SWAP-Assembler: Scalable and Efficient Genome Assembly towards Thousands of Cores

Jintao Meng[1,2,3], Bingqiang Wang[4] , Yanjie Wei*[1] , Shengzhong Feng[1], Pavan Balaji[5]

[1]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, 518055, P.R. China
[2]Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100190, P.R.China
[3]University of Chinese Academy of Sciences, Beijing, 100049, P.R.China
[4]Beijing Genomics Institute, Shenzhen, 518083, P.R. China
[5]Mathematics and Computer Science Division, Argonne National Laboratory, 60439-4844, USA

Email: Jintao Meng - meng.jintao@gmail.com; Bingqiang Wang - wangbingqiang@genomics.cn; Yanjie Wei - yj.wei@siat.ac.cn; Shengzhong Feng - sz.feng@siat.ac.cn; Pavan Balaji - balaji@mcs.anl.gov;

*Corresponding author

Abstract

Background: There is a widening gap between the throughput of massive parallel sequencing machines and the ability to analyze these sequencing data. Traditional assembly methods requiring long execution time and large amount of memory on a single workstation limit their use on these massive data.

Results: This paper presents a highly scalable assembler named as SWAP-Assembler for processing massive sequencing data using thousands of cores, where SWAP is an acronym for Small World Asynchronous Parallel model. In the paper, a mathematical description of multi-step bi-directed graph (MSG) is provided to resolve the computational interdependence on merging edges, and a highly scalable computational framework for SWAP is developed to automatically preform the parallel computation of all operations. Graph cleaning and contig extension are also included for generating contigs with high quality. Experimental results show that SWAP-Assembler scales up to 2048 cores on Yanhuang dataset using only 26 minutes, which is better than several other parallel assemblers, such as ABySS, Ray, and PASHA. Results also show that SWAP-Assembler can generate high quality contigs with good N50 size and low error rate, especially it generated the longest N50 contig sizes for Fish and Yanhuang dataset.

Conclustions: In this paper, we presented a highly scalable and efficient genome assembly software, SWAP-Assembler. Compared with several other assemblers, it showed very good performance in terms of scalability and contig quality. This software is available at: https://sourceforge.net/projects/swapassembler.

Background

To cope with massive sequence data generated by next-generation sequencing machines, a highly scalable and efficient parallel solution for fundamental bioinformatic applications is important [1,2]. With the help of high performance computing, cloud computing [3,4], and many-cores in GPU [5], successful scalable examples have been seen in many embarrassingly parallel applications: sequence alignment [6–8], SNP searching [9,10], expression analysis [11], etc. However, for tightly coupled graph related problems, such as genome assembly, a scalable solution is a still a big challenge [12,13].

State-of-the-art trials on parallel assemblers include ABySS [14], Ray [15], PASHA [16], and YAGA [17–19]. ABySS adopts the traditional de Bruijn graph data structure proposed by Pevzner et. al. [20] and follows the similar assembly strategy as EULER_SR [21] and Velvet [22]. The parallelization is achieved by distributing k-mers to multi-servers to build a distributed de Bruijn graph, and error removal and graph reduction are implemented over MPI communication primitives. Ray extends k-mers (or seeds) into contigs with a heuristical greedy strategy by measuring the overlapping level of reads in both direction. Based on the observation that the time consuming part of genome assembly are generating and distributing k-mers, constructing and simplifying the distributed de Bruijn graph, PASHA concentrates its effort on parallelizing these two stages to improve its efficiency. However, PASHA allows only single process for each unanimous path, and this limit its degree of parallelism. In their experiments, ABySS and PASHA take about 87 hours and 21 hours to assembly the Yoruban male genome with a coverage of 42X.

To avoid merging k-mers on two different servers, which can result in too many small inter-process messages and the communication latency, YAGA constructs a distributed de Bruijn graph by maintaining edge tuples in a community of servers. Reducible edges belonging to one unanimous path are grouped into one server using a list rank algorithm [23], then these unanimous paths are reduced locally on separated servers. The complexity of YAGA is bounded by $O(\frac{n}{p})$ computing time, $O(\frac{n}{p})$ communication volume, and $O(log^2(n))$ communication rounds, where n is the number of nucleotides in all reads, and p denotes the number of processors. Due to the fact that the recursive list ranking algorithm used in YAGA has a memory usage of $O(\frac{nlogn}{p})$, this will use large amount of memory and cause low efficiency.

Our previous work [24] tries to avoid access collision of merging two neighbor edges. In this work, 1-step bi-directed graph and a computational model named as SWAP are proposed for edge merging operation. In its experiments, the prototype of edge merging algorithm using SWAP can scale to 640 cores on both Yeast and C.elegans dataset. However this exploratory work only focuses on the edge merging operation of genome assembly, some other important problems are not addressed, for example, contig extension, complexity analysis etc.

The scalability of previous assemblers is affected by the computational interdependence on merging k-mers/edges in unanimous paths. Sequential assemblers, for example Velvet and SOAP-denovo, process each path sequentially. Parallel assemblers can process several paths in parallel, however k-mers/edges sharing one path are merged one by one. SWAP-Assembler resolves the computational interdependence on merging edges sharing the same path with MSG. For each path, at most half of its edges can be merged concurrently in each round, and merging multiple edges on the same path can be done in parallel using SWAP computational framework. In Figure 1, the parallel strategy of SWAP-Assembler is compared with other assemblers using an example of two linked paths, we can see that a deeper parallelism on edge merging can be achieved by SWAP-Assembler.

In this paper, we present a highly scalable and efficient genome assembler named as SWAP-Assembler, which can scale to thousands of cores on processing massive sequencing data such as Yanhuang (500G). SWAP-Assembler includes five fully parallelized steps: input parallelization, graph construction, graph cleaning, graph reduction and contig extension. Compared with our previous work, two fundamental improvements have been made for graph reduction. Firstly MSG is presented as a comprehensive mathematical abstraction for genome assembly. Using MSG and semi-group theory, the computational interdependence on merging edges is resolved. Secondly, we have developed a scalable computational framework for SWAP, this framework triggers the parallel computation of all operations with no interference. In this paper, complexity of this framework and SWAP-Assembler is also analyzed and proved in detail. In addition, two steps in SWAP-Assembler are used to improve the quality of contigs. One is graph cleaning, which adopts the traditional strategy of removing k-molecules and edges with low frequency, and the other one is contig extension, which resolves special edges and some cross nodes using a heuristic method. Experimental results show that SWAP-Assembler can scale up to 2048 cores on Yanhuang dataset using only 26 minutes, which is the fastest compared with other assemblers, such as ABySS, Ray and PASHA. Conitg evaluation results confirm that SWAP-Assembler generates good results on N50 size with lowest error rate for *S. aureus* and *R. sphaeroides* datasets. When processing larger datasets (Fish and Yanhuang) without using external error correction tools, SWAP-Assembler generates the longest N50 contig sizes of 1309 bp and 1368 bp for these two datasets.

Methods

In this section, our method for genome assembly towards thousands of cores is presented. We first abstract the genome assembly problem with MSG. Generating longer sequences (contigs) from shorter sequences corresponds to merging semi-extended edges to full-extended edges in MSG. In addition, computational interdependence of edge merging is resolved by introducing a semi-group over a closed edge set $E_s \bigvee \mathbf{0}$ in MSG. The edge set $E_s \bigvee \mathbf{0}$ is proved to be a semi-group with respect to edge merging operation. According to the associativity law of semi-group, the final results will be the same as long as all edges have been merged regardless of the merging order, thus these edge merging operations can be computed in parallel.

In order to maximally utilize the potential parallelism resolved by MSG, a scalable SWAP computational framework is developed. As one edge may be accessed by two merging operations in two different processes at the same time, a lock-computing-unlock mechanism introduced in [24] is adopted for avoiding the conflict. For the problems which can be abstracted with semi-group, the corresponding operations can be done in parallel, and SWAP computational framework can achieve linearly scale up for these problems.

Based on MSG and SWAP computational framework, SWAP-Assembler is developed with five steps, including input parallelization, graph construction, graph cleaning, graph reduction, and contig extension. In the following, we first present MSG and the SWAP computational framework, then details of SWAP-Assembler's five steps will be followed.

Mathematical formulation of genome assembly using MSG

Given a biological genome sample with reference sequence $w \in \mathbb{N}^g$, where $\mathbb{N} = \{A, T, C, G\}$, $g = |w|$, a large number of short sequences called **reads**, $S = \{s_1, s_2, ..., s_h\}$, can be generated from the sequencing machines. Genome assembly is the process of reconstructing the reference genome sequence from these reads. Unfortunately, the genome assembly problem of finding the shortest string with all reads as its substring falls into a NP-hard problem [25].

Finding the original sequence from all possible Euler paths cannot be solved in polynomial time [26]. In real cases, gaps and branches caused by uneven coverage, erroneous reads and repeats prevent obtaining full length genome, and a set of shorter genome sequences called **contigs** are generated by merging unanimous paths instead. Our method focuses on finding a mathematical and highly scalable solution for the following standard genome assembly (SGA) problem, which is also illustrated in Figure 2.

Problem of Standard Genome Assembly (SGA)

Input: Given a set of reads without errors $S = \{s_1, s_2, ..., s_h\}$

Output: A set of contigs $C = \{c_1, c_2, ..., c_w\}$

Requirement: Each contig maps to an unanimous path in the De Bruijn graph constructed from the set of reads S.

Preliminaries

We first define some variables. Let $s \in \mathbb{N}^l$ be a DNA sequence of length l. Any substring derived from s with length k, is called a k-mer of s, and it is denoted by $\alpha = s[j]s[j+1]\ldots s[j+k-1], 0 \leqslant j < l-k+1$. The set of k-mers of a given string s can be written as $\mathbb{Z}(s, k)$, where k must be odd. The reverse complement of a k-mer α, denoted by α', is obtained by reversing α and complementing each base by the following bijection of $\mathbb{M}$, $\mathbb{M} : \{a \to t, t \to a, c \to g, g \to c\}$. Note that $\alpha = \alpha''$.

A k-molecule $\hat{\alpha}$ is a pair of complementary k-mers $\{\alpha, \alpha'\}$. Let $\geqslant$ be the partial ordering relation between the strings of equal length, and $\alpha \geqslant \beta$ indicates that α is lexicographically larger than β. We designate the lexicographically larger one of two complementary k-mers as the positive

k-mer, denoted as α^+, and the smaller one as the negative k-mer, denoted as α^-, where $\alpha^+ \gtrdot \alpha^-$. We choose the positive k-mer α^+ as representative k-mer for k-molecule $\{\alpha, \alpha'\}$, denoted as α^+, implying that $\hat{\alpha} = \alpha^+ = \{\alpha^+, \alpha^-\} = \{\alpha, \alpha'\}$. The relationship between k-mer and k-molecule is illustrated in Figure 3. The set of all k-molecules of a given string s is known as k-spectrum of s, and it can be written as $\mathbb{S}(s, k)$. Noted that $\mathbb{S}(s, k) = \mathbb{S}(s', k)$.

Notation $suf(a, l)(pre(a, l)$, respectively) is used to denote the length l suffix (prefix) of string a. The symbol $\circ$ is introduced to denote the concatenation operation between two strings. For example, if $s_1 =$ "*abc*"$, s_2 =$ "*def*", then $s_1 \circ s_2 =$ "*abcdef*". The number of edges attached to k-molecule $\hat{\alpha}$ is denoted as $degree(\hat{\alpha})$. All notations are listed in Table 1.

