Arioc User Guide

written by
Richard Wilton

richard.wilton@jhu.edu

in collaboration with
Tamás Budavári
Ben Langmead
Sarah Wheelan
Steven L. Salzberg
Alexander S. Szalay

Copyright and other notices

Arioc User Guide

Introduction

Arioc is a set of computer programs that carry out the alignment of short DNA sequences to a comparatively large reference sequence or genome.

In the field of biological sequence alignment, there have been numerous attempts to use GPU (general-purpose graphics processing unit) hardware to accelerate successful CPU-based algorithms and implementations. A decade of such efforts has demonstrated that this is not easy. Arioc is interesting because it was designed and engineered to use GPU hardware to maximize the speed of DNA sequence alignment without sacrificing accuracy.

GPU acceleration of short read alignment

GPU programming requires software-development techniques that aren't needed for programs that run exclusively on a CPU. GPU hardware can run tens of thousands of threads of execution concurrently, but speed improvements in GPU software are not obtained simply by replicating source code that runs efficiently on one or two dozen CPU threads.

There are two fundamental reasons for this. One has to do with the way GPUs manage threads. In Nvidia's CUDA environment (for which Arioc is implemented), threads execute in fixed groups of 32, in which each thread simultaneously executes the same instruction on different data. This SIMD (single instruction, multiple data) model influences the layout of data in memory as well as the structure of source code, so algorithms that operate efficiently in CPU threads are often suboptimal when executed in SIMD threads on a GPU. Algorithms that contain a great deal of branching logic must be carefully redesigned to accommodate SIMD parallelism.

The second, and perhaps more important, reason why GPU application speeds don't scale in direct proportion to the number of available GPU threads is that there isn't enough memory bandwidth to keep up with all of those threads. GPU programming involves writing code that conforms to a variety of memory-layout and memory-addressing constraints that can fundamentally change assumptions that are taken for granted in algorithms that run efficiently on CPUs. Of course, memory optimization techniques are important in CPU software engineering as well, but the idiosyncrasies of GPU memory management tend to dominate other considerations in building a successful GPU application.

This all means that, in practice, a 10-fold speedup in a GPU-based sequence-alignment application (compared to a multi-threaded CPU-based implementation that performs the same computational task) represents a reasonable "return on investment" [Anderson, 2011]. (A hundred-fold GPU-versus-CPU speedup would be extraordinary and make one wonder whether the corresponding CPU application had been properly optimized!) As far as Arioc is concerned, we did not release the software until we reached this 10x threshold.

How Arioc is different

The central problem in read alignment is essentially that of quantifying the similarity between two strings of symbols. This problem has attracted a great deal of attention since the early 1980s, and several successful algorithms have been developed in response. Of these, two dynamic-programming techniques — one using weighted scores for mismatches and gaps [Smith and Waterman, 1981] and the

other using Levenshtein edit distance [Ukkonen, 1983] — are by far the most commonly used algorithms in modern read aligners. Arioc uses the Smith-Waterman algorithm, which consistently finds better mappings for reads that contain multiple insertions and deletions [Wilton et al, 2015], even though it requires more computational effort than edit-distance alignment.

Unfortunately, these string similarity computations are time-consuming and use a large amount of memory. With a reference sequence the size of the human genome, it is impractical to align a read by computing similarity at every possible reference-sequence location. For this reason, read aligners perform a great deal of preliminary work to narrow down the number of reference-sequence locations at which they must compute alignments. This work cannot easily be reduced to a single algorithmic description. Instead, read aligners rely on heuristics and on software-engineering considerations to limit the number of reference-sequence locations they examine (and ultimately the number of string-similarity computations they perform) for each read.

It is here that GPU-based parallelism can greatly improve performance. Arioc implements a well-known technique known as "seed and extend" [Altschul, 1992] using parallel sorting and adjacent-neighbor comparisons on long, one-dimensional arrays of integers. Such operations are well suited to SIMD implementation on GPU hardware. GPU acceleration of this aspect of the read-alignment process is crucial to Arioc's speed and accuracy.

What Arioc is (and isn't)

Arioc is a read aligner that uses GPU acceleration in an interesting way, namely, to find high-priority locations at which to compute alignments. The process of identifying high-priority alignment locations can be a significant bottleneck in a read aligner. Arioc uses GPU hardware to speed up this process and thereby achieve higher throughput [Wilton et al, 2015].

What Arioc does

Arioc accepts as input a large number — typically, hundreds of millions — of short sequencer reads. For each such read, Arioc reports the location (or locations) within a given reference sequence where a highly-similar subsequence is found.

Ideally, Arioc would find a perfect and unique mapping for every read; there would be exactly one location in the reference sequence where each read sequence precisely matches the reference sequence. In practice, of course, many reads do not map perfectly or uniquely within a given reference sequence. For such reads, Arioc reports what it considers to be the "best" mapping (or mappings). Arioc does these things accurately and with high throughput.

Nongapped and gapped alignment

For each sequencer read, Arioc first tries to find alignments that contain no gaps (insertions or deletions) and no more than a few differences between the read sequence and the reference sequence. The "spaced seed" algorithm that Arioc uses for these alignments is very fast, and Arioc's speed is greatly improved by identifying nongapped mappings prior to computing Smith-Waterman alignments.

Arioc computes a valid alignment score (AS) and mapping quality score (MAPQ) for every mapping it finds. Arioc's output can thus be used by any software tool that can handle alignment results from CPU-only aligners such as BWA [Li, 2013] or Bowtie 2 [Langmead and Salzberg, 2012].

Input and output data formats

Arioc expects FASTA-formatted reference sequences and FASTQ-formatted sequencer reads. It generates SAM-formatted output.

Arioc has several "database-oriented" features in its implementation. One of Arioc's original design goals was to carry out read alignment on sequence data residing in relational database tables. For this reason, Arioc can create output files in the binary data format used by the Microsoft SQL Server for high-speed "bulk" input.

Arioc was designed to handle large amounts of sequencer data and genomes the size of the human reference genome. We remembered the notion of "scarcity thinking" [Cooper, 2004] and made a conscious effort to avoid it! If you are processing sequencer reads on a terabyte or petabyte scale, we assume that you have computers with the compute and memory resources required by Arioc. We also provide a variety of parameterization options that let you tune Arioc's performance to your specific hardware and data-analysis requirements. (See Tuning for speed and sensitivity, page 28.)

What Arioc does not do

Arioc is not a tool for searching a database of short sequences and ranking those sequences by their similarity to a given sequence. (CUDASW++ [Liu et al, 2013] is the best-known GPU-accelerated tool for this.) Arioc does not operate with any sequences other than DNA; it does not understand any sequence "alphabet" other than ACGTN. It does not report SNPs or recognize structural variations. It does not do multiple sequence alignment or assemble sequencer reads in any way.

Finally, Arioc is not by any means a "simple", "turnkey", or "one size fits all" tool. Instead, Arioc exposes a variety of user-configurable parameters that let you experiment to optimize speed and sensitivity (or trade one for the other). Although this approach exposes the complexity of the read-alignment process, it provides the ability to fine-tune Arioc's performance to high-performance computing hardware and to the specific characteristics of sequencer read data.

The GPU software-development landscape continues to evolve. Since its initial release, Arioc has grown and adapted to newer, faster GPU hardware and software libraries. We think that Arioc's approach to the read-alignment problem is sound, and that it has the potential to prove its utility as sequencers generate ever-increasing numbers of reads to align.

Installation

Arioc is written in C++ and compiled for the Nvidia CUDA environment. The same code base is used for both Windows and Linux.

Hardware requirements

Arioc requires the following minimum hardware:

- a multi-core 64-bit Intel- or PowerPC-compatible CPU
- at least 80 gigabytes of free memory; more memory required for larger reference genomes
- sufficient disk space to store reference sequences, reads, and alignment results
- at least one Nvidia GPU of compute capability 3.5 or greater, with at least 5 gigabytes of GPU memory

We have tested Arioc on the following Nvidia GPU microarchitectures:

- Kepler
- Maxwell
- Pascal
- Volta

CUDA

Arioc has been tested with the following minimum CUDA versions:

- Nvidia video driver v354.56
- CUDA runtime version 7.5

We recommend using the most recent stable versions of these products to run Arioc.

