5 have fruitful information about the matching positions, we used them as the other clues to decide the movement direction to overcome such long gaps.

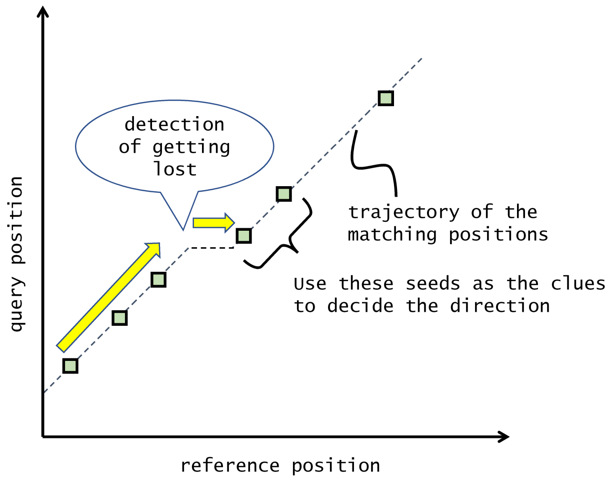
detection of getting lost
trajectory of the matching positions
Use these seeds as the clues to decide the direction
query position
reference position

Figure 17: Alignment with long gap.

As illustrated in Figure 17, when a long gap causes the alignment to get lost, the nearby seeds can be used to decide the next moving direction. How to judge the state of getting lost and how to get an optimal clue among the nearby seeds are not simple. They are remained as important issues. Currently the tendency of the highest score in the working buffer is used to judge the state. When the highest score keeps low for a certain period, we consider the alignment as getting lost.

List 5: Judging a lost state.

```
s_new = max_score_prev < max_score ? 1.0 : 0.0
s = lerp(s, s_new, s_coef)
s = clamp(s, SMIN, 1)
max_score_prev = max_score
got_lost = s < S_THRESHOLD
```

To find a lost state, a variable `s` is introduced as shown in List 5. `s` increases if the maximum score in the working buffer is higher than previous one, otherwise it decreases.

Here, the `lerp(x, y, t)` function returns the value linearly interpolated between `x` and `y` by the interpolant `t`, and the `clamp(x, min, max)` function clamps `x` between `min` and `max`. `max_score` is the maximum score in the working buffer of current step, and `max_score_prev` is of previous step. `s_coef` is a coefficient in the range 0 to 1 which relates to the change speed of `s`.

List 6: Calculating `s_coef`.

```
s_coef = 2 * S_NORMALIZED_SPEED / (align_width - 1)
s_coef = clamp(s_coef, 0, 1)
```

`s_coef` is calculated as List 6 so that `s` changes quickly if `align_width` is narrow and slowly if it is wide.

We set `SMIN` as 0.75, `S_THRESHOLD` as 0.8 and `S_NORMALIZED_SPEED` as 0.8.

Since a wrong clue will mislead the alignment, we have to select an adequate seed more carefully than the filtering in (C) phase. In our interim implementation, a seed is selected as a clue only when it is not too far from current position and does not seem to be very strange.

List 7: Selecting an anchor.

```
seed_next1 = seed_cur.next
if seed_next1 != null
```

```
    seed_next2 = seed_next1.next
    if seed_next2 != null
        diag_dist_cur = seed_cur.pos_ref - seed_cur.pos_query
        diag_dist_next1 = seed_next1.pos_ref - seed_next1.pos_query
        diag_dist_next2 = seed_next2.pos_ref - seed_next2.pos_query
        if abs(diag_dist_cur - diag_dist_next1) <= abs(diag_dist_cur -
            diag_dist_next2)
            anchor = seed_next1
        else
            anchor = seed_next2
    else
        anchor = seed_next1
else
    anchor = null
```

List 7 is a pseudo code selecting a seed as `anchor`. The seeds are supposed to be sorted along with their reference position, and `seed_cur` is the seed placed just prior to the center position of the working buffer. `seed_next1` is the seed placed just after `seed_cur` and `seed_next2` is just after `seed_next1`.