1-*step Bi-directed Graph*

Definition 1: 1-step bi-directed graph. The 1-step bi-directed de Bruijn graph of order k for a given string s can be presented as,

$$G_k^1(s) = \{V_s, E_s^1\} \tag{1}$$

In the rest of the paper, 1-step bi-directed de Bruijn graph of order k is abbreviated as **1-step bi-directed graph**. In equation (1), the vertex set V_s is the k-spectrum of s,

$$V_s = \mathbb{S}(s, k) \tag{2}$$

and the 1-step bi-directed edge set E_s^1 is defined as follows,

$$\begin{gathered} E_s^1 = \{e_{\alpha\beta}^1 = (\alpha, \beta, d_\alpha, d_\beta, c_{\alpha\beta}^1) | \forall \hat{\alpha}, \hat{\beta} \in \mathbb{S}(s, k), suf(\alpha, k-1) = \\ pre(\beta, k-1) \wedge (\alpha \circ \beta[k-1] \in (\mathbb{Z}(s, k+1) \vee \mathbb{Z}(s', k+1)))\} \end{gathered} \tag{3}$$

Equations (3) declares that any two overlapped k-molecules can be connected with one 1-step bi-directed edge when they are consecutive in sequence s or the complementary sequence s'. Here d_α denotes the direction of k-mer α, if $\alpha = \alpha^+$, $d_\alpha =$'+', otherwise $d_\alpha =$ '-'. $c_{\alpha\beta}^1$ is the content or **label** of the edge, and is initialized with $\beta[k-1]$, that is $c_{\alpha\beta}^1 = \beta[k-1]$, and we have $suf(\alpha \circ c_{\alpha\beta}^1, k) = \beta$.

Lemma 1. Given two k-molecules $\hat{\alpha}, \hat{\beta} \in \mathbb{S}(s, k)$, there are four possible connections, and for each type of connection exactly two equivalent 1-step bi-directed edge representations exist,

1. $e_{\alpha^+\beta^+}^1 = (\alpha^+, \beta^+, +, +, c_{\alpha^+\beta^+}^1)$, $e_{\beta^-\alpha^-}^1 = (\beta^-, \alpha^-, -, -, c_{\beta^-\alpha^-}^1)$

2. $e_{\alpha^+\beta^-}^1 = (\alpha^+, \beta^-, +, -, c_{\alpha^+\beta^-}^1)$, $e_{\beta^+\alpha^-}^1 = (\beta^+, \alpha^-, +, -, c_{\beta^+\alpha^-}^1)$

3. $e^1_{\alpha^-\beta^+} = (\alpha^-, \beta^+, -, +, c^1_{\alpha^-\beta^+})$, $e^1_{\beta^-\alpha^+} = (\beta^-, \alpha^+, -, +, c^1_{\beta^-\alpha^+})$

4. $e^1_{\alpha^-\beta^-} = (\alpha^-, \beta^-, -, -, c^1_{\alpha^-\beta^-})$, $e^1_{\beta^+\alpha^+} = (\beta^+, \alpha^+, +, +, c^1_{\beta^+\alpha^+})$

In each type of connection, the first bi-directed edge representation and the second one are equivalent. The first bi-directed edge is associated with k-molecule $\hat{\alpha}$, and the second one is associated with $\hat{\beta}$. Figure 4 illustrates all four possible connections. For example in figure 4-(a), a positive k-mer "TAG" points to positive k-mer "AGT" with a label "A", and the corresponding edge is $e^1_{TAG\ AGT} = (TAG, AGT, +, +, T)$.

Given a set of reads $S = \{s_1, s_2, \ldots, s_{\mathrm{h}}\}$, a 1-step bi-directed graph derived from S with order k is,

$$G^1_k(\mathrm{S}) = \{V_S, E^1_S\} = \{\bigcup_{1 \leqslant i \leqslant h} V_{s_i}, \bigcup_{1 \leqslant i \leqslant h} {E_{s_i}}^1\} \tag{4}$$

Each read s_i corresponds to a path in $G^1_k(S)$, and read s_i can be recovered by concatenating $(k-1)$-prefix of the first k-molecule and the edge labels on the path consisted by $\mathbb{S}(s_i, k)$. As an example, an 1-step bi-directed de Bruijn graph derived from $S = \{$"$TAGTCG$", "$AGTCGA$", "$TCGAGG$"$\}$ is plotted in Figure 5.

Multi-step Bi-directed Graph and Its Properties

Definition 2: edge merging operation. Given two 1-step bi-directed edges $e^1_{\alpha\beta} = (\alpha, \beta, d_\alpha, d_\beta, C^1_{\alpha\beta})$ and $e^1_{\beta\gamma} = (\beta, \gamma, d_\beta, d_\gamma, C^1_{\beta\gamma})$ in a 1-step bi-directed graph, if $e^1_{\alpha\beta}.d_\beta = e^1_{\beta\gamma}.d_\beta$ and $degree(\hat{\beta}) = 2$, we can obtain a 2-step bi-directed edge $e^2_{\alpha\gamma} = (\alpha, \gamma, d_\alpha, d_\gamma, c^2_{\alpha\gamma})$ by merging edges $e^1_{\alpha\beta}$ and $e^1_{\beta\gamma}$, where $c^2_{\alpha\gamma} = c^1_{\alpha\beta} \circ c^1_{\beta\gamma}$. Using symbol $\otimes$ to denote **edge merging operation** between two bi-directed edges attached to the same k-molecule with the same direction, and the 2-step bi-directed edge is written as,

$$e^1_{\alpha\beta} \otimes e^1_{\beta\gamma} = e^2_{\alpha\gamma} \quad or \quad e^1_{\gamma\beta} \otimes e^1_{\beta\alpha} = e^2_{\gamma\alpha} \tag{5}$$

Two edges $e^2_{\alpha\gamma}$ and $e^2_{\gamma\alpha}$ in equation (5) are equivalent, indicating it is same to apply edge merging operation on $e^1_{\alpha\beta}$ and $e^1_{\beta\gamma}$, and to apply edge merging operation on $e^1_{\gamma\beta}$ and $e^1_{\beta\alpha}$. Figure 6 shows an example on edge merging operation.

Zero edge 0 is defined to indicate all non-existing bi-directed edges. Note that $\mathbf{0} \otimes e^x_{\alpha\beta} = \mathbf{0}$, $e^x_{\alpha\beta} \otimes \mathbf{0} = \mathbf{0}$. A z-step bi-directed edge can be obtained by,

$$e^{z}_{\alpha\gamma} = \begin{cases} e^{x}_{\alpha\beta} \otimes e^{y}_{\beta\gamma}, & \text{if } \exists\beta, e^{x}_{\alpha\beta} \neq \mathbf{0}, e^{y}_{\beta\gamma} \neq \mathbf{0}, z = x + y, \\ & e^{x}_{\alpha\beta}.d_{\beta} = e^{y}_{\beta\gamma}.d_{\beta}, degree(\hat{\beta}) = 2 \\ \mathbf{0} & \text{otherwise} \end{cases} \tag{6}$$

Definition 3: Multi-step Bi-directed Graph(MSG). A MSG derived from a read set $S = \{s_1, s_2, \ldots, s_h\}$, is written as,

$$G_k(S) = \{V_S, E_S\} = \{ \bigcup_{1\leqslant i\leqslant h} V_{s_i}, \bigcup_{1\leqslant j\leqslant g} (\bigcup_{1\leqslant i\leqslant h} E_{s_i}{}^{j})\} \tag{7}$$

where g is the length of reference sequence w, $E^{j}_{s_i} = \{e^{j}_{\alpha\beta} | \forall\hat{\alpha}, \hat{\beta} \in \mathbb{S}(s_i, k)\}$. A MSG is obtained through edge merging operations.

Given an x-step bi-directed edge $e^{x}_{\alpha\beta} = (\alpha, \beta, d_\alpha, d_\beta, c^{x}_{\alpha\beta})$, if there exists edge $e^{y}_{\gamma\alpha}$ or $e^{z}_{\beta\gamma}$ satisfying $e^{y}_{\gamma\alpha} \otimes e^{x}_{\alpha\beta} \neq \mathbf{0}$ or $e^{x}_{\alpha\beta} \otimes e^{z}_{\beta\gamma} \neq \mathbf{0}$, then we call edge $e^{x}_{\alpha\beta}$ as a **semi-extended edge**, and the corresponding k-molecule $\hat{\alpha}$ or $\hat{\beta}$ as **semi-extended k-molecule**. If $e^{x}_{\alpha\beta}$ cannot be extended by any edge, this edge is called as **full-extended edge**, and k-molecule $\hat{\alpha}$ and $\hat{\beta}$ are **full-extended k-molecules**. In Figure 5 and 6, semi-extended k-molecule and full-extended k-molecule are plotted with different colors (yellow for semi-extended k-molecules and blue for full-extended k-molecules), semi-extended edge and full-extended edge are drawn with different lines (broken line for semi-extended edges and real line for full-extended edges).

Property 1. If the set of full-extended edges in the MSG defined in equation 7 is denoted as E^{*}_{S}, then the set of labels on all edges in E^{*}_{S} can be written as,

$$L^{*}_{S} = \{c^{x}_{\alpha\beta} | e^{x}_{\alpha\beta} = (\alpha, \beta, d_\alpha, d_\beta, c^{x}_{\alpha\beta}), e^{x}_{\alpha\beta} \in E^{*}_{S}\} \tag{8}$$

and we have $L^{*}_{S} = C$, C is the set of contigs. The proof is presented in Appendix 1.

Property 2. Edge merging operation $\otimes$ over the multi-step bi-directed edge set $E_S \bigvee \mathbf{0}$ is associative, and $Q(E_S \bigvee \mathbf{0}, \otimes)$ is a semigroup. The proof is presented in Appendix 1.

The key property of 1-step bi-directed graph $G^{1}_{k}(S)$ is that each read s corresponds to a path starting from the first k-molecule of s and ending at the last k-molecule. Similarly, each chromosome can also be regarded as a path. However because of sequencing gaps, read errors, and repeats in the set of reads, chromosome will be broken into pieces, or contigs. Within a MSG, each contig corresponds to one full-extended edge in $G_k(S)$, and this has been presented and proved in Property 1. Property 2 ensures that the edge merging operation $\otimes$ over the set of multi-step bi-directed edges has formed a semi-group, and this connects the standard genome assembly (SGA) problem with edge merging operations in semi-group. According to the associativity law of semi-group, the final full-extended edges or contigs will be the same as long as all edges have been merged regardless of the merging order, thus these edge merging operations can be computed in parallel. Finally in order to reconstruct the genome with a large set of contigs, we need to merge all semi-extended edges into full-extended edges in semi-group $Q(E_S \bigvee \mathbf{0}, \otimes)$.

SWAP computational framework

The lock-computing-unlock mechanism of SWAP was first introduced in our previous work [24], where no implementation details and complexity analysis is given. In this section, we present the mathematical description of the problems which can be solved by SWAP, then a scalable computational framework for SWAP model and its programming interface are presented. Its complexity and scalability is analyzed in Appendix 3.

Definition 4: small world of operations. Semi-group $SG(A, R)$ is defined on set A with an associative operation $R : A \times A \to A$. $R(a_i, a_j)$ is used to denote the associative operation on a_i and a_j, a_i, $a_j \in A$. The elements a_i and a_j, together with the operation $R(a_i, a_j)$ are grouped as a **small world** of the operation R(a_i, a_j). We denote this small world as $[a_i, a_j]$, and $[a_i, a_j] = \{R(a_i, a_j), a_i, a_j\}$. **Activity** $ACT(A, \sigma)$ are given on a semi-group $SG(A, R)$ as the computational works performed by a graph algorithm, where operation set σ is a subset of R.