Operating systems

Arioc requires a 64-bit operating system. It has been tested with the following minimum operating-system versions:

- Windows 7 (64-bit)
- Windows Server 2008 R2 datacenter
- RHEL Scientific Linux v7.2

Arioc should be compatible with more recent versions of these operating systems.

Installation

Windows

Install using the Windows installer and the Arioc setup package (`AriocSetup.msi`). For source code only, unzip the `Arioc.w.140.zip` archive.

Linux

Unzip the `Arioc.x.140.zip` archive and build from source code.

Building from source code

The source code for is available in an archive whose name embodies the operating system and release version number. The sources for Windows release 1.40 are found in `Arioc.w.140.zip`; the Linux sources for the same release are in `Arioc.x.140.zip`.)

The same sources can be compiled for Windows and for Linux:

- In Windows, use Microsoft Visual Studio 2019 or later to load the Arioc solution (`Arioc.sln`). You may need to edit the associated `.vsproj` files to reference the version of the CUDA SDK installed in your development environment.
- In Linux, use GNU make and the makefile in the top-level directory of the source-code distribution. Build each Arioc binary separately:

```
make clean
make AriocE
make AriocU
make AriocP
```

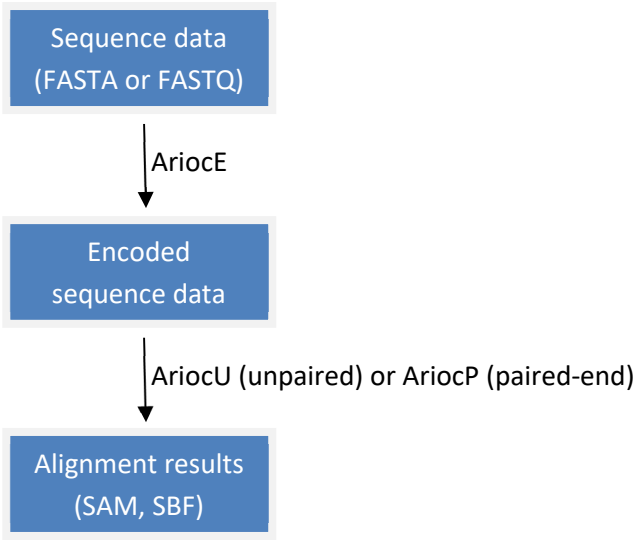
- You may verify your Arioc installation using the test data distributed in the `Arioc.RQA.140.zip` archive. The `readme.txt` file in that archive contains additional details.

Arioc runtime roadmap

There are three executable binaries in the Arioc distribution:

- AriocE: encodes sequence data
- AriocU: aligns unpaired reads
- AriocP: aligns paired-end reads

Here is how they are used:



The three executables are described in detail in the sections that follow.

Arioc command-line syntax

Each of the Arioc executables obtains its runtime configuration from an XML-formatted configuration file, so the command-line syntax for running them is simple:

```
AriocE config_filename
AriocU config_filename
AriocP config_filename
```

where `config_filename` is the name of the configuration file to use. (You should, of course, use fully-qualified filenames whenever it makes sense to do so.)

Arioc error messages

In general, Arioc error messages contain condensed information that identifies the source-code location that detected the error, along with human-readable text that explains the nature of the error. For example, if Arioc cannot open the configuration file you specify, it emits an error message like the following:

```
44:37.416 [00001704] ApplicationException ([0x5892] AriocAppMainBase.cpp 715):
XMLDocument::Loadfile( "AriocP.cfg" ) failed with error 3: file not found
XML error details:
AriocP.cfg
```

If Arioc produces an error message whose origin is not immediately apparent, you can usually find the cause of the problem by taking additional debugging and internal monitoring steps (see Troubleshooting, page 36).

Arioc input and output

Arioc writes status and error messages to the `stderr` device. In practice, this means that all output appears in the console by default. If you want to redirect console output to a file, use command-line redirection syntax. For example:

```
AriocU test23.cfg 2> test23.log
```

For sequence input and alignment results, Arioc uses the files you specify in the configuration (`.cfg`) file.

AriocE: encoding sequence data

The AriocE component transforms raw sequence data from FASTA or FASTQ format into the binary format read by the AriocU and AriocP executables. Encoded sequence data resides in separate files whose contents can be distinguished by filename extension:

- `$a21.sbf, $a21.rc.sbf`: encoded forward and reverse-complement bases (3 bits per base, 20 bases per 64-bit value)
- `$raw.sbf, $raw.rc.sbf`: ASCII representation of the sequence data (forward and reverse complement)
- `$sqm.sbf, $sqm.rc.sbf`: sequence metadata (FASTA or FASTQ description line)
- `$sqq.sbf, $sqq.rc.sbf`: sequence quality scores (for FASTQ-formatted input only)

For example, encoded sequence data for human chromosome 1 might reside in the following set of files:

```
hs_ref_GRCh38_chr1$a21.sbf
hs_ref_GRCh38_chr1$a21.rc.sbf
hs_ref_GRCh38_chr1$raw.sbf
hs_ref_GRCh38_chr1$raw.rc.sbf
hs_ref_GRCh38_chr1$sqm.sbf
hs_ref_GRCh38_chr1$sqm.rc.sbf
```

Before using the Arioc aligners, AriocU and AriocP, you must encode both a reference sequence and a set of reads to align. Encoding is a one-time-only procedure. You can re-use the encoded files for multiple invocations of either the AriocU or AriocP executables.

Encoding unpaired reads

Encoding reads with AriocE is straightforward. For each FASTA file, AriocE generates $a21, $raw, and $sqm files; for FASTQ input, AriocE generates a $sqq file as well.

The configuration file you provide to AriocE, in addition to specifying input and output filenames, contains "metadata" that is passed to the AriocU or AriocP aligners:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE>

  <dataIn sequenceType="Q">
    <file subId="1">C:\test103\i100p_1.fastq</file>
  </dataIn>

  <dataOut>
    <path>C:\BulkData\Q\Mason\test103</path>
  </dataOut>

</AriocE>
```

Simple configuration file for AriocE (unpaired reads).

In the XML configuration file, the `sequenceType=` attribute of the `<dataIn>` element indicates that the input sequences are to be encoded as reads. The `<dataIn>` element also contains one or more `<file>`

elements, each of which specifies the fully-qualified name of an input file. Similarly, the <dataOut> element contains a <path> element that contains the fully-qualified name of the output directory.

AriocE requires that you use the subId= attribute of each <file> element to assign a "subunit ID" number to each input file. (Think of the subunit ID as a chromosome number for a reference genome or a sample subset identifier for a set of reads.) The subunit ID can be any integer between 1 and 127. AriocE combines the subunit ID, the data-source ID (specified as srcId= in the <dataIn> element), and an ordinal to create a unique 64-bit integer sequence identifier for every input sequence in every file you specify.

Although it was originally intended to simplify the automation of the sequence-alignment process, the use of an XML configuration file provides flexibility in parameterization as well as an opportunity to include human-readable documentation alongside runtime parameters. Here is an example of a more complex configuration file:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE maxDOP="2">

  <dataIn sequenceType="Q" srcId="1" samplingRatio="0.1">
    <!-- The following rg element defines read-group attributes common to all files -->
    <rg CN="JHU PHA" DS="test 103" PL="ILLUMINA" />

    <!-- Each file is associated with different read group ID and SM attributes -->
    <file subId="1" ID="RG1" SM="sample 1: 11221">C:\test103\i100p_1.fastq</file>
    <file subId="2" ID="RG2" SM="sample 1: 11222">C:\test103\i100p_2.fastq</file>
  </dataIn>

  <dataOut>
    <path>C:\BulkData\Q\Mason\test103</path>
  </dataOut>

</AriocE>
```

Configuration file for AriocE (unpaired reads) with comments, optional parameters (XML attributes), and SAM-style read groups.