List 8: Deciding the direction of movement.

```
if got_lost == false
    if max_score_id < left_threshold
        direction = VERTICAL
    else if right_threshold < max_score_id
        direction = HORIZONTAL
    else
        direction = DIAGONAL
else
    # when got lost, use the anchor as a clue for the movement
    if anchor != null
        dpos_ref = max(anchor.pos_ref - working_buffer_center.pos_ref, 0)
        dpos_query = max(anchor.pos_query - working_buffer_center.pos_query, 0)
        if dpos_ref < dpos_query
            direction = VERTICAL
        else if dpos_query < dpos_ref
            direction = HORIZONTAL
        else
            direction = DIAGONAL
    else
        direction = DIAGONAL
```

Then, the moving direction is decided as List 8. Here, `max_score_id` is the id of the cell of maximum score in the working buffer, and `working_buffer_center` is the center cell of the working buffer.

These algorithms seem to work well in many cases, however it needs further studies and improvements. The conditional branches used in these algorithms are frequently inefficient on a SIMD processor, on the other hand, they are efficiently conducted on a MIMD processor.

3.1.3 Alignment width depending on the seeds variation

Although we improved the alignment algorithms for long sequences as described in section 3.1.2, yet there are cases that need to widen the alignment width. There is a trade-off between the alignment width and the processing time, therefore, it is effective to change the alignment width for each sequence pair. To decide the alignment width, we used the variation of the seed positions.

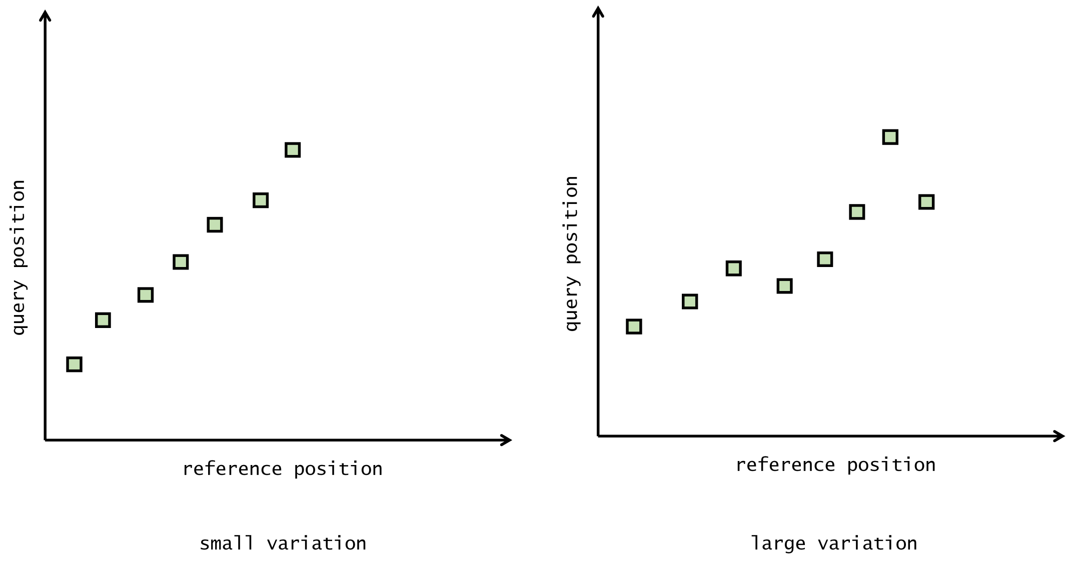

Figure 18: Variation of the seeds.

Left of Figure 18 is an example of the layout of seeds with small variation. In such case, a narrow width is sufficient for the alignment. While in the right of this figure, the position of the seeds varies greatly, and we select a wide width for such case.

List 9: Calculating alignment width.

```
align_width = 0
for seed in all_seeds_in_an_alignment
    seed_next = seed.next
    if seed_next != null
        diag_dist = seed.pos_ref - seed.pos_query
        diag_dist_next = seed_next.pos_ref - seed_next.pos_query
        align_width = max(align_width, 2 * abs(diag_dist - diag_dist_next) + 1)
align_width = clamp(align_width, MIN_ALIGN_WIDTH, MAX_ALIGN_WIDTH)
```

As shown in List 9, the alignment width is calculated comparing the positions of a seed and its following seed. Here, all seeds are supposed to be sorted along with their reference position, and `seed_next` is a seed just behind `seed`.

