Lan *et al. BMC Bioinformatics* 2016, **17**(Suppl 9):267
DOI 10.1186/s12859-016-1128-0

BMC Bioinformatics

RESEARCH Open Access

# Parallel algorithms for large-scale biological sequence alignment on Xeon-Phi based clusters

Haidong Lan[1†], Yuandong Chan[1†], Kai Xu[1], Bertil Schmidt[2], Shaoliang Peng[3] and Weiguo Liu[1*]

*From* IEEE International Conference on Bioinformatics and Biomedicine 2015
Washington, DC, USA. 9-12 November 2015

## Abstract

**Background:** Computing alignments between two or more sequences are common operations frequently performed in computational molecular biology. The continuing growth of biological sequence databases establishes the need for their efficient parallel implementation on modern accelerators.

**Results:** This paper presents new approaches to high performance biological sequence database scanning with the Smith-Waterman algorithm and the first stage of progressive multiple sequence alignment based on the ClustalW heuristic on a Xeon Phi-based compute cluster. Our approach uses a three-level parallelization scheme to take full advantage of the compute power available on this type of architecture; i.e. cluster-level data parallelism, thread-level coarse-grained parallelism, and vector-level fine-grained parallelism. Furthermore, we re-organize the sequence datasets and use Xeon Phi shuffle operations to improve I/O efficiency.

**Conclusions:** Evaluations show that our method achieves a peak overall performance up to 220 GCUPS for scanning real protein sequence databanks on a single node consisting of two Intel E5-2620 CPUs and two Intel Xeon Phi 7110P cards. It also exhibits good scalability in terms of sequence length and size, and number of compute nodes for both database scanning and multiple sequence alignment. Furthermore, the achieved performance is highly competitive in comparison to optimized Xeon Phi and GPU implementations. Our implementation is available at https://github.com/turbo0628/LSDBS-mpi.

**Keywords:** Smith-Waterman, Dynamic programming, Pairwise sequence alignment, Multiple sequence alignment, Xeon Phi clusters

## Background

Calculating similarity scores between a given query protein sequence and all sequences of a database and computing multiple sequence alignments are two common tasks in bioinformatics. Both tasks include iterative calculations of pairwise local alignments as a basic building block. This can lead to high runtimes for large-scale input data sets. Since biological sequence databases are continuously growing, finding fast solutions is of high importance. An approach to reduce associated runtimes is the implementation of basic alignment algorithms on parallel computer architectures [1–3]. More recently, the usage of modern massively parallel accelerator architectures such as CUDA-enabled GPUs has gained momentum [4]. In this paper we are investigating how a Xeon Phi-based compute cluster can be used as a computational platform to accelerate alignment algorithms based on dynamic programming for two applications:

(i) databases scanning of protein sequence databases with the Smith-Waterman algorithm, and

*Correspondence: weiguo.liu@sdu.edu.cn
†Equal contributors
[1]School of Computer Science and Technology, Shandong University, Shunhua Road 1500, Jinan, Shandong, China
Full list of author information is available at the end of the article

(ii) distance matrix computation for multiple sequence alignment (i.e. the first stage of the popular ClustalW heuristic).

Three levels of parallelization are required in order to exploit the compute power available in a cluster of Xeon Phis. Parallelization within a Xeon Phi is usually based on the "scale-and-vectorize" approach: scaling across the up to 61 cores requires the usage of several hundred threads while exploiting the 512-bit wide vector units requires SIMD vectorization within each core. Recent examples of efficient parallelization on Xeon Phis include scientific computing [5], bioinformatics [6–10], and database operations [11]. Furthermore, parallelization between Xeon Phis adds another level of message passing based parallelism. This level needs to consider data partitioning, load balancing, and task scheduling. The accelerator-based approach is motivated by the fact that the performance of many-core architectures is growing. For example, the 2nd generation Xeon Phi processor named "Knight's Landing" has already been announced.

The rest of this paper is organized as follows. The "Related work" Section provides important background information about the Xeon Phi programming model, pairwise and multiple sequence alignment, and hardware accelerated alignment algorithms. Our single-node parallel algorithms are presented in the "Algorithms on a single node" Section. The "Cluster level data parallelization" Section describes our cluster-level parallelization. Section "Results and discussion" evaluates performance. Some conclusions are drawn in Section "Conclusion".

## Related work

### Programming models on Xeon Phi coprocessor

Xeon Phi is a coprocessor connected via the PCI express (PCIe) bus to a host CPU. From a hardware perspective, it contains up to 61 86 compatible cores. Each core features a 512-bit vector processing unit (VPU) based on a new instruction set. The cache hierarchy contains a L1 data cache of size 32 KB and a 512 KB per core L2 cache. The cores are connected via a bidirectional ring bus which enables L2 cache coherence based on a directory based protocol. Each core can execute up to four threads at the same time.

Assuming a Xeon Phi with 61 usable cores running at 1.238 GHz, we can determine the peak performance for 32-bit integer (integer arithmetic is commonly used for sequence alignment calculations) operations as follows: 16 (#SIMD lanes) $\times$ 1 integer operation $\times$ 1.238 GHz $\times$ 61 (#cores) = 1.208 Tera integer operations per second.

From a software perspective, three programming models can be used in order to harness the compute power of the Xeon Phi: (i) native model, (ii) offload model, and (iii) symmetric model. In this paper, we have chosen the offload model. In this model, code sections and data can be offloaded from the host CPU to the Xeon Phi. Using OpenMP pragmas, offload regions can be specified. When encountering such a region during program execution, the necessary data transfers between host and Xeon Phi are performed and the code inside the (parallelized) region is executed on the Xeon Phi.

### Pairwise sequence alignment and database search

The database search application considered in this paper scans a protein sequence database using a single protein sequence as a query (similar to BLASTP). Different to the BLASTP heuristic, we calculate the score of an optimal local alignment between the query and each subject sequence using the Smith-Waterman algorithm with affine gap penalties (instead of a seed-and-extend approach). The subject sequences are ranked in terms of this score. Actual alignments are only computed for the top-ranked database sequences which only takes a negligible amount of time in comparison to the score-only search procedure. Note that the score-only Smith-Waterman computation can be performed in linear space and quadratic time with respect to the length of the alignment targets.

Consider two protein sequences $Q$ and $S$ and length $q$ and $s$, respectively. We want to compute the score of an optimal local alignment of $Q$ and $S$ with respect to a given scoring scheme consisting of a gap opening penalty $\alpha$, a gap extension penalty $\beta$ and an amino acid substitution matrix $sbt()$. The well-known Smith-Waterman algorithm solves this problem by computing a dynamic programming matrix iteratively based on the following recurrence relations:

$$\begin{aligned} H_A(i,j) &= max\{0, E(i,j), F(i,j), H_A(i-1,j-1) \\ &\quad + sbt(Q[i], S[j])\} \\ E(i,j) &= max\{H_A(i,j-1) - \alpha, E(i,j-1) - \beta\} \\ F(i,j) &= max\{H_A(i-1,j) - \alpha, F(i-1,j) - \beta\} \end{aligned} \quad (1)$$

The iterative computation of theses matrices is started with the initial values: $H_A(i,0) = H_A(0,j) = E(i,0) = F(0,j) = 0$ for all $0 <= i <= q, 0 <= j <= s$.

### Progressive multiple sequence alignment

The time complexity of computing an optimal multiple alignment of more than two sequences grows exponentially in terms of the number of input sequences. Thus, heuristic approaches with polynomial complexities must be used in practice for large inputs to approximate the (generally unknown) optimal multiple alignment.