This configuration encodes two different files. It also specifies an optional parameter (maxDOP=) that controls the number of concurrent CPU threads used by AriocE; with a multi-core CPU and multiple input files, AriocE can encode multiple files concurrently and thereby run faster than it would on a single CPU thread. Finally, the samplingRatio= parameter causes AriocE to encode a randomly-sampled subset of the input reads.

This example also illustrates how AriocE associates read group information with reads. The read group associated with each file is the union of the SAM-conformant read group attributes (ID, CN, DS, PL, and SM) from the <rg> and <file> elements. (The <rg> element defines read group attributes that are common to all the files; the <file> element defines attributes specific to individual files.) AriocE copies the consolidated read group attributes for each file into an associated metadata file in the output directory. The Arioc aligner (AriocU or AriocP) subsequently uses these metadata files to generate appropriate @RG headers and RG tags in SAM-formatted output.

Here is a summary of the optional XML elements and attributes that control the runtime configuration of AriocE:

Element	Attribute	Description
<AriocE>	maxDOP	Number of concurrent CPU threads
<dataIn>	srcId	Data source ID (between 0 and 4,194,303)
	samplingRatio	Sampled fraction of reads encoded (maximum 1.0)
	QNAME	Encodes only the parenthesized fields in the FASTQ defline
	qualityScoreBias	Bias (offset) for encoding base quality scores (33 or 64)
<rg>	ID CN DS DT FO KS LB PG PI PL PU SM	SAM read group fields (common to all input files)
<file>	SN	SAM sequence name field ID
	ID CN DS DT FO KS LB PG PI PL PU SM	SAM read group fields (merged with fields in the <rg> element)

Optional elements and attributes in the AriocE configuration file for encoding unpaired reads.

Both the QNAME= attribute in the <dataIn> element and the ID= attribute in the <rg> element accept a parenthesized wildcard specification that parses subfields from the FASTQ defline associated with each read. This makes it possible to generate meaningful SAM-formatted QNAME and RG (read-group) identifiers using the information in the FASTQ defline for each read. For example, given a read with a CASAVA-formatted FASTQ defline like this [Illumina, 2015]:

```
ERR037901.323683141 509.8.45.13203.181161/1
```

you can specify `<dataIn ... QNAME="(*) *(/*)">` to cause AriocE to generate the QNAME string `ERR037901.323683141/1`. Similarly, you can use `<rg ... ID="* (*.*).">` to associate the read with a read group whose ID is `509.8`.

Encoding paired-end reads

The AriocE configuration file for paired-end reads uses the same parameters as the configuration file for unpaired reads, but it uses an additional XML attribute (mate=) to indicate the files in which mate 1 and mate 2 are found.

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE>

  <dataIn sequenceType="Q">
    <file subId="1" mate="1">C:\test103\paired\i100p_1.fastq</file>
    <file subId="1" mate="2">C:\test103\paired\i100p_2.fastq</file>
  </dataIn>

  <dataOut>
    <path> C:\BulkData\Q\Mason\test103</path>
  </dataOut>

</AriocE>
```

Simple configuration file for AriocE (paired-end reads).

Multiple input files

It is possible to encode multiple input files (or, more exactly, pairs of input files) by using the `subId=` and `mate=` attributes of the `<file>` element:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE maxDOP="6">

  <dataIn sequenceType="Q" srcId="110114" samplingRatio="0.01">
    <file subId="5" mate="1">E:\yh110114\s_5_1_sequence.txt</file>
    <file subId="5" mate="2">E:\yh110114\s_5_2_sequence.txt</file>
    <file subId="6" mate="1">E:\yh110114\s_6_1_sequence.txt</file>
    <file subId="6" mate="2">E:\yh110114\s_6_2_sequence.txt</file>
    <file subId="7" mate="1">E:\yh110114\s_7_1_sequence.txt</file>
    <file subId="7" mate="2">E:\yh110114\s_7_2_sequence.txt</file>
  </dataIn>

  <dataOut>
    <path>C:\BulkData\Q\yh110114</path>
  </dataOut>

</AriocE>
```

Configuration file for AriocE (paired-end reads) with multiple input files and optional parameters.

Nevertheless, you should exercise caution when you encode multiple input files (or pairs of input files) because AriocE cannot recognize different layouts among the input files. In particular, files that originate from different sequencer runs may have different defline layouts or may use a different bias to encode base quality scores (e.g., 33 for Sanger, 64 for Illumina v1.3 and later).

Encoding a reference sequence

Encoding a reference sequence or genome is like encoding reads in that AriocE creates binary files that encode the sequence data and metadata. For a reference sequence or genome, however, AriocE also builds a set of hash tables that are used in both AriocU and AriocP.

Input files

If you are encoding a genome that consists of two or more reference sequences (chromosomes or other genome subunits), place each reference sequence in a separate file prior to encoding. For example, to encode the human genome, use a separate input file for each chromosome rather than concatenating all the chromosome sequences into a single file.

AriocE recognizes reference sequence files in FASTA format only.

Two-pass encoding

AriocE builds hash tables in a straightforward manner: At every location in each reference, the encoder extracts a seed whose length and pattern is specified in the configuration file. A numerical hash function is applied to each seed and the reference-sequence location is appended to a list of such locations for the seed's hash value. Other configuration-file parameters control the number of bits in a hash value (and thus the size of the hash table) as well as the maximum number of reference-sequence locations recorded for a given hash value.

Internally, both the AriocU (unpaired) and AriocP (paired-end) aligners implement a pipeline in which nongapped alignments for each read are computed prior to gapped alignments. The aligners use different hash tables for nongapped (spaced-seed) alignment and for gapped (seed-and-extend) alignment. This means that you must execute AriocE twice to fully encode a reference sequence. The idea is to make it possible to use AriocE multiple times to create hash tables with different characteristics and then choose among those hash tables when you execute the AriocU or AriocP aligners.

Here is an example of an AriocE configuration file that builds a nongapped (spaced-seed) hash table for the human reference genome:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocE seed="ssi84_2_30" maxDOP="32" maxJ="*">

  <dataIn sequenceType="R" srcId="0" filePath="/dss004/R/GRCh38.p6" AS="GRCh38"
uriPath="ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA_000001405.21_GRCh38.p6/GCA_000001405.21_GR
Ch38.p6_assembly_structure/Primary_Assembly/assembled_chromosomes/FASTA">
    <file SN="chr1" subId="1">chr1.fna</file>
    <file SN="chr2" subId="2">chr2.fna</file>
    <file SN="chr3" subId="3">chr3.fna</file>
    <file SN="chr4" subId="4">chr4.fna</file>
    <file SN="chr5" subId="5">chr5.fna</file>
    <file SN="chr6" subId="6">chr6.fna</file>
    <file SN="chr7" subId="7">chr7.fna</file>
    <file SN="chr8" subId="8">chr8.fna</file>
    <file SN="chr9" subId="9">chr9.fna</file>
    <file SN="chr10" subId="10">chr10.fna</file>
    <file SN="chr11" subId="11">chr11.fna</file>
    <file SN="chr12" subId="12">chr12.fna</file>
    <file SN="chr13" subId="13">chr13.fna</file>
    <file SN="chr14" subId="14">chr14.fna</file>
    <file SN="chr15" subId="15">chr15.fna</file>
    <file SN="chr16" subId="16">chr16.fna</file>
    <file SN="chr17" subId="17">chr17.fna</file>
    <file SN="chr18" subId="18">chr18.fna</file>
    <file SN="chr19" subId="19">chr19.fna</file>
    <file SN="chr20" subId="20">chr20.fna</file>
    <file SN="chr21" subId="21">chr21.fna</file>
    <file SN="chr22" subId="22">chr22.fna</file>
    <file SN="chrX" subId="23">chrX.fna</file>
    <file SN="chrY" subId="24">chrY.fna</file>
    <file SN="chrMT" subId="25"
uriPath="ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCA_000001405.21_GRCh38.p6/GCA_000001405.21_GR
Ch38.p6_assembly_structure/non-nuclear/assembled_chromosomes/FASTA">chrMT.fna</file>
  </dataIn>

