GraphSeq: Accelerating String Graph Construction for De Novo Assembly on Spark

Chung-Tsai Su[1,*], Ming-Tai Chang[1], Yun-Chian Cheng[1], Yun-Lung Li[1] and Yao-Ting Wang[1]

[1]Atgenomix, Taipei, Taiwan.

*To whom correspondence should be addressed.

System Flow of GraphSeq

The system flow of GraphSeq is illustrated in Fig. S1. First of all, a single compressed FASTQ file will be divided into hundreds of small FASTQ files. Then, those small files will be loaded by ADAM in parallel and transformed into the alignment records in Parquet format. GraphSeq leverage ADAM to load sequencing data efficiently via the columnar Apache Parquet format. After that, all of suffixes will be generated and dispatched to the corresponding partition according to their prefix string. The detailed algorithm for string graph construction is described in the following session.

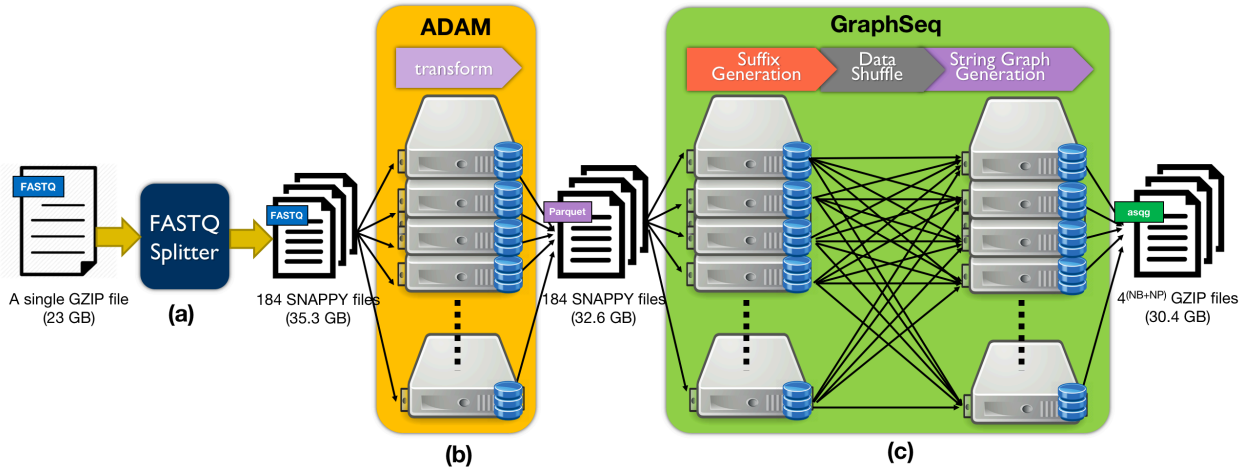


Fig. S1. System flow. (**a**) a lightweight program to split a single big FASTQ file into several small FASTQ files. (**b**) ADAM transforms FASTQ data into alignment record in parquet format. (**c**) GraphSeq loads all of reads in parallel and generates all suffixes of the given reads. Then, each suffix will be shuffled into a specific partition by its' prefix. After that, string graph construction can be applied within each partition.

Method

GraphSeq leverages Spark to construct string graph in parallel and provides many useful parameters to adjust the resource requirement and expected turnaround time. Before going to algorithm details, the parameter of GraphSeq and the definitions used in algorithm are introduced.

Parameter

GraphSeq provides many parameters to adopt to diverse data and heterogenous hardware environment.

Table. S1. Parameter of GraphSeq.

Name	Type	Default	Description
NB	int	1	Number of Batch [4^{NB}]
NP	int	7	Number of Partitions [4^{NP}]
BS	int	100	Bucket size
MAX_RL	int	151	Maximal read length
MIN_OL	Int	85	Minimal overlapping length
CACHED	boolean	false	Keep all reads in memory
RMDUP	boolean	false	Remove duplicated reads

Notation

The definitions used in the following session is listed as follows:

DS_r : dataset of reads;

R : dataset of reads with their reverse complement;

bucket(i) : the i-th bucket to keep byte-aligned 2-bits data;

SA : dataset of all suffixes;

SSA : array of suffixes with LCP;

ED : array of edges;

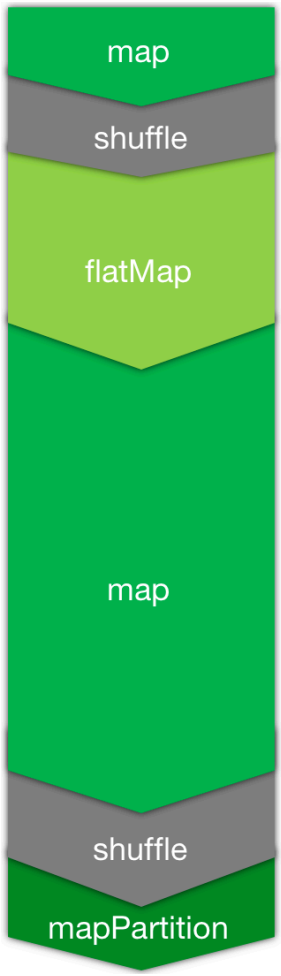
RP : array of reverse prefix of each suffix in ED;

IRR : array of irreducible edges;

Algorithm

The pseudo code of GraphSeq is shown with the corresponding stages on Spark in Fig. S2. Since GraphSeq support to remove duplicates before constructing string graph, we leverage RDD.distinct() to achieve the task and introduce data shuffling in this stage. We know the data shuffling is the major

bottleneck of performance on Spark. Therefore, there is only another data shuffling encountered when grouping all suffixes with the same prefix string together. Furthermore, so many small data elements are generated when suffix expansion. The bucket mechanism is adopted to reduce the number of elements for data shuffling.