Changing the alignment width for each sequence pair causes a significant difference in the processing time between the threads. Flexible scheduling with MIMD features described in section 3.1.1 is also effective for reducing the imbalance between the threads.

3.2 Problems in Many-core processor

In section 2.2, we introduced the whole computation flow in CLAST. Here are issues in related to the efficient use of extremely many threads in parallel.

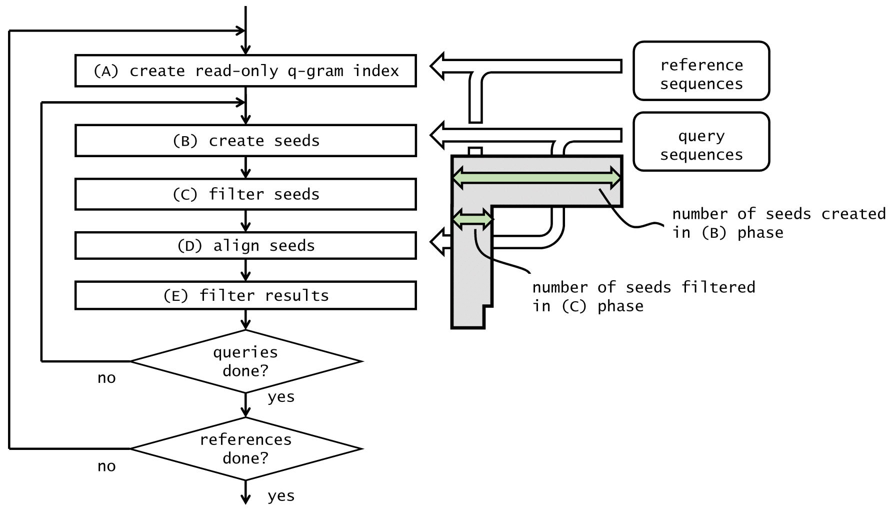

Figure 19: Typical change in the number of seeds in each phase of CLAST.

The gray area in Figure 19 sketches the typical change in the number of seeds in each phase of CLAST. This figure highlights two problems in applying an ultra-parallel computation to a general application. One difficulty is in deciding the optimal amount of inputs for each iteration. In CLAST, the maximum number of reference and query sequences per iteration are fixed at startup, and never changes in runtime. The number of seeds created in (B) phase varies dependent on the characteristics of inputs, thus deciding the appropriate amount of inputs before the run is extremely difficult and usually even impossible. In the worst case, enormous number of seeds causes overflow in the device memory. In ultra-parallel computations, reallocating the buffer in runtime is costly since the device memory is shared by many threads. Therefore in CLAST, it is necessary to allocate a sufficiently large area for storing seeds and decide the maximum amount of inputs per iteration prior to the run so that overflow never occurs. In addition, (C) phase needs to sort the seeds by their position prior to filtering them. Even if the overflow does not occur, the efficiency of sorting extremely declines if the number of sorting seeds explodes. The number of seeds created in runtime is unpredictable and usually it is difficult or impossible to decide the optimal conditions. Another difficulty is in keeping up the parallelism efficiently throughout the whole computation. Usually the seeds created in (B) phase are dramatically reduced in (C) phase. If the number of seeds after filtering becomes few, sufficient seeds cannot be supplied to the subsequent phases and the parallelism of many threads cannot be kept well.

3.3 Adaptive Flow Control and Seeds Accumulation

In section 3.2, we pointed out two problems related to the implementation on an ultra-parallel computation. They are summarized as follows.

(1) Deciding the optimal amount of inputs prior to the run is extremely difficult and usually even impossible.

(2) Keeping up the parallelism efficiently throughout the whole computation is not always possible.

