SWAP-Assembler 2: Optimization of De Novo Genome Assembler at Extreme Scale

Jintao Meng*, Sangmin Seo,† Pavan Balaji,† Yanjie Wei,* Bingqiang Wang,‡ and Shenzhong Feng*

*Shenzhen Institutes of Advanced Technology,
Chinese Academy of Sciences, Shenzhen, China 518055
† Mathematics and Computer Science Division
Argonne National Laboratory, Lemont, IL 60439-4844
‡Beijing Genomics Institute, Shenzhen, China 518083
jt.meng@siat.ac.cn, {sseo,balaji}@anl.gov, yj.wei@siat.ac.cn, wangbingqiang@gmail.com, sz.feng@siat.ac.cn

Abstract—**In this paper, we analyze and optimize the most time-consuming steps of the SWAP-Assembler, a parallel genome assembler, so that it can scale to a large number of cores for huge genomes with the size of sequencing data ranging from terabyes to petabytes. According to the performance analysis results, the most time-consuming steps are input parallelization, k-mer graph construction, and graph simplification (edge merging). For the input parallelization, the input data is divided into virtual fragments with nearly equal size, and the start position and end position of each fragment are automatically separated at the beginning of the reads. In k-mer graph construction, in order to improve the communication efficiency, the message size is kept constant between any two processes by proportionally increasing the number of nucleotides to the number of processes in the input parallelization step for each round. The memory usage is also decreased because only a small part of the input data is processed in each round. With graph simplification, the communication protocol reduces the number of communication loops from four to two loops and decreases the idle communication time.**

The optimized assembler is denoted as SWAP-Assembler 2 (SWAP2). In our experiments using a 1000 Genomes project dataset of 4 terabytes (the largest dataset ever used for assembling) on the supercomputer Mira, the results show that SWAP2 scales to 131,072 cores with an efficiency of 40%. We also compared our work with both the HipMER assembler and the SWAP-Assembler. On the Yanhuang dataset of 300 gigabytes, SWAP2 shows a 3X speedup and 4X better scalability compared with the HipMer assembler and is 45 times faster than the SWAP-Assembler. The SWAP2 software is available at https://sourceforge.net/projects/swapassembler.

I. Introduction

Scientists increasingly want to assemble and analyze very large genomes [1], metagenomes [2], [3], and large numbers of individual genomes for personalized healthcare [4], [5], [6], [7]. In order to meet the demand for processing these huge datasets [8], parallel genome assembly seems promising, but in fact the genome assembly problem is very hard to scale for the following reasons [9], [10].

First, state-of-art parallel assembly solutions are dominantly utilizing the de Bruijn graph (DBG) strategy [11]. This strategy is a variant of traveling salesman problem or equivalent to the Euler path problem, which is a well-known NP-hard problem [12]. Second, the number of nodes in the graph representing the genome data is enormous. One base pair in the sequencing data can generate a k-mer (node) in the de Bruijn graph. For example, a 1000 Genomes dataset [13], [14] with 200 terabytes data can generate about 2^{47} k-mers (or nodes), which is 128 times larger than the problem size of the top rank result in the Graph 500 list (as of March, 2016) [15]. Third, sequencing machines are not accurate. About 50% to 80% of k-mers are erroneous [16], [17], thus the nodes and edges in the graph may not be considered trustable depending on what error the user is willing to tolerate. What is more, k-mers located in the low coverage gaps are hard to be distinguished from erroneous k-mers, which makes it hard to recover DNA in the gap. Lastly, species related features, such as repeats, GC distribution, and polyploid make the genome assembly itself more complex and even harder for parallelization.

Previously we developed the SWAP-Assembler [10], which can assemble the Yanhuang genome [18] in 26 minutes using 2,048 cores on TianHe 1A [19]. The work in this paper further improves the SWAP-Assembler by analyzing and optimizing its most time-consuming steps—input parallelization, k-mer graph construction, and graph simplification (edge merging)—with the aim of developing a much faster assembly tool that can scale to hundreds of thousands of cores and challenging the largest genome assembly with a dataset of 4 terabytes. Before our work, the record of largest assembly to date was kept by Kiki [20], which has assembled nearly 2.3 terabytes.

We optimize these three steps to keep the percentage of time usage in each step constant as the number of cores increases. In the input parallelization step, a fragment adjustment algorithm (FAA) and an adjustable I/O *data block* size are used to explore the largest I/O efficiency and at the same time keep a balance between the memory usage and I/O efficiency. In k-mer graph construction, two methods are used to prevent communication efficiency from degrading with increasing numbers of cores. One method keeps the message size independent of the varying number of cores in order to prevent communication with tiny messages. The other method is a data pool designed to separate the I/O process in the input parallelization and communication process in the k-mer graph construction step. With graph simplification, the communication protocol of the lock-compute-unlock mechanism in the SWAP-Assembler is optimized by minimizing the number of communication loops from four loops to two loops, which helps to keep the idle time constant with increasing numbers of cores.

The optimized assembler is called SWAP-Assembler 2 (SWAP2) in this work. In our experiments, a 1000 Genomes dataset of 4 terabytes [13], [14] is used—the largest dataset ever used for assembly. The results on the supercomputer Mira show that SWAP2 scales to 131,072 cores (the highest scalability ever reached) with an efficiency of 40%. The total execution time is about 4 minutes (including 2.5 minutes I/O time). With the Yanhuang dataset of 300 gigabytes, SWAP2 shows a 3X speedup and 4X better scalability compared with HipMer and is 45 times faster than the SWAP-Assembler.

The rest of the paper is organized as follows. Section II briefly introduces the problem of genome assembly. Section III discusses previous works on parallel genome assembly. Section IV presents the optimization methods for each time consuming step. The performance evaluation results for SWAP2 are given in Section V. We summarize the conclusion in Section VI.

II. Background: Genome Assembly

Given one biological genome sample with reference sequence $w \in \mathbb{N}^g$, where $\mathbb{N} = \{A, T, C, G\}$ and $g = |w|$, a large number of short sequences called **reads**, $S = \{s_1, s_2, ..., s_h\}$, are sequenced by the sequencing machines. s_i is a substring of w with some editorial errors introduced by sequencing machines, here $1 \leq i \leq h$. Genome assembly problem is to recover the reference sequence w with S.

In genome assembly, graphs are utilized for representing the genomic data. A directed graph $G = (V, E)$ consists of a set of vertices (k-mers) V and a set of arcs (directed relationships) $E = (V \times V)$. The k-mers are generated by cutting the reads S with a sliding window of length k. The arcs are used to connect any two k-mers cut by two continuous sliding windows on some read $s_i \in S$.