DS_r = load reads from parquet files in parallel;

DS_r.distinct() if enable(RMDUP)

foreach read $r_k \in DS_r$
  assign unique ID to r_k and create reverse complement r_k^{rc};
  add r_k and r_k^{rc} into R

foreach i = 0..4^{NB}
  //initialize bucket
  foreach j = 0..4^{NP}
    bucket(j) = new ArrayByte[BS * MAX_RL]
  //suffix generation
  foreach read $r_k \in R$
    for idx = 0 ..(length(r_k) – MIN_OL - 1)
      (bid,pid) = checkSuffix(r_k, idx, NB, NP)
      if (bid == i)
        append the idx-th suffix of r_k into bucket(pid)

//partitionBy(pid)

stringGraphBuild()

Fig. S2. Pseudo code of GraphSeq.

When all of suffixes with the same prefix string are collected into a partition, we are able to literate each suffix from the data buckets one-by-one. The following function **stringGraphBuild()** is to unpack all of suffixes from data buckets, sort them, identify longest common substring, sort them again by special order and then apply irreducible edge identification in **edgeIdentification()**.

```
def stringGraphBuild(bucket: Dataset) {
  SA = ∅
  foreach b in bucket
    for j=0..length(b)
      add b[j] into SA
```

```
  sort SA
  SSA = ∅
  foreach s in SA
    identify longest common substring (LCS)
    if s is candidate for string graph and LCS(s) > MIN_OL
      add s with its' LCS into SSA
  sort SSA by $ < (LCS) < A < C < G < T
  edgeIdentification(SSA)
}
```

In **edgeIdentification()**, all of suffixes matched the criterion of edges of string graph will be collected as input. Since those suffixes are sorted by alphabetical order, those suffixes completely overlapped from the start of any read are located before it. The threshold of length of overlapping is configurable. Therefore, we have to go backwards to identify those qualified suffixes for each read. After that, those qualified suffixes will be sorted by their prefix sequences (just like the prefix mentioned in Fig. S3). Then, transitive edge reduction can be applied in the function **transitiveEdgeReduction()**.

```
def edgeIdentification(SSA: array) {
  j = len(SSA)
  while (j > 0)
    if the index of SSA[j] is 0
      ED = ∅
      k = j - 1
      while (SSA[k] is prefix of SSA[j])
        add SSA[k] into ED
        k--
      RP = reverse prefix of ED
      sort RP
      IRR = transitiveEdgeReduction(RP)
      output IRR
    j--
}
```

In **transitiveEdgeReduction()**, the irreducible edges can be identified by comparing their prefix sequence. If an edge is transitive, its' prefix should be covered by the shorter prefix sequence of other irreducible edge and vice versa. Using the characteristic, the algorithm of string graph construction is entirely fitted into the parallelization framework of Spark.

```
def transitiveEdgeReduction(RP: array) {
  IRR = ∅
  prev = first of RP
  add prev in IRR
  prev_len = prev.length
  foreach (curr <- prefixes)
    if (curr.length < prev_len)
       add curr in IRR
      prev_len = curr.length
    else
      cmp = strncmp(prev, curr, prev_len)
      if (cmp != prev_len)
        add curr in IRR
        prev_len = curr.length
      else if ((cmp == prev.lengh) &&(prev.length == curr.length))
        add curr in IRR
        prev_len = curr.length
    prev = curr
  return IRR
}
```

Data Structure

Since comparing any two reads by their suffixes and prefixes is required in our algorithm, the read is encoded by the following byte-aligned format in Fig. S3. The encoding method allows us to not only reduced memory consumption but also speedup the comparing function.

Assume read length as 22 and take the read at Partition 0 (AAA) for example

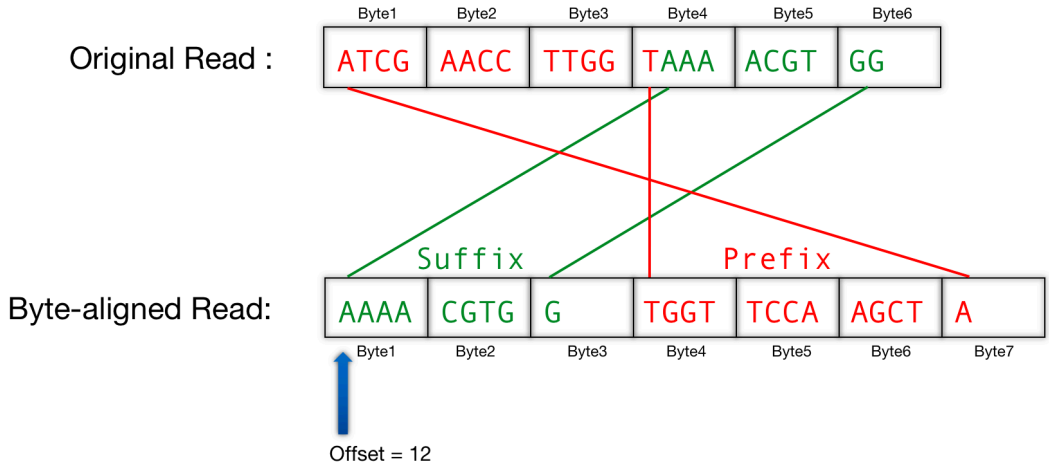


Fig. S3. Data structure of byte-aligned compression.