In real application, an operation corresponds to a basic operation of a given algorithm. For example, for MSG based genome assembly application, an operation can be defined as edge merging. For topological sorting, re-ordering a pair of vertices can be defined as an operation.

For any two small worlds $[a_1, a_2], [b_1, b_2]$, where $a_1 \neq b_1, a_1 \neq b_2, a_2 \neq b_1, a_2 \neq b_2$, the corresponding operations $R(a_1, a_2)$ and $R(b_1, b_2)$, can be computed independently, thus, there exists potential parallelism in computing activity induced from the semi-group $SG(A, R)$. We use SWAP for such parallel computing. The basic schedule of SWAP is **Lock-Computing-Unlock**. For an operation $R(a, b)$ in σ, the three-steps of SWAP are listed below:

1. **Lock** action is applied to lock $R(a, b)$'s small world $[a, b]$.

2. **Computing** is performed for operation $R(a, b)$, and the values of a, b are updated accordingly. In MSG, this corresponds to merging two edges.

3. **Unlock** action is triggered to release operation $R(a,b)'s$ small world $[a,b]$.

In SWAP computational framework, all operations σ in activity $ACT(A,\sigma)$ can be distributed among a group of processes. Each process needs to fetch related elements, for example a and b, to compute operation $R(a,b)$. At the same time, this process also has to cooperate with other processes for sending or updating local variables. Each process should have two threads, one is SWAP thread, which performs computing tasks using the three-steps schedule of SWAP, and the other is service thread, which listens and replies to remote processes. In the implementation of our framework, we avoid multi-threads technology by using nonblocking communication and finite-state machine in each process to ensure its efficiency.

The activity $ACT(A,\sigma)$ on set A with operations in σ can be treated as a graph $G(\sigma,A)$ with σ as its vertices and A as its edges. Adjacent list is used to store the graph $G(\sigma,A)$, and a hash function $hashFun(x)$ is used to distribute the set σ into p subset for p processes, where $\sigma = \bigcup_{i=0}^{p-1} \sigma_i$, and A_i is associated with σ_i. Note that $ACT(A,\sigma) = \bigcup_{i=0}^{p-1} ACT_i(A_i,\sigma_i)$, which is illustrated in Figure 7. In Appendix 2, the pseudocodes of SWAP thread and service thread are demonstrated in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 describes the three-steps of SWAP computational framework, and Algorithm 2 on remote processers can cooperate with Algorithm 1 for running this schedule. The message functions, internal functions, and user-defined functions in Algorithm 1 and Algorithm 2 are listed in Table 2, where user-defined functions can be redefined for user-specific computational problems.

Similar to CSMA/CA in 802.11 protocol [27], Algorithm 1 adapts random backoff algorithm to avoid lock collision. A variety of backoff algorithms can be used, without loss of generality, binary exponential backoff [27] is used in SWAP thread. Note that all collided operations in σ_i share only one binary backoff, so the cost can be ignored as long as the number of relations in σ_i is huge.

The complexity and scalability analysis for SWAP computational framework are presented in Appendix 3. When the number of processes is less than the number of operations in σ, which is true for most cases, equation (3.8) shows that SWAP computational framework can linearly scale up with the increasing number of cores. When the number of processes is larger than the number of operations, according to equation (3.7) the running time will be dominated by the communication round, which is bounded by $log(d_{max})$, where d_{max} is the diameter of graph $G(\sigma,A)$.

Implementation of SWAP-Assembler

Based on MSG and SWAP computational framework, SWAP-Assembler consists with five steps, including input parallelization, graph construction, graph cleaning, graph reduction, and contig extension. Complexity analysis of SWAP-Assembler are presented in the end of this section.

Input parallelization

As the size of data generated by next generation sequencing technology generally has hundreds of Giga bytes, loading these data with one process costs hours to finish [14, 16]. Similar to Ray [15] and YAGA [18], we use input parallelization to speedup the loading process. Given input reads with n nucleotides from a genome of size g, we divide the input file equally into p virtual data block, p is the number of processes. Each process reads the data located in its virtual data block only once. The computational complexity of this step is bounded by $O(\frac{n}{p})$. For E.coli dataset of 4.4G bytes, SWAP-Assembler loads the data into memory in 4 seconds with 64 cores while YAGA uses 516.5 seconds [18], and for Yanhuang dataset SWAP-Assembler loads the data in 10 minutes while Ray costs 2 hour and 42 minutes.

Graph construction

This step aims to construct a 1-step bi-directed graph $G_k^1(S) = \{V_s, E_s^1\}$, where V_s and E_s^1 are k-molecule set and 1-step bi-directed edge set. In this step, input sequences are broken into overlapping k-molecules by sliding a window of length k along the input sequence. A k-molecule can have up to eight edges, and each edge corresponds to a possible one-base extension, $\{A, C, G, T\}$ in either direction. The adjacent k-molecule can be easily obtained by adding the base extension to the source k-molecule. The generated graph has $O(n)$ k-molecules and $O(n)$ bi-directed edges distributed among p processors. Graph construction of 1-step bi-directed graph can be achieved in $O(\frac{n}{p})$ parallel computing time, and $O(\frac{n}{p})$ parallel communication volume.

An improvement to our previous work [24] is that the time usage on graph construction is overlapped with the previous step. As CPU computation and network communication can be performed when only partial data are loaded from the first step, they can be overlapped and combined as a pipeline. Computation and communication time used in this step are hid behind the time used on disk I/O in previous step.

Graph cleaning

This step cleans the erroneous k-molecules, based on the assumption that the erroneous k-mers have lower frequency compared with the correct ones [19]. Assuming that the errors are random, we identify the k-molecules with low frequency as erroneous k-molecules, and delete them from the vertex set. SWAP-Assembler also removes all edges with low frequency in the 1-step bi-directed graph, and the k-molecules without any attached edges. The frequency threshold can be set by users, or our method will calculate it automatically based on the average coverage of k-molecules. In our case, we prefer $3 \sim 10\%$ of the average coverage as the threshold depending on the species.

All the operations in this step can be finished in $O(\frac{n}{p})$ parallel computation time, and about $60 \sim 80\%$ of the k-molecules can be removed from our graph.

Graph reduction

In order to recover contigs, all semi-extended edges in MSG need to be merged into full-extended edges. This task can be defined as edge merging computing activity and denoted as $ACT(E_s \bigvee \mathbf{0}, \sigma)$, where the edge merging operation set σ is,

$$\sigma = \{(e^u_{\beta\alpha}, e^v_{\alpha\gamma}) | e^u_{\beta\alpha} \otimes e^v_{\alpha\gamma} \neq \mathbf{0}, e^u_{\beta\alpha}, e^v_{\alpha\gamma} \in E_s\} \tag{9}$$

in which $e^u_{\beta\alpha}$ indicates an u-step bi-directed edge connecting two vertices β and α. All semi-extended edges of E_s will be merged into full-extended edges finally.

In order to compute edge merging operations in σ_i using our SWAP computational framework, two user-defined functions in Table 2 are described as Algorithm 3 and Algorithm 4 in Appendix 2. For each process, the edge merging step has a computing complexity of $O(\frac{n}{p})$, communication volume of $O(\frac{n\log(\log(g))}{p})$, and communication round of $O(\log(\log(g)))$. The proposed methods has much less computation round of $O(loglog(g))$ than YAGA with $O(log(n)^2)$ [18, 19]. The detailed complexity analysis is provided in Appendix 4.

Contig extension

In order to extend the length of contigs while maintaining as less errors as possible, three types of special edges and two type of special vertices are processed in our method.

The first type of special edge is tip edge, which is connected with an terminal vertex and has a length less than k, where k is the k-mer size. These tip edges are deleted from the graph. The second type is self-loop edge, whose beginning vertex and terminal vertex are same. If this vertex has another edge which can be merged with this self-loop edge, they will be merged, otherwise it will be removed. The last type is multiple edge, whose two vertices are directly connected by two different edges. In this case the edge with lower coverage will be removed.

In addition, processing two special vertices can help further improve the quality of contigs. The first is cross vertex shown in Figure 8-(e), which has more than two edges on both sides. For each cross vertex, we sort all its edges according to their coverage. When the coverage difference between the two edges is less than 20%, then these two edges are merged as long as they can be merged regardless of other edges. The second vertex is virtual cross vertex shown in Figure 8-(f). We treat edge e_0 with its two end vertices as one virtual vertex A^*, and A^* has more than two edges on both sides. All its edges are ranked according to their coverage. When the coverage difference between two edges on different nodes is less than 20%, these two edges will be merged with the edge e_0 regardless of other edges. By processing these two special vertices using the heuristic method, we can partly resolve some of the repeats satisfying our strict conditions at the cost of introducing errors and mismatches into contigs occasionally.

The graph reduction step and contig extension step need to be iterated in a constant number of rounds to extend full-extended edges or stop when no special edges and special vertices can be found. The number of errors and mismatches introduced in contigs can be controlled by the percentage of special edges and vertices processed in contig extension stage. In our method, we process all edges and vertices aiming at obtaining longer contigs. The computing complexity for contig extension step will be bounded by $O(\frac{n}{p})$. As the graph shrinks greatly after graph reduction and contig extension step, all the remaining edges are treated as contigs.

The complexity of SWAP-Assembler is dominated by graph reduction step, which is bounded by $O(\frac{n}{p})$ parallel computing time, $O(\frac{n\cdot\log(\log(g))}{p})$ communication volume, and $O(\log(\log(n)))$ communication round. According to complexity analysis results of graph reduction step in equation (4.3), when the number of processors is less than the length of longest path d_{max}, the speedup of SWAP-Assembler can be calculated as follows,

$$Speedup = \frac{n}{RunTime} = \frac{p}{bL\log(3cq\cdot\log(g)) + rS + 1} \tag{10}$$

Equation (10) indicates that, for a given genome with fixed length g, the speedup is proportional to the number of processors; while for a given number of processors p, the speedup is inversely proportional to logarithm of the logarithm of the genome size g. However when the number of processors is larger than the length of longest path d_{max}, the running time will be bounded by the number of communication round, which is presented in equation (4.1) in Appendix 4. In either situation, we can conclude that the scalability or the optimal number of cores will increase with larger genomes.

Results

SWAP-Assembler is a highly scalable and efficient genome assembler using multi-step bi-directed graph (MSG). In this section, we perform several computational experiments to evaluate the scalability and assembly quality of SWAP-Assembler. In the experiments, TianHe 1A [28] is used as the high performance cluster. 512 computing nodes are allocated for the experiment with 12 cores and 24GB memory on each node. By comparing with several state-of-the-art sequential and parallel assemblers, such as Velvet [22], SOAPdenovo [29], Pasha [16], ABySS [14] and Ray [15], we evaluate the scalability, quality of contigs in terms of N50, error rate and coverage for SWAP-Assembler.

Experimental data

Five datasets in Table 3 are selected for the experiments. *S. aureus*, *R. sphaeroides* and *human chromosome 14* (Hg14) datasets are taken from GAGE project [30], Fish dataset is downloaded from the Assemblathon 2 [31,32], and Yanhuang dataset [33] is provided by BGI [34].