As for (1), for a fail-safe purpose, PZLAST removes the seeds that have overflowed. Although this is useful to prevent the accidental illegal memory accesses, it is just a minimal prescription and we have introduced Adaptive Flow Control as an additional strategy. Furthermore, for (2), we have introduced Seeds Accumulation strategy to keep up the parallelism throughout the whole computation.

3.3.1 Adaptive Flow Control

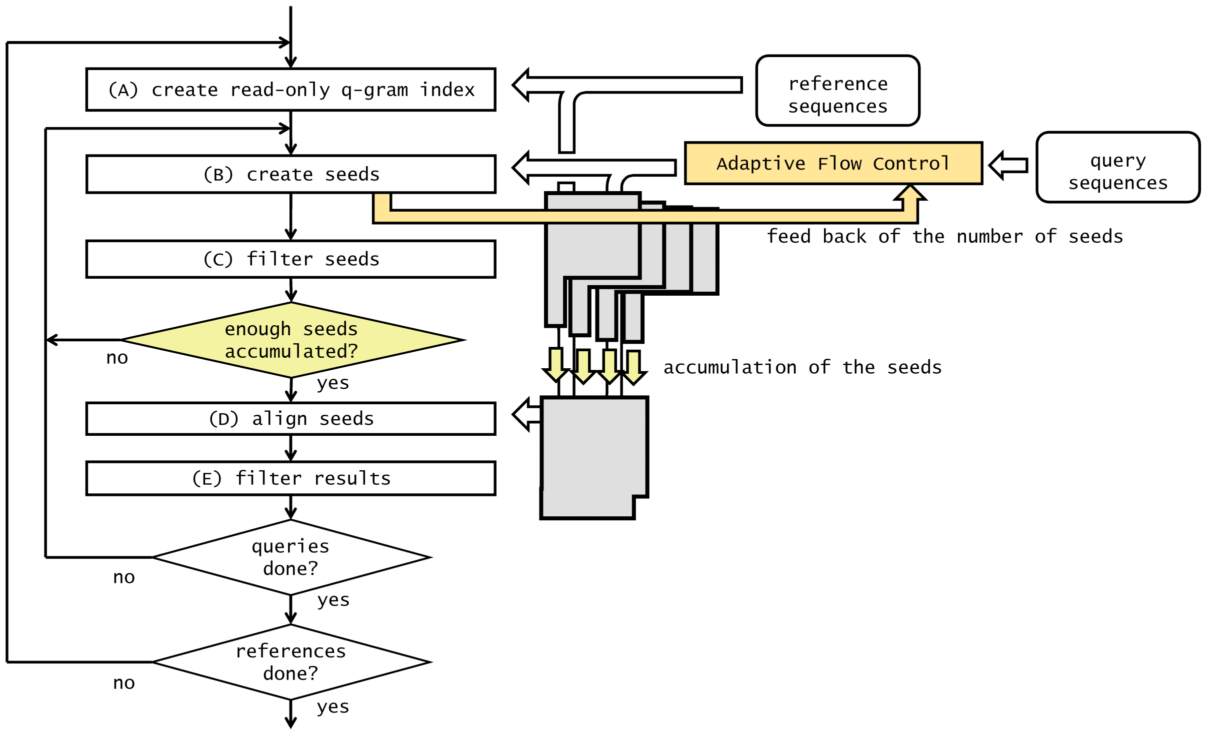

Figure 20: PZLAST whole computation flow.

Figure 20 shows the whole computation flow of PZLAST. This figure includes two types of gray areas, which schematically represent the number of seeds related to (B)-(C) phase and (D)-(E) phase respectively. Adaptive Flow Control adjusts the number of seeds after seed creation to an appropriate value. Further, by Seeds Accumulation, the imbalance in the number of seeds is alleviated compared to the CLAST case illustrated in Figure 19.

The key idea of Adaptive Flow Control is that by constructing a feedback system and controlling the amount of inputs in runtime, it may be possible to keep the number of seeds around the appropriate value. Since the number of seeds created in (B) phase is directly related not to the total number of sequences but to the total number of letters, it is reasonable to use the maximum number of letters as a control parameter. Within the scope of this paper, for ease of implementation, we selected as a control parameter the maximum number of letters in only query side and did not use the one in reference side.