Experiment

Data Preparation

The WGS data is downloaded from http://s3-us-west-2.amazonaws.com/10x.files/samples/genome/2.0.0/NA12878_WGS/NA12878_WGS_fastqs.tar. We follow the best practice of SGA to prepare the qualified reads for string graph construction. The preprocessing flow is illustrated in Fig. S4.

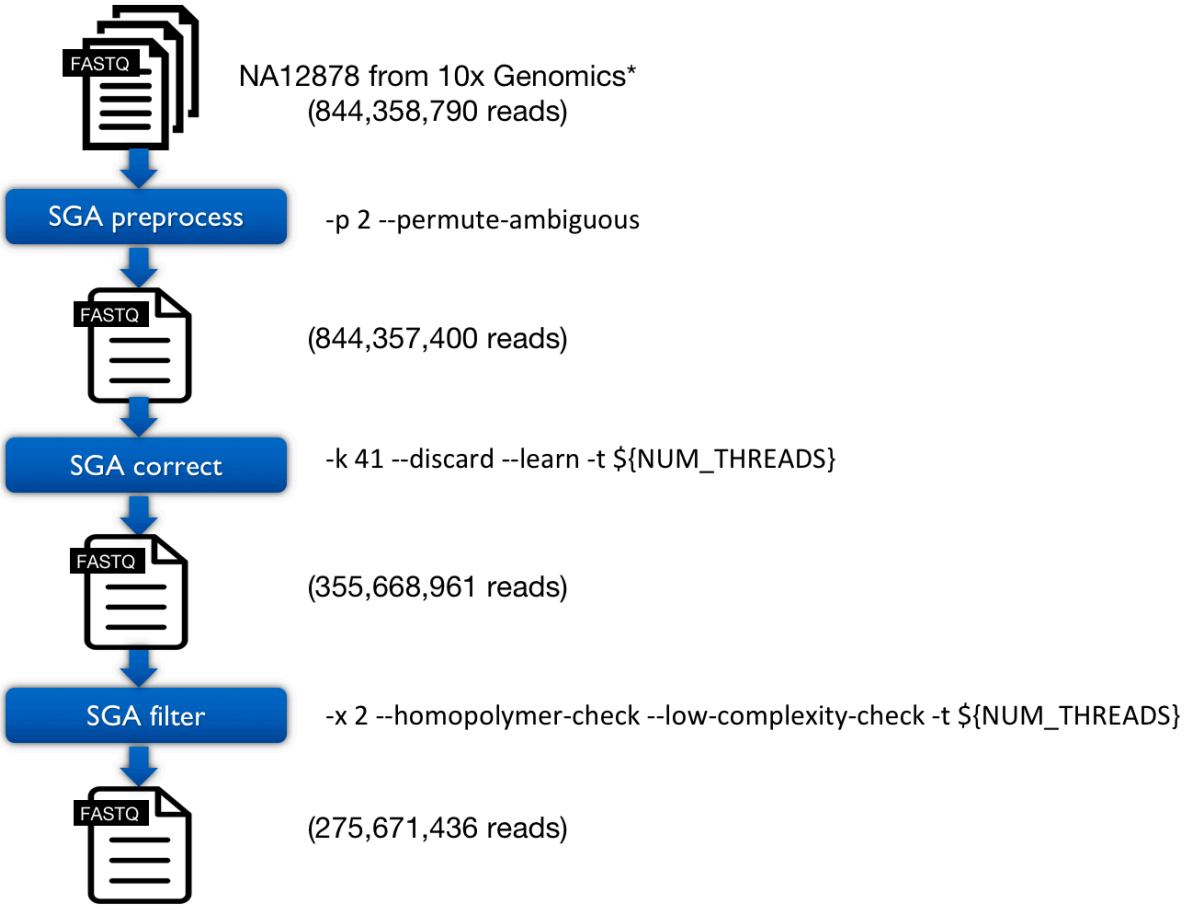


Fig. S4. Data preparation by using SGA.

Discussion

In Fig. S5, the long-tail distribution is observed when showing the number of suffixes in partition. It means that the execution time will be bounded by some partitions with huge number of suffixes. For example, Fig. S5 is applied by (NB, NP) = (1, 7) and the total of partitions is $4^{(1+7)}$ = 65,536. The maximum, minimum and average size is 42061542, 3712 and 563659, respectively.