  <dataOut>
    <path>/dss004/BulkData/R/NCBI/GRCh38.p6</path>
  </dataOut>

</AriocE>
```

Configuration file for AriocE (reference sequences) for a nongapped (spaced-seed) hash table.

To build the corresponding gapped (spaced-seed) hash tables, change the parameters specified in the XML attributes of the <AriocE> element and re-execute AriocE:

```
<AriocE seed="hsi20_0_30" maxDOP="32" maxJ="200">
```

Here is a list of additional configuration-file parameters for reference-sequence encoding:

Element	Attribute	Description
<AriocE>	maxDOP	Number of concurrent CPU threads
	gpuMask	Bits corresponding to GPU devices
	maxJ	Maximum hash table bucket size ("big bucket" threshold)
	seed	One of the following values: ssi84_2_*bb* — Seed width: 84; Mismatches: 2; Hash bits: *bb* between 29 and 32 ssi84_2_*bb*_CT — Seed width: 84; Mismatches: 2; Hash bits: *bb* between 29 and 32 hsi*ww*_0_*bb* — Seed width: *ww* between 18 and 42; Mismatches: 0; Hash bits: *bb* between 29 and 32 hsi*ww*_0_*bb*_CT — Seed width: *ww* between 18 and 42; Mismatches: 0; Hash bits: *bb* between 29 and 32 Examples: seed="ssi84_2_30" seed="hsi21_0_32" seed="hsi25_0_31_CT"
<dataIn>	srcId	Data source ID (between 0 and 4,194,303)
	AS	SAM genome assembly identifier (common to all input files)
	filePath	Parent directory for files specified in <file> elements
	uriPath	Original URI for reference sequences (copied to the UR field in the @SQ records in SAM-formatted output)
<file>	SN	Reference-sequence subunit (chromosome) name (e.g., "chr1")
	AS	SAM genome assembly identifier (overrides AS in <dataIn>)
	uriPath	Original URI for reference sequence (overrides uriPath in <dataIn>)
	TP	Reference-sequence topology: "linear" (default) or "circular"

Optional elements and attributes in the AriocE configuration file for encoding reference sequences or a genome.

Most of these parameters are self-evident, but the seed= and maxJ= values deserve careful attention.

seed=. The seed= value for the nongapped hash table should generally be ssi84_2_30 (seed width 84, up to 2 mismatches, 30-bit hash values). For the gapped hash table, a seed width of 20 or 21 bits is

usually appropriate for WGS reads, whereas a 25-bit seed width is generally a better choice for bisulfite-treated (WGBS) reads.

For both WGS and WGBS reads, there is a tradeoff between hash table size and the number of hash-value bits: 29-bit hash values require a 2.5GB table and the table size doubles with each additional bit in the hash value. If enough memory is available, "wider" hash values and correspondingly larger hash tables may provide up to 5% additional speed and about 0.5% additional sensitivity.

`maxJ=`. Because it limits the number of reference-sequence locations associated with a seed, the `maxJ=` parameter trades speed for sensitivity during read alignment. For the nongapped hash table, however, this effect is barely noticeable. In this case, you may simply omit the `maxJ=` parameter so that the maximum size of a hash table bucket is unlimited.

In contrast, the value of `maxJ=` has a tangible effect on the sensitivity of seed-and-extend (gapped) read alignment. In general, maximum throughput is obtained with a value of about 16 (but with less sensitivity), whereas maximum sensitivity can be achieved by using values of 200 and up (but with a corresponding decrease in throughput).

When encoding a genome such as the human genome that contains numerous repetitive regions, you should always specify a value for `maxJ=`. This causes AriocE to optimize the hash table by pruning repetitive regions of the genome where adjacent seeds hash to "big buckets"; this optimization improves throughput significantly with very little decrease in sensitivity for reads that map to repetitive regions.

`maxDOP=`. The `maxDOP=` value controls the number of concurrent threads AriocE uses for encoding and validating reference sequences. Additional threads can cause AriocE to run faster due to increased parallelism, but for a reference genome the size of the human reference genome the optimal value for `maxDOP=` may be constrained by the amount of available memory and by memory bandwidth.

`gpuMask=`. The `gpuMask=` parameter is a bit mask (specified as a hexadecimal value) that indicates which GPU device is to be used by AriocE for sorting hash-table bins. Each bit in the parameter corresponds to a GPU device identifier. Because AriocE uses only one GPU for sorting, only one bit in the parameter should be set. If `gpuMask=` is omitted or specified as zero, AriocE uses CPU threads to sort.

`TP=`. Certain reference sequences (e.g., human mitochondrial chromosome, bacteriophage lambda, EBV) have circular topology. When you specify `TP="circular"`, AriocE encodes these reference sequences in a way that enables AriocU and AriocP to report read mappings that straddle the origin of the reference sequence. This can improve mapping coverage near the origin of a circular reference sequence, although not all software tools can recognize the SAM representation of such mappings.

Output directory layout

The output from each invocation of AriocE should occupy a single directory that contains encoded sequence files as well as a subdirectory for each hash table. If, for example, you run AriocE twice (once

for a nongapped hash table and once for a gapped hash table) and configure it to write its output to a directory named `/GRCh38`, the resulting directory layout would look like this:

`/GRCh38`	encoded reference sequence files (`*.sbf`)
`/GRCh38/ssi84_2_30`	LUTs for nongapped (spaced-seed) aligner
`/GRCh38/hsi20_0_30`	LUTs for gapped (seed-and-extend) aligner

If you use a set of previously encoded reference sequence files, ensure that the directory structure follows this pattern. The lookup tables for both the nongapped and the gapped aligner must reside in subdirectories of the directory that contains the encoded reference sequences, and the subdirectory names must correspond to the "seed" strings that describe their format, seed width, and hash-value size.

AriocU: aligning unpaired reads

Before running AriocU, use AriocE to encode a reference sequence and hash tables (see AriocE: encoding sequence data, page 11).

To run AriocU, use a configuration file that specifies where to find encoded reference-sequence and read files, where to write alignment-result files, and how to parameterize the aligner:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocU gpuMask="0x00000001" batchSize="64k">

  <R>C:\BulkData\R\NCBI\GRCh38.p6</R>

  <nongapped seed="ssi84_2_30" />

  <gapped seed="hsi20_0_30" Wmxgs="2_6_5_3" Vt="400" />

  <Q filePath="C:\BulkData\Q\mason\test203">
    <unpaired subId="1">
      <file>i250p_1</file>
    </unpaired>
  </Q>

  <A overwrite="true">
    <sam report="mu">C:\BulkData\A\mason</sam>
  </A>

</AriocU>
```

Simple configuration file for AriocU.