Genome assembly with the de Bruijn graph (DBG) strategy is the process of reconstructing the reference genome sequence from these reads using the above graph consisted with k-mers (Figure 1). However this strategy is a variant of traveling salesman problem or equivalent to the Euler path problem, which is an NP-hard problem [12]. Finding the original reference sequence from all possible Euler paths cannot be solved in polynomial time. What is more, gaps and branches caused by uneven coverage as well as erroneous reads and repeats prevent from obtaining a full length genome. In real cases, a set of shorter genome sequences called **contigs** are generated by merging unanimous paths instead. Thus, our work focuses on finding a scalable solution for the following genome assembly problem.

Problem of Genome Assembly

Input: A set of reads without errors $S = \{s_1, s_2, ..., s_h\}$
Output: A set of contigs $C = \{c_1, c_2, ..., c_t\}$
Requirement: Each contig corresponds to an unanimous path in the de Bruijn graph constructed from the set of reads S

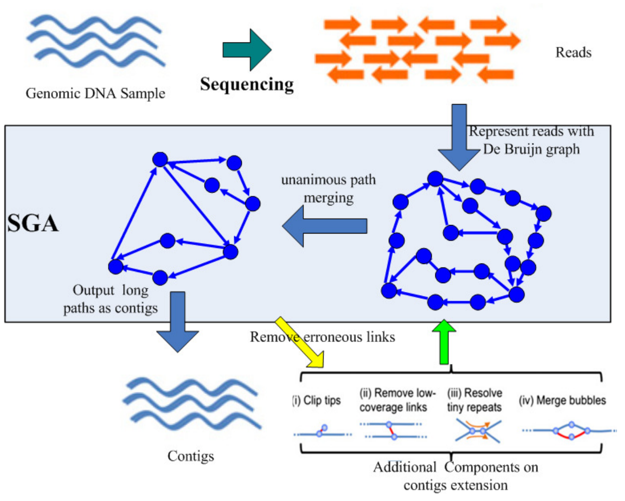
Genomic DNA Sample
Sequencing
Reads
Represent reads with De Bruijn graph
SGA
unanimous path merging
Output long paths as contigs
Remove erroneous links
Contigs
(i) Clip tips
(ii) Remove low-coverage links
(iii) Resolve tiny repeats
(iv) Merge bubbles
Additional Components on contigs extension

Fig. 1: The work flow of genome assembly using de Bruijn graph (DBG) strategy.

III. Related Work

Several state-of-art parallel assemblers have been proposed [9], [10], [21], [22], [23], [24], [25], [26], [27]. Most of them follow the de Bruijn graph (DBG) strategy proposed by Pevzner et al. in 2001 [11].

In ABySS [9], the parallelization is achieved by distributing k-mers to multiservers in order to build a distributed de Bruijn graph. Error removal and graph reduction are implemented over MPI communication primitives.

Ray [2], [28] is a general distributed engine proposed by Boisvert for traditional de Bruijn graphs, that extends k-mers (or seeds) into contigs with a heuristically greedy strategy by measuring the overlapping level of reads in both directions. Performance results on the Hg14 dataset, however indicate that Ray is 12 times slower than the SWAP-Assembler on 512 cores [10].

PASHA [21] focuses on parallelizing k-mer generation and distribution and DBG simplification in order to improve its efficiency with multithreads technology. However, PASHA allows only a single process for each unanimous path, thus limiting its degree of parallelism. Performance results [22] show that PASHA can scale to 16 cores on a machine with 32 cores on three different datasets.

YAGA [23], [24], [25] 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 with a list ranking algorithm [29]. These unanimous paths are reduced locally on separate servers. The recursive list ranking algorithm used in YAGA has a memory usage of $O(n \times \log(np))$, however, that induces large memory usage and causes low efficiency. Here n is the input data size, and p is the number of processes.

HipMer [26], [27] is an end-to-end parallel de novo genome assembler developed in the UPC language [30]. With a graph partition method provided by Oracle, HipMer achieves a scalability of 15,360 cores on both human genome sequencing data (290 Gbp) and wheat genome sequencing data (477 Gbp).

On the other hand, the SWAP-Assembler [10] that we previously developed presents a multi-step bi-directed graph (MSG), a variant of de Bruijn graph, to resolve the computational interdependence on merging edges that belong to the same path. A scalable computational framework SWAP [17], [10] was developed to perform the computation of all edge merging operations in parallel. Experimental results show that the SWAP-Assembler can scale up to 2,048 cores on Yanhuang dataset (300 Gbp). The SWAP-Assembler is demonstrated as a solution with the lowest communication complexity. It is also the first assembler using more than 1,000 CPU cores.

IV. Optimizations

We first evaluate the SWAP-Assembler to identify its performances bottlenecks. We examine every time-consuming step, find the bottlenecks, and discuss the reasons for these performance degradations. Optimization methods and strategies are then presented to resolve these problems. Experiments are presented later in order to confirm the efficiency of these strategies.

Based on **multi-step bi-directed graph** (MSG) and the SWAP computational framework [10], the major time usage of the SWAP-Assembler can be divided into five steps: input parallelization, k-mer graph construction, k-mer filtering, MSG graph construction, and graph simplification (edge merging). Figure 2 shows the time (in seconds) consumed by each step of the SWAP-Assembler when assembling a test genomic dataset from the 1000 Genomes project [13], [14]. This project has more than 200 terabytes of sequencing data from 1,000 people from all over the world. The results show that the most time-consuming steps are input parallelization, k-mer graph construction, and graph simplification. In particular, the most time-consuming step—input parallelization—uses more than half the total time. The time usage of the k-mer graph construction step increases steadily with an increasing number of cores, thus seriously impacting the scalability of SWAP. The time usage of the graph simplification step decreases slightly with the number of cores. To further improve both the efficiency and scalability of the SWAP-Assembler, we optimize the three steps by keeping the percentage of time usage in each step constant with an increasing number of cores and input data size (or **weak-scaling** test).

A. Input Parallelization

Loading the terabytes to petabytes of genomic data into memory with multiple processes faces significant challenges [9], [22]. The SWAP-Assembler [10] adapted a strategy similar to that of Ray [21] and YAGA [23], [24], [25]. Given input reads with n nucleotides from a genome of size g, we divide the input file equally into p virtual data blocks, where p is the number of processes. Each process reads the data located in

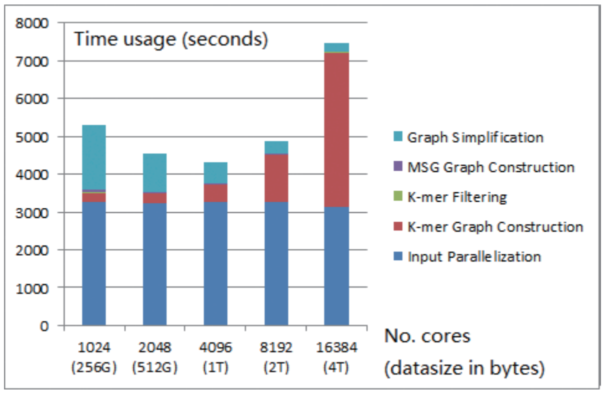

Fig. 2: Time usage for a weak-scaling test of the SWAP-Assembler on processing the data from the 1000 Genomes project. Here each computing node has been allocated 4 processes, and each process has been allocated 256 megabytes of input data. The supercomputer Mira at Argonne National Laboratory was used in this test.

its virtual data block only once. The computational complexity of this step is bounded by $O(n/p)$. However, two restrictions affect the performance of this step. One is that there is no format-sensitive partition strategy for the biology data; the other is that no adjustable parameters are available to boost the I/O performance close to the system limit.