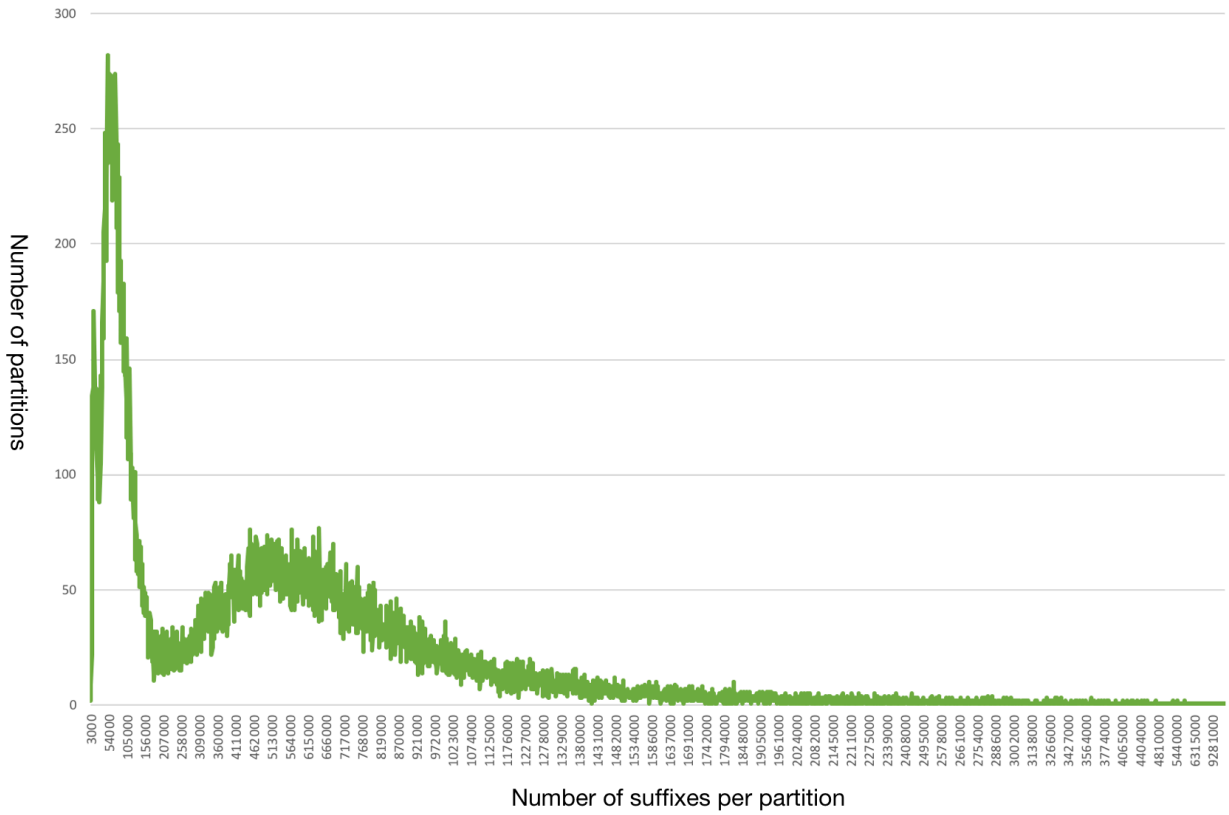


Fig. S5. Distribution of suffixes per partition.

In Fig. S6, the long-tail distribution is also observed when showing the number of edges in partition. means that the execution time will be bounded by some partitions with huge number of irreducible edges. For example, Fig. S6 is applied by (NB, NP) = (1, 7) and the total of partitions is $4^{(1+7)}$ = 65,536. The maximum, minimum and average size is 97921678, 14 and 20795, respectively. The effect of long tail distribution is even worse.