The multiple (protein) sequence alignment application considered in this paper is the first stage of the popular

ClustalW heuristic [12]. ClustalW is based on the classical progressive alignment approach [13] featuring a 3-step pipeline (see Fig. 1):

(a) *Distance matrix*: For each input sequence pair, a distance values is computed based on the Smith-Waterman algorithm
(b) *Guide tree*: Using the distance matrix computed in the previous step is taken as an input to compute an evolutionary tree using the neighbor-joining method [14].
(c) *Progressive alignment*: Following the branching order of the tree a multiple sequence alignment is build progressively.

#### Hardware accelerated alignment algorithms

We briefly review some previous work on accelerating pairwise alignment (based on Smith Waterman) and progressive multiple sequence alignment (based on ClustalW) on a number of parallel computer architectures. A number of SIMD implementations have been designed in order to harness the vector units of common multi-core CPUs (e.g. [15–21]) or the the Cell/BE (e.g. [22, 23]). Recent years has seen increased interests in acceleration of sequence alignment on massively parallel GPUs. Initially, programming these graphics chips for bioinformatics application still required programming with shaders using languages such as OpenGL [24]. The release of CUDA in 2007 made the usage GPUs for general purpose computing more accessible and subsequently a number of CUDA-enabled Smith-Waterman implementation have been presented in recent years [4, 25–33]. A number of MPI-based solutions for progressive multiple sequence alignments are targeted towards PC clusters [34–37]. Another attractive architecture for sequence analysis are FPGAs [38–41] which are based on reconfigurable hardware. However, in comparison to the other mentioned architectures, FPGAs are often less accessible and generally more difficult to program.

The solution in this paper is based on a cluster of Xeon Phis. Compared to common CPUs, a Xeon Phi contains significantly more cores and often a wider vector unit. Different from CUDA-enabled GPUs, a Xeon Phi provides x86 compatibility, which often simplifies the implementation process. Nevertheless, achieving near-optimal performance is still a challenge which needs to be addressed by parallel algorithm design and efficient implementation. In this paper we demonstrate how this can be done for protein sequence database search and distance matrix computation for multiple sequence alignment.

Compared to our previously presented LSBDS [9], we introduce the following new contributions in this paper:

- We have designed new algorithms which can handle searching tasks for large-scale protein databases on Xeon Phi clusters.
- We have designed new algorithms for calculating large-scale multiple sequence alignments on Xeon Phi clusters.
- We have implemented our multiple sequence alignment algorithm using the offload model to make full use of the compute power of both the multi-core CPUs and the many-core Xeon Phi hardware.

## Methods

### Algorithms on a single node

#### *Protein sequence database search*

We have observed two facts: (1) protein sequence database search has inherent data parallelism; (2) each VPU on Xeon Phi can execute multiple integer operations in an SIMD parallel way efficiently. Based on these two facts, we have partitioned the database search process on a single node into two data parallel parts: device level and thread level. The device level data parallel part is encoded on the host CPU. It splits the subject database into multiple batches that can be distributed to CPU and Xeon Phi devices. The thread level data parallel part is used to process data batches locally. In order to support search tasks for large-scale databases, we have designed a dynamic data distribution framework to distribute these batches to both the host CPU device and the Xeon Phi devices. In order to solve the performance loss problem for searching long query sequences, we have also proposed a multi-pass algorithm where long query sequences are partitioned into

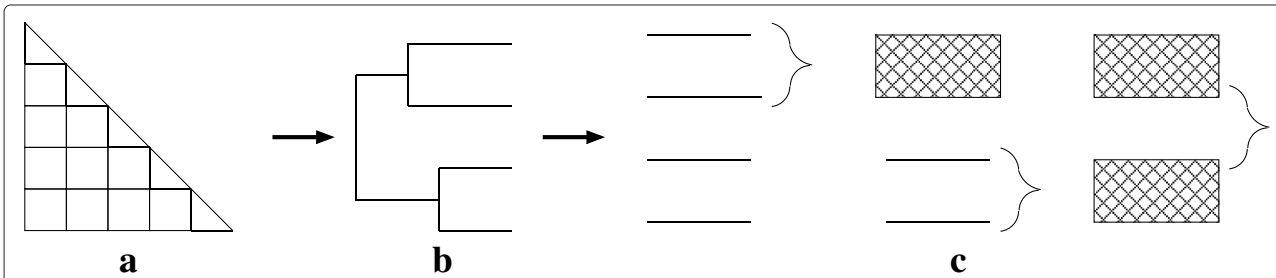


**Fig. 1** Illustration of the three stages of progressive multiple alignment (see text for details)

multiple short subsequences for consecutive searching passes. We have presented more implementation details of our algorithm in [9].

#### *MSA*

The distance matrix computation stage of ClustalW is typically a major runtime bottleneck. Thus, in our work we have only concentrated on designing a parallel algorithm for this stage. ClustalW bases the distance computation between two protein sequences on the following concept [24]:

**Definition 1.** Consider two sequences $S_i, S_j \in S = \{S_1, \ldots S_n\}$. The following equation defines their distance $d(S_i, S_j)$:

$$d(S_i, S_j) = 1 - \frac{nid(S_i, S_j)}{min\{l_i, l_j\}}$$

whereby $nid(S_i, S_j)$ is defined as the number of exact matches in an optimal local alignment between $S_i$ and $S_j$. $l_i$ ($l_j$) is the length of $S_i$ ($S_j$).

The value $nid(S_i, S_j)$ can be calculated in the Smith-Waterman traceback procedure by counting the number of exact character matches. Figure 2 illustrates this method. However, this direct method does not work well for long sequences and large-scale datasets because it needs to store the whole DP matrix. In order to solve this problem, we have adapted the method presented in [24] to do the *nid*-value computation on the Xeon Phi architecture. That is we have used the following definition and theorem to calculate the *nid*-value without doing the actual traceback.

**Definition 2.** Consider two protein sequences $S_1$ and $S_2$, affine gap penalties $\alpha$, $\beta$, and substitution matrix *sbt*. The matrix $N_A(i,j)$ $(1 \le i \le l_1, 1 \le j \le l_2)$ is defined in terms of the following recurrence relations:

| | \ | A | T | C | T | C | G | T | A | T | G | A | T |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| \ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 2 | 1 | 0 | 3 | 2 | 1 |
| T | 0 | 0 | 3 | 2 | 3 | 2 | 2 | 6 | 5 | 4 | 3 | 2 | 5 |
| C | 0 | 0 | 2 | 6 | 5 | 6 | 5 | 5 | 5 | 4 | 3 | 2 | 4 |
| T | 0 | 0 | 3 | 5 | 9 | 8 | 7 | 8 | 7 | 8 | 7 | 6 | 5 |
| A | 0 | 3 | 2 | 4 | 8 | 8 | 7 | 7 | 11 | 10 | 9 | 10 | 9 |
| T | 0 | 2 | 3 | 3 | 7 | 7 | 7 | 10 | 10 | 14 | 13 | 12 | 13 |
| C | 0 | 1 | 2 | 6 | 6 | 10 | 9 | 9 | 9 | 13 | 13 | 12 | 12 |
| A | 0 | 3 | 2 | 5 | 5 | 9 | 9 | 8 | 12 | 12 | 12 | 16 | 15 |
| C | 0 | 2 | 2 | 5 | 4 | 8 | 8 | 8 | 11 | 11 | 11 | 15 | 15 |