First we decided the optimal number of seeds as 0x80000(=524,288). This value depends on the algorithm of sorting, the number of available threads, total device memory size and so on, therefore it could not be derived theoretically but was decided experimentally specialized for our PEZY-SC2 implementation. Adaptive Flow Control always watches the number of seeds created in (B) phase, predicts the appropriate maximum number of query letters, and then restricts the amount of query input to the predicted value. Assuming that the distribution of the reference and query sequences are uniform and the number of reference letters at each step does not vary significantly, we can treat the number of seeds almost proportional to the number of query letters, thus a simple calculation strategy can be used for this purpose.

List 10: Predicting maximum number of query letters.

```
function PREDICT_MAXIMUM_QUERY_LETTERS(max_letters, num_seeds)
    rate = OPTIMAL_NUMBER_OF_SEEDS / max(num_seeds, 1)
    n1 = rate * max_letters
    n1 = clamp(n1, MIN_QUERY_LETTERS, MAX_QUERY_LETTERS)
    return lerp(max_letters, n1, LERP_COEF)
```

List 10 shows the pseudo function code to predict the maximum number of query letters. Here, `max_letters` represents the maximum number of query letters (i.e. control parameter) and `num_seeds` represents the number of seeds created in (B) phase (i.e. control target) in the previous tion. The function receives the value of control parameter and target, and predicts the control parameter for the next iteration. The prediction is done according to the ratio of the optimal number of seeds to the current one. The other parts of this code are to avoid zero division, to clamp extremely strange values, and to suppress sudden changes in the number of seeds.

3.3.2 Seeds Accumulation

Another strategy named Seeds Accumulation is designed to solve (2). In this strategy, the resultant seeds of (C) phase are accumulated until the total number of seeds achieves a sufficient amount. As shown in Figure 20 , a checkpoint is placed just after (C) phase. At the checkpoint, the seeds just after (C) phase are accumulated. If the number of accumulated seeds becomes larger than a predefined threshold, (D) and (E) phases are processed, otherwise (B) and (C) phases are repeated again without going down to the subsequent phases. Here, the threshold is set to 0x80000, which is the same as the optimal number of seeds used in Adaptive Flow Control. By accumulating the intermediate seeds in this way, it becomes possible to supply a sufficient number of seeds to the following phases and perform a large number of tasks in parallel throughout the computation.

4 Results

4.1 Performance evaluation in short queries

Table 2: Comparison of BLAST+, CLAST and PZLAST.

	BLAST+	CLAST	PZLAST
Processor	Xeon Gold 6134	NVIDIA Tesla V100	PEZY-SC2
Frequency(GHz)	3.2	1.53	0.733
Number of cores	8	5,120	2,048
Rpeak(TFlops/DP)	0.8	7.8	2.8
Memory BW(GB/s)	127.8	900	76.8
Memory size(GB)	96	16	64
Total processing time(sec)	**3,213**	**517**	**314**

The total processing times of BLAST+, CLAST and PZLAST for short queries are compared in Table 2. Here, PZLAST is conducted with Adaptive Flow Control and Seeds Accumulation previously described. To evaluate the performance with the same functionalities, the alignment algorithms for long sequences described in section 3.1.2 and 3.1.3 are not adopted in this evaluation. Since the other part of the alignment search algorithms for these three tools are almost the same as BLAST and we used the same parameters for these three tools, the accuracy and sensitivity also are about the same.

Throughout this section except for Figure 21, we fixed the k-length (i.e. exact string matching length) as 15. As an experimental dataset, we used 1Gbp (base pair) nucleotide 500 bacterial genome (2Mbp average length) for the reference, and 750Mbp 5M reads (150bp) of Illumina HiSeq metagenome for the query [7]. In spite that CLAST runs on V100 whose theoretical performance and memory bandwidth are higher than PEZY-SC2, PZLAST achieves a higher performance than CLAST. While there may be various reasons for this result including the differences in optimization or the differences between CUDA Thrust library and PZCL, the use of MIMD features, Adaptive Flow Control and Seeds Accumulation have largely contributed to the improvement of performance.