Previous data partition methods can possibly divide the data fragment in the middle of any DNA reads in FASTA or FASTQ format in the SWAP-Assembler [9], [21], [23]. In order to resolve this problem, a location function is used to check the start symbol one by one after reading each byte from the beginning point of each fragment [10]. However this method has an additional I/O overhead on reading the data.

To overcome this drawback, we propose a fragment adjustment algorithm (FAA) in Algorithm 1 to replace the current location function. Every process reads one *data block* and adjusts the start and end position of its own data fragment in the beginning. Each process with rank $procID$ updates the starting point of its fragment to the position of the beginning points of any reads and sends this value to the process $procID - 1$. Then process $procID - 1$ updates the end point of its fragment to this value. After this operation, each DNA read is automatically allocated to only one process in FASTA or FASTQ format without spanning multiple fragments. The FAA algorithm keeps its I/O overhead and the communication overhead constant. Moreover, the size of the *data block* used in our algorithm is an additional parameter for tuning the I/O performance; specifically, by adjusting the *data block* size, the I/O efficiency can be maximized to approach the system limit.

To evaluate the I/O performance improvement with the FAA algorithm and *data block* size tuning, we created a weak-scaling dataset from the 1000 Genomes project [13], [14]. The input data increases proportionally from 256 GB to 4,096 GB as the number of cores increases from 1024 to 16,384 cores; the problem size for each process is kept constant at 256 MB.

Algorithm 1: Fragment Adjustment Algorithm.

```
Input: Dataset $S$ in FASTA or FASTQ format, the rank of local
       process $procID$ and the total number of processes $p$.
Output: Virtual fragments $S_1, S_2, \ldots, S_p$.
begin
    $size$ = the file size of dataset $S$;
    $step = size/p$;
    $start = procID * step$;
    $end = (procID + 1) * step$;
    $end = end < size\ ?\ end : size$;
    $readBuf$ = Read one data block* starting from $start$;
    $i = 0$;
    while $readBuf[i] \neq$ '>' do
        $i$++;
    $sendAdjustDelta = i$;
    if $procID \neq 0$ then
        Send $sendAdjustDelta$ to process $procID - 1$;
    if $procID \neq p - 1$ then
        receive $recvAdjustDelta$ from process $procID + 1$;
    $start$ += $sendAdjustDelta$;
    $end$ += $recvAdjustDelta$;
    $S_{procID} = (start, end)$;
* Here the data block size will be larger than the length of
reads to ensure that every data block contains at least one start
symbol of the read '>'.
```

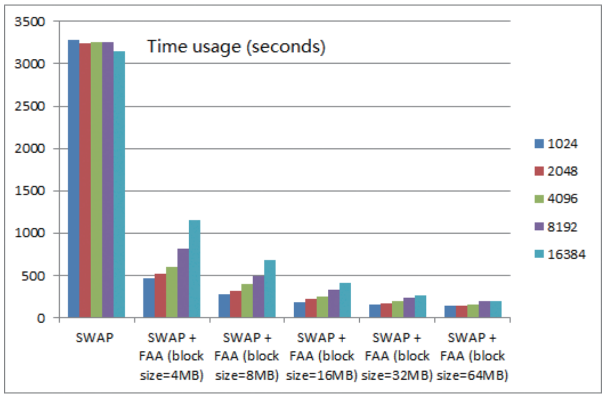

Fig. 3: Time usage statistics of input parallelization using FAA with varying *data block* sizes and numbers of cores.

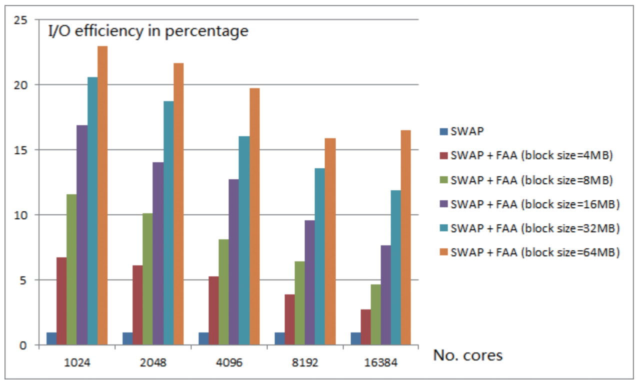

Fig. 4: I/O efficiency of FAA with varying *data block* sizes and numbers of cores. The blue bar is the I/O efficiency of the SWAP-Assembler.

The time usage results are shown in Figure 3. As one can see, compared with the SWAP-Assembler, FAA generally saves more than 60% of the time usage on the input parallelization. By increasing the *data block* size from 4 MB to 64 MB, the time usage is further decreased with increasing *data block* size. The results also confirm that the larger block size can benefit the performance with less I/O operations and better streaming effect, but larger block size also causes several times larger memory usage in the postprocessing steps. Therefore, to balance between memory usage and efficiency, the block size will be fixed to be 64 MB in the following test. With 16,384 cores, SWAP2 achieves 16X speedup with the optimization of the input parallelization.

The I/O efficiency of the input parallelization is presented in Figure 4. Each I/O drawer in Mira has an I/O bandwidth of 32 GB/s [31], [32], [33], [34]; the I/O efficiency of one rack can be estimated by dividing the real I/O bandwidth with 32 GB/s. Figure 4 shows that with 16,384 cores, the I/O efficiency of SWAP2 achieves about 16.5% of the system efficiency.

B. K-Mer Graph Construction

The second step in the SWAP-Assembler aims to construct a graph with its vertices that are k-mers or k-molecules (containing two complementary k-mers) [10]. This step has three phases: k-molecule generation, k-molecule distribution, and k-molecule storage. In the first phase, input sequences are broken into overlapping k-molecules by sliding a window of length k along the input sequence. In the k-molecule distribution phase, each nucleotide can generate one k-molecule. Because the input involves terabytes of data, the number of generated k-molecules is huge for distribution and communication. In the last phase, each process allocates a container to store these k-molecules according to a given hash function.

Figure 5 shows the time usage of these three phases in processing the data generated by the 1000 Genomes project with the SWAP-Assembler. The results show that the bottleneck is the k-molecule distribution; the percentage of time usage used in distribution increases from 53% to 97% when the number of cores increases from 1,024 to 16,384. The dominant workload in the distribution phase is communication, during which each process needs to send the k-molecules to the remote process according to a given hash function and receives all the k-molecules belonging to it. In the SWAP-Assembler, the total data volume of messages communicated between all processes is fixed; but when the number of processes doubles, the message size between each pair of processes is reduced by half. Communication with tiny messages thus will induce low efficiency and directly affect the performance of this step [35], [36].