**Fig. 2** An example of how to compute the *nid*-value in the traceback procedure. The matrix $H_A(i,j)$ is shown for a linear gap penalty $\alpha = 1$, and a substitution score +3 for the exact match and −1 otherwise. The *nid*-value here is five

$$N_A(i,j) = \begin{cases} 0, & \text{if } H_A(i,j) = 0 \\ N_A(i-1,j-1) + m(i,j), & \text{if } H_A(i,j) = H_A(i-1,j-1) + sbt(S_1[i], S_2[j]) \\ N_E(i,j), & \text{if } H_A(i,j) = E(i,j) \\ N_F(i,j); & \text{if } H_A(i,j) = F(i,j) \end{cases}$$

where

$$m(i,j) = \begin{cases} 1, & \text{if } S_1[i] = S_2[j] \\ 0; & otherwise \end{cases}$$

$$N_E(i,j) = \begin{cases} 0, & \text{if } j = 1 \\ N_A(i,j-1), & \text{if } E(i,j) = H_A(i,j-1) - \alpha \\ N_E(i,j-1); & \text{if } E(i,j) = E(i,j-1) - \beta \end{cases}$$

$$N_F(i,j) = \begin{cases} 0, & \text{if } i = 1 \\ N_A(i-1,j), & \text{if } F(i,j) = H_A(i-1,j) - \alpha \\ N_F(i-1,j); & \text{if } F(i,j) = F(i-1,j) - \beta \end{cases}$$

It can be shown that

$$nid(S_1, S_2) = N_A(i_{max}, j_{max})$$

where $(i_{max}, j_{max})$ denote the coordinates of the maximum value in the corresponding pairwise local alignment DP matrix $H_A$.

Input data set sizes for MSA are typically smaller than for database search (protein sequence databases typically contain may millions of sequences while large-scale MSAs are computed for a few thousand protein sequences) making the subject sequence set for distance matrix computation comparatively small. In order to design an efficient parallel distance matrix computation algorithm on Xeon Phi, we have used the task partitioning method shown in Fig. 3. In our method, the sequences are sorted by their lengths and then partitioned into smaller sized batches. In an alignment task, a query sequence will be aligned to the corresponding sequence batch. This procedure will continue until all task batches are calculated. We have implemented the whole process into two parallel parts: the thread level and the VPU level. On the thread level, the process aligning $S_i$ to $S = \{S_{(i+1)}, \ldots, S_n\}$ is grouped to $task_i$, and each task is processed by a thread. On the VPU level, multi-pairwise comparisons are performed in parallel on VPUs. In our method, $S = \{S_{(i+1)}, \ldots, S_n\}$ is packed into a 2D buffer which has 16 channels, meaning that sequence $S_i$ can be aligned to 16 different sequence in the 16-channel buffer in parallel. We have used Knights Corner instructions to implement this part. Figure 4 shows the pseudo-code of our algorithm framework. In order to take advantage of both CPUs and Xeon Phis in a node to process MSA for large-scale datasets, we have implemented our algorithm framework

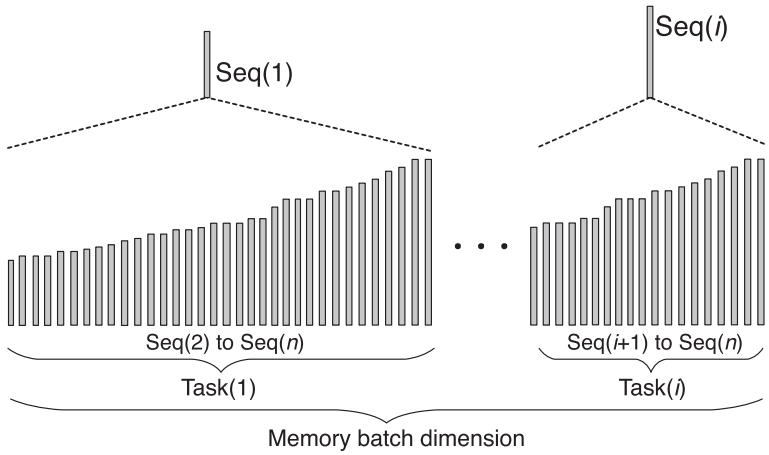


**Fig. 3** Illustration of our task partitioning scheme

using the offload model. We have implemented the arithmetic operations specified by the equations in Definition 2 using a number of Knights Corner instructions (see Fig. 5) for Xeon Phis. These instructions are executed on VPUs to calculate the sixteen residue vectors of alignment matrices according to Definition 2. For CPUs, VPUs fetch 8 residues each time. The core instructions used on CPUs are identical with Xeon Phis, whereas they have been implemented using different 256-bit AVX intrinsic instructions.

Before performing the alignment process, two temporary score vectors (the *sprofile* and the *mprofile* in Fig. 4) are created to help improve the IO efficiency for loading the substitution matrix values and the $m(i,j)$ (see Definition 2) values in parallel. Figure 6 shows an example of how to create these two temporary vectors. From Fig. 6 we can see that the substitution score matrix, the current database sequence vector, and the query sequence will be used to create the *sprofile* and the *mprofile*. VPUs will make use of these two score vectors to load substitution values and $m(i,j)$ values quickly. The shuffling procedure in Fig. 6 is used to help VPUs fetch corresponding values from the substitution matrix in parallel [7].

In our implementation, the size of these two temporary vectors for Xeon Phi and CPU is 16 and 8 separately.

We have designed and implemented a device level dynamic task distribution framework to distribute tasks to both the CPU device and the Xeon Phi device. Figure 7 shows our framework. In this framework, the task distributor is implemented as a critical section to prevent the concurrent access to shared tasks. It is also used to perform the dynamic distribution of tasks to CPUs and Xeon Phis. In Fig. 7, both CPUs and Xeon Phis fetch and process multiple tasks in parallel. After the allocated tasks are processed, both devices will send requirements to the data distributor to request for new tasks. All new task requirements will first be identified and queued by the data distributor. It then distributes tasks to the queue in order.

1. Initialize and create global variable on CPUs and Xeon Phis, including *Matrix*, *Query*, *GapOpen*, *GapExtend*, *Ratio*
2. Load sequence dataset into the CPU memory
3. Partition the sequence dataset into a set of tasks.
4. Create a distributor to assign tasks to both CPUs and Xeon Phis.
5. For each task, we pack the subject sequences into a 8-channel *buffer* for CPUs or a 16-channel *buffer* for Xeon Phis respectively.
6. Transfer *buffers* onto CPUs and Xeon Phis.
7. Launch computing threads on CPUs and Xeon Phis to process these tasks in parallel.
8. For $i = 0$ to $buffer\_size$ `/*For each buffer*/`;
9. Fetch data from a buffer, calculate the *sprofile* and *mprofile*.
10. Use 25 core instructions to implement the algorithm in *Definition 2*.
    `/*For Xeon Phis, these instructions are implemented using KCI instructions which are shown in Figure 5.*/`
    `/*CPUs use the same instructions which have been implemented using AVX intrinsics.*/`
11. Write back calculated distance matrix;