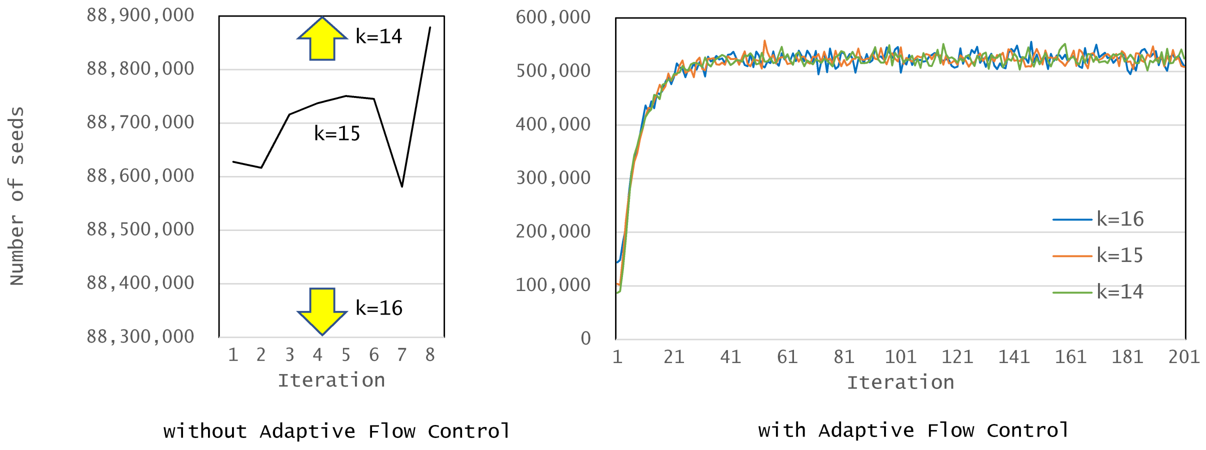

Figure 21: The number of seeds created in (B) phase with and without Adaptive Flow Control.

Table 3: Processing time (sec) of each phase with and without Adaptive Flow Control.

	without Adaptive Flow Control	with Adaptive Flow Control
(A)	69.68	69.69
(B)	184.44	167.91
(C)	**154.50**	**73.53**
(D)	1.27	1.28
(E)	2.16	1.16

Figure 21 and Table 3 show the effect of Adaptive Flow Control in more detail.

Figure 21 illustrates the number of seeds created in (B) phase for every iterations. Left of this figure shows the number of seeds without Adaptive Flow Control at k-length=15, which drastically fluctuates around 88,700,000. When we change the k-length to 14, the number of seeds grows larger than 290,000,000, which causes a high risk of memory overflow and a slowdown of seed sorting process. Also when we set the k-length to 16, the number of seed decreases smaller than 30,000,000. These indicate that the number of seeds is very sensitive to both the data and parameters. On the other hand, right of this figure shows the number of seeds with Adaptive Flow Control at the k-length of 14, 15 and 16. They are controlled well to almost the same range regardless the k-length, and the possibility of the seeds overflow is extremely reduced. Moreover, the number of seeds is remained to the preferable range from the viewpoint of the processing time of the seeds sorting. As for the initial value of the maximum number of query letters, we assigned a small value so that no overflow of seeds will occur. Therefore, the number of seeds created in (B) phase remains low in the early iterations and then gradually increases to the preferable value.

Table 3 compares the processing time of each phase at k-length=15, with and without Adaptive Flow Control. The shortening of the processing time in (C) phase is remarkable.

Table 4: Average number of seeds with and without Seeds Accumulation.

	without Seeds Accumulation	with Seeds Accumulation
(B) (C)	521,442.9	521,442.9
(D) (E)	**7,811.3**	**506,244.2**

Table 5: Processing time (sec) of each phase with and without Seeds Accumulation.

	without Seeds Accumulation	with Seeds Accumulation
(A)	69.64	69.69
(B)	167.42	167.91
(C)	73.40	73.53
(D)	**2.56**	**1.28**
(E)	**6.52**	**1.16**

Table 4 and Table 5 show the effect of Seeds Accumulation.