Scalability evaluation

The scalability of our method is first evaluated on a share memory machine with 32 cores and 1T memory. Five other assemblers including Velvet, SOAPdenovo, Pasha, ABySS and Ray, are included for comparison. Only the first three small datasets in Table 3 are used in this test due to the memory limitation. The results are presented in Table 4, and the corresponding figures are plotted in Figure 9. According to Table 4, SWAP-Assembler has the lowest running time on all three datasets for 16 cores and 32 cores, and SOAPdenovo has the lowest time usage on 4 cores and 8 cores. According to Figure 9, SOAPdenovo, Ray and SWAP-Assembler can scale to 32 cores, however Pasha and ABySS can only scale to 16 cores. Figure 9 also shows that Ray and SWAP-Assembler can achieve nearly linear speedup and SWAP-Assembler is more efficient than Ray.

The time usage for each step of SWAP-Assembler on the share memory machine is also presented in Table 5 and Figure 10. The input parallelization step is overlapped with graph construction, thus we treat these two steps as one in this experiment. According to Table 5, for all three datasets the most time-consuming step is graph reduction, and the fastest steps are graph cleaning and contig extension. Figure 10 shows that input parallelization & graph construction, graph cleaning and graph reduction can achieve nearly linear speedup when the number of cores increases from 4 to 32 cores, whereas the contig extension step does not benefit as much as other steps.

To further evaluate the scalability of SWAP-Assembler from 64 to 4096 cores on TianHe 1A, we have compared our method with two parallel assemblers, ABySS and Ray, and the results are included in Table 6. According to Table 6, SWAP-Assembler is 119 times and 73 times faster than ABySS and Ray for 1024 cores on the *S. aureus* dataset. On the same dataset, ABySS and Ray cannot gain any speedup beyond 64 and 128 cores, respectively. However SWAP-Assembler scales up to 512 cores. For the *R. sphaeroides* dataset, ABySS, Ray, and SWAP-Assembler can scale up to 128, 256, and 1024 cores, respectively.

For three larger datasets, Table 6 shows that scalability of SWAP-Assembler is also better than the other two methods. On Hg14 dataset, SWAP-Assembler is 280 times faster than ABySS, and 38 times faster than Ray when using 1024 cores. Similar to the results on *R. sphaeroides* dataset, three assemblers still hold their turning point of scalability at 128, 256 and 1024 cores, respectively. Fish and Yanhuang dataset cannot be assembled by ABySS and Ray in 12 hours, so their running times are not recorded in Table 6. For 1024 cores, SWAP-Assembler assembles Fish dataset in 16 minutes, while it takes 26 minutes with 2048 cores to assemble the Yanhuang dataset. The speedup curves of SWAP-Assembler on processing five datasets are shown in Figure 11. It shows that the speedup of assembling two small datasets have a turning point at 512 cores, and linear speedup to 1024 cores is achieved for other three larger datasets. SWAP-Assembler can still benefits from the increasing cores up to 2048 cores on processing Yanhuang dataset.

Memory footprint is a bottleneck for assembling large genomes, and parallel assemblers is a solution for large genome assembly by using more memory on the computational nodes. For our case, Fish and Yanhuang genome assembly needs 1.6T bytes and 1.8T bytes memory, respectively. As in Tianhe 1A each server has 24G memory, Fish genomes cannot be assembled on a cluster with 64 servers. The same reasoning applies to Yanhuang dataset.

SWAP-Assembler has better scalability compared with Ray and ABySS due to two important improvements. Firstly, computational interdependence of edge merging operations on one single unanimous path is resolved by MSG. Secondly, SWAP computational framework can trigger parallel computation of all operations without interference, and the communication latency is hidden by improving the computing throughput. Ray and ABySS cannot merge the k-mers in a single linear chain in parallel, and PASHA can only parallelize the k-mer merging work on different chains, which limits their degree of parallelism.

Assembly quality assessment

This part evaluate the assembly quality of SWAP-Assembler. To be compatible with the comparison results from GAGE, we follow the error correction method of GAGE. ALLPATH-LG [35] and Quake [36] are used to correct errors for *S. aureus* and *R. sphaeriodes* datasets. The corrected reads are used as the input to ABySS, Ray and SWAP-Assembler. In addition, two other sequential genome assemblers, Velvet and SOAPdenovo, are selected in this experiment for comparison, and a machine with 1TB memory is used. The k-mer size for all assemblers varies between 19 and 31, and best assembly results from the experiments of different k-mer sizes for each assembler are reported in Table 7 and Table 8.

Table 7 presents the results of four metrics for evaluation: the number of contigs, N50, number of erroneous contigs and error-corrected N50. Error-corrected N50 is used to exclude the misleading assessment of larger N50 by correcting erroneously concatenated contigs. Each erroneous contig is broken at every misjoin and at every indel longer than 5 bases. From Table 7, SWAP-Assembler generates 3 and 7 error contigs on *S. aureus* and *R. sphaeriodes* datasets, respectively, which are the smallest compared with other assemblers. N50 size and error-corrected N50 size for SWAP-Assembler are also longer than two other parallel assemblers, Ray and ABySS. SOAPdenovo has minimal number of contigs and largest N50 size for both datasets.

Table 8 summarizes the statistics of contigs generated for these two datasets. Three metrics in [30] are used to evaluate the quality of the contigs. In this table, “assembly size” close to its genome size is better. Larger “chaff size” can be indicative of the assembler being unable to integrate short repeat structures into larger contigs or not properly correcting erroneous bases. Coverage can be measured by the percentage of reference bases aligned to any assembled contig, which is “100%-Unaligned ref bases” [30]. SWAP-Assembler and Ray both have the smallest chaff size of 0.13% on *R. sphaeroides* dataset, and they show very close coverage and assembly sizes. In terms of *S. aureus* dataset, Ray has a lower chaff size of 0.10% compared with SWAP-Assembler, however, SWAP-Assembler generates better assembly size of 99.3% and larger coverage of 99.2%.

We also analyzed the contig statistics for three larger datasets and the results are presented in Table 9. Because these datasets do not have a standard reference set and the original script provided by GAGE requires a reference set, we wrote a script to analyze the assembly results using the number of contigs, N50 size, max length of contigs and bases in the contigs for evaluation. The original data of three datasets are directly processed by five assemblers with a fixed k-mer size of 31. According to Table 9, the N50 size of contigs generated by SWAP-Assembler is longest for all three datasets. For Fish and Yanhuang datasets, SWAP-Assembler also performs best in the number of contigs and max length of contigs. However for SWAP-Assembler on Hg14 dataset, whose reads are extracted from the human dataset by mapping the human chromosome 14, the number of contigs, max length of contigs and bases in contigs have a rank of second, third, and second, respectively. SWAP-Assembler has a best N50 size for all datasets. This is because it has efficient graph cleaning and contig extension steps, which can handle sequencing errors efficiently. Four other assemblers, without the help from external tools on error correction, are affected by the quality of input reads on larger datasets.

In conclusion, SWAP-Assembler is a highly scalable and efficient genome assembler. The evaluation shows that our assembler can scales up to 2048 cores, which is much better than other parallel assemblers, and the quality of contigs generated by SWAP-Assembler is the best in terms of error rate for several small datasets and N50 size for two larger data sets.

Conclusion

In this paper, SWAP-Assembler, a fast and efficient genome assembler scaling up to thousands of cores, is presented. In SWAP-Assembler, two fundamental improvements are crucial for its scalability. Firstly, MSG is presented as a comprehensive mathematical abstraction for genome assembly. With MSG the computational interdependence is resolved. Secondly, SWAP computational framework triggers the parallel computation of all operations without interference. Two additional steps are included to improve the quality of contigs. One is graph cleaning, which adopts the traditional methods of removing k-molecules and edges with low frequency, and the other is contig extension, which resolves special edges and some cross nodes with a heuristic method. Results show that SWAP-Assembler can scale up to 2048 cores on Yanhuang dataset using only 26 minutes, which is the best compared to other parallel assemblers, such as ABySS and Ray. Conitg evaluation results confirm that SWAP-Assembler can generate good results on contigs N50 size and retain low error rate. When processing massive datasets without using external error correction tools, SWAP-Assembler is immune from low data quality and generated longest N50 contig size.

For large genome and metagenome data of Tara bytes, for example the human gut microbial community sequencing data, highly scalable and efficient assemblers will be essential for data analysis. Our future work will extend our algorithm development for massive matagenomics dataset with additional modules.

The program can be downloaded from https://sourceforge.net/projects/swapassembler.

Acknowledgements

This work is supported by National Science Foundation of China under grant No. 11204342, the Science Technology and Innovation Committee of Shenzhen Municipality under grant No. JCYJ20120615140912201, and Shenzhen Peacock Plan under grant No. KQCX20130628112914299.

The calculations of this work were performed on TianHe-1A at National Supercomputer Center in Tianjin.

References

1. Schatz MC, Langmead B, Salzberg SL: **Cloud computing and the DNA data race**. Nature biotechnology 2010, **28**(7):691.
2. Stein LD, et al.: **The case for cloud computing in genome informatics**. Genome Biol 2010, **11**(5):207.
3. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, et al.: **A view of cloud computing**. Communications of the ACM 2010, **53**(4):50–58.
4. Dean J, Ghemawat S: **MapReduce: simplified data processing on large clusters**. Communications of the ACM 2008, **51**:107–113.
5. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC: **GPU computing**. Proceedings of the IEEE 2008, **96**(5):879–899.
6. Matsunaga A, Tsugawa M, Fortes J: **Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications**. In eScience, 2008. eScience'08. IEEE Fourth International Conference on, IEEE 2008:222–229.
7. Wall DP, Kudtarkar P, Fusaro VA, Pivovarov R, Patil P, Tonellato PJ: **Cloud computing for comparative genomics**. BMC bioinformatics 2010, **11**:259.
8. Liu CM, Wong T, Wu E, Luo R, Yiu SM, Li Y, Wang B, Yu C, Chu X, Zhao K, et al.: **SOAP3: ultra-fast GPU-based parallel alignment tool for short reads**. Bioinformatics 2012, **28**(6):878–879.
9. Schatz MC: **CloudBurst: highly sensitive read mapping with MapReduce**. Bioinformatics 2009, **25**(11):1363–1369.
10. Langmead B, Schatz MC, Lin J, Pop M, Salzberg SL: **Searching for SNPs with cloud computing**. Genome Biol 2009, **10**(11):R134.
11. Langmead B, Hansen KD, Leek JT, et al.: **Cloud-scale RNA-sequencing differential expression analysis with Myrna**. Genome Biol 2010, **11**(8):R83.
12. McPherson JD: **Next-generation gap**. Nature Methods 2009, **6**:S2–S5.
13. Shendure J, Ji H: **Next-generation DNA sequencing**. Nature biotechnology 2008, **26**(10):1135–1145.
14. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, Birol: **ABySS: a parallel assembler for short read sequence data**. Genome research 2009, **19**(6):1117–1123.
15. Boisvert S, Laviolette F, Corbeil J: **Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies**. Journal of Computational Biology 2010, **17**(11):1519–1533.
16. Liu Y, Schmidt B, Maskell DL: **Parallelized short read assembly of large genomes using de Bruijn graphs**. BMC bioinformatics 2011, **12**:354.
17. Jackson BG, Aluru S: **Parallel construction of bidirected string graphs for genome assembly**. In Parallel Processing, 2008. ICPP'08. 37th International Conference on, IEEE 2008:346–353.
18. Jackson BG, Schnable PS, Aluru S: **Parallel short sequence assembly of transcriptomes**. BMC bioinformatics 2009, **10**(Suppl 1):S14.
19. Jackson BG, Regennitter M, Yang X, Schnable PS, Aluru S: **Parallel de novo assembly of large genomes from high-throughput short reads**. In Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, IEEE 2010:1–10.
20. Pevzner PA, Tang H, Waterman MS: **An Eulerian path approach to DNA fragment assembly**. Proceedings of the National Academy of Sciences 2001, **98**(17):9748–9753.