**Fig. 4** The pseudo-code of our MSA algorithm framework on a single computing node

1: $vH_A[i] = \_mm512\_add\_epi32(vH_A[i], sProfile);$
$//H_A = H_A + S(i,j)$
2: $vN_A[i] = \_mm512\_add\_epi32(vN_A[i], mProfile);$
$//N_A = N_A + m(i,j)$
3: $k_1 = \_mm512\_cmp\_epi32\_mask(vF[i], vH_A[i], \mathrm{GREATER});$
$//$set mask when $F > H_A$
4: $vH_A[\mathrm{i}] = \_mm512\_max\_epi32(vH_A[i], vF[i]);$
$//H_A = max(H_A, F)$
5: $vN_A[\mathrm{i}] = \_mm512\_mask\_mov\_epi32(vN_A[i], k_1, vN_F[i]);$
$//N_A = N_F$ according to mask $k_1$
6: $k_2 = \_mm512\_cmp\_epi32\_mask(vE, vH_A[i], \mathrm{GREATER});$
$//$set mask when $E > H_A$
7: $vH_A[i] = \_mm512\_max\_epi32(vH_A[i], vE);$
$//H_A = max(H_A, E)$
8: $vN_A[i] = \_mm512\_mask\_mov\_epi32(vN_A[i], k_2, vN_E[i+2]);$
$//N_A = N_E$ according to mask $k_2$
9: $k_3 = \_mm512\_cmp\_epi32\_mask(vZero, vH_A[i], \mathrm{GREATER});$
$//$set mask when $0 > H_A$
10: $vH_A[i] = \_mm512\_max\_epi32(vH_A[i], vZero);$
$//H_A = max(H_A, 0)$
11: $vN_A[i] = \_mm512\_mask\_mov\_epi32(vN_A[i], k_3, vZero);$
$//N_A = 0$ according to mask $k_3$
12: $k_4 = \_mm512\_cmp\_epi32\_mask(vH_A[i], vS, \mathrm{GREATER});$
$//$set mask when $H_A > S$
13: $vS = \_mm512\_max\_epi32(vS, vH_A[i]);$
$//S = max(S, H_A)$
14: $vN_S[i] = \_mm512\_mask\_mov\_epi32(vN_S[i], k4, vN_A[i]);$
$//N_S = N_A$ according to mask $k_4$
15: $vF[i] = \_mm512\_sub\_epi32(vF[i], \beta);$
$//F = F - \beta$
16: $vE = \_mm512\_sub\_epi32(vE, \beta);$
$//E = E - \beta$
17: $vH_{A2}[i] = \_mm512\_mask\_mov\_epi32(vH_{A2}[i], \mathrm{0xFFFF}, vH_A[i]);$
$//H_A = H'_A$
18: $vH_A[i] = \_mm512\_sub\_epi32(vH_A[i], \alpha);$
$//H_A = H_A - \alpha$
19: $vN_{A2}[i] = \_mm512\_mask\_mov\_epi32(vN_{A2}[i], \mathrm{0xFFFF}, vN_A[i]);$
$// N'_A = N_A$
20: $k_5 = \_mm512\_cmp\_epi32\_mask(vH_A[i], vE, \mathrm{GREATER});$
$//$set mask when $H_A > E$
21: $vE = \_mm512\_max\_epi32(vH_A[i], vE);$
$//E = max(H_A, E)$
22: $vN_E[i+2] = \_mm512\_mask\_mov\_epi32(vN_E[i+2], k_5, vN_A[i]);$
$//N_E = N_A$ according to mask $k_5$
23: $k_6 = \_mm512\_cmp\_epi32\_mask(vH_A[i], vF[i], \mathrm{GREATER});$
$//$set mask when $H_A > F$
24: $vF[i] = \_mm512\_max\_epi32(vH_A[i], vF[i]);$ $//F = max(H_A, F)$
25: $vN_F[i] = \_mm512\_mask\_mov\_epi32(vN_F[i], k_6, vN_A[i]);$
$//N_F = N_A$ according to mask $k_6$

**Fig. 5** Xeon Phi vectorized implementation of pairwise alignment according to Definition 2 by dynamic programming using 25 core instructions. The variables in these instructions can be divided into two classes. One class includes $vH_A$, $vE$, $vF$, and $vS$ which are used in the Smith-Waterman algorithm. Another class contains $vN_A$, $vN_E$, $vN_F$ and $vN_S$ which are defined in Definition 2. Here $vN_A$ is the target vector and $vN_S$ is the value $nid(S_i, S_j)$

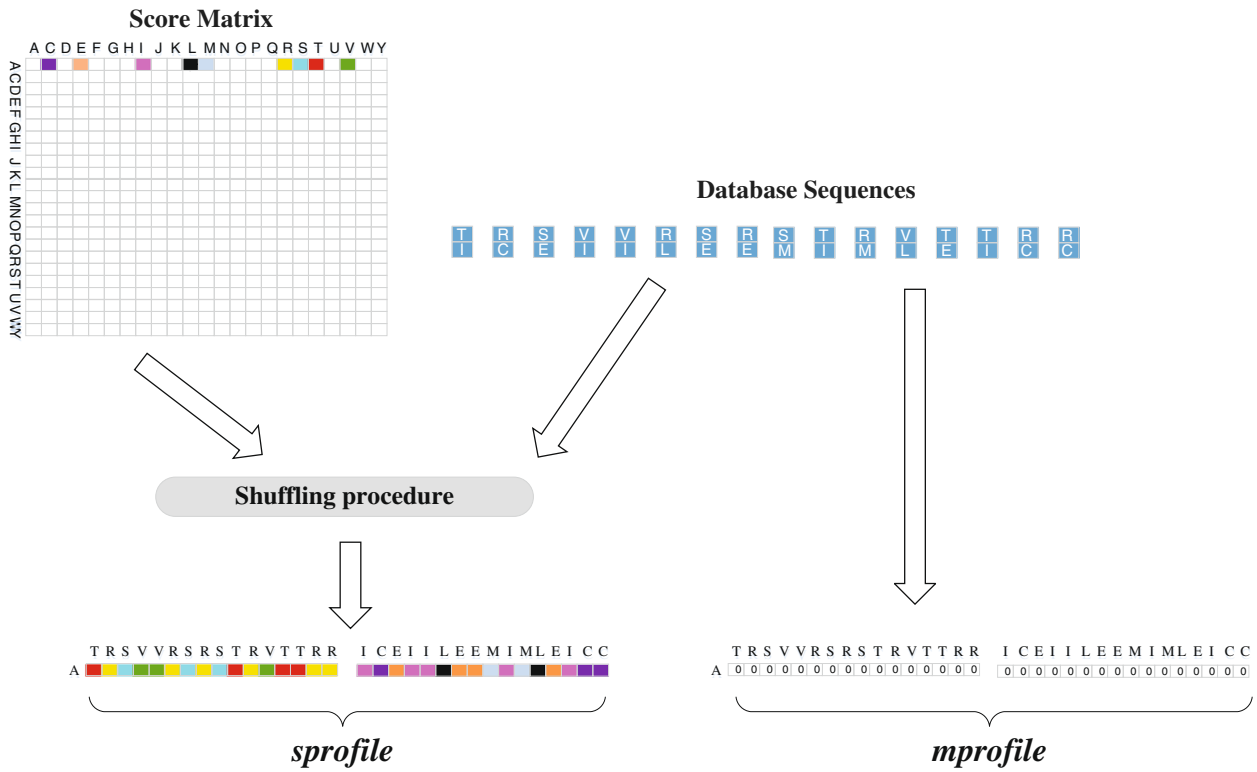


**Fig. 6** An example of how to create the *sprofile* and the *mprofile* for two sequence vectors to match the 'A' residue

## Cluster level data parallelization

Our approach is based on the fact that both subject database batches (for database searching) and MSA tasks can be scanned in parallel. Thus we have implemented the cluster level data parallel algorithm for these two alignment applications. The cluster level data parallel algorithm is encoded on the master node. The master-node partitions the subject database or the MSA tasks into a number of chunks that will be sent to different compute nodes. Our approach is implemented using the following modules:

**Dispatcher (Master):** Partitions subject database or MSA tasks into a number of chunks in a preprocessing steps and sends them to compute nodes.