In Table 4, the average values derived from dividing the total number of seeds by the number of repetitions are compared. The number of repetitions of (B) and (C) phase does not change with and without Seeds Accumulation. With Seeds Accumulation, the number of repetitions of (D) and (E) phase decreases because the execution of these phases is delayed until sufficient seeds have been accumulated. The average value in (D) and (E) phases becomes larger with Seeds Accumulation, which allows PZLAST to take full advantage of thread parallelism.

Table 5 is a comparison of the processing time of each phase, with and without Seeds Accumulation. The effects in (D) and (E) phases are conspicuous although the percentage to the total time is not large. This indicates the parallel efficiency of the multi-threads has improved with Seeds Accumulation. While we have tried short query sequences (i.e. 150bp) in this evaluation, in the case of long length queries or more sensitive alignment, the effect of this improvement becomes more crucial.

4.2 Alignment result of long sequences

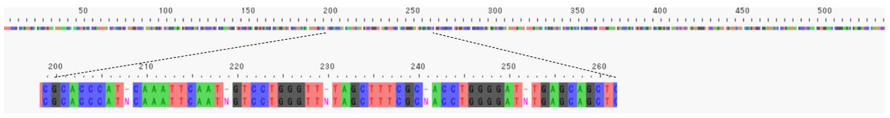

Figure 22: Alignment result in the case of many short gaps.

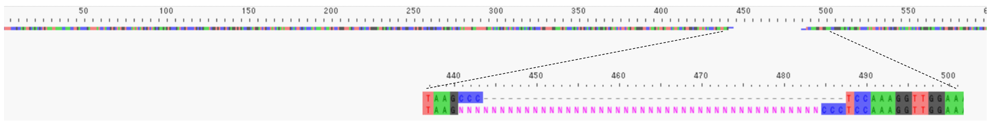

Figure 23: Alignment result in the case of a long gap.

Figure 22 is an alignment result in the case where the reference sequence include many gaps, and Figure 23 is one where the reference include a long gap. In these evaluations, we set `align_width` as a fixed value 17, which is too narrow to treat such sequences with the alignment algorithm described in section 2.2. `left_threshold` is set as 9 and `right_threshold` is set as 11. Although the delay in the judgement of getting lost shown in Figure 17 causes a few misalignment at the start and end point of a long gap, totally they are well aligned with the two phase alignment discussed in section 3.1.2.

To confirm the effectivity of the new method described in section 3.1.2 and 3.1.3, we have aligned 20,000 COVID-19 related nucleotide sequences downloaded from NCBI site (https://www.ncbi.nlm.nih.gov) against a reference sequence (NC_045512.2 : Wuhan-Hu-1). The length of these sequences are around 30,000 and they include many insertions, deletions or mutations. Here, we set k-length as 20. `align_width` was fixed as 33 for the original method. The new method calculates `align_width` as described in section 3.1.3. `left_threshold` and `right_threshold` are calculated as $(\texttt{align\_width}+1)/2-1$ and $(\texttt{align\_width}+1)/2+1$ respectively.

Table 6: Alignment score distribution in the original and new method.

alignment score	original method	new method
[-40,000, -30,000)	0	0
[-30,000, -20,000)	35	0
[-20,000, -10,000)	0	0
[-10,000, 0)	0	0
[0, 10,000)	0	0
[10,000,20,000)	0	0
[20,000,30,000)	3	0
[30,000,40,000)	16	0
[40,000,50,000)	6	1
[50,000,60,000)	19940	19999
[60,000,70,000)	0	0

Table 6 shows the distribution of resultant alignment scores. Although both the original and new method could align all of 20,000 sequences against NC_045512.2, the new method takes higher scores, which shows this new method is effective in the alignment of long sequences.

5 Conclusion and discussion

We have developed an ultra-fast sequence similarity search tool which utilizes the MIMD features efficiently. The merit of MIMD features in reducing the imbalance among the threads was shown in this paper. Also we have developed the alignment algorithms for long sequences fully utilizing the MIMD features. Although these algorithms work well in many cases, further studies are remained in deciding the direction of movement.