To improve the communication efficiency and prevent communication with tiny messages, we include three optimization strategies in this step.

Data compressing In the k-molecule generation, we have compressed two arcs sharing the same k-molecule into one,

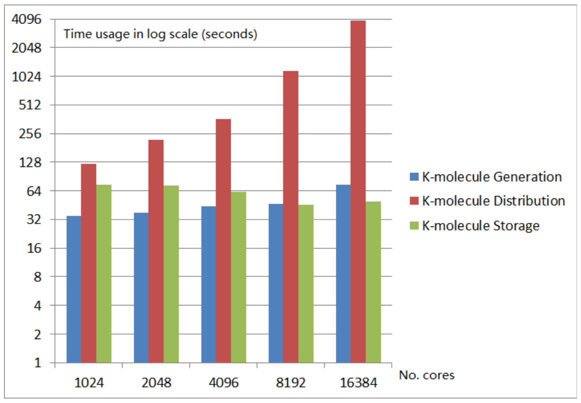

Fig. 5: Time usage statistics for the three phases of the k-mer graph construction step.

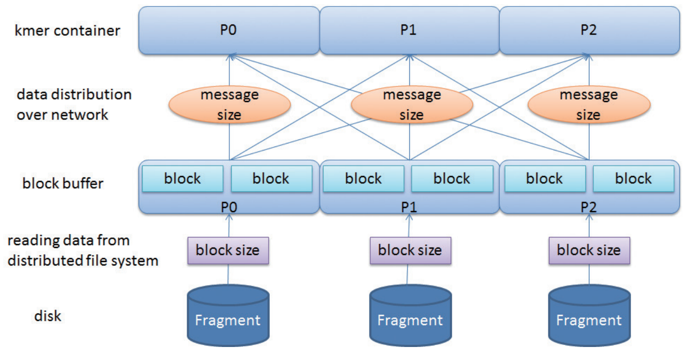

Fig. 6: Data pool designed to separate the I/O process in the input parallelization step and communication in the k-mer graph construction step.

thus reducing the communication data volume and memory usage by half.

Initial message size tuning To prevent communication with tiny messages, we have to keep the message size independent of the increasing number of cores. In each communication round, the number of nucleotides (in DNA reads) processed in every process is fixed to be L. The data thus can generate at most L k-molecules distributed across p processors. The number of nucleotides L is designed to increase proportionally with the number of processers; in this case, the number of generated k-molecules or message size between any two processers is a constant of $L \times B_k/p_0$, where B_k is the data structure size of k-molecules and p_0 is the number of cores used for the performance baseline. In our case, $p_0 = 1024$.

With this method the message size between any two processers is constant at run-time. However, the initial message size can be adjusted with the number of nucleotides L, enabling higher communication efficiency. Arbitrarily varying the number of nucleotides and the I/O *data block* size, however, can induce interference between these two steps. For example, if the I/O block size is set to 1 MB and the number of nucleotides L is set to 1K, when the number of processes increases beyond 1,024, the total number of k-molecules needed for communication will be more than 1 million. In this case, the data is not enough, and the message size will decrease.

I/O and Communication Isolation To fix the cited problem, we used a data pool, shown in Figure 6, to separate the I/O process in the input parallelization step and the communication process in the k-mer graph construction step. The data pool is a shared-memory space for the two steps. The communication phase can continuously read data from this data pool, and the data pool will be large enough to keep the message size constant. Here the data pool acts as a blocking queue, the I/O process in the input parallelization step acts as a producer, and the communication process acts as a consumer. With this data pool, the communication part and the I/O part are isolated, and the input data (reads) can be automatically refilled from disk to the data buffer by calling the I/O functions. The communication process automatically reads these reads from this pool. The advantage is that we can select the best message size and I/O *data block* size to achieve peak performance in both steps.

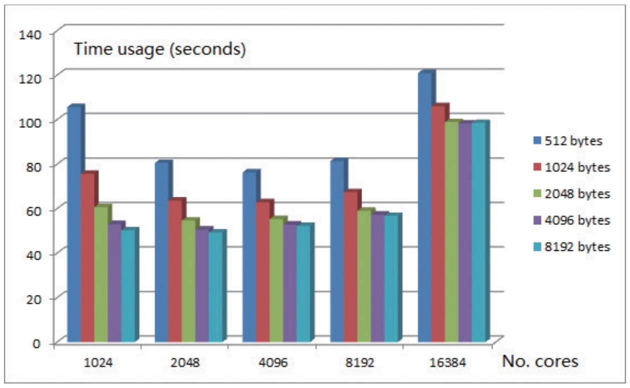

Fig. 7: Time usage of the communication routines for k-mer graph construction step on processing the 1000 Genomes project dataset.

We designed a weak-scaling experiment to find the best value for the initial number of nucleotides L processed in one round. Here we increased L from 512 bytes to 8,192 bytes. To collect the time usage on data communication, we inserted tags before and after the MPI communication routines delivering the data. The results are plotted in Figure 7. As the figure shows, for a fixed number of nucleotides L the running time increases with the increasing number of cores. For a run using more CPU cores (in this case 16,384 cores), the efficiency is decreased. Increasing the initial number of nucleotides L can save running time, but this trend is weakened by the increasing number of cores. The best value, 8,192 bytes, is used as the initial number of nucleotides L in the following experiments.

The time usage for these three phases before and after optimization is presented in Figure 8. For the first phase, compared with original version, the running time on cutting reads decreases steadily with the increasing number of cores, and a 5.2X speedup is achieved. The time usage in distribution is almost fixed when the data size increases with the number

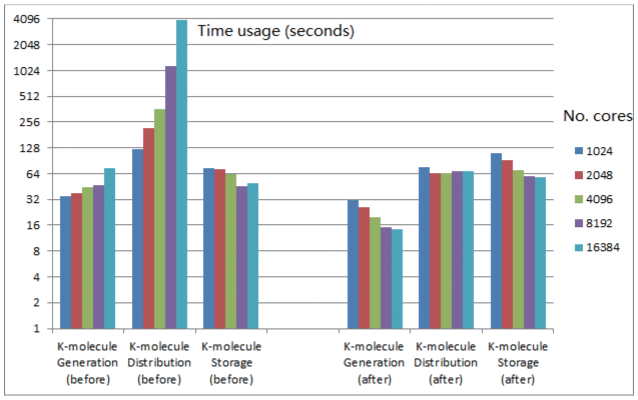

Fig. 8: Time usage statistics for the three phases of the k-mer graph construction step before and after optimization.