Each of the XML elements in the configuration file controls a different aspect of the read-alignment pipeline:

- `<AriocU>`: GPU devices, GPU memory, and CPU threads
- `<R>`: encoded reference sequence files
- `<nongapped>`: seed type for nongapped alignments
- `<gapped>`: seed type and scoring parameters for gapped alignments
- `<Q>`: encoded read files
- `<A>`: output files and formats

The following table summarizes the basic configuration parameters:

Element	Attribute	Description
<AriocU>	gpuMask	Bits corresponding to GPU devices
	batchSize	Number of reads per batch per GPU
	maxDOP	Number of concurrent CPU threads
<nongapped>	seed	Seed used for AriocE nongapped reference-sequence encoding
<gapped>	seed	Seed used for AriocE gapped reference-sequence encoding
	Wmxgs	Smith-Waterman scoring parameters
	Vt	Minimum reportable alignment score
<unpaired>	subId	Subunit ID used for AriocE read encoding
<A>	overwrite	"true" to overwrite existing result file(s)
	cigarFormat	=XIDS [default], MIDS, or MID
	mdFormat	standard: [default] conforms with SAM format specification regex compact: omits placeholder '0' characters for adjacent mismatches
	mapqVersion	0: BWA model (0 ≤ MAPQ ≤ 60) 1: Bowtie 2 model (0 ≤ MAPQ ≤ 44) 2: Bowtie 2 model (0 ≤ MAPQ ≤ 44), optimized for AriocU
	maxAperRead	Maximum number of output records per read
	AtN, AtO	Minimum number of proper mappings per read (see Tuning for speed and sensitivity, page 28)
	seedDepth	Limits the number of seed iterations (see Tuning for speed and sensitivity, page 28)
	maxAperFile	Maximum number of read alignments per output file
<sam> <sbf>	report	Alignment result type(s) to report in SAM or SBF format in the specified output directory

XML elements and attributes in the AriocU configuration file.

Some of these configuration parameter values are self-evident, but others demand careful attention:

gpuMask=. The gpuMask= parameter is a bit mask (specified as a hexadecimal value) that indicates which GPU device(s) are to be used by AriocU. Each bit in the parameter corresponds to a GPU device identifier. If there is only one GPU in the computer, the value of gpuMask= should be 1. To run AriocU on two or more GPUs, set the corresponding bits in gpuMask=. For example, to use GPUs 0, 1, and 2, specify gpuMask="0x0007".

batchSize=. The batchSize= parameter determines how many reads AriocU can process concurrently per GPU. Batch sizes are limited by available GPU memory, but the amount of GPU memory needed to compute alignments cannot be precisely predicted because it depends on the number of Smith-

Waterman computations AriocU must perform in its search for satisfactory mappings for every read — and this is something you don't know until you try it.

With some data, we have observed overall speed increases up to 10% with well-chosen batch sizes, but batch size has less effect on speed than other "tuning" parameters (see Tuning for speed and sensitivity, page 28). If you do want to try to find an optimal batch size for your data, start with a batch size of about 32k. If AriocU runs without memory-allocation errors, progressively increase the batch size and re-run AriocU until it runs out of GPU memory or until its throughput does not improve with increasing batch size.

The `batchSize=` parameter may be specified either as a positive integer (e.g., `"25000"`) or as a multiple of 1024 (e.g., `"128K"`).

`maxDOP=`. AriocU uses CPU threads for file input/output and for post-processing alignment data (computing mapping qualities, formatting alignment results, and so on). If you omit `maxDOP=`, AriocU uses as many CPU threads as are available. You should specify a value for `maxDOP=` only if you wish to reserve CPU threads for a task other than AriocU.

`Wmxgs=`. This parameter specifies four Smith-Waterman scoring parameters as non-negative integers separated by commas or underscores. The four values indicate the score for a match, mismatch, gap open, and gap space respectively. For example, "2,6,5,3" means match=+2; mismatch=−6; gap open=−5; gap space=−3.

`Vt=`. This parameter indicates the minimum reportable Smith-Waterman alignment score. You can specify `Vt=` in one of two ways: as a non-negative integer threshold score (e.g., "100") or as a function of read length described in a Bowtie-style function parameter: "[LSG],b,a", where the initial letter indicates the function type (L: linear; S: square root; G: natural log), b is a constant term and a is the coefficient. For example, `Vt="S,75,2.5"` specifies a reportable alignment score threshold of `75 + 2.5*sqrt(`*`read_length`*`)`.

`mapQversion=`. The `mapqVersion=` parameter specifies the computational model to use when reporting mapping qualities (MAPQ) for reads. AriocU can compute MAPQ using a model similar to either BWA (numerical range 0-60) or Bowtie 2 (numerical range 0-44).

`maxAperRead=`. The `maxAperRead=` parameter limits the number of alignment-result records output by AriocU. This is independent of the `AtN=` and `AtO=` parameters whose purpose is to establish a minimum number of proper mappings to compute. Thus, for example, you might use `AtO="2"` (to increase accuracy by searching for at least two proper mappings for every read) and `maxAperRead="1"` (to report only the primary mapping for each read, regardless of how many proper mappings were found).

`report=`. Use the `report=` parameter to filter reported alignment results. You can use any combination of `m` (mapped) and `u` (unmapped). For example, to report both mapped and unmapped results in the same SAM file, use a `<sam>` element with `report="mu"`. To report the same results to two separate SAM files, use two `<sam>` elements, one with `report="m"` and one with `report="u"`.

AriocP: aligning paired-end reads

You must use AriocE to encode a reference sequence and hash tables (see AriocE: encoding sequence data, page 11) before you run AriocP.

The AriocP configuration file resembles the one that AriocU uses. The important differences have to do with parameters that control the alignment of paired-end reads:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocP gpuMask="0x00000001" batchSize="48k">

  <R>/BulkData/R/NCBI/GRCh38.p6</R>

  <nongapped seed="ssi84_2_30" />

  <gapped seed="hsi20_0_30" Wmxgs="2,6,5,3" Vt="100" />

  <Q filePath="/BulkData/Q/mason/test103">
    <paired subId="1">
      <file>i100p_1</file>
      <file>i100p_2</file>
    </paired>
  </Q>

  <A overwrite="true" pairOrientation="c" pairCollision="oc">
    <sam report="cdru">/BulkData/A/mason</sam>
  </A>