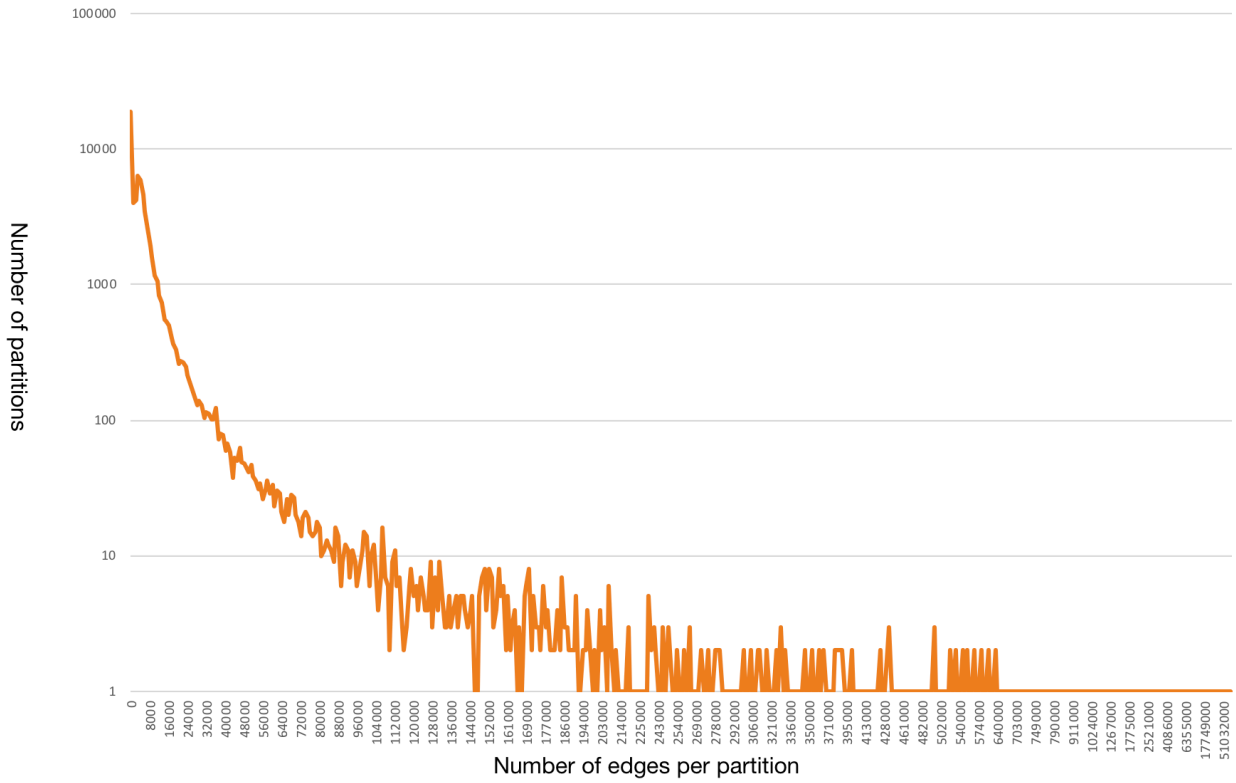


Fig. S6. Distribution of irreducible edges per partition.

To discuss the insignificant improvement when increasing number of cores to 128 in Fig. 2, Table S2 show the total execution time and the time for the longest task by batch. The reason is the total execution time is bounded by the longest task. According to Table S2, the longest task takes more than 75% of batch time in 128 cores comparing to 50% in 64 cores. If we have unlimited cores, the execution time is bounded in the total time of the longest task of each batch. However, GraphSeq provide the parameters NB and NP to adjust the number of parallelization and the data size of each partition for customization.

Table. S2. Performance Profiling.

#cores	Batch0		Batch1		Batch2		Batch3		Total		%
	batch time	longest task	batch time	longest task	batch time	longest task	batch time	longest task	batch time	longest task	
16	16	3.8	12	2.1	13	1.8	17	1.6	58	9.3	16.0
32	8.1	3.9	5.4	2.1	6.4	1.7	7.8	2	27.7	9.7	35.0
64	6.2	3.6	3.6	2.2	3.9	1.7	4.6	1.7	18.3	9.2	50.3
128	5.3	4.4	3.6	2.6	3.1	2.2	3.7	2.8	15.7	12	76.4

For example, Fig. S7 shows the relation between the execution time and memory requirement. If hardware resource is limited in memory, NB of GraphSeq should be larger and the consequence is longer execution time. If lots of CPU cores are available, NP of GraphSeq should be larger to get more benefit on parallelization.

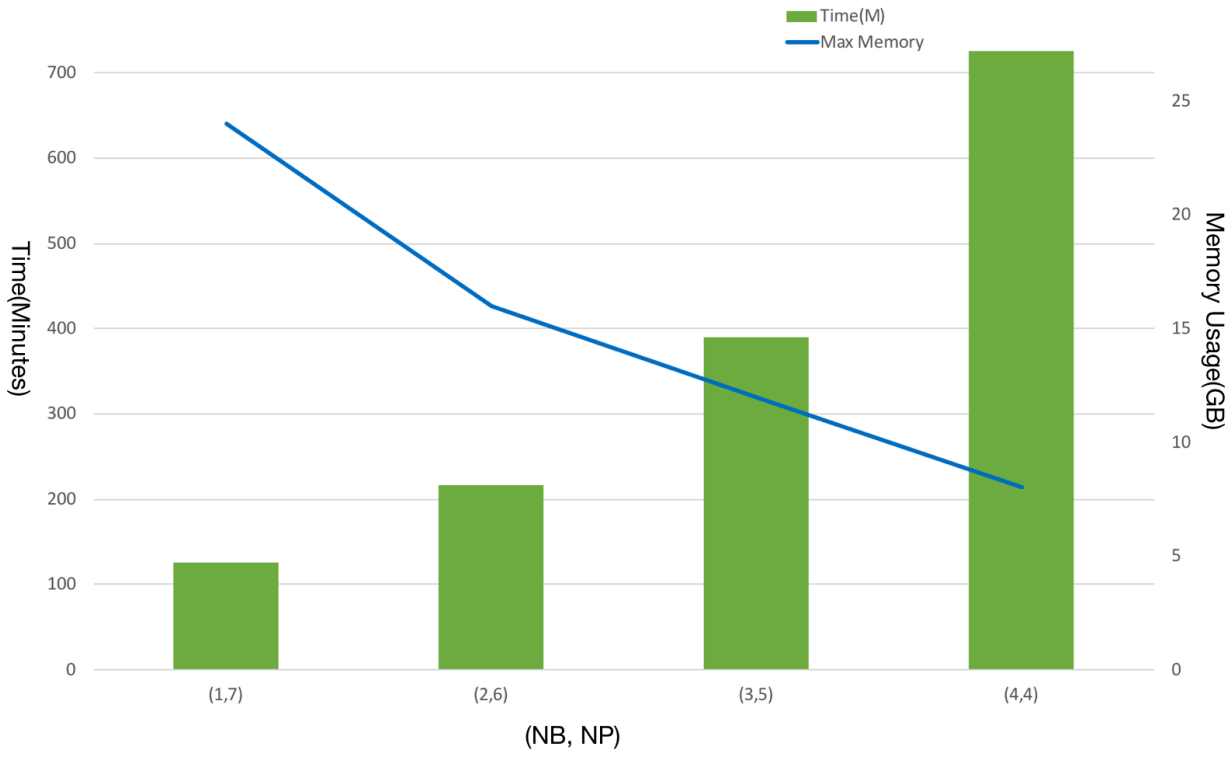


Fig. S7. Memory usage and performance in different configuration.