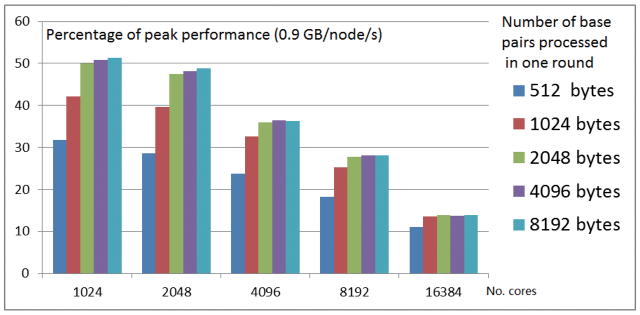

Fig. 9: Communication efficiency of the k-mer graph construction step on processing the 1000 Genomes project dataset. Here the theoretical peak communication performance of Mira is 0.9 GB per node per second.

of cores. With 16,384 cores, the speedup is about 64X that of its previous version. In the last phase, the time usage in these two subfigures share the same trends.

We also evaluated the communication efficiency of the optimized SWAP. The peak all-to-all bandwidth of a 5D-torus network is limited by the length of its longest dimension [35], [36]. Because the longest dimension Dim in Mira with 4,096 nodes is 16, the peak user data communication per-node for all-to-all bandwidth is $8/Dim * 1.8$ GB/s, which is 0.9 GB/s [35], [36]. The time usage for data communication and the communication efficiency was calculated and is plotted in Figure 9. The results show that the communication bandwidth has improved slightly by increasing the number of nucleotides L processed in each round. With the increasing number of cores from 1,024 (1/4 rack) to 16,384 (4 rack), however, the communication efficiency decreases from 50% to 15%, which follows the general trend of decreasing efficiency with increasing number of cores.

C. Graph Simplification

In the graph simplification step, the SWAP computational framework with two user-defined functions is used to merge edges into contigs. Algorithm 2 describes the lock-computing-unlock schedule in the SWAP computational framework [10].

Algorithm 2: Communication protocol for lock-computing-unlock schedule in SWAP. Here the protocol is divided into two routines for the SWAP thread and service thread [10].

```
begin
    Routine in SWAP thread;
    Lock Stage:
    Post MPI_Isend($compReq$);
    Post MPI_Irecv($compReq + 1$);
    Reply Call RecvProc(2, $compReq$);
    Notify Stage:
    Post MPI_Isend($compReq$);
    Post MPI_Isend($compReq + 1$);
    Call RecvProc(2, $compReq$);

    Routine in service thread;
    while $true$ do
        Post MPI_Testall(2, $compReq$, $\&flag$);
        if $flag$ then
            break;
        Post MPI_Test($\&globalReq$, $\&flag$);
        if $flag == 0$ then
            continue;
        Doing computation work here $\ldots$;
        Post MPI_Irecv($\&globalReq$);
```

In Algorithm 2, the communication protocol is divided into two routines for the SWAP thread and service thread. In the SWAP thread, a vertex needs to send a lock message to its neighbors. This vertex can move to the notify stage only after it collects all lock replies. In the notify stage, this vertex sends computing commands and associated data to its neighbors if all these lock replies have success tags; otherwise, the protocol will send unlock messages to release the lock for all its neighbors automatically. In the service thread, a while loop is used to detect the completion of the communication and revoke the computing work or restart the routine in SWAP thread.

Figure 10 illustrates how the communication protocol in Algorithm 2 works. Node 0 sends a lock message to node 1 and waits for its reply. After that, another lock message is sent to node 2 for its reply. Here two communication loops are used for node 0 to communicate with node 1 and node 2. After receiving the two reply messages with lock success tags, node 0 sends the notify message together with the related data to node 1 and node 2 in two communication loops. Overall, Algorithm 2 needs four loops to complete the schedule in SWAP. The overall time usage of Algorithm 2 is affected by two factors: the waiting time of a round-trip reply message in one loop and the number of communication loops. In Algorithm 2, the service routine is active only after receiving the reply message. For large supercomputers such as Mira [33], [34], more cores indicate longer latency, as confirmed by the left graph of Figure 11, where the waiting time increases steadily with the increasing number of cores.

To minimize the number of communication loops and

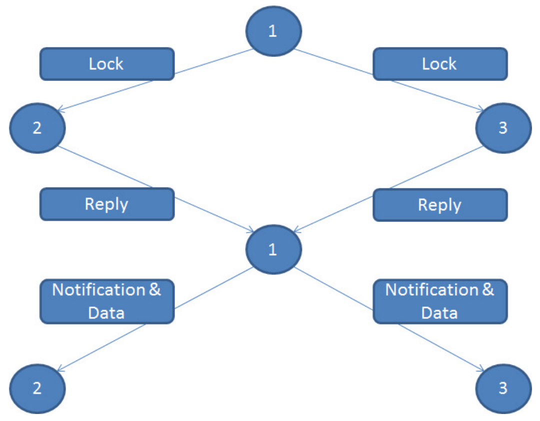

Fig. 10: Two neighbors sharing the same loop in both the lock and notify stages.

providing possibilities for sharing the idle time in Algorithm 2, we introduced an optimized communication protocol in Algorithm 3. In this algorithm, node 0 can send the lock messages to node 1 and node 2 and receive the replies at the same time. After collecting all the replies, node 0 can send the notify message and data to node 1 and node 2 at the same time. Only two loops therefore are needed for the communication protocol of SWAP.

Algorithm 3: Optimized communication protocol for lock-computing-unlock schedule. Here the calls to the compute routine on two vertices have been integrated into one routine.

```
begin
    Routine in SWAP thread;
    Lock Stage:
    Post MPI_Isend(compReq) ;
    Post MPI_Irecv(compReq+1) ;
    Post MPI_Isend(compReq+2) ;
    Post MPI_Irecv(compReq+3) ;
    Call RecvProc(4,compReq) ;
    Notify Stage:
    Post MPI_Isend(compReq);
    Post MPI_Irecv(compReq+1);
    Post MPI_Isend(compReq+2);
    Post MPI_Irecv(compReq+3);
    Call RecvProc(4, compReq);

    Routine in service thread;
    while $true$ do
        Post MPI_Testall(2, $compReq$, &$flag$);
        if $flag$ then
            break;
        Post MPI_Test(&$globalReq$, &$flag$);
        if $flag == 0$ then
            continue;
        Doing computation work here ...;
        Post MPI_Irecv(&$globalReq$);
```

We conducted an experiment to test the improvement of the optimized protocol for SWAP. A weak-scaling data from the 1000 Genomes project is used in this experiment. The input data increases proportionally from 512 GB to 4,096 GB with the increasing number of cores in order to keep the problem size for each process constant. The time usage results are shown in Figure 11. The left panel in the figure shows that the idle time in the communication protocol of SWAP increases with the increasing number of cores and reaches 85% of the total time at 32,768 cores. The right panel shows that with the optimization on the communication protocol, the idle time is kept constant at about 40% in all cases.