Furthermore, we applied two strategies against the problems found in implementing on an ultra-parallel computation. These strategies are not limited to a sequence similarity search tool, but effective for creating the other general applications utilizing an ultra-parallel computation. In using a huge number of threads in the applications, how to control their parallelism is an essential issue. The feedback strategy and the accumulation strategy presented in this paper have the ability to solve this issue in many cases. While the strategies described in this paper are versatile enough, their software designs are not well modularized and not always optimal for adapting to any other applications. Further studies are needed to modularize and make them applicable to any other general applications.

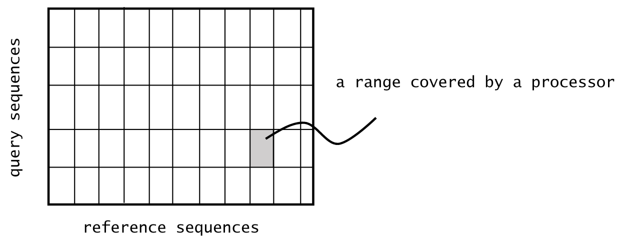

Figure 24: Parallelization to multiple processors by splitting reference and query sequences.

In this paper, we introduced PZLAST as a sequence similarity search tool on a single PEZY-SC2 processor. The approaches proposed in this paper parallelizes single alignment of a reference and query sequence pair and accelerate it. On the other hand, since an alignment of a reference and query pair is independent of that of the other pair, a sequence similarity searching between multiple sequences can be easily parallelized with splitting them by the sequence id (Figure 24). In [2], we have already proposed a web-based metagenomic PZLAST service using multiple PEZY-SC2s in parallel. The reference database is larger than 2 terabytes, which has been too large for the traditional systems to treat it. Although we have not yet apply the two types of strategies and the alignment algorithms for long sequences to the system, the MIMD features of PEZY-SC2 processors have been fully utilized there. Currently we are building the other version of PZLAST systems on multiple PEZY-SC2s utilizing our novel algorithms described in this paper.

Acknowledgement

We thank Masahiro Yano for the CLAST algorithm discussion. We thank Yoshinori Kimura and Toshimitsu Murakami for the help with developments of the PZLAST.

References

[1] Camacho C., Coulouris G., and Avagyan V. et al. BLAST+: architecture and applications. *BMC Bioinformatics*, 10(421), 2009. https://doi.org/10.1186/1471-2105-10-421.

[2] Mori H., Ishikawa H., Higashi K., Kato Y., Ebisuzaki T., and Kurokawa K. PZLAST: an ultra-fast amino acid sequence similarity search server against public metagenomes. *Bioinformatics*, 37(21):3944–3946, 2021. https://doi.org/10.1093/bioinformatics/btab492.

[3] Illumina Inc. Introduction to NGS. https://www.illumina.com/science/technology/next-generationsequencing.html (Retrieved on Jun 30, 2021).

[4] Yano M., Mori H., and Akiyama Y. et al. CLAST: CUDA implemented large-scale alignment search tool. *BMC Bioinformatics*, 15(406), 2014. https://doi.org/10.1186/s12859-014-0406-y.

[5] Torii S. and Ishikawa H. ZettaScaler: Liquid immersion cooling Many-core based Supercomputer. https://www.pezy.co.jp/wpcontent/uploads/2019/02/Keynote_candar2017.pdf (Retrieved on Jun 30, 2021).

[6] Altschul SF., Gish W., Miller W., Myers EW., and Lipman DJ. Basic local alignment search tool. *J Mol Biol*, 215:403–410, 1990. https://doi.org/10.1016/S0022-2836(05)80360-2.

[7] E. Singer, Andreopoulos B., and Bowers R. et al. Next generation sequencing data of a defined microbial mock community. *Sci Data*, 3(160081), 2016. https://doi.org/10.1038/sdata.2016.81.

[8] Hishinuma T. and Nakata M. pzqd: PEZY-SC2 Acceleration of Double-Double Precision Arithmetic Library for High-Precision BLAS. *Mechanisms and Machine Science*, 75, 2019. https://doi.org/10.1007/978-3-030-27053-7_61.