21. Chaisson MJ, Pevzner PA: **Short read fragment assembly of bacterial genomes**. Genome research 2008, **18**(2):324–330.

22. Zerbino DR, Birney E: **Velvet: algorithms for de novo short read assembly using de Bruijn graphs**. Genome research 2008, **18**(5):821–829.

23. Dehne F, Song SW: **Randomized parallel list ranking for distributed memory multiprocesors**. In Concurrency and Parallelism, Programming, Networking, and Security, Springer 1996:1–10.

24. Meng J, Yuan J, Cheng J, Wei Y, Feng S: **Small World Asynchronous Parallel Model for Genome Assembly**. In Network and Parallel Computing, Springer 2012:145–155.

25. Pop M, Salzberg SL, Shumway M: **Genome sequence assembly: Algorithms and issues**. Computer 2002, **35**(7):47–54.

26. Kapun E, Tsarev F: **De Bruijn Superwalk with Multiplicities Problem is NP-hard**. BMC bioinformatics 2013, **14**(Suppl 5):S7.

27. Andrew S: Computer Networks. The Netherlands: Prentice Hall 2003.

28. Yang XJ, Liao XK, Lu K, Hu QF, Song JQ, Su JS: **The TianHe-1A supercomputer: its hardware and software**. Journal of Computer Science and Technology 2011, **26**(3):344–351.

29. Li R, Zhu H, Ruan J, Qian W, Fang X, Shi Z, Li Y, Li S, Shan G, Kristiansen K, et al.: **De novo assembly of human genomes with massively parallel short read sequencing**. Genome research 2010, **20**(2):265–272.

30. Salzberg SL, Phillippy AM, Zimin A, Puiu D, Magoc T, Koren S, Treangen TJ, Schatz MC, Delcher AL, Roberts M, et al.: **GAGE: A critical evaluation of genome assemblies and assembly algorithms**. Genome Research 2012, **22**(3):557–567.

31. Bradnam KR, Fass JN, Alexandrov A, Baranay P, Bechner M, Birol I, Boisvert10 S, Chapman JA, Chapuis G, Chikhi R, et al.: **Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species**. arXiv preprint arXiv:1301.5406 2013.

32. **Assemblathon 2** [http://assemblathon.org/assemblathon2].

33. Luo R, Liu B, Xie Y, Li Z, Huang W, Yuan J, He G, Chen Y, Pan Q, Liu Y, et al.: **SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler**. GigaScience 2012, **1**:18.

34. **Yan Huang - The first Asian diploid genome** [http://yh.genomics.org.cn].

35. Gnerre S, MacCallum I, Przybylski D, Ribeiro FJ, Burton JN, Walker BJ, Sharpe T, Hall G, Shea TP, Sykes S, et al.: **High-quality draft assemblies of mammalian genomes from massively parallel sequence data**. Proceedings of the National Academy of Sciences 2011, **108**(4):1513–1518.

36. Kelley DR, Schatz MC, Salzberg SL, et al.: **Quake: quality-aware detection and correction of sequencing errors**. Genome Biol 2010, **11**(11):R116.

Figures

Figure 1 - Parallel strategy comparison on k-mers/edges merging for different assemblers.

Two linked paths with 3 nodes and 5 nodes are given as an example, merging a linked path with 3 nodes needs 2 operations/rounds, and merging a path with 5 nodes needs 4 operations/rounds. To assemble these two paths, sequential assemblers need 6 operations, and parallel assemblers need 4 rounds. For SWAP-Assembler different processes can merge several edges on the same path in parallel using the SWAP computational framework, and merging of these 2 paths can be finished in 2 rounds. For a given sequencing data, if we treat the sequencing coverage as an constant number, the upper bound of the three assembly strategies on merging k-mers/edges are bounded by $O(g)$, $O(log(g))$, and $O(log(log(g)))$ respectively, here g denotes the genome size and the longest path for a genome of length g will be bounded by $O(log(g))$. The upper bound of edge merging operations in SWAP-Assembler and expected length of longest path are proved in Appendix 3.

Reference of the k-mers merging strategy for these assemblers can be found in their papers or codes. For velvet 1.1.04, the k-mers merging method can be found in its code "./src/concatenatedGraph.c"; for SOAPdenovo-V1.05, its method is in the code "./src/31mer/contig.c"; for ABySS 1.3.5, the method can be found in "./Parallel/NetworkSequenceCollection.cpp"; for YAGA, the method has been described in the last paragraph in the methods section [18]; for Pasha, its method is presented in the last paragraph at the graph simplification subsection [16].

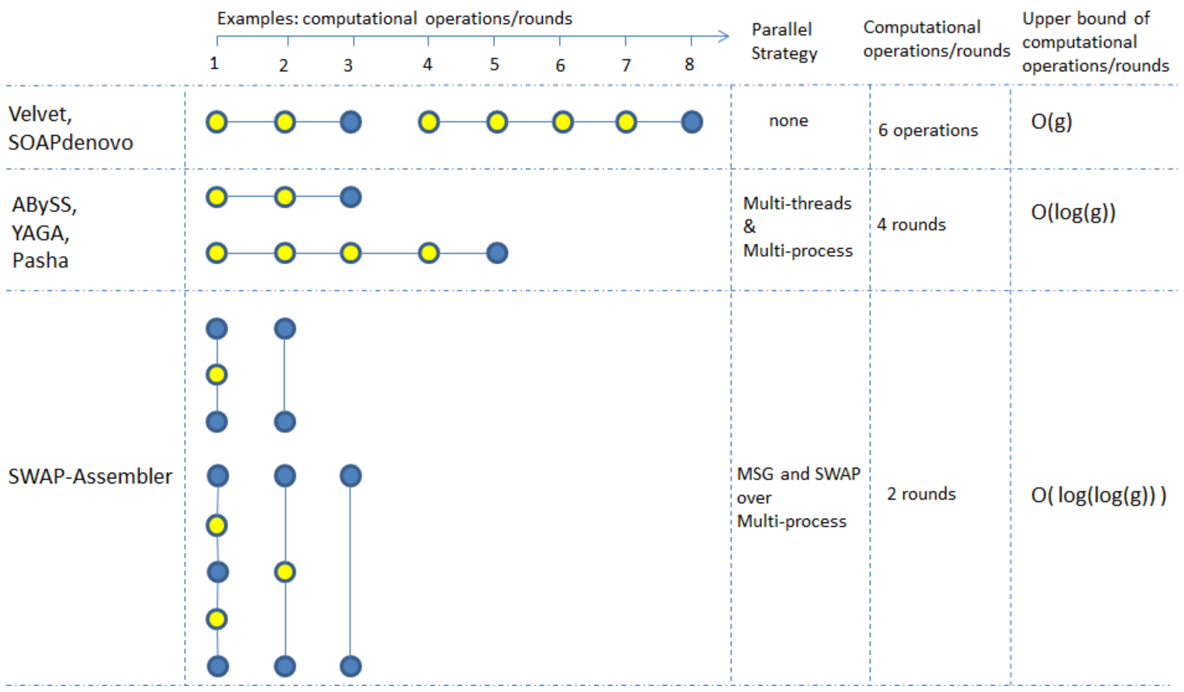

Examples: computational operations/rounds
1
2
3
4
5
6
7
8
Parallel Strategy
Computational operations/rounds
Upper bound of computational operations/rounds
Velvet, SOAPdenovo
none
6 operations
O(g)
ABySS, YAGA, Pasha
Multi-threads & Multi-process
4 rounds
O(log(g))
SWAP-Assembler
MSG and SWAP over Multi-process
2 rounds
O(log(log(g)))

Figure 2 - The whole process of genome assembly and the standard genome assembly (SGA) problem.

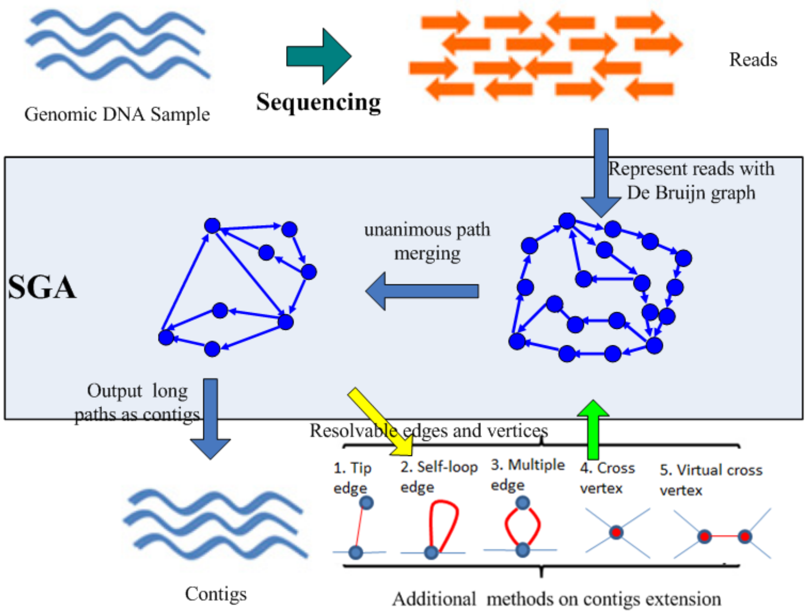


Figure 3 - Each k-molecule consists of one positive k-mer and one negative k-mer.

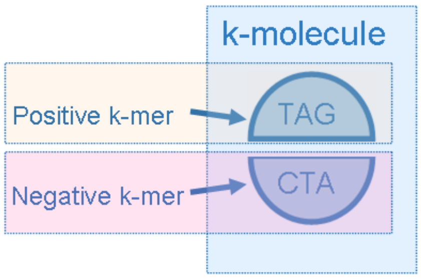


Figure 4 - The illustration of four possible connections.

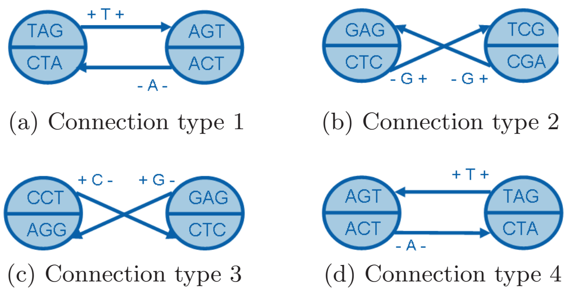


Figure 5 - An example of 1-step bi-directed graph. Here semi-extended k-molecules are colored with yellow, and semi-extended edges are plotted with dashed line.

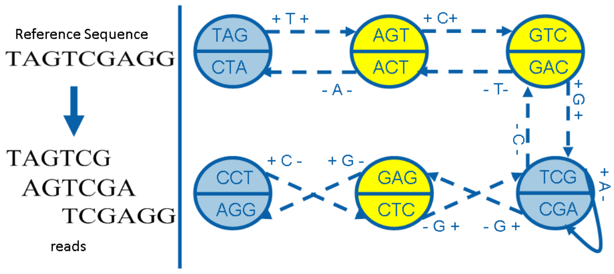


Figure 6 - An example for edge merging operation.

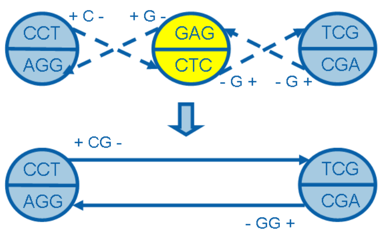


Figure 7 - Illustration of the distribution of an activity $ACT(A, \sigma)$ on a cluster using SWAP computational framework.