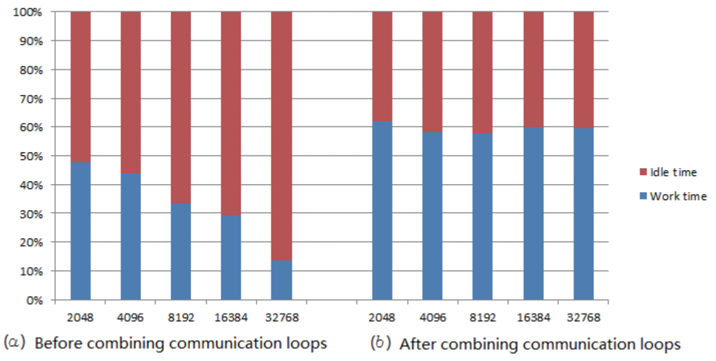

Fig. 11: Graphs showing constant idle time after the communication optimization.

TABLE I: Time usage (seconds) of SWAP2 on weak-scaling test with the data from the 1000 Genomes project.

Data Size	*256 GB*	*512 GB*	*1 TB*	*2 TB*	*4 TB*
No. Cores	1024	2048	4096	8192	16384
Input Parallelization	138.27	145.2	154.81	183.35	208.41
K-mer Graph Const	139.62	136.67	129.68	119.39	177.87
K-mer Filtering	20.88	14.46	15.78	12.6	13.41
MSG Graph Const	174.37	98.3	54.23	30.28	15.73
Graph Simplification	1443.87	843.64	438.9	231.33	123.88
Total Time Usage	1948.77	1256.13	803.57	582.59	543.58

V. Performance Evaluation

SWAP2 has integrated all the cited optimization methods and is available online in SourceForge [37]. For performance evaluation, Mira at Argonne National Laboratory [33] was used as the test cluster; 32,768 computing nodes were allocated for this experiment. Each compute node is equipped with 16 cores and 16 GB of memory; all nodes are connected with a high-speed 5D-torus network with the bidirectional bandwidth of 10 GB/s. The I/O storage system of Mira uses the IBM GPFS system; it supports parallel file I/O defined in MPI-3.

First, a weak-scaling comparison between SWAP-Assembler (SWAP for short) and SWAP2 was made with the data selected from the 1000 Genomes project. In this experiment the data size was increased from 256 GB to 4 TB as the number of cores increased from 1,024 to 16,384. Figure 2 and Figure 12 show that SWAP2 has the following three performance improvements over SWAP.

Scalability: SWAP2 scales to 16,384 cores, whereas SWAP scales only to 4,096. We can see that excluding the time

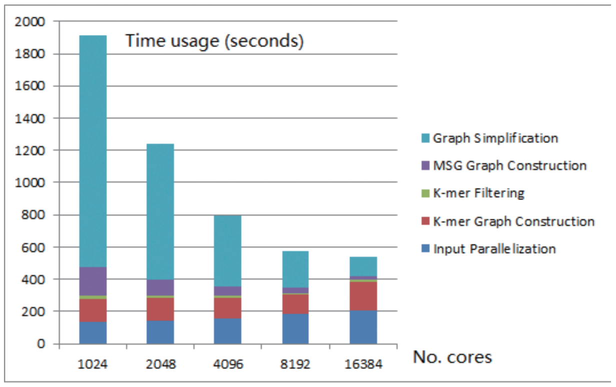

Fig. 12: Time usage for each step of SWAP2 in processing the data from the 1000 Genomes project. Here each computing node has been allocated 4 processes.

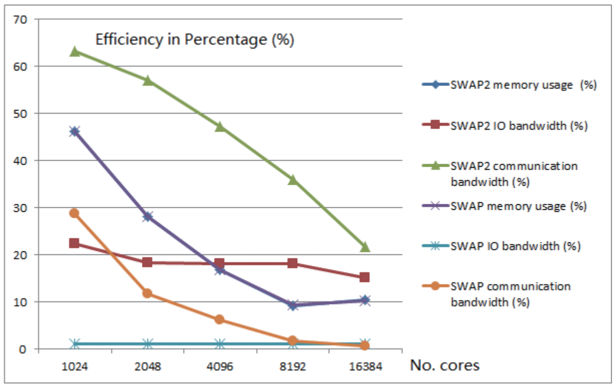

Fig. 13: Ratio of I/O bandwidth, communication bandwidth, and memory usage with the system peak performance in theory. The peak performance of I/O bandwidth and communication bandwidth are 32 GB/rack/s [33], [34], [31], [32] and 0.9 GB/node/s, respectively [35], [36]. Each computing node equipped 16 GB memory has been allocated 4 processes; the percentage of memory usage therefore is calculated by the memory usage of each process divided by 4 GB.

used in graph simplification and distributed MSG graph construction, the time usage for the other three steps by SWAP2 increases slightly with the increasing number of cores. The data in Table I also confirms that the percentage of time usage on these three steps is almost constant. Because the 1000 Genomes project has a fixed genome size of 3 billion nucleotides, after the k-mer filtering step, the de Bruijn graph has approximately the same number of nodes with the genome size. With a fixed problem size, the time usage of the last two steps is decreased almost in half when the number of cores doubles.

Speedup: The time usage of SWAP2 is orders of magnitude less than that of SWAP. With the fragment adjustment algorithm and I/O *data block* size tuning, the input parallelization step gains a 15X speedup over its previous version. In the k-mer graph construction step, the communication efficiency degradation has been resolved with a fixed communication message size and a data pool isolating the communication and I/O process. With these two solutions, a 23X speedup is achieved. In the graph simplification step, by compressing the communication protocol of SWAP's lock-computing-unlock schedule from 4 loops to 2 loops and sharing the idle time between these loops, the time usage is 1.75 times less than that of the previous version. The overall speedup of SWAP2 is 14 times faster than that of SWAP.

Efficiency: In order to compare the performance after optimization with the system peak performance, the percentage of I/O bandwidth, communication bandwidth, and memory usage are illustrated in Figure 13. The I/O bandwidth of SWAP2 has been improved from 1% to 18% on 4,096 cores (one rack), and the communication bandwidth has been improved from 5% to 47%. Room for improvement remains, however, particularly in memory usage, which shows the same trend as does SWAP.