**Algorithms on a Single Node (Worker):** Receives sequence chunks from master and performs the corresponding DP calculations.

**Result Collector (Master):** Performs additional operations required to further process the returned results.

### Protein sequence database search

In our work, we have implemented a static dispatcher for our cluster level parallel database searching algorithm. Figure 8 illustrates our method. In Fig. 8, the static dispatcher in the preprocess stage first divides the database into several chunks with respect to the total number of nodes. The database chunks are then sent to the corresponding node for local searching. Since the compute power of all compute nodes may vary, the size of each database subset can also vary. In order to achieve load balancing among all nodes, we have implemented a sample test method. In our method, at the preprocess stage (see Fig. 8), firstly a sample test is performed to explore the compute power of all compute nodes. Performance factors of different nodes are then automatically generated. In our work, we name this factor the compute power $P_i$ for node $i$. With the performance factor $P_i$, we can then calculate the appropriate size of the database subset allocated to node $i$.

### MSA

We have designed and implemented a cluster level dynamic dispatcher to distribute tasks to compute nodes. Figure 9 illustrates our method. In this method, the dynamic dispatcher first divides the dataset into a set of tasks which are organized as a task pool. Then, multiple tasks are sent to each node for local distance matrix computation. After the allocated tasks are processed, each node will send requirements to the dispatcher to ask for

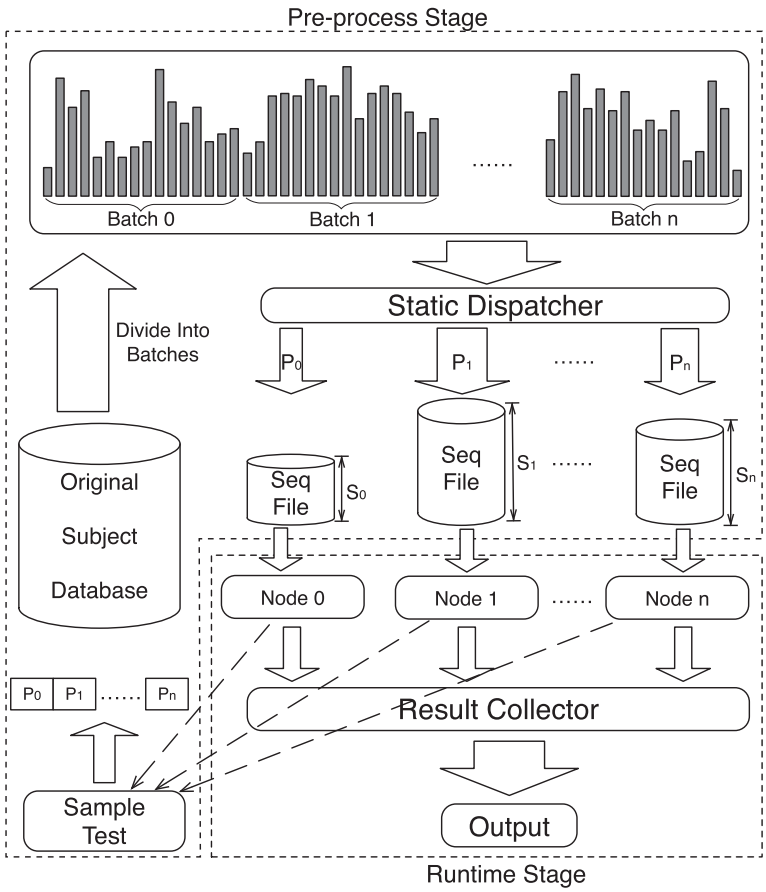


**Fig. 7** Our device level dynamic task distribution framework. The *black dots* denote tasks

new tasks to process. This procedure will continue until all tasks are processed.

## Results and discussion

### Test platforms

We have implemented the proposed methods using C++ and evaluated them on compute nodes with the following Xeon Phi cards (with ECC enabled) installed:

- *Intel Xeon Phi 7110P*: 61 hardware cores, 1.1 GHz processor clock speed, 8 GB GDDR5 device memory.
- *Intel Xeon Phi 31S1P*: 57 hardware cores, 1.1 GHz processor clock speed, 8 GB GDDR5 device memory.

Tests have been conducted on a Xeon Phi cluster with three compute nodes that are connected by an Ethernet switch. There are two Xeon E5 CPUs and 16GB RAM on each compute node. The cluster runs Centos 6.5 with the Linux kernel 2.6.32-431.17.1.el6.x86_64. The CPU configuration on each node varies, as is listed in Table 1. We also have SSD hard disks installed on each compute node.

### Protein sequence database search

A performance measure commonly used in computational biology to evaluate Smith-Waterman implementations is *cell updates per second* (*CUPS*). A CUPS represents the time for a complete computation of one entry of the DP matrix, including all comparisons, additions and maxima operations.

We have scanned three protein sequence databases: (i) the 7.5 GB UniProtKB/Reviewed and Annotated (5,943,361,275 residues in 16,110,751 sequences), (ii) the 18 GB UniProtKB/TrEMBL (13,630,914,768 residues in 42,821,879 sequences), and (iii) the 37 GB merged Non-Redundant plus UniProtKB/TrEMBL (24,323,686,690 residues in 73,401,766 sequences) for query sequences with varying lengths. Query sequences used in our tests have the accession numbers P01008, P42357, P56418, P07756, P19096, P0C6B8, P08519, and Q9UKN1.

#### *Performance on a single node*

We have firstly compared the single-node performance of our methods to SWAPHI [8] and CUDASW++ 3.1 [26].

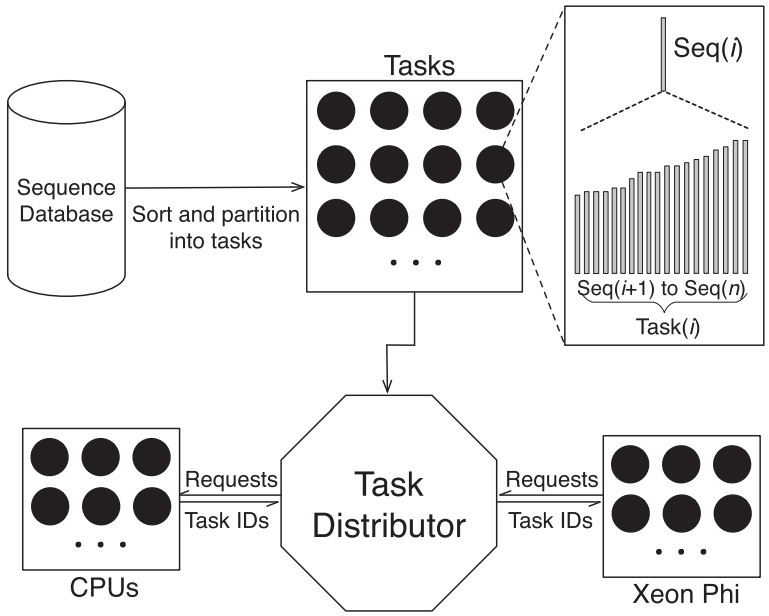


**Fig. 8** Illustration of our method to dispatch database subsets to all nodes. The node who has more computing power will be dispatched more sequences, which will finally balance the workload at runtime