Operation set σ is distributed over p processors, here $\sigma = \bigcup_{i=0}^{p-1} \sigma_i$, $\bigcap_{i=0}^{p-1} \sigma_i = 0$ and $ACT(A, \sigma) = \bigcup_{i=0}^{p-1} ACT_i(A_i, \sigma_i)$.

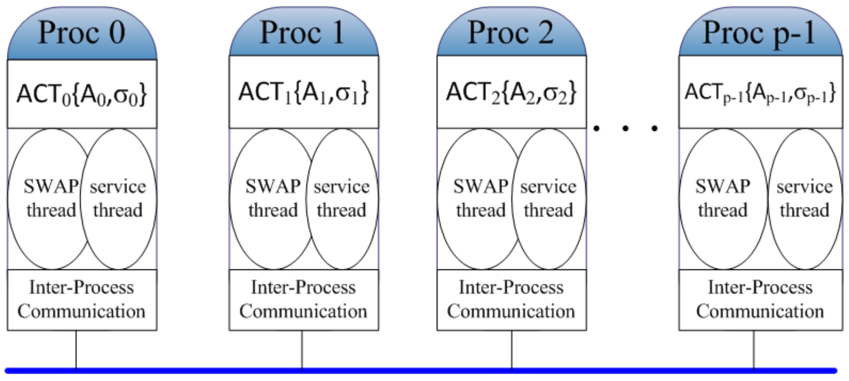
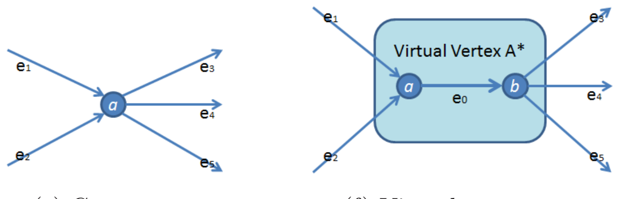


Figure 8 - Two type of special vertex defined in contig extension step.



(e) Cross vertex (f) Virtual cross vertex

Figure 9 - Time usage comparison on a share memory machine for three small datasets. (Time unit: seconds in logarithmic scale)

In this test, the length of k-mer for all assemblers is set to be 31 and the k-mers cutoff threshold is set to 3. For the three datasets, the sequencing data filtered by ALLPATH-LG is their input data. The time usage is recorded until the contig is generated. The horizontal axis is marked with the name of assemblers and the number of cores used.

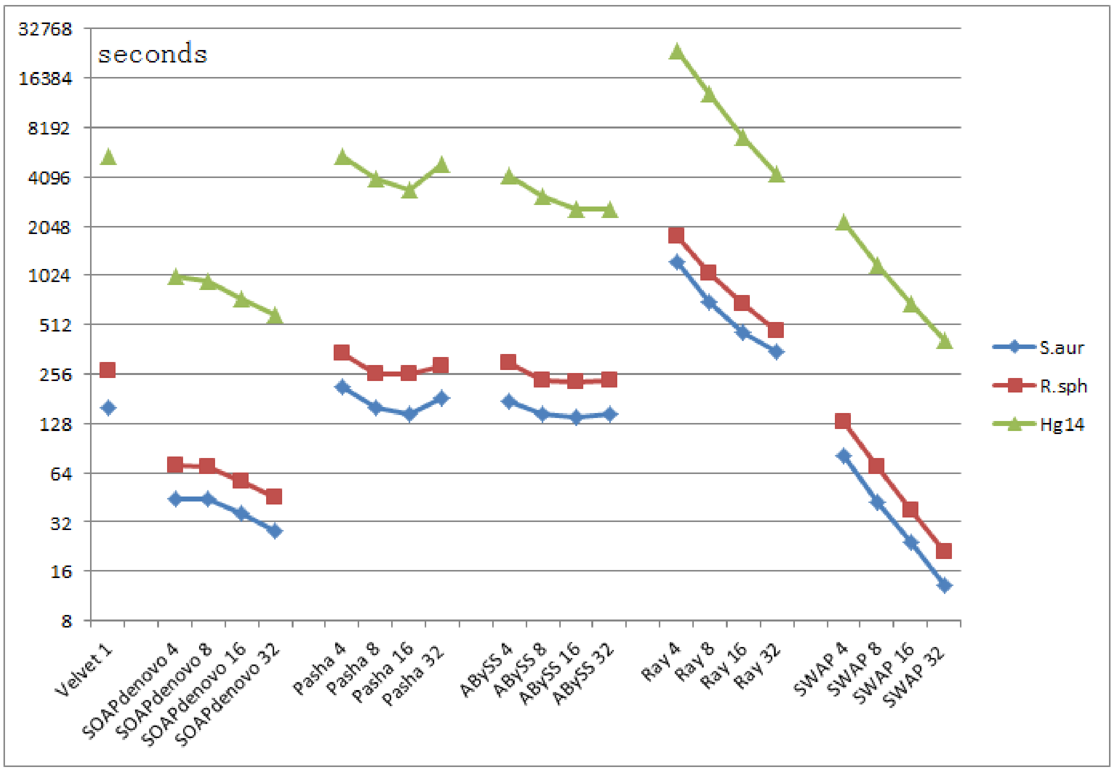


Figure 10 - Time usage details of SWAP-Assembler's five steps on processing three small datasets using a share memory machine with 32 cores. (Time unit: seconds in logarithmic scale)

In this test, the length of k-mer for SWAP-Assembler is set to be 31 and the k-mers cutoff threshold is set to 3. For the three datasets, the sequencing data filtered by ALLPATH-LG is their input data. The horizontal axis is marked with the name of datasets and the number of cores used.

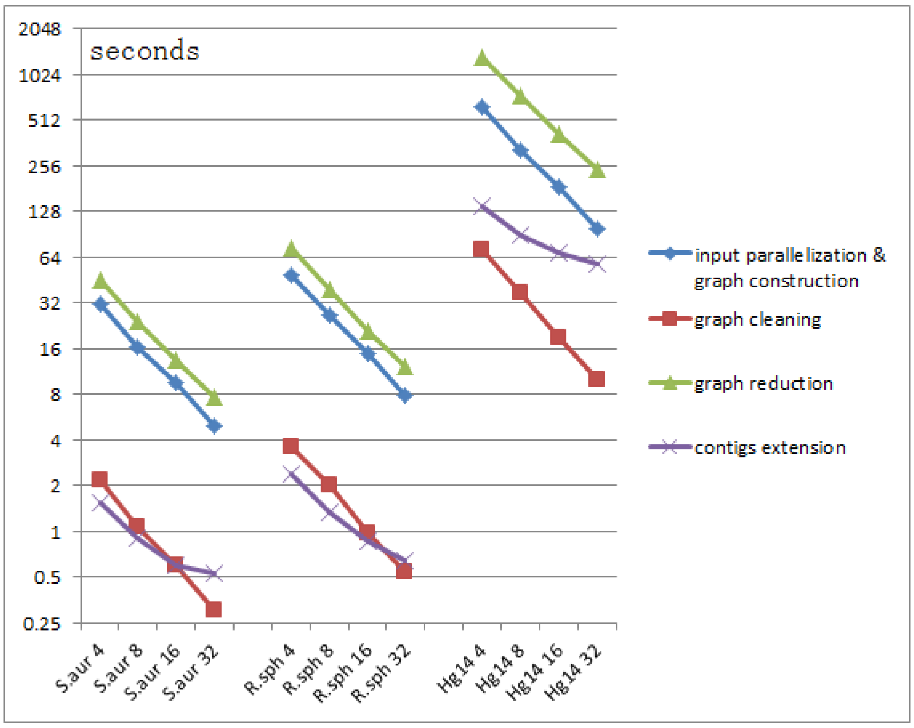
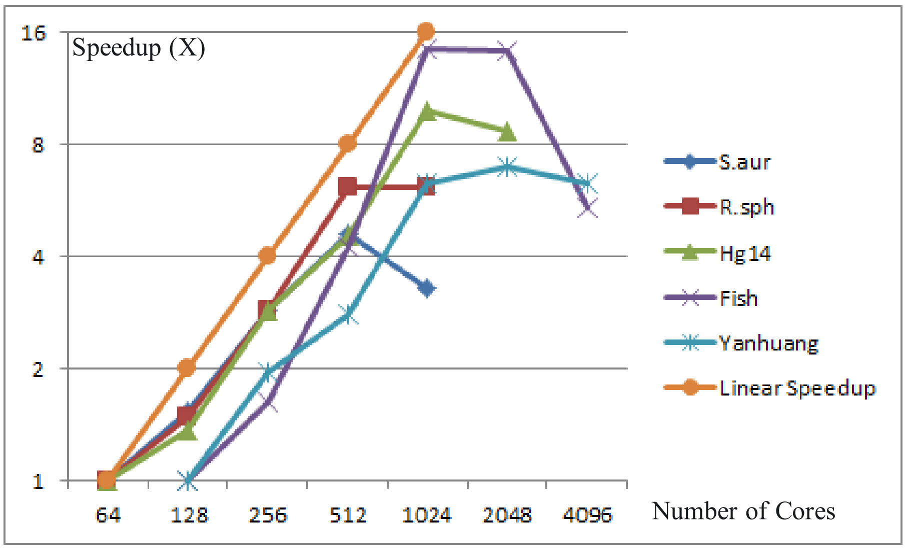


Figure 11 - Linear speedup of SWAP-Assembler on processing five datasets.



Tables

Table 1 - List of Notations.

Definition	Notation	Example
set of nucleotides	$\mathbb{N}$	$\mathbb{N} = \{A, T, C, G\}$
reference sequence	w	$w =$ “$TAGTCGAGG$”
read set	S	$S = \{$ “$TAGTCG$”, “$AGTCGA$”, “$TCGAGG$” $\}$
k-mer	α or α'	$\alpha =$ “TAG”$, \alpha' =$ “CTA”
positive k-mer	α^+	$\alpha^+ =$ “TAG”
negative k-mer	α^-	$\alpha^- =$ “CTA”
representative k-mer	α^+	$\alpha^+ =$ “TAG”
k-molecule	$\hat{\alpha}=\alpha^+ = \{\alpha^+, \alpha^-\}$	$\hat{\alpha} = \{$“TAG”, “CTA”$\}$
set of k-mers	$\mathbb{Z}(s, k)$	$\mathbb{Z}($“$TAGTCG$”$, 3) =\{$“TAG”,“AGT”,“GTC”,“TCG”$\}$
set of k-molecules	$\mathbb{S}(s, k)$	$\mathbb{Z}($“$TAGTCG$”$, 3) = \{\{$“TAG”, “CTA”$\}, \{$“AGT”, “ACT”$\}, \{$“GTC”,”GAC”$\}, \{$“TCG”, “CGA”$\}\}$

Table 2 - Description of message functions, internal functions, and user-defined functions used in Algorithm 1 and 2.

Class	Function Name	Function Description
Message Functions	Msg_Lock(a, p)	Lock a in process p
	Msg_Unlock(a, p)	Unlock a in process p
	Msg_Read(a, p)	Fetch associated values of a
	Msg_Write(a, $newa$, p)	Update associated values of a with $newa$
	Msg_Locksuccess(a, $R(a, b)$, p)	Send Locksuccess Message back to $R(a, b)$
	Msg_Lockfailed(a, $R(a, b)$, p)	Send Lockfailed Message back to $R(a, b)$
	Msg_ReadBack(a, $R(a, b)$, p)	Send associated value of a back to $R(a, b)$
	Msg_End()	Command to stop the service thread
Internal Functions	proc(a)	Get process ID of a
	trylock(a)	Lock a
	unlock(a)	Unlock a
User-defined Functions	GetSmallWorld($R(a, b)$)	Get small world $[a, b]$ from operation $R(a, b)$
	Operation(a,b)	Compute the operation $R(a, b)$

Table 3 - Details about the five short read datasets.