Programs and Toolsets

The following session is the commands used in the paper.

GraphSeq

```
# /usr/local/spark/bin/spark-submit   --master spark://XXX:7077   --class
com.atgenomix.seqslab.cli.SparkSTMain   /src/graphseq-1.0.0.jar overlap
    INPUT           : Input path (generated by Adam transform)
    OUTPUT          : Output path
    -cache          : Cache the reads in memory to speedup data processing
```

```
    -h (-help, --help, -?) : Print help
    -max_edges N      : Maximal number of edges per read [default =
    Integer.MAX_VALUE]
    -max_read_length N: Maximal read length [default = 151]
    -mlcp N           : Minimal longest common prefix [default = 45]
    -packing_size N    : The number of reads will be packed together [default
= 100]
    -pl_batch N       : Prefix length for number of batches [default=1]
    -pl_partition N   : Prefix length for number of partitions [default=7]
    -print_metrics    : Print metrics to the log on completion
    -profiling        : Enable performance profiling and output to
$OUTPUT/STATS
    -rmdup            : Remove duplication of reads
    -stats            : Enable to output statistics of String Graph to
$OUTPUT/STATS
```

ADAM

```
/usr/local/spark/bin/spark-submit --master spark://graphseq-master-
portal:7077 --class org.bdgenomics.adam.cli.ADAMMain  --driver-cores 1   --
driver-memory 1g   --num-executors 8  --executor-memory 50g
/usr/local/seqslab/adam/adam-assembly/target/adam-assembly-spark2_2.11-
0.24.0-SNAPSHOT.jar  transformAlignments -force_load_fastq -
parquet_compression_codec SNAPPY  /10x_NA12878  /full.adam
```

SGA

```
#!/bin/bash

TMPFILE=NA12878
FOLDER="NA12878_WGS_fastqs"
OUTPUT_FOLDER="out"

INFILES=""
```

```
for entry in `ls ${FOLDER}/read-RA*.gz`
do
  INFILES="$INFILES /$entry"
done

SGA_BIN=./sga/src/SGA/sga
NUM_THREADS=32
BWA_BIN=./bwa/bwa
SAMTOOLS_BIN=./samtools

#preprocess
$SGA_BIN preprocess -p 2 --permute-ambiguous -o
$FOLDER/${OUTPUT_FOLDER}/$TMPFILE.fq.gz $INFILES

#index
$SGA_BIN index -a ropebwt --no-reverse -t ${NUM_THREADS}
$FOLDER/${OUTPUT_FOLDER}/$TMPFILE.fq.gz

#correct
$SGA_BIN correct -k 41 --discard --learn -t ${NUM_THREADS} -o
$FOLDER/${OUTPUT_FOLDER}/${TMPFILE}_ec.gz
$FOLDER/${OUTPUT_FOLDER}/$TMPFILE.fq.gz

#index
$SGA_BIN index -a ropebwt -t ${NUM_THREADS}
$FOLDER/${OUTPUT_FOLDER}/${TMPFILE}_ec.gz

#filter
$SGA_BIN filter -x 2 --homopolymer-check --low-complexity-check -t
${NUM_THREADS} -o $FOLDER/${OUTPUT_FOLDER}/${TMPFILE}_ft.gz
$FOLDER/${OUTPUT_FOLDER}/${TMPFILE}_ec.gz

#index
$SGA_BIN index -a ropebwt -t ${NUM_THREADS}
$FOLDER/${OUTPUT_FOLDER}/${TMPFILE}_ft.gz
```

```
#overlap
$SGA_BIN overlap -m 85 -t ${NUM_THREADS}
$FOLDER/${OUTPUT_FOLDER}/${TMPFILE}_ft.gz
```