To evaluate SWAP2's strong-scaling scalability, we performed an experiment with fixed problem size and increasing number of cores. Here we selected 4 terabytes of data from the 1000 Genomes project, and the number of cores was increased from 1,024 (or 512 nodes) to 131,072 (or 16,384 nodes). The

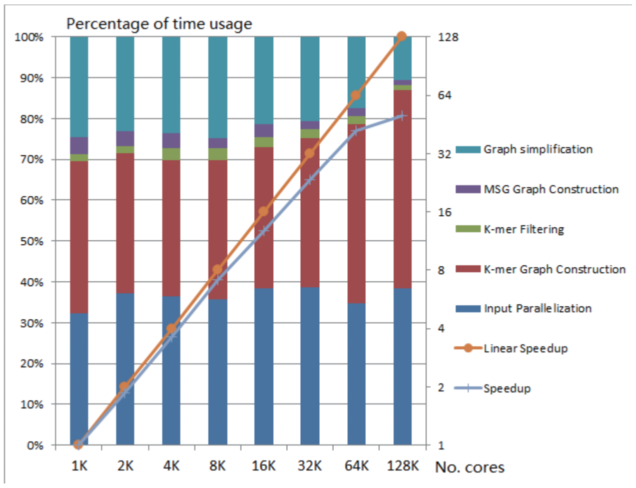

Fig. 14: Time consumption results for the strong-scaling experiment.

time usage results are presented in Table II and plotted in Figure 14. In the figure, each step keeps a fixed proportion of time usage as the number of cores increases; all five steps are highly parallelized and scale at almost the same ratio. The runtime results are presented in Table II. SWAP2 takes 2 minutes with 131,072 cores to assemble the 4 terabytes of sequencing data, the largest dataset that has ever been tested. The speedup of SWAP2 increases steadily and reaches 8.8 when the number of cores is 131,072, corresponding to an efficiency of about 40%.

We also compared our work with another highly parallel assembler, HipMer. The Yanhuang dataset of about 300 GB was used [18], [38]. The runtime results in Table III show that SWAP2 can assemble the dataset in 163 seconds using 16,384 cores on Mira. Because HipMer is not an open source project,

TABLE II: Time usage of SWAP2 collected for the strong-scaling test on a 4-terabyte dataset from the 1000 Genomes project. Each computing node was allocated 4 processes ($ppn = 4$); time is measured in seconds.

No. Cores	*Input Parallelization*	*K-mer Graph Construction*	*K-mer Filtering*	*MSG Graph Construction*	*Graph Simplification*	*Total Time Usage*
1,024	681	880.54	14.85	18.73	130.12	1725.24
2,048	1372.33	1268.96	61.33	140.66	850.58	3721.01
4,096	691.88	633.04	53.35	70.28	446.68	1906.96
8,192	346.23	328.91	26.61	23.79	240.32	972.55
16,384	207.15	184.86	13.2	17.33	115.43	541.46
32,768	114.26	107.16	6.6	6.06	60.57	297.38
65,536	56.2	70.76	3.28	3.1	28.24	165.01
131,072	51.53	64.71	1.63	1.62	14.22	138.39

TABLE III: Time usage of SWAP2 collected for the strong-scaling test on the human genome (Yanhuang genome dataset) [18]. Each computing node was allocated 4 processes ($ppn = 4$); time is measured in seconds.

No. Cores	*Input Parallelization*	*K-mer Graph Construction*	*K-mer Filtering*	*MSG Graph Construction*	*Graph Simplification*	*Total Time Usage*
1,024	117.55	281.44	19.08	185.05	1630.32	2266.72
2,048	59.81	140.56	9.63	92.56	838.18	1157.72
4,096	20.81	71.77	6.1	46.3	429.71	583.51
8,192	13.46	39.29	3.15	22.49	223.92	307.06
16,384	8.35	23.9	1.55	11.39	115.44	163.36
32,768	5.08	18.99	0.88	5.71	63.77	96.51
65,536	6.5	20.85	0.67	2.92	27.74	64.55

we directly took the results in [27] for comparison. Note that Cray Edison [39], [40] is used by HipMer [27]. Compared with Mira, Edison is 7.8 times faster with its dragonfly network, and its single CPU core is 1.5 times faster [39], [40]. Using about 15,000 CPU cores on Edison, HipMer assembled (including scaffolding) the Yanhuang dataset in 8 minutes. SWAP2 is about 3 times faster than HipMer. Moreover, for the same Yanhuang dataset, SWAP2 can further scale to 65,536 cores on Mira and assemble the dataset in 64 seconds with a parallel efficiency of 55%. The scalability of HipMer strongly depends on the effectiveness and scalability of the oracle graph partition mechanism, whereas SWAP2 is optimized with a fully parallelized algorithm in every step, resulting in better scalability and system efficiency than achieved by HipMer.

Although the work in this paper on SWAP2 is optimized based on Mira, the strategies used are general and focus on the algorithm level. We use no special instructions designed for any special CPU or network architectures. Therefore SWAP2 can be used with other supercomputers with a particular value for the data block size and initial message size L in order to approach that supercomputer's peak performance.

VI. Conclusion

In this paper, the most time-consuming steps of the SWAP-Assembler—input parallelization, k-mer graph construction, and graph simplification—were optimized in order to keep the percentage of time usage in each step constant when the number of cores increases. With these optimizations, the I/O bandwidth improved from 2% to 18% on 4,096 cores (one rack), and the communication improved from 5% to 47%. In the experiment on the 1000 Genomes project dataset, the weak-scaling results show that newly optimized assembler, called SWAP2, scales to 16,384 cores; and the strong-scaling results show that SWAP2 scales to 131,072 cores. The total assembly time with 131,072 cores is 2 minutes. For the Yanhuang dataset, SWAP2 shows a 3X faster execution time and 4X better scalability than does HipMer. The experiments show that the optimized SWAP2 can both scale up (3 times faster than HipMer) and scale out to 131,072 cores. The program can be downloaded from
https://sourceforge.net/projects/swapassembler.

Authors' Contributions

Meng carried out the optimization and development of SWAP2 and drafted the manuscript. Seo and Balaji conceived of the study, participated in its design and coordination, and helped draft the manuscript. Wei participated in the development of the SWAP-Assembler and modification of this manuscript. Wang participated in the design and optimization of the SWAP-Assembler. Shengzhong participated in the design of the study and design of the performance test. All authors read and approved the final manuscript.

Acknowledgments

We thank Rob Latham from Argonne National Laboratory (ANL) for sharing his optimization experience in improving SWAP2's I/O library, Huiwei Lv from ANL (currently in Tencent) for his work in optimizing the communication routines, Sheng Di from ANL and Dong Chen from IBM T.J. Watson for sharing their optimization strategies on 5D torus network in Mira, Jiefeng Cheng from Huawei Noah's Ark Lab for his cooperation work on graph computing platforms, and Gail Pieper from ANL for her editorial work on this manuscript.

Declarations

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357 and by the National Science Foundation of China under Grant No.11204342, Shenzhen Peacock Plan under grant No. KQCX20130628112914299, the Science Technology and Innovation Committee of Shenzhen Municipality under grant No. JCYJ20140901003939036 and National High Technology Research and Development Program of China under grant No. 2015AA020109 and No. 2014AA01A302. The calculations were performed on Mira at the Argonne Leadership Computing Facility.

References

[1] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen et al., "De novo assembly of human genomes with massively parallel short read sequencing," Genome research, vol. 20, no. 2, pp. 265–272, 2010.