-	*S. aureus*	*R. sphaeroides*	Hg14	Fish	YanHuang
fastq data size (Gbytes)	0.684	0.906	14.2	425	495
read length (bp)	37, 101	101	101	101	80-120
no. of reads (million)	4.8	4.1	62	1910	1859
coverage	90X	90X	70X	192X	57X
reference size (Mbp)	2.90	4.60	88.6	1000	3000

Table 4 -Time usage results of three small datasets on a share memory machine with 32 cores. (Time unit: seconds)

assembler	cores	*S. aureus*	*R. sphaeroides*	Hg14
Velvet	1	159	265	5432
SOAPdenovo	4	44	71	1004
	8	44	69	933
	16	36	57	742
	32	38	45	582
Pasha	4	215	342	5494
	8	159	255	3938
	16	147	255	3436
	32	183	289	4852
ABySS	4	174	302	4138
	8	146	234	3079
	16	139	226	2588
	32	147	235	2596
Ray	4	1247	1778	24145
	8	707	1050	13116
	16	454	688	7222
	32	351	467	4235
SWAP-Assembler	4	81	129	2167
	8	42	69	1187
	16	24	38	685
	32	13	21	408

Table 5 - Time usage details of SWAP-Assembler's five steps on processing three small datasets using a share memory machine with 32 cores. (Time unit: seconds)

	steps	4 cores	8 cores	16 cores	32 cores
S. aureus	input parallelization & graph construction	31.29	16.32	9.55	4.91
	graph cleaning	2.14	1.07	0.6	0.3
	graph reduction	45.75	24.2	13.23	7.67
	contig extension	1.54	0.89	0.6	0.52
R. sphaeroides	input parallelization & graph construction	49.19	26.42	14.88	7.79
	graph cleaning	3.63	1.99	0.97	0.54
	graph reduction	73.78	39.31	20.83	12.19
	contig extension	2.37	1.33	0.85	0.64
Hg14	input parallelization & graph construction	628	327	184	98
	graph cleaning	71	37	19	10
	graph reduction	1328	734	413	242
	contig extension	140	89	69	58

Table 6 - Scalability evaluation on parallel assemblers. (Time unit: seconds)

This table records the time usage on assembling all five datasets with different number of cores, and the length of k-mer is set to be 23. For ABySS and Ray, the time is recorded until contigs are generated.

dataset	software	64	128	256	512	1024	2048	4096
S. aureus (2.87Mb)	ABySS	248	269	334	554	831	-[1]	-
	Ray	244	198	202	266	510	-	-
	SWAP-Assembler	23	15	8	5	7	13	-
R. sphaeroides (4.60Mb)	ABySS	492	454	522	718	1312	-	-
	Ray	287	183	181	190	285	-	-
	SWAP-Assembler	43	29	15	7	7	7	-
Hg14 (88.29Mb)	ABySS	6472	5299	6935	9045	16530	-	-
	Ray	2926	1746	1288	1517	2266	-	-
	SWAP-Assembler	585	428	203	128	59	67	-
Fish (1Gb)	ABySS	* [2]	+ [3]	+	+	+	-	-
	Ray	*	+	+	+	+	-	-
	SWAP-Assembler	+	13941	8622	3263	962	971	2582
Yanhuang (3Gb)	ABySS	*	+	+	+	+	-	-
	Ray	*	+	+	+	+	-	-
	SWAP-Assembler	*	11243	5761	4021	1783	1608	1778

[1] − denotes assembler with this parameter has not been tested.
[2] $*$ denotes assembler with this parameter has run out of memory.
[3] + denotes assembler with this parameter has run out of time, the time limit is 12 hours.

Table 7 - Assembly results of S. aureus and R. sphaeriodes datasets.

software	*S. aureus* Contigs				*R. sphaeroides* Contigs			
	Num	N50(kb)	Errors	N50 corr. (kb)	Num	N50(kb)	Errors	N50 corr.(kb)
Velvet	162	48.4	28	41.5	583	15.7	35	**14.5**
SOAPdenovo	**107**	**288.2**	48	**62.7**	**204**	**131.7**	414	14.3
ABySS	302	29.2	14	24.8	1915	5.9	55	4.2
Ray	221	36.6	15	35.6	752	11.5	17	11.2
SWAP-Assembler	183	51.1	**3**	37.8	529	16.2	**7**	12.3

Table 8 - Contig statistics on the assembly results of S. aureus, R. sphaeroides datasets. (Unit: Percentage (%)

dataset	software	Assembly size	Chaff size	Unalign ref bases	Unalign asm bases	Duplicate ref bases	Compress ref bases
S. aureus (2.9 Mb)	Velvet	99.2	0.45	0.79	0.03	0.10	1.28
	SOAPdenovo	101.3	0.35	**0.38**	**0.01**	1.44	1.41
	ABySS	127.0	66.00	1.37	<**0.01**	23.30	**0.99**
	Ray	98.4	**0.10**	0.88	0.04	**0.08**	1.26
	SWAP-Assembler	**99.3**	0.28	0.80	0.02	1.29	1.45
R. sphaeroides (4.6 Mb)	Velvet	97.8	0.54	1.60	0.01	0.29	0. 92
	SOAPdenovo	**99.9**	0.45	**0.88**	0.02	1.07	0.51
	ABySS	108.0	1.65	3.01	0.15	10.04	**0.04**
	Ray	99.0	**0.13**	1.03	<**0.01**	**0.27**	0.73
	SWAP-Assembler	99.1	**0.13**	1.08	0.11	0.83	0.75

Table 9 - Contig statistics of Hg14, Fish and Yanhuang datasets.

dataset	software	contigs			
		Num	N50 (bp)	Max (bp)	BasesInContig (Mbp)
Hg 14 (88.3 Mb)	Velvet	90,784	1688	25,729	**83.3**
	SOAPdenovo	200,153	836	21,144	96.4
	ABySS	190,693	1914	**26,697**	107.4
	Ray	**76,950**	964	14,399	68.4
	SWAP-Assembler	88,609	**2036**	21,246	96.4
Fish (1Gb)	Velvet	-	-	-	-
	SOAPdenovo	3291290	378	7181	1,134.4
	ABySS	-	-	-	-
	Ray	-	-	-	-
	SWAP-Assembler	**2881443**	**1309**	**35,962**	**1,097.9**
Yanhuang (3Gb)	Velvet	-	-	-	-
	SOAPdenovo	8,584,515	841	23,782	**3396.2**
	ABySS	9,218,967	1059	24,428	3691.8
	Ray	3,755,103	266	6,765	1620.1
	SWAP-Assembler	**2,379,151**	**1368**	**31,152**	2434.3

Appendix 1 — Property proof of MSG

Property 1. If the set of full-extended edges in the MSG defined in equation (7) is denoted as E_S^*, then the set of labels on all edges in E_S^* can be written as,

$$L_S^* = \{c_{\alpha\beta}^x | e_{\alpha\beta}^x = (\alpha, \beta, d_\alpha, d_\beta, c_{\alpha\beta}^x), e_{\alpha\beta}^x \in E_S^*\}$$

and we have $L_S^* = C$, C is the set of contigs.

Proof. (1). $\forall a \in C$, a is a contig. Each contig maps to an unanimous path P_a in 1-step bi-directed graph $G_k^1(S) = \{V_S, E_S^1\}$. Let $P_a = t_1 t_2 \ldots t_m$, where $t_i \in V_S$, t_1, t_m are full-extended k-molecules, and the ones in between are semi-extended k-molecules, then we have $e_{t_1t_2}^1 \oplus e_{t_2t_3}^1 \oplus e_{t_3t_4}^1 \oplus \ldots \oplus e_{t_{m-1}t_m}^1 = e_{t_1t_m}^{m-1}$. As t_1, t_m are full-extended k-molecules, then $e_{t_1t_m}^{m-1}$ is full-extended edge, $e_{t_1t_m}^{m-1} \in E_S^*$. The contig a can be recovered by concatenate labels of all 1-step bi-directed edges along the path P_a, and that's $c_{t_1t_m}^{m-1}$, so $a = c_{t_1t_m}^{m-1} \in L_S^*$.

(2). $\forall e_{t_1t_m}^{m-1} = \{t_1, t_m, d_{t_1}, d_{t_m}, c_{t_1t_m}^{m-1}\} \in E_S^*$, here $c_{t_1t_m}^{m-1} \in L_S^*$. This full-extended edge $e_{t_1t_m}^{m-1}$ corresponds to one path P_a in the original 1-step bi-directed graph. With no loss of generality, let $P_a = t_1 t_2 \ldots t_m$, and $t_2, t_3, \ldots t_{m-1}$ are semi-extended k-molecules, t_1, t_m are full-extended k-molecules. Clearly this path P_a is an unanimous path, which will corresponds to a contig. In SGA problem, we can recover this contig by concatenating the labels along the path, and this will be $c_{t_1t_m}^{m-1}$, finally we get $c_{t_1t_m}^{m-1} \in C$.

To combine the conclusions of (1) and (2), we have $L_S^* = C$.

Property 2. Edge merging operation $\otimes$ over the multi-step bi-directed edge set $E_S \bigvee \mathbf{0}$ is associative, and $Q(E_S \bigvee \mathbf{0}, \otimes)$ is a semigroup.

Proof. For any three multi-step bi-directed edge e_{ab}^x, e_{cd}^y and e_{ef}^z in $E_S \bigvee \mathbf{0}$,

Case 1. if $e_{ab}^x \otimes e_{cd}^y \otimes e_{ef}^z \neq \mathbf{0}$, then $(e_{ab}^x \otimes e_{cd}^y) \otimes e_{ef}^z = e_{ab}^x \otimes (e_{cd}^y \otimes e_{ef}^z) = e_{af}^{x+y+z}$.

Case 2. if $e_{ab}^x \otimes e_{cd}^y \otimes e_{ef}^z = \mathbf{0}$, then $(e_{ab}^x \otimes e_{cd}^y) \otimes e_{ef}^z = e_{ab}^x \otimes (e_{cd}^y \otimes e_{ef}^z) = \mathbf{0}$.

So edge merging operation $\otimes$ is associative, and $Q(E_S \bigvee \mathbf{0}, \otimes)$ is a semigroup.