FASTQ Splitter

```
import sys
import zlib
import subprocess
import os
import uuid
import shutil
import io
import logging
import traceback

logger = logging.getLogger('chunkwise_logger')


# constant
################################################

SNZIP = '/usr/local/bin/snzip'
HADOOP = '/usr/local/hadoop/bin/hadoop'
TMP_DIR = '/tmp'
BUFFER_READ_SIZE = 16384
GZ_SPLIT_CHUNK_SIZE = 256 * 1024 * 1024
SUBPROCESS_NUM = 3
PLAIN_TEXT_LENGTH_LIMIT = 600 * 1024 * 1024
HDFS_NAME_TEMPLATE = '{}/chunk_{}.fastq.snappy'
FS_NAME_TEMPLATE = '{}/tmp_chunk-{}.fq'
FS_NAME_PAIRED_TEMPLATE = '{}/tmp_chunk-{}-{}.fq'
```

```
CONTENT_BUFFER_SIZE = 10240
SNZIP_N_UPLOAD_RETRY = 20
PAIRED_TWO_SEARCH_SCOPE = 1 * 1024 * 1024
PAIR_TWO_SEARCH_BUF_SIZE = 10 * 1024

# class
################################################


class DatasetsUploadError(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return self.value


class GzChunkFileWrapper:
    def __init__(self, source, is_dir=True,
input_chunk_size=GZ_SPLIT_CHUNK_SIZE, handle_buff_size=1024 * 1024):
        self.decompressor = zlib.decompressobj(zlib.MAX_WBITS | 16)
        self.input_chunk_size = input_chunk_size
        self.index = 0
        self.source = source
        self.is_dir = is_dir

        if not os.path.isdir(self.source) and is_dir:
            logger.error("provided source not a directory {} when
is_dir==True".format(self.source))
            raise Exception("provided source not a directory {} when
is_dir==True".format(self.source))
        if os.path.isdir(self.source) and not is_dir:
            logger.error("provided source not a file {} when
is_dir!=True".format(self.source))
```

```
            raise Exception("provided source not a file {} when
is_dir!=True".format(self.source))

        self.path_list, self.path_list_size, self.list_last_index =
self.init_input_file_list()
        self.source_handle, self.buff, self.handle =
self.init_file_handle(self.input_chunk_size, handle_buff_size)
        logger.info('path_list {}; path_list_length {}'.format(self.path_list,
self.path_list_size))

    def __del__(self):
        self.buff.close()
        self.handle.close()

    def init_input_file_list(self):
        if self.is_dir:
            tmp = [f for f in os.listdir(self.source)]
            logger.debug('input chunk list -- {}'.format(tmp))
            tmp.sort(key=int)
            chunk_list = ["{}/{}".format(self.source, f) for f in tmp]
        else:
            chunk_list = [self.source]
        return chunk_list, len(chunk_list), len(chunk_list) - 1

    def init_file_handle(self, chunk_size, handle_buff_size):
        f = open(self.path_list[self.index], "rb")
        buff = io.BytesIO(f.read(chunk_size))
        return f, buff, io.BufferedReader(buff, handle_buff_size)

    def read(self, length=BUFFER_READ_SIZE):
        try:
            total = 0
            buffer = self.handle.read(length)
            # return length < read length => self.buff is consumed, and need to
load next chunk to self.buff
```

```
            if len(buffer) < length:
                logger.debug('buffer.len -- {}'.format(len(buffer)))
                self.append_next_chunk()
                buffer += self.handle.read(length)

            # decompress buffer (gz.compressed) to outstr (binary string)
            outstr = self.decompressor.decompress(buffer)
            total += len(outstr)

            # loop through all the unused_data, and append decompressed portion
into outstr
            while self.decompressor.unused_data != b'':
                unused_data = self.decompressor.unused_data
                self.decompressor = zlib.decompressobj(zlib.MAX_WBITS | 16)
                tmp = self.decompressor.decompress(unused_data)
                total += len(tmp)
                outstr += tmp
        except:
            logger.error('input decompress error --
{}'.format(traceback.format_exc()))
            raise DatasetsUploadError('input decompress error --
{}'.format(traceback.format_exc()))

        return len(outstr) == 0, outstr

    def append_next_chunk(self):
        # check whether source_handle has been consumed
        if self.index > self.list_last_index:
            return

        content = self.source_handle.read(self.input_chunk_size)

        if not content:
            # remove consumed chunk file
            self.source_handle.close()
```

```
            if self.is_dir:
                os.unlink(self.path_list[self.index])

            # about to append next chunk
            self.index += 1

            if self.index > self.list_last_index:
                return

            self.source_handle = open(self.path_list[self.index], 'rb')

            # append chunk routine
            remain_buf = self.buff.read(sys.getsizeof(self.buff) -
self.buff.tell())
            self.buff.truncate(0)
            self.buff.seek(0)
            self.buff.write(remain_buf)
            self.buff.write(self.source_handle.read(self.input_chunk_size))
            logger.debug("__appended chunk No. {}, with buf length --
{}".format(self.index, sys.getsizeof(self.buff)))
            self.buff.seek(0)
        else:
            remain_buf = self.buff.read(sys.getsizeof(self.buff) -
self.buff.tell())
            self.buff.truncate(0)
            self.buff.seek(0)
            self.buff.write(remain_buf)
            self.buff.write(content)
            logger.debug("appending next source file content, with buf length -
- {}".format(sys.getsizeof(self.buff)))
            self.buff.seek(0)


class FastqUploader:
    def __init__(self, file_dir_list, dest_path,
```

```
                fastq_txt_chunk_size=PLAIN_TEXT_LENGTH_LIMIT,
                input_chunk_size=GZ_SPLIT_CHUNK_SIZE):
        self.upload_retried_count = 0
        self.source_handle = []
        logger.info(file_dir_list)
        logger.info(dest_path)
        for item in file_dir_list:
            if os.path.isdir(item):
                self.source_handle.append(GzChunkFileWrapper(item, True,
input_chunk_size))
            elif os.path.isfile(item):
                self.source_handle.append(GzChunkFileWrapper(item, False,
input_chunk_size))
            else:
                logger.error('input source {} is not a directory nor a