</AriocP>
```

Simple configuration file for AriocP.

Each of the XML elements in the configuration file controls a different aspect of the read-alignment pipeline:

- <AriocP>: GPU devices, GPU memory, and CPU threads
- <R>: encoded reference sequence files
- <nongapped>: seed type for nongapped alignments
- <gapped>: seed type and scoring parameters for gapped alignments
- <Q>: encoded read files
- <A>: output files and formats

The following table summarizes the basic configuration parameters:

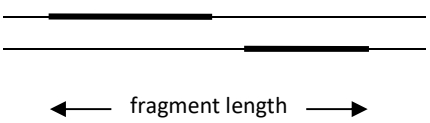
Element	Attribute	Description
<AriocP>	gpuMask	Bits corresponding to GPU devices
	batchSize	Number of reads per batch per GPU
	maxDOP	Number of concurrent CPU threads
<nongapped>	seed	Seed used for AriocE nongapped reference-sequence encoding
<gapped>	seed	Seed used for AriocE gapped reference-sequence encoding
	Wmxgs	Smith-Waterman scoring parameters
	Vt	Minimum reportable alignment score
<paired>	subId	Subunit ID used for AriocE read encoding
<A>	overwrite	"true" to overwrite existing result file(s)
	pairFragmentLength	Distance between the ends of a paired-end mapping
	pairOrientation	Specifies the relative orientation (convergent/divergent/same) of the mates in a pair.
	pairCollision	Indicates whether mates may overlap, cover, or dovetail.
	cigarFormat	=XIDS [default], MIDS, or MID
	mdFormat	standard: [default] conforms with SAM format specification regex compact: omits placeholder '0' characters
	mapqVersion	0: BWA model (0 ≤ MAPQ ≤ 60) 1: Bowtie 2 model (0 ≤ MAPQ ≤ 44) 2: Bowtie 2 model (0 ≤ MAPQ ≤ 44), optimized for AriocP
	maxAperRead	Maximum number of output records per read
	AtN, AtO	Minimum number of proper mappings per read (see Tuning for speed and sensitivity, page 28)
	seedDepth	Limits the number of seed iterations (see Tuning for speed and sensitivity, page 28)
	maxAperFile	Maximum number of read alignments per output file
<sam> <sbf>	report	Alignment result type(s) to report in SAM or SBF format in the specified output directory

XML elements and attributes in the AriocP configuration file.

Most of the AriocP configuration parameters control the same behaviors as they do with AriocU. The important differences are related to the management of paired-end alignments:

Vt=. The Vt= parameter is the minimum reportable alignment score for each individual mate in a pair, not the combined alignment scores for both mates.

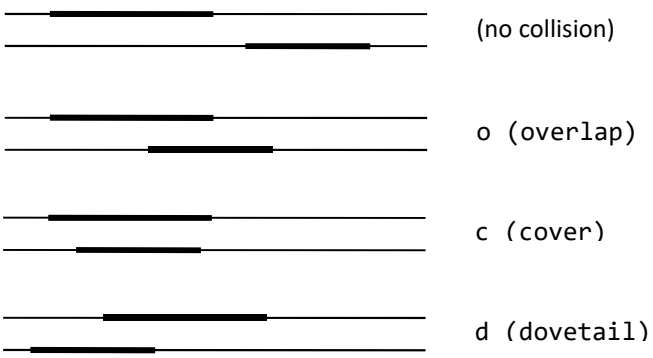
pairFragmentLength=. The pairFragmentLength= parameter specifies a range of acceptable fragment-length values. AriocP computes the pair fragment length as a SAM TLEN, that is, the number of mapped reference-sequence positions between the first ("upstream") mapped base and the last ("downstream") mapped base in a pair of mappings:



pairOrientation=. This parameter specifies the expected orientation of the mates in a pair:

- convergent: the "upstream" mate (that is, the mate at the lower-numbered location on the reference sequence) maps to the forward strand and the opposite mate maps to the reverse-complement strand
- divergent: the upstream mate maps to the reverse-complement strand and the opposite mate to the forward strand
- same: both mates map to the same strand

pairCollision=. The pairCollision= parameter indicates whether the mappings of a pair of mates may overlap, cover, or dovetail each other:



report=. The report= parameter value can be any combination of the following (similar to Bowtie's paired-end alignment result categories):

- c (concordant): both mates in a pair have reportable mappings and meet the user-specified constraints for orientation and fragment length
- d (discordant): each mate has a unique mapping, but the mappings do not meet the user-specified paired-end constraints
- r (rejected): both mates have mappings and at least one of the mates has two or more mappings, but no combination of the mappings meets the user-specified paired-end constraints
- u (unmapped): one or both mates have no reportable mapping

You can write alignment results to two, three, or four different files by using multiple <sam> elements, each with a different, mutually exclusive subset of the four alignment-result categories.

Alignment results

Both AriocU and AriocP can generate alignment results in two different formats: SAM [SAM/BAM Format Specification Working Group, 2017] and SBF (the binary format that the Microsoft SQL Server uses for "native" bulk inserts). The `<sam>` and `<sbf>` elements in the configuration file control whether the Arioc aligners emit results in SAM, SBF, or both formats.

SAM results

The Arioc aligners emit the following tab-separated fields. (See the SAM optional fields specification document for details):

Field name	Description
QNAME	Query-sequence (read) metadata
FLAG	Flag bits
RNAME	Reference sequence name
POS	Mapping position
MAPQ	Mapping quality
CIGAR	CIGAR string
RNEXT	Opposite mate reference sequence name
PNEXT	Opposite mate mapping position
TLEN	Paired-end fragment length
SEQ	Query sequence
QUAL	Phred-scaled base quality scores
AS	Alignment score
XS	Second-highest alignment score
YS	Opposite mate alignment score
NM	Edit distance
MD	MD string
YT	Alignment result category
Na	Number of distinct mappings found by the aligner
Nb	Number of maximum-score Smith-Waterman mappings
RG	Read group
id	Unique sequence identifier
af	Aligner flags
qf	Query flags

Tab-separated fields in Arioc SAM files.

Here are some additional details about the SAM fields reported by AriocU and AriocP:

RNAME: By default, the Arioc aligners use a 3-digit decimal string in RNAME. This numerical identifier corresponds to the subId= parameter that you associate with each reference sequence at the time the sequence is encoded (see Encoding a reference sequence, page 14). The metadata associated with each reference sequence (that is, in the original FASTA file for the reference sequence) appears in the rm= field of the corresponding @SQ record in the SAM file.

You can override this default behavior by running AriocE with a configuration file that specifies an SN= value (for example, a chromosome name) in each <file> element in the .cfg file. (See the example on page 15).

YT: AriocP assigns a 2-character "mapping type" (based on the YT field in Bowtie 2) to each SAM record:

YT	Description
CP	Concordant (meets user-specified criteria for mate orientation and fragment length)
DP	Discordant (unique mapping that does not meet user-specified criteria for orientation and fragment length)
RP	Rejected (both mates mapped without meeting user-specified criteria for orientation and fragment length, and one or both mates had multiple mappings)
UP	Unmapped (one or both mates unmapped)

Paired-end mapping types reported by AriocP.

Na: Because the Arioc aligners evaluate potential alignments in parallel, the number of mappings they discover may be significantly greater than the number of mappings they report. The Na field indicates the total number of distinct mappings that were found by the aligner.

Nb: In the scoring matrix computed for Smith-Waterman alignment, two or more different cells may contain the same maximum alignment score. The Nb field records the number of different cells that contained the same maximum score for the reported mapping.

c3: The c3 field contains a "triplet complexity" score for the read sequence that estimates the information content of a read sequence as a weighted sum of the counts of each three-base triplet in the sequence based on that used in DUST [Morgulis et al, 2006]:

$$\text{sum}((c_t*(c_t-1))/2)$$

The score is normalized to the sequence length N and represented as a phred score so that low-complexity sequences have lower c3 scores and 95% of sequences from the human reference genome have c3 scores between 45 and 60. See the AriocE source code distribution for details.

id: The AriocE encoder assigns a unique 64-bit sequence identifier to each sequence it reads from a FASTA or FASTQ file. The AriocU and AriocP aligners emit this unique identifier in the id field. The identifiers are assigned in the order in which they are encountered in the input FASTA or FASTQ file, although this ordering is not preserved in SAM output.

af, qf: These fields contain status flags used internally by the Arioc aligners.

SBF results

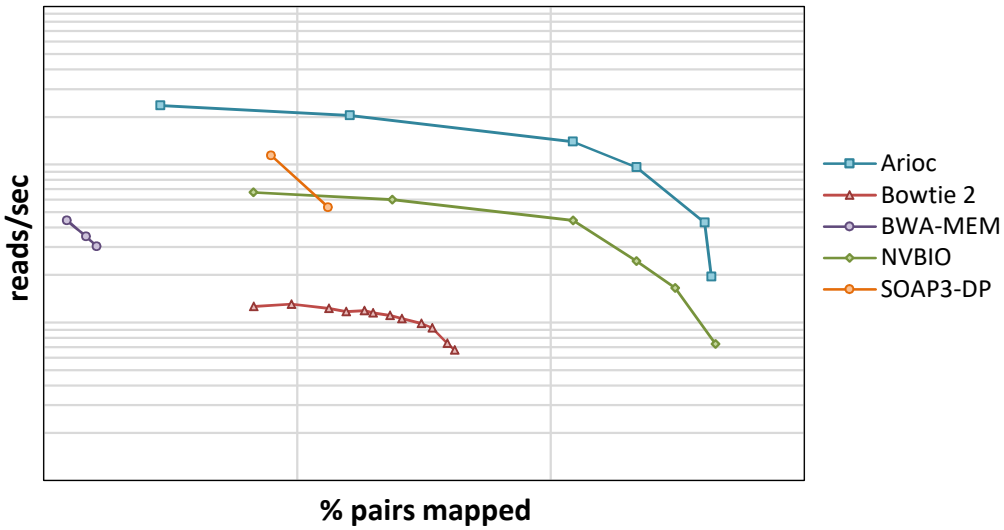
The binary format of SBF files generated by the Arioc aligners corresponds to the following SQL table definition (where N is a placeholder for a user-specified maximum size for variable-length strings):