Appendix 2 — Algorithms for SWAP-Assembler

Algorithm 1: Pseudocode of SWAP thread

```
Input: The activity $ACT_i(A_i, \sigma_i)$.
Output: $A_i^*$
begin
    while $\sigma_i \neq \Phi$ do
        for $R(a,b)$ in $\sigma_i$ do
            if trylock($a$) $\neq$ Locksuccess or trylock($b$) $\neq$ Locksuccess then
                unlock($a$);
                unlock($b$);
                continue;
            end
            [a,b] $\leftarrow$ GetSmallWorld($(a,b)$);
            $Replya \leftarrow Msg\_Lock(a, proc(a))$;
            $Replyb \leftarrow Msg\_Lock(b, proc(b))$;
            if $Replya \neq Locksuccess$ or $Replyb \neq Locksucess$ then
                if $Replya = Locksuccess$ then
                    $Msg\_Unlock(a, proc(a))$ ;
                end
                if $Replyb = Locksuccess$ then
                    $Msg\_Unlock(b, proc(b))$ ;
                end
                unlock($a$);
                unlock($b$);
                continue;
            end
            Recva $\leftarrow Msg\_Read(a, proc(a))$;
            Recvb $\leftarrow Msg\_Read(b, proc(b)$;
            $(Recva, Recvb) \leftarrow Operation(Recva, Recvb)$;
            $Msg\_Write(a, Recva, proc(a))$;
            $Msg\_Write(b, Recvb, proc(b))$;
            $Msg\_Unlock(a, proc(a))$;
            $Msg\_Unlock(b, proc(b))$;
            $\sigma_i \leftarrow \sigma_i$ - $R(a,b)$;
        end
        Wait for a random backoff time;
    end
end
```

Algorithm 2: Pseudocode of Service thread

```
Input: Subset $A_i$
begin
    while $true$ do
        Receive a message $Msg$ to $a$ from Operation $R(a,b)$ at $ProcID$;
        if $Msg = Msg\_Lock$ then
            if trylock $(a) = success$ then
                $Msg\_Locksuccess(a, R(a,b), ProcID)$;
                continue;
            end
            $Msg\_Lockfailed(a, R(a,b), ProcID)$;
            continue;
        end
        if $Msg = Msg\_Unlock$ then
            unlock($a$) ;
            continue;
        end
        if $Msg = Msg\_Read$ then
            $Msg\_ReadBack(a, R(a,b), ProcID)$ ;
            continue;
        end
        if $Msg = Msg\_Write$ then
            $x \leftarrow y$;
            continue;
        end
        if $Msg = Msg\_End^*$ then
            break;
        end
    end
end
* When all SWAP threads finish, process 0 will broadcast an End message to stop all Service threads.
```

Algorithm 3: Pseudocode of GetSmall-World(R(a,b))

```
Input: $R(e^u_{\beta\alpha}, e^v_{\alpha\gamma})$
Output: small world $[e^u_{\beta\alpha}, e^v_{\alpha\gamma}]$
begin
    $[e^u_{\beta\alpha}, e^v_{\alpha\gamma}] \leftarrow \{R(e^u_{\beta\alpha}, e^v_{\alpha\gamma})^*, e^u_{\beta\alpha}, e^v_{\alpha\gamma}\}$ ;
    return $[e^u_{\beta\alpha}, e^v_{\alpha\gamma}]$;
end
```

* $R(e^u_{\beta\alpha}, e^v_{\alpha\gamma})$ is the operation of two edges.

Algorithm 4: Pseudocode of Operation(a, b)

```
Input: $e^u_{\beta\alpha}, e^v_{\alpha\gamma}$
begin
    $e^{u+v}_{\beta\gamma} \leftarrow e^u_{\beta\alpha} \otimes e^v_{\alpha\gamma}$;
    $e^{u+v}_{\gamma'\beta'} \leftarrow e^v_{\gamma'\alpha'} \otimes e^u_{\alpha'^*\beta'}$;
    delete $k$-molecule $\alpha$ ;
end
```

* α' is the reverse complement of k-mer α.

Appendix 3 — Complexity Analysis of SWAP computational framework

In this section, the complexity of SWAP computational framework is analyzed. We first assumes the used cluster with following components:

1. A number of computing nodes, with each having a processing unit and local memory.

2. A router that delivers point to point messages between pairs of computing nodes.

The main cluster specifications are listed as follows:

L: An upper bound on the latency, or delay, incurred in communication of 1 unit data by a point to point message,

S: The startup cost, or the latency on putting a message from the memory to the cable,

p: The number of computing nodes.

We make the following assumptions on computing activity $ACT(A, \sigma)$:

1. The number of operations in σ is m. Operations in σ are randomly distributed over p computing components, so $|\sigma_i| \approx m/p$,

2. The number of elements in A is e. Elements in set A_i, which are associated with σ_i, are stored on i-th computing components, and $|A_i| \approx e/p$,

3. Each message incurs a latency of L per bit with a startup time of S seconds. In Algorithm 1 and 2, messages can be classified into two categories, command message and data message. Command message includes *Lock*, *Unlock* and the reply message of *Lock*, and data message includes *Read*, *Write* and the reply message of *Read*. The total data communication volume transmitted by data message is denoted as Γ. As a command message has a fix length, the command data are ignored to simplify our analysis,

4. Each operation in σ needs a computing time of H, Which is the computing complexity of user-defined function $Operation(a, b)$,

5. For a graph $G(\sigma, A)$, the maximum diameter of all connected components of this graph is noted as d_{max}.

According to assumptions 1 and 2, the memory usage for each computing component is $O(\frac{m+e}{p})$. When the number of computing nodes p is far less than m which is true for most applications, the running time is dominated by the computing time and communication time,

$$RunTime = CompTime + CommTime \tag{3.1}$$

Computing each operation in σ_i involves r number of command/data message communication, where r is a number between 4 and 12 in Algorithms 1 and 2. The startup time for each computing nodes is given by,

$$StartTime = \frac{rmS}{p}. \tag{3.2}$$

To simplify our analysis, it is assumed that the total data volume $\digamma$ is evenly transmitted by every computing nodes. The transition time is given by,

$$TranTime = \frac{\digamma}{p} \times L \tag{3.3}$$

Combining equations (3.2) and (3.3), the communication time per computing node can be described as,

$$CommTime = StartTime + TranTime = \frac{rmS + \digamma L}{p} \tag{3.4}$$

According to assumption 1 and 4, each computing nodes has m/p operations. If each operation needs a computing time of H, the computing time is,

$$CompTime = \frac{m}{p} \times \mathrm{H} \tag{3.5}$$

With equations (3.4) and (3.5), the running time per computing node can be written as,

$$RunTime = \tfrac{rmS + \digamma L + m\mathrm{H}}{p} \tag{3.6}$$

When the number of computing nodes is close to or even larger than m, *RunTime* follows different equation rather than equation (3.6). For most applications, all the operations in σ cannot

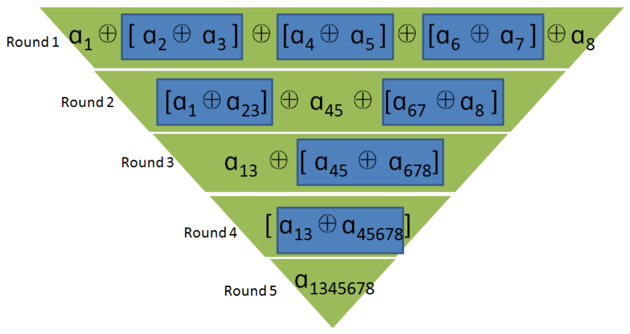

Round 1 a1 ⊕ [a2 ⊕ a3] ⊕ [a4 ⊕ a5] ⊕ [a6 ⊕ a7] ⊕ a8
Round 2 [a1 ⊕ a23] ⊕ a45 ⊕ [a67 ⊕ a8]
Round 3 a13 ⊕ [a45 ⊕ a678]
Round 4 [a13 ⊕ a45678]
Round 5 a1345678

Figure 3.1 - An example for a given list of operations taken from activity $ACT(A, \oplus)$. $A = \{a_1, a_2, \ldots, a_8\}$, and $ACT(A, \oplus) = \{(a_1, a_2), (a_2, a_3), (a_3, a_4), (a_4, a_5), (a_5, a_6), (a_6, a_7), (a_7, a_8)\}$. In each round, nearly half of these operations are operable and the number of operations will be reduced by approximately half.

be finished in one round, since the lock mechanism disallows the element to interact with other elements during its computing period. For a given list of operations $a_1 \oplus a_2 \oplus \ldots \oplus a_z$, as each operation (a_i, a_{i+1}) needs to lock the small world $[a_i, a_{i+1}]$ first, only half of these operations are operable in each communication round. The number of operations is reduced by half at each round, and the number of communication round required for computing all operations in $a_1 \oplus a_2 \oplus \ldots \oplus a_z$ is $b \log(z)$, where b is a constant. An example for a given list can be seen in Figure 3.1. In graph $G(\sigma, A)$, multiple lists for a particular task exist, and the longest list is the maximum diameter d_{max} of graph $G(\sigma, A)$. The maximum number of communication round on processing $ACT(A, \sigma)$ is,

$$CommRound = b \cdot \log(d_{max}) \tag{3.7}$$

The number of communication round in this case depends on the structure of graph $G(\sigma, A)$. In real applications, the number of computing nodes p is far less than the number of operation m in σ. so equation (3.6) is used to analyze the running time.

As the computing time on processing $ACT(A, \sigma)$ in a share memory machine is $O(m\mathrm{H})$, the speedup of using a cluster with p processors can be calculated with the following formula according to equation (3.6),

$$Speedup = \frac{pm\mathrm{H}}{rmS + FL + m\mathrm{H}} \tag{3.8}$$

Equation (3.6) and (3.8) can be used for analysis the performance of semigroup computation. Parameters, such as F, H and m, are further determined by the problem itself. For example, if an operation has constant number of computing time, here H = O(1), SWAP computational framework will linearly scale up with the increasing number of cores.

Appendix 4 — Complexity analysis of graph reduction

To analyze the complexity of graph reduction using the complexity results of SWAP computational framework, the diameter d_{max} of graph $G(\sigma, A)$ and total communication data volume $\digamma$ need to be addressed first.

If the gaps, sequencing errors, and repeats are randomly distributed over the genome sequence w, we have the following corollary according to Theorem 4 in [23],

Corollary 1. Given a reference genome sequence w with length g, if the percentage of k-molecules in the area of gaps, errors, and repeats is q, then the sequence w can be broken into $qw + 1$ subsequences or contigs, and the length of the longest contigs C_{max} satisfies $Prob\{C_{\max} \geqslant 3c \cdot q\log(g)\} \leqslant \frac{1}{g^c}, c > 2$.

Corollary 1 shows that the length of longest contigs is bounded by $3c \cdot q\log(g)$ with high probability, where c is a constant number and it varies for different species.

Graph $G(\sigma, A)$ consists of a series of separated long lists of edges, and each list can be merged into one contigs. The length of the longest list in graph $G(\sigma, A)$ can be regarded as the diameter of $G(\sigma, A)$, which is bounded by $3cq \cdot \log(g)$. According to equation (3.7), when the number of cores is closer or larger than the number of operations m, the maximum communication round on computing edge merging operations is,

$$CommRound = b \cdot \log(3cq \cdot \log(g)) \tag{4.1}$$

The total communication data volume is the same for a particular parallel job, independent of the number of cores used. We can calculate the total communication volume for the case when the number of cores is close or larger than the number of operations. In figure 8, every edge in 1-step bi-directed graph carries a label about the edge and is initialized with one nucleotide. The label on each edge is the main data transmitted by data messages. In each edge merging operation, only two edges are merged into one, and the label length of the max length of new edges will be almost doubled. The total data volume transmitted by data messages for all round can be calculated as follows,

$$\digamma = \sum_{i=0}^{CommRound} 2^i \times \frac{n}{2^i} = b \cdot \log(3cq \cdot \log(g)) \times n \tag{4.2}$$

From Algorithm 4, the computation complexity of *Operation* function H is $O(1)$. Combining the equation (3.6), (20) and (21), we can obtain the running time of SWAP-Assembler in graph reduction step as,

$$RunTime = \frac{bnL\log(3cq\cdot\log(g)) + rnS + n}{p} \tag{4.3}$$

With equations (20), (21) and (22), we can conclude that for each computing component the graph reduction step has a computing complexity of $O(\frac{n}{p})$, $O(\frac{n\log(\log(g))}{p})$ communication volume, and $O(\log(\log(g)))$ communication round.