[2] S. Boisvert, F. Raymond, É. Godzaridis, F. Laviolette, J. Corbeil et al., "Ray Meta: scalable de novo metagenome assembly and profiling," Genome Biol, vol. 13, no. 12, p. R122, 2012.

[3] T. Namiki, T. Hachiya, H. Tanaka, and Y. Sakakibara, "MetaVelvet: an extension of Velvet assembler to de novo metagenome assembly from short sequence reads," Nucleic acids research, vol. 40, no. 20, pp. e155–e155, 2012.

[4] E. Le Chatelier, T. Nielsen, J. Qin, E. Prifti, F. Hildebrand, G. Falony, M. Almeida, M. Arumugam, J.-M. Batto, S. Kennedy et al., "Richness of human gut microbiome correlates with metabolic markers," Nature, vol. 500, no. 7464, pp. 541–546, 2013.

[5] M. Arumugam, J. Raes, E. Pelletier, D. Le Paslier, T. Yamada, D. R. Mende, G. R. Fernandes, J. Tap, T. Bruls, J.-M. Batto et al., "Enterotypes of the human gut microbiome," Nature, vol. 473, no. 7346, pp. 174–180, 2011.

[6] J. Qin, R. Li, J. Raes, M. Arumugam, K. S. Burgdorf, C. Manichanh, T. Nielsen, N. Pons, F. Levenez, T. Yamada et al., "A human gut microbial gene catalogue established by metagenomic sequencing," nature, vol. 464, no. 7285, pp. 59–65, 2010.

[7] S. R. Gill, M. Pop, R. T. DeBoy, P. B. Eckburg, P. J. Turnbaugh, B. S. Samuel, J. I. Gordon, D. A. Relman, C. M. Fraser-Liggett, and K. E. Nelson, "Metagenomic analysis of the human distal gut microbiome," science, vol. 312, no. 5778, pp. 1355–1359, 2006.

[8] J. Shendure and H. Ji, "Next-generation DNA sequencing," Nature biotechnology, vol. 26, no. 10, pp. 1135–1145, 2008.

[9] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "ABySS: a parallel assembler for short read sequence data," Genome research, vol. 19, no. 6, pp. 1117–1123, 2009.

[10] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "SWAP-Assembler: scalable and efficient genome assembly towards thousands of cores," BMC bioinformatics, vol. 15, no. Suppl 9, p. S2, 2014.

[11] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," Proceedings of the National Academy of Sciences, vol. 98, no. 17, pp. 9748–9753, 2001.

[12] P. Pevzner, Computational molecular biology: an algorithmic approach. MIT press, 2000.

[13] N. Siva, "1000 Genomes project," Nature biotechnology, vol. 26, no. 3, pp. 256–256, 2008.

[14] Data provided by the 1000 genomes project. [Online]. Available: ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/data

[15] Brief introduction graph 500. [Online]. Available: www.graph500.org

[16] Y. Peng, H. C. Leung, S.-M. Yiu, and F. Y. Chin, "Idba–a practical iterative de bruijn graph de novo assembler," in Research in Computational Molecular Biology. Springer, 2010, pp. 426–440.

[17] J. Meng, J. Yuan, J. Cheng, Y. Wei, and S. Feng, "Small world asynchronous parallel model for genome assembly," in Network and Parallel Computing. Springer, 2012, pp. 145–155.

[18] G. Li, L. Ma, C. Song, Z. Yang, X. Wang, H. Huang, Y. Li, R. Li, X. Zhang, H. Yang et al., "The YH database: the first Asian diploid genome database," Nucleic acids research, vol. 37, no. suppl 1, pp. D1025–D1028, 2009.

[19] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su, "The TianHe-1A supercomputer: its hardware and software," Journal of computer science and technology, vol. 26, no. 3, pp. 344–351, 2011.

[20] Kiki download site. [Online]. Available: https://github.com/GeneAssembly/kiki

[21] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: simultaneous assembly of reads from a mix of high-throughput sequencing technologies," Journal of Computational Biology, vol. 17, no. 11, pp. 1519–1533, 2010.

[22] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de Bruijn graphs," BMC bioinformatics, vol. 12, no. 1, p. 1, 2011.

[23] B. G. Jackson and S. Aluru, "Parallel construction of bidirected string graphs for genome assembly," in Parallel Processing, 2008. ICPP'08. 37th International Conference on. IEEE, 2008, pp. 346–353.

[24] B. G. Jackson, P. S. Schnable, and S. Aluru, "Parallel short sequence assembly of transcriptomes," BMC bioinformatics, vol. 10, no. Suppl 1, p. S14, 2009.

[25] B. G. Jackson, M. Regennitter, X. Yang, P. S. Schnable, and S. Aluru, "Parallel de novo assembly of large genomes from high-throughput short reads," in IEEE International Symposium on Parallel and Distributed Processing. IEEE, 2010, pp. 1–10.

[26] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de Bruijn graph construction and traversal for de novo genome assembly," in International Conference for High Performance Computing, Networking, Storage and Analysis, SC14. IEEE, 2014, pp. 437–448.

[27] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "HipMer: an extreme-scale de novo genome assembler," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2015, p. 14.

[28] E. Godzaridis, S. Boisvert, F. Xia, M. Kandel, S. Behling, B. Long, C. P. Sosa, F. Laviolette, and J. Corbeil, "Human analysts at superhuman scales."

[29] F. Dehne and S. W. Song, "Randomized parallel list ranking for distributed memory multiprocesors," in Concurrency and Parallelism, Programming, Networking, and Security. Springer, 1996, pp. 1–10.

[30] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, Introduction to UPC and language specification. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[31] K. Kumaran, "Introduction to Mira," in Code for Q Workshop.

[32] J. Milano, P. Lembke et al., IBM system Blue Gene solution: Blue Gene/Q hardware overview and installation planning. IBM Redbooks, 2013.

[33] Mira - IBM BG/Q supercomputer machine overview. [Online]. Available: https://www.alcf.anl.gov/user-guides/machine-overview

[34] Blue gene/q overview and update. [Online]. Available: https://www.alcf.anl.gov/files/IBM_BGQ_Architecture_0.pdf

[35] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q interconnection network and message unit," in High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for. IEEE, 2011, pp. 1–10.

[36] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow et al., "Looking under the hood of the IBM Blue Gene/Q network," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, 2012, p. 69.

[37] SWAP2 download site. [Online]. Available: http://sourceforge.net/projects/swapassembler

[38] Dataset provided by the 1000 Genomes project. [Online]. Available: ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/data

[39] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," Cray Inc., White Paper WP-Aries01-1112, 2012.

[40] M. J. Cordery, B. Austin, H. Wassermann, C. S. Daley, N. J. Wright, S. D. Hammond, and D. Doerfler, "Analysis of Cray XC30 performance using Trinity-NERSC-8 benchmarks and comparison with Cray XE6 and IBM BG/Q," in High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation. Springer, 2013, pp. 52–72.