SWAPHI is another parallel Smith-Waterman algorithm on Xeon Phi-based neo-heterogeneous architectures. It is also implemented using the offload model. However, SWAPHI can only run search tasks on Xeon Phi; i.e. it does not exploit the computing power of multi-core CPUs. SWAPHI cannot handle search tasks for large-scale biological databases. In our tests, we find that the database size limitation for SWAPHI is less than the available RAM size; i.e. 16 GB. CUDASW++ 3.1 is currently the fastest available Smith-Waterman implementation for database searching. It makes use of the compute power of both the CPU and GPU. At the CPU side, CUDASW++ 3.1 carries out parallel database searching by invoking the SWIPE [18] program. It employs CUDA

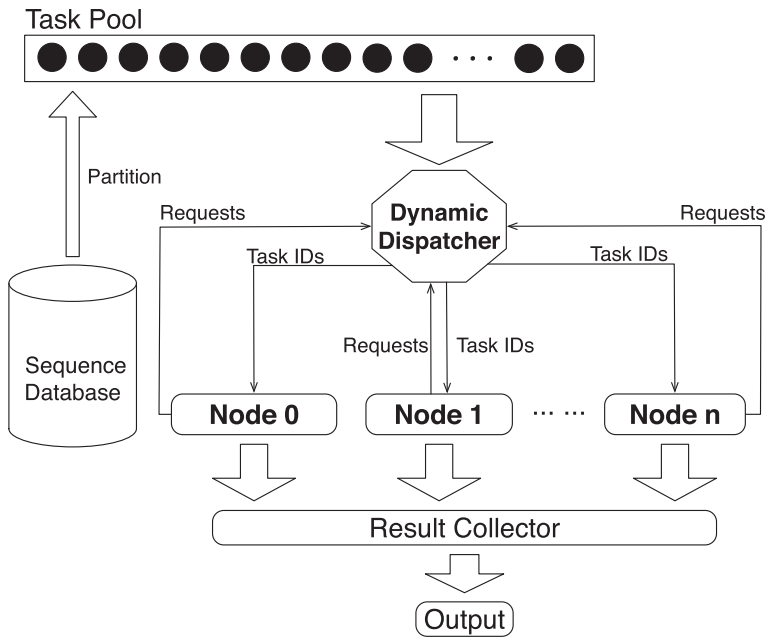


**Fig. 9** Illustration of our method to dispatch tasks dynamically to all nodes. The task partition method is illustrated in Fig. 3

**Table 1** Test cluster configurations

| Node | CPU | Coprocessor |
|---|---|---|
| $N_1$ | Xeon E5-2620 (6 cores) * 2 | Xeon Phi 7110p * 1 |
| $N_2$ | Xeon E5-2620v2 (6 cores) * 2 | Xeon Phi 7110p * 2 |
| $N_3$ | Xeon E5-2650v2 (8 cores) * 2 | Xeon Phi 31s1p * 4 |

PTX SIMD video instructions to gain the data parallelism at the GPU side. The database size supported by CUDASW++ 3.1 is less than the memory size available on the GPU. Neither SWAPHI nor CUDASW++ 3.1 supports clusters.

For single-node tests, we have used the $N_2$ node (see Table 1) as test platform. In our experiments, we run our methods with 24 threads on two Intel E5-2620 v2 six-core 2.0 GHz CPUs and 240 threads on each Intel Xeon Phi 7110P respectively. We execute SWAPHI with 240 threads on each Xeon Phi 7110P. We have executed CUDASW++ 3.1 on another server with the same two Intel E5-2620 v2 six-core 2.0 GHz CPUs plus two Nvidia Tesla Kepler K40 GPUs with ECC enabled. 24 CPU threads are also used for CUDASW++ 3.1. If not specified, default parameters are used for both SWAPHI and CUDASW++ 3.1. Furthermore, all available compiler optimizations have been enabled. The parameters $\alpha = 10$, and $\beta = 2$ have been used in our experiments. The substitution matrix used is BLOSUM62.

We have measured the time to compute the similarity matrices to calculate the *computing CUPS* values in our experiments. Figure 10a shows the corresponding computing GCUPS values of our methods, SWAPHI and CUDASW++ 3.1 for searching the 7.5 GB UniProtKB/Reviewed and Annotated protein database using different query sequences. From Fig. 10a we can see that the computing GCUPS of our multi-pass method is comparable to CUDASW++ 3.1. Both of them achieve better performance than SWAPHI.

SWAPHI and CUDASW++ 3.1 cannot support search tasks for the 18 GB and 37 GB databases. Thus, we only use our methods to search them. Figure 10a also reports the performance of our methods for searching these two databases. The results show that our methods can handle large-scale database search tasks efficiently.

#### *Performance on a cluster*

Figure 10b shows the performance of our methods using all three cluster nodes. The result indicates that our methods exhibit good scalability in terms of sequence length and size, and number of compute nodes. Our method achieves a peak overall performance of 730 GCUPS on the Xeon Phi-based cluster.

### MSA

A set of performance tests have been conducted using different protein sequence datasets to evaluate the processing time for the distance matrix computation step of our implementation in comparison to MSA-CUDA [32]. The datasets are extracted from the UniProtKB/Reviewed database, whose details are listed in Table 2. We have used two groups of datasets in our tests. Datasets $S_1$ to $S_6$ are used to compare the performance of our method and MSA-CUDA, where the sequence numbers are small since MSA-CUDA can not handle datasets with large sequence number. Datasets $L_1$ to $L_6$ are used to evaluate

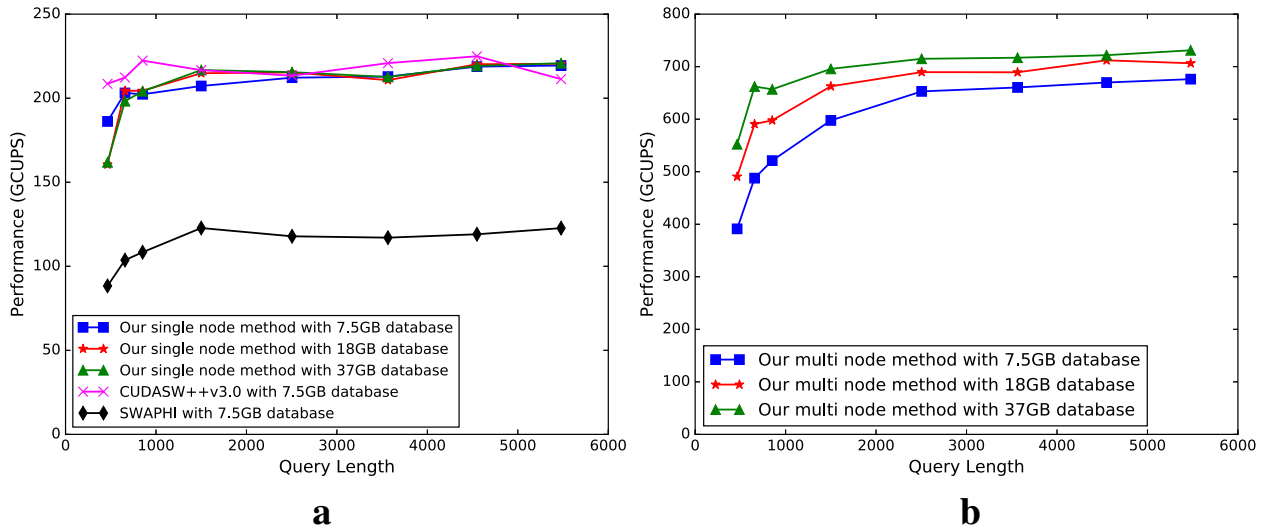


**Fig. 10 a** performance comparison on a single node ($N_2$) between our method, CUDASW++v3.1 and SWAPHI. **b** performance results of our method using all three compute nodes