file'.format(item))
                raise DatasetsUploadError('input source {} is not a directory
nor a file'.format(item))

        self.result_dest = dest_path
        self.tmp_dir = '{}/{}'.format(TMP_DIR, uuid.uuid4())
        try:
            os.mkdir(self.tmp_dir)
        except FileExistsError:
            logger.error("{} -- already exist".format(self.tmp_dir))
        self.init_dest_dir()
        # for single-end file, self.fastq_txt_chunk_size = fastq_txt_chunk_size
        self.fastq_txt_chunk_size = int(fastq_txt_chunk_size /
len(self.source_handle))
        logger.debug('source dir: {}, dest dir: {}, chunk fastq plain text size
{}'.format(
            file_dir_list, format(dest_path), self.fastq_txt_chunk_size))

    def init_dest_dir(self):
```

```
        return_code = subprocess.call([HADOOP, 'fs', '-test', '-e',
self.result_dest])
        if return_code != 0:
            subprocess.check_call([HADOOP, 'fs', '-mkdir', self.result_dest])
        logger.info('result dest dir -- {}'.format(self.result_dest))

    @staticmethod
    def find_split_point(shortfall, content):
        content_size = len(content)
        idx1 = shortfall
        while idx1 < content_size:
            if content[idx1] == ord('\n') and content[idx1 + 1] == ord('@'):
                idx2 = idx1 + 1
                while idx2 < content_size:
                    if content[idx2] == ord('\n'):
                        if content[idx2 + 1] == ord('@'):
                            return idx2 + 1
                        else:
                            return idx1 + 1
                    idx2 += 1
            idx1 += 1
        logger.error("cannot find proper position for partitioning fastq
files")
        return -1

    def process_file(self, chunk_file_path, chunk_index, pid):
        # compress and upload file
        cmd = '{0} -k -t hadoop-snappy {1} && {4} fs -put {2} {3}'.format(
            SNZIP,
            chunk_file_path,
            '{}.snappy'.format(chunk_file_path,),
            HDFS_NAME_TEMPLATE.format(self.result_dest,
str(chunk_index).zfill(5)),
            HADOOP)
        logger.debug(cmd)
```

```
        pid.append((subprocess.Popen(cmd, shell=True), cmd, [chunk_file_path,
'{}.snappy'.format(chunk_file_path)]))
        if len(pid) > SUBPROCESS_NUM:
            return self.join_subprocess(pid)
        return pid

    def join_subprocess(self, pid):
        retry_pid = []
        for item in pid:
            return_code = item[0].wait()
            if return_code != 0:
                if self.upload_retried_count < SNZIP_N_UPLOAD_RETRY:
                    self.upload_retried_count += 1
                    logger.error('retry {} with command
{}'.format(self.upload_retried_count, item[1]))
                    retry_pid.append((subprocess.Popen(item[1], shell=True),
item[1], item[2]))
                else:
                    logger.error('compress process error failed {}
time'.format(SNZIP_N_UPLOAD_RETRY))
                    raise DatasetsUploadError('compress process error failed {}
time'.format(SNZIP_N_UPLOAD_RETRY))
            else:
                for f in item[2]:
                    try:
                        os.remove(f)
                    except:
                        logger.error('failed to remove {}'.format(f))
        pid = retry_pid
        return pid

    def run(self):
        try:
            chunk_index = 0
            written_length = 0
```

```
          chunk_file_path = FS_NAME_TEMPLATE.format(self.tmp_dir,
chunk_index)
          chunk_f = open(chunk_file_path, 'wb')
          pid = []
          while True:
              eof, content = self.source_handle[-1].read()
              # shortfall_length = self.fastq_txt_chunk_size - written_length
              # len(content) > shortfall_length => reach current chunk length
threshold, and find the
              # chunk-segmentation at the start of the 1st read right after
content[:shortfall_length]
              shortfall_length = self.fastq_txt_chunk_size - written_length

              # CONTENT_BUFFER_SIZE is a buffer size of content, which is set
to be 10240,
              # to guarantee that content is at least 10240 characters longer
than shortfall_length for the following
              # split point search operation.
              # CONTENT_BUFFER_SIZE should be configurable based on size of
reads.
              # For Illumina short reads scenario, CONTENT_BUFFER_SIZE can be
set to 512 since size of Illumina
              # short reads rarely exceeds 300 byte.
              if len(content) > shortfall_length + CONTENT_BUFFER_SIZE:
                  if shortfall_length < 0:
                      shortfall_length = 0
                  split_point = self.find_split_point(shortfall_length,
content)
                  written_length += split_point
                  chunk_f.write(content[:split_point])
                  logger.debug('done {} writing, file length
{}'.format(chunk_file_path, written_length))
                  chunk_f.close()
                  pid = self.process_file(chunk_file_path, chunk_index, pid)
```

```
                chunk_index += 1
                chunk_file_path = FS_NAME_TEMPLATE.format(self.tmp_dir,
chunk_index)
                chunk_f = open(chunk_file_path, 'wb')
                chunk_f.write(content[split_point:])
                written_length = len(content) - split_point

            else:
                written_length += len(content)
                chunk_f.write(content)

            if eof:
                # eof chunk writing
                written_length += len(content)
                chunk_f.write(content)
                logger.debug('done eof chunk - {} writing, file length
{}'.format(chunk_file_path, written_length))
                chunk_f.close()
                self.process_file(chunk_file_path, chunk_index, pid)
                self.join_subprocess(pid)
                break

        finally:
            pass
            # clean tmp files and source chunks
            shutil.rmtree(self.tmp_dir, ignore_errors=True)


def main(args):
    uploader = FastqUploader([sys.argv[1]], sys.argv[2])
    uploader.run()


if __name__ == '__main__':
    main(sys.argv)
```