**Table 2** Test datasets for MSA

| Dataset | Avg. Length | #Sequences | Workload (GCells) |
|---|---|---|---|
| $S_1$ | 465 | 200 | 4.35 |
| $S_2$ | 472 | 400 | 17.84 |
| $S_3$ | 474 | 600 | 40.52 |
| $S_4$ | 476 | 800 | 72.56 |
| $S_5$ | 476 | 1000 | 113.54 |
| $S_6$ | 480 | 1200 | 164.13 |
| $L_1$ | 150 | 30000 | 10891 |
| $L_2$ | 382 | 16000 | 18692 |
| $L_3$ | 935 | 10000 | 39148 |
| $L_4$ | 274 | 40000 | 60246 |
| $L_5$ | 1350 | 10000 | 88013 |
| $L_6$ | 700 | 24000 | 133112 |

the performance of our method for handling large-scale datasets. These datasets consist at least 10,000 sequences.

The workload for computing a distant matrix grows quadratically with respect to the number of input sequences. The average sequence length of the dataset also has a great impact on the computing workload. We have used the following equation to measure the workload needed to process a dataset.

$$W = \sum_{i=1}^{n} \left( L_i * \sum_{j=i+1}^{n} L_j \right)$$

where $L_i$ denotes the length of the $i$th sequence in the dataset. Thus, the workload $W$ is actually the total number of matrix cells to be calculated. As our method utilizes the constant 25 instructions for calculating each cell (as is listed in Fig. 5), the execution time grows linearly with $W$. Table 2 also lists the workload needed for processing each dataset.

#### *Performance for processing medium-scale datasets*

For the medium-scale datasets $S_1$ to $S_6$, MSA-CUDA is benchmarked on a Tesla K40 GPU with default options and all available compiler optimizations enabled. Our implementation runs on an Intel Xeon Phi 7110P with 240 threads. Figure 11 shows the performance comparison between our method and MSA-CUDA. From Fig. 11 we can find our implementation achieves significantly better performance compared to MSA-CUDA.

#### *Performance for processing large-scale datasets*

For the large-scale datasets $L_1$ to $L_6$, MSA-CUDA cannot work normally. We have run our methods on a single Intel Xeon Phi 7110P, the $N_2$ node and the cluster respectively. The performance results are shown in Fig. 12. Figure 12 indicates that our methods exhibit very good scalability in terms of workload and number of compute nodes. Although the nodes in our cluster have different compute power, our dynamic task dispatching scheme still works efficiently. Moreover, our method on the cluster is able to process large-scale datasets that are rarely seen in other MSA implementations, whereas the runtime is still acceptable.

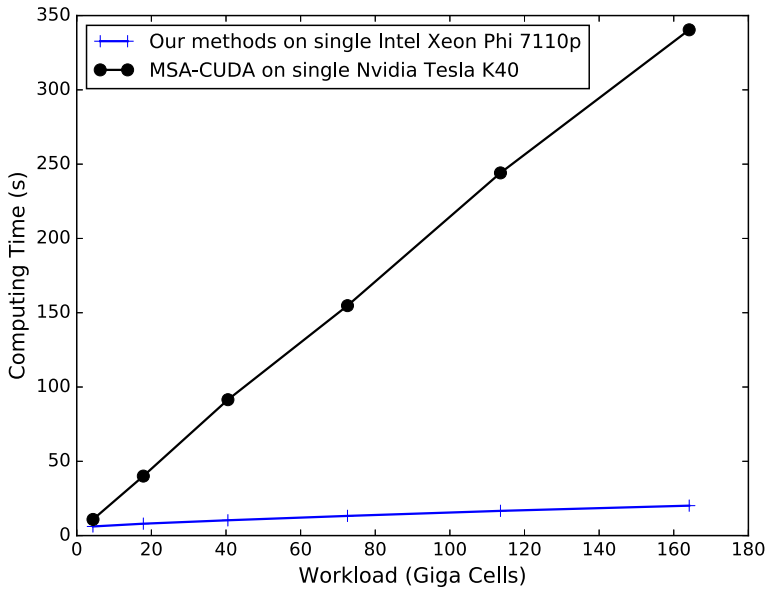


**Fig. 11** Runtime (in seconds) for processing datasets $S_1$ to $S_6$. Our method runs on a Xeon Phi 7110P. MSA-CUDA runs on a Tesla K40 GPU

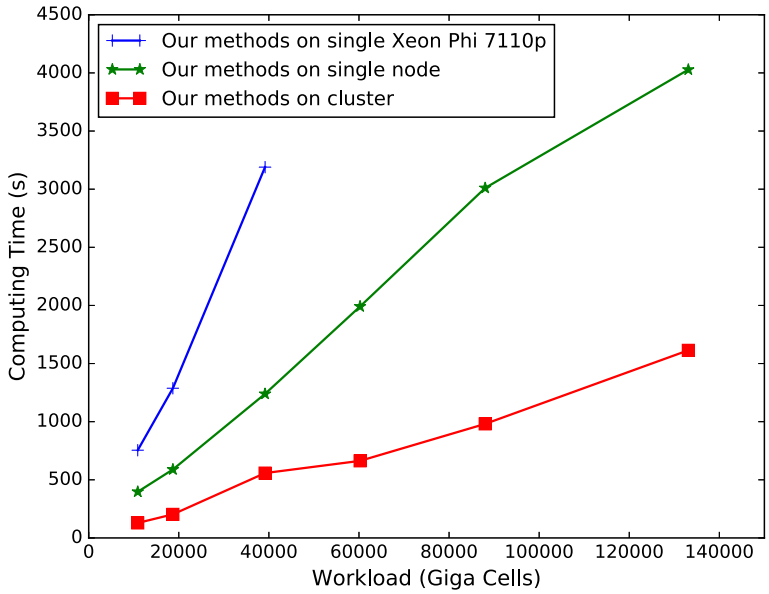


**Fig. 12** Runtime (in seconds) for processing datasets $L_1$ to $L_6$. We have run our method on an Intel Xeon Phi 7110P, the $N_2$ node and the cluster, respectively

## Conclusion

We have presented two parallel algorithms for protein sequence alignment based on the dynamic programming concept which can be efficiently mapped onto Xeon Phi clusters. Our methods exhibit good performance on a single compute node as well as good scalability in terms of sequence length and size, and number of compute nodes for both protein sequence database search and distance matrix computation employed in multiple sequence alignment. Furthermore, the achieved performance is highly competitive in comparison to other optimized Xeon Phi and GPU implementations. Biological sequence databases are continuously growing establishing the need for even faster parallel solutions in the future. Hence, our results are especially encouraging since performance of many-core architectures grows much faster than Moore's law as it applies to CPUs. For instance, the performance improvement with at least a factor of 3 can be expected on the already announced next-generation Xeon Phi product.

**Declarations**

Publication of this article was funded by the PPP project from CSC and DAAD, Taishan Scholar, and NSFC Grants 61272056 and U1435222.

This article has been published as part of *BMC Bioinformatics* Vol 17 Suppl 9 2016: Selected articles from the IEEE International Conference on Bioinformatics and Biomedicine 2015: genomics. The full contents of the supplement are available online at http://bmcbioinformatics.biomedcentral.com/articles/supplements/volume-17-supplement-9.

**Availability of data and materials**

**Project name:** LSDBS-mpi
**Project homepage:** https://github.com/turbo0628/LSDBS-mpi
**Operating System:** Linux
**Programming Language:** C++

**Authors' contributions**

HL, BS, and WL designed the study, wrote and revised the manuscript. HL, YC, and KX implemented the algorithm, performed the tests, analysed the results. BS, SP, and WL contributed the idea of using Knights Corner instructions and Xeon Phi clusters, participated in the algorithm optimization, analysed the results. All authors read and approved the final manuscript.

**Competing interests**

The authors declare that they have no competing interests.

**Consent for publication**

Not applicable.

**Ethics approval and consent to participate**

Not applicable.

**Author details**

[1]School of Computer Science and Technology, Shandong University, Shunhua Road 1500, Jinan, Shandong, China. [2]Johannes Gutenberg University, Mainz, Germany. [3]School of Computer Science, National University of Defense Technology, Changsha, Hunan, China.

Published: 19 July 2016

**References**

1. Schmidt B, Schröder H, Schimmler M. Massively parallel solutions for molecular sequence analysis. International Parallel and Distributed Processing Symposium parallel solutions for molecular sequence analysis. IEEE; 2002. p. 0186.
2. Bader DA. Computational biology and high-performance computing. Commun ACM. 2004;47(11):34–41.
3. Rajko S, Aluru S. Space and time optimal parallel sequence alignments. IEEE Trans Parallel Distrib Syst. 2004;15(11):1070–81.
4. Liu Y, Schmidt B. SWAPHI: Smith-waterman protein database search on Xeon Phi coprocessors. Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on. IEEE; 2014. p. 184–5.
5. Heinecke A, Vaidyanathan K, Smelyanskiy M, et al. Design and implementation of the linpack benchmark for single and multi-node systems based on intel xeon phi coprocessor. Parallel & Distributed

Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE; 2013. p. 126–37.
6. Pennycook SJ, Hughes CJ, Smelyanskiy M, et al. Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE; 2013. p. 1085–97.
7. Wang L, Chan Y, Duan X, et al. XSW: Accelerating biological database search on xeon phi. Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International. IEEE; 2014. p. 950–7.
8. Liu Y, Maskell DL, Schmidt B. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. BMC Res Notes. 2009;2(1):73.
9. Lan H, Liu W, Schmidt B, et al. Accelerating large-scale biological database search on Xeon Phi-based neo-heterogeneous architectures. Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on. IEEE; 2015. p. 503–10.
10. Rucci E, García C, Botella G, Degiusti A, Naiouf M, Prieto-Matías M. An energy-aware performance analysis of swimm: Smith—waterman implementation on i ntel's m ulticore and m anycore architectures. Concurr Comput Pract Experience. 2015;22(6):865–72.
11. Lu M, Zhang L, Huynh HP, et al. Optimizing the mapreduce framework on intel xeon phi coprocessor. Big Data, 2013 IEEE International Conference on. IEEE; 2013. p. 125–30.
12. Thompson J, Higgins D, Gibson T. ClustalW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting position specific gap penalties and weight matrix choice. Nucleic Acids Res. 1994;22:4673–680.
13. Feng D, Doolittle R. Progressive sequence alignment as a prerequisite to a correct phylogenetic trees. J Mol Evol. 1987;25:351–60.
14. Saitou N, Nei M. The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol Biol Evol. 1987;4:406–25.
15. Wozniak A. Using video-oriented instructions to speed up sequence comparison. Comput Appl Biosci. 1997;13(2):145–50.
16. Rognes T, Seeberg E. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics. 2000;16(8):699–706.
17. Alpern B, Carter L, Su Gatlin K. Microparallelism and high-performance protein matching. Proceedings of the 1995 ACM/IEEE conference on Supercomputing. ACM; 1995. p. 24.
18. Rognes T. Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation. BMC Bioinforma. 2011;12:.
19. Edgar RC. Muscle: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Res. 2004;32(5):1792–7.
20. Notredame C, Higgins D, Heringa J. T-coffee: A novel method for fast and accurate multiple sequence alignment. J Mol Biol. 2000;302:205–17.
21. Chaichoompu K, Kittitornkun S, Tongsima S. MT-ClustalW: multithreading multiple sequence alignment. Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. IEEE; 2006. p. 8.
22. Wirawan A, Kwoh CK, Hieu NT, et al. CBESW: sequence alignment on the playstation 3. BMC Bioinforma. 2008;9(1):377.
23. Szalkowski A, Ledergerber C, Krähenbühl P, et al. SWPS3–fast multi-threaded vectorized Smith-Waterman for IBM Cell/BE and x86/SSE2. BMC Res Notes. 2008;1(1):107.
24. Liu W, Schmidt B, Voss G, Mueller-Wittig W. Streaming algorithms for biological sequence alignment on gpus. IEEE Trans Parallel Distrib Syst. 2007;18(9):1270–81.
25. Liu Y, Schmidt B, Maskell DL. CUDASW++ 2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. BMC Res Notes. 2010;3(1):93.
26. Liu Y, Wirawan A, Schmidt B. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinforma. 2013;14(1):117.
27. Manavski S, Valle G. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinforma. 2008;9(2):1.
28. Ligowski L, Rudnicki W. An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. 2009 International Parallel and Distributed Processing Symposium. IEEE; 2009. p. 1–8.
29. Khajeh-Saeed A, Poole S, PJ B. Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors. J Comput Phys. 2010;229(11):4247–58.
30. Blazewicz J, Frohmberg W, Kierzynka M, Pesch E, Wojciechowski P. Protein alignment algorithms with an efficient backtracking routine on multiple gpus. BMC Bioinforma. 2011;12:181.
31. Hains D, Cashero Z, Ottenberg M, et al. Improving CUDASW++, a parallelization of Smith-Waterman for CUDA enabled devices. Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on. IEEE; 2011. p. 490–501.
32. Liu Y, Schmidt B, Maskell DL. MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on. IEEE; 2009. p. 121–8.
33. Hung CL, Lin YS, Lin CY, Chung YC, Chung YF. CUDA ClustalW: An efficient parallel algorithm for progressive multiple sequence alignment on multi-gpus. Comput Biol Chem. 2015;58:62–8.
34. Li K. ClustalW analysis using parallel and distributed computing. Bioinformatics. 2003;19:1585–6.
35. Ebedes J, Datta A. Multiple sequence alignment in parallel on a workstation cluster. Bioinformatics. 2004;20:1193–5.
36. Cheetham J, Dehne F, Pitre S, et al. Parallel clustal w for pc clusters[M]. Computational Science and Its Applications—ICCSA 2003. Berlin Heidelberg: Springer; 2003, pp. 300–9.
37. Tan J, Feng S, Sun N. Parallel multiple sequences alignment in SMP cluster. Int Conf High Perform Comput Asia Reg. 2005;20:425–31.
38. Oliver T, Schmidt B, Maskell D. Hyper customized processors for bio-sequence database scanning on FPGAs. Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays. ACM; 2005. p. 229–37.
39. Li ITS, Shum W, Truong K. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). BMC Bioinforma. 2007;8(1):1.
40. Oliver T, Schmidt B, Nathan D, Clemens R, Maskell D. Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW. Bioinformatics. 2005;21:3431–432.
41. Boukerche A, Correa JM, de Melo ACMA, et al. An FPGA-based accelerator for multiple biological sequence alignment with DIALIGN[M]. High Performance Computing-HiPC 2007. Berlin Heidelberg: Springer; 2007. p. 71–82.