```
create table AriocSBFTable
( qname   varchar(N)      not null,
  flag    smallint        not null,
  rname   smallint        not null,
  pos     int             not null,
  mapq    tinyint         not null,
  cigar   varchar(N)      not null,
  rnext   smallint        not null,
  pnext   int             not null,
  tlen    int             not null,
  seq     varchar(N)      not null,
  qual    varbinary(N)    not null,
  sqId    bigint          not null,
  V       smallint        not null,
  XS      smallint        not null,
  NM      smallint        not null,
  MD      varchar(N)      not null,
  NA      smallint        not null,
  NB      smallint        not null,
  RG      smallint        not null,
  af      smallint        not null,
  qf      smallint        not null
)
```

SQL table definition corresponding to the Arioc SBF file format.

Tuning for speed and sensitivity

Like other short-read aligners, Arioc exhibits a decrease in speed when you configure it for higher sensitivity, that is, when you parameterize AriocU or AriocP to compute more alignments in order to discover acceptable mappings for read sequences that contain larger numbers of mismatches and indels:



The simplest way to mitigate this is to use faster hardware! In particular, increasing the number of GPU threads available to Arioc increases overall throughput. You can accomplish this either by using multiple GPU devices or by running Arioc on newer GPU hardware with a larger number of compute cores and a higher degree of parallelism.

Before you process multiple samples with Arioc, we recommend that you perform a few test runs to establish a reasonable balance between speed and sensitivity. Arioc supports a variety of parameters you can use to control the tradeoff between speed and sensitivity.

The configuration files distributed with Arioc contain settings for reference-sequence encoding (AriocE) and alignment (AriocU, AriocP) that are reasonable for the human reference genome and for typical Illumina whole-genome sequencing (WGS) reads. You can use these settings as a starting point to adjust Arioc's performance to the reference genome you are using and to the content of your own sequencer reads.

The `verboseMask=` and `maxQ=` parameters

The `verboseMask=` parameter (used with the `<AriocE>`, `<AriocU>`, or `<AriocP>` element) controls the amount of detail in the output of each of the Arioc executables. Specify the parameter value as a 32-bit hexadecimal bitmap:

Bit	Hexadecimal	Description
31	0x80000000	Output to stderr
30	0x40000000	Output to Windows debugger
29	0x20000000	Emit timestamps
11	0x00000800	Trace main loop iterations
9	0x00000200	Trace CUDA (GPU) memory management
8	0x00000100	Trace host (CPU) memory management
4	0x00000010	Detailed trace
3	0x00000008	Basic trace
2	0x00000004	Detailed performance metrics
1	0x00000002	Basic performance metrics
0	0x00000001	Banners and exceptions

Bit flags in the verboseMask= parameter.

For performance tuning, use `verboseMask=0xE0000807` to see time-stamped intermediate results and summary timings.

The `maxQ=` parameter (used with the `<Q>` element) halts read alignment after the specified number of reads (for AriocU) or pairs (for AriocP) have been processed per GPU.

Tuning for sensitivity

Arioc finds more high-scoring mappings when it evaluates potential alignments at more reference-sequence locations. Here are several ways to broaden Arioc's "search space":

AriocE: Build the seed-and-extend lookup table (hash table) with a higher value for `maxJ=`. This affects "big bucket" hash values, that is, seeds that derive from highly repetitive regions and that consequently are associated with many reference-sequence locations. Increasing `maxJ=` causes AriocU and AriocP to compute more potential alignments for reads that contain "big bucket" seeds.

Increasing `maxJ=` has no effect on most hash table buckets because seeds that are not associated with highly repetitive regions of the reference hash to very few reference-sequence locations anyway. Thus, with increased `maxJ=`, AriocU and AriocP find additional mappings primarily for a few noisy reads. (By "noisy", we mean that a read contains enough mismatches and indels that only a few of its seeds remain intact. When those intact seeds happen to be associated with "big buckets" in the hash table, the aligner is more likely to find a mapping when you use a larger value for `maxJ=`.)

AriocU and AriocP spend more time evaluating additional seed locations when they use a seed-and-extend hash table built with larger values for `maxJ=` in AriocE. For example, with the human reference genome, speed starts to decrease sharply with `maxJ=` values above 500 or 1000, with relatively small improvements in sensitivity. With a larger, highly repetitive reference such as the bread wheat genome, a `maxJ=` value of 100 or less might provide a better tradeoff between speed and sensitivity.

This effect is much less noticeable with the spaced-seed (nongapped) lookup table because the spaced-seed alignment algorithm is much faster than the Smith-Waterman algorithm used in Arioc's seed-and-extend (gapped) aligner. (More about this below.) If you do not notice a significant improvement in speed by limiting `maxJ=` for the spaced-seed lookup table, then simply omit `maxJ=` or use `maxJ="*"` in AriocE.

Finally, you can encode the reference lookup tables (hash tables) with up to 32-bit hash values. Larger hash values (and their correspondingly larger lookup tables) provide additional sensitivity because there are fewer hash collisions, so in general you should use the largest hash tables that fit into available memory.

AriocU, AriocP: To obtain greater sensitivity from the AriocU and AriocP aligners, you can increase the value of any or all of the following parameters:

Element	Attribute	Description
`<gapped>`	`AtN`	Per-read threshold for nongapped proper mappings
	`AtO`	Per-read threshold for overall (nongapped + gapped) proper mappings
	`seedDepth`	Number of seed iterations

Configuration parameters useful for AriocU and AriocP performance tuning.

To understand how to use these parameters, consider how the AriocU and AriocP aligners process each read.

First, the aligner uses periodic spaced seeds to find nongapped mappings that contain no more than two mismatches. The value of `AtN=` is a threshold for the number of nongapped mappings; if the aligner finds at least that number of nongapped mappings for a read, it searches no further for additional mappings. Higher `AtN=` values cause the aligner to spend more time searching for nongapped mappings. The additional mappings discovered in this way can improve the accuracy of the aligner's mapping quality (MAPQ) estimates because the aligner computes a lower MAPQ score for reads with multiple high-scoring mappings.

After searching for nongapped mappings, the aligner uses seed-and-extend to find gapped mappings for the remaining reads (i.e., those that do not have the minimum number of nongapped mappings). For each such read, the aligner iteratively chooses seeds, starting with the set of seeds that are immediately adjacent to each other. (For example, with 20nt seeds, the aligner starts with the seeds at positions 0, 20, 40... in the read). If none of these seeds leads to a successful mapping, the seed interval is halved and the new set of seeds is evaluated.

With typical seed sizes, six such "interval halving" iterations are needed to evaluate every possible seed location in the read. You can alter this behavior by setting the `seedDepth=` parameter, which limits the number of seed iterations to a number between 1 and 6. The Arioc aligners run significantly faster with smaller values of `seedDepth=`, but they also find high-scoring mappings for fewer reads. Also, as read length increases, more seeds are generated in each iteration, so you may find that fewer seed iterations are necessary to find a sufficient number of mappings for longer reads.

Arioc abandons the search for seed-and-extend mappings for a read when it finds a sufficient number of high-scoring mappings. You can use the `AtO=` parameter to force Arioc to keep searching for higher-scoring mappings; the aligner will not give up on a read until it finds at least the number of mappings specified in `AtO=` (or until it runs out of seeds).

Because Arioc computes potential alignments in parallel, neither `AtN=` nor `AtO=` is precisely deterministic. For example, even with the default settings of `AtN="1"` and `AtO="1"`, Arioc finds multiple high-scoring mappings for many reads because those mappings are all discovered in parallel.

Tuning for speed

Arioc runs faster when it searches for fewer mappings. In general, choosing lower values for the same set of tuning parameters leads to higher throughput.

AriocE: Build the seed-and-extend hash table with a lower value for `maxJ=`. This causes AriocE to be more aggressive in identifying "big bucket" regions of the reference genome and pruning the hash table accordingly. You can use values for `maxJ=` as low as 16 and still obtain acceptable alignment results with AriocU and AriocP.

AriocU, AriocP: The AriocU and AriocP aligners also recognize the `maxJ=` parameter. (Use `maxJ=` on the `<nongapped>` and `<gapped>` elements in the configuration file.) In the aligners, `maxJ=` specifies a limit to the number of reference-sequence locations that the aligners examine for each seed. Changing `maxJ=` has little effect when used with the nongapped aligner (which is very fast anyway) but using lower values for `maxJ=` can tangibly increase the speed of the gapped aligner.

You can also increase throughput by using smaller values for the `seedDepth=`, `AtN=`, and `AtO=` parameters.

- `seedDepth`: AriocU and AriocP iterate through overlapping subsets of the seeds for each read sequence. By limiting the number of iterations, you limit the number of reference-sequence locations at which each read can potentially be mapped.
- `AtN, AtO`: these parameters specify the minimum number of nongapped (`AtN`) and overall (`AtO`) mappings that AriocU or AriocP must find before it stops computing alignments for a read. The default values (`AtN="1"`, `AtO="1"`) favor speed but overestimate MAPQ for some reads that have two or more high-scoring mappings. Consider using larger values if you prefer to avoid this inaccuracy, if you want the aligner to report secondary mappings for such reads, or if you want to favor the reporting of read mappings that contain insertions or deletions.

As you decrease these parameter values, you may notice that speed is limited by factors other than the intrinsic speed of the aligners. Disk I/O bandwidth and contention for the PCIe bus can each limit overall throughput, especially when you run AriocU or AriocP on multiple GPUs.

Multiple input files

It is possible to align multiple input files (or, more exactly, pairs of input files) by using the `subId=` and `mate=` attributes of the `<paired>` or `<unpaired>` element in the configuration file. This may provide a

marginal improvement in overall speed because GPU initialization, which takes several minutes in some computers, is performed only once.

Nevertheless, you should exercise caution when you align multiple input files (or pairs of input files) because AriocE cannot recognize different layouts among the input files. For example, files that originate from different sequencer runs may use a different bias to encode base quality scores (e.g., 33 for Sanger, 64 for Illumina v1.3 and later), but AriocU and AriocP cannot recognize this on the fly.

It is preferable simply to encode and align one sequencer run at a time.

Extra ("X") parameters

AriocU and AriocP recognize a number of additional configuration parameters that may be useful for optimizing performance with certain hardware configurations.

The X parameters are documented in file `Xparams.txt` in the `Arioc.RQA.zip` archive.

GPU-specific builds

More recent iterations of the Nvidia GPU microarchitecture support a more powerful instruction set than do older Nvidia GPUs. Nvidia identifies instruction set versions with an identifier it calls the "compute capability." For example, the Kepler K80 device, released in 2014, has compute capability 3.7, whereas the V100 GPU, released in 2017, has compute capability 7.0. (Complete lists of compute capabilities for all CUDA-enabled Nvidia GPUs are available online.)

To ensure broad hardware compatibility, the makefile shipped with the Linux distribution of Arioc specifies that the Nvidia compiler generate code that targets an older (lower-numbered) compute capability. If you will be using Arioc with a GPU with a more recent (higher-numbered) compute capability, you can build AriocU and AriocP specifically for that hardware by specifying an additional parameter, `CUDA_CC`, on the make command line. For example, to target V100 GPUs, you could build AriocP like this:

```
make AriocP CUDA_CC=70
```

The resulting executable binary will be incompatible with GPU devices that do not support the targeted compute capability, but it may run up to 5% faster on the same GPU device than a binary that is built for an older (lower-numbered) compute capability.

BS-seq alignment

The Arioc aligners natively support alignment of bisulfite-treated DNA reads. By "natively" we mean that the additional logic involved in computing BS-seq alignments is integrated into Arioc. Although this approach allows for significant GPU acceleration of the code as well as increased sensitivity [Wilton et al., 2017], Arioc nevertheless aligns BS-seq reads several times more slowly than it does non-bisulfite-treated reads of similar length.

Arioc's BS-seq functionality is compatible with that of the Bismark aligner [Krueger and Andrews, 2011]. In particular, the Arioc aligners produce SAM-formatted output that is recognized by Bismark's downstream processing tools, so Arioc is functionally compatible with Bismark in a BS-seq read-processing workflow.

Preparing for BS-seq alignment

In terms of sequencer output, bisulfite-treated DNA sequences represent a six-letter alphabet using only five letters, so T in a BS-seq read may represent either cytosine or thymine in the original DNA sequence:

	original DNA	sequencer read
adenine	A	A
cytosine	C	T
methylcytosine	C_m	C
guanine	G	G
thymine	T	T
(unknown)	N	N

The Arioc aligners use hash tables in which the reference sequence (genome) is similarly encoded. They then disambiguate Ts in each mapped sequence by comparing them to the original reference sequence at each point in the mapping.

To encode the reference sequence for BS-seq alignment, specify a CT-converted seed type in the `seed=` attribute of the `<AriocE>` element when you execute `AriocE`. For example, the following creates a gapped (spaced-seed) hash table with seeds of length 25:

```
<AriocE seed="hsi25_0_30_CT" maxDOP="32" maxJ="200">
```

(See also Encoding a reference sequence, page 14.)

Running the aligner

When executed using CT-converted hash tables, the Arioc aligners perform CT-converted alignments to identify and report methylcytosines in context. The following configuration file illustrates how to do this:

```
<?xml version="1.0" encoding="utf-8"?>

<AriocU gpuMask="0x00000007" batchSize="8K" verboseMask="0xE0000007">

  <R>C:\data\raid5-2\BulkData\R\NCBI\GRCh38.p6\maxJ 2048</R>

  <nongapped seed="ssi84_2_30_CT" />

  <gapped seed="hsi25_0_30_CT" Wmxgs="2_6_5_3" Vt="L,-50,2" />

  <Q filePath="C:\data\data2\BulkData\Q\Sherman">
    <unpaired srcId="1" subId="1">
      <file>test00</file>
    </unpaired>
  </Q>

  <A overwrite="true" cigarFormat="MID" mapqVersion="2">
    <sam report="mu">C:\data\data2\BulkData\A\Sherman</sam>
  </A>

</AriocU>
```

BS-seq configuration file for AriocU.

In the above example, the `cigarFormat=` and `mapqVersion=` attributes of the `<A>` element are set so that the `CIGAR` and `MAPQ` fields in the SAM-formatted output will be formatted as expected by `bismark_methylation_extractor`, Bismark's methylation-context identification tool.

Additional SAM output fields for BS-seq reads

When computing BS-seq alignments, Arioc emits three additional fields in SAM-formatted output:

Field name	Description
XM	Methylation context map
XR	Reference-sequence encoding (always CT)
XG	Read sequence encoding (CT or GA)

Additional tab-separated fields in Arioc SAM files for BS-seq alignments.

These three fields are modeled after the identically-named fields in SAM output from the Bismark read aligner. Although the format of each field in Arioc SAM output matches the Bismark format, there are small differences in the way the fields are generated.

The `XM` field is a character map of the position of each methylcytosine (C_m) identified by the read aligner. For each mapped read, Arioc encodes the methylation context for each cytosine by examining the two symbols that follow it in the read sequence:

XM encoding	methylation context	symbols following C
Z, z	CpG	G
H, h	CAG, CTG, CCG	AA AC AT CA CC CT TA TC TT
X, x	CHG	AG CG TG
U, u	unknown	N AN CN TN (or end of read)

Uppercase letters (ZHXU) represent methylcytosine. Lowercase letters (zhxu) represent unmethylated cytosine.

Arioc differs from Bismark in that the Arioc aligners determine methylation context exclusively by examining the read sequence, whereas Bismark uses the corresponding reference sequence whenever possible. This introduces rare differences between the XM maps generated by Bismark and by Arioc for identically mapped reads.

Arioc does not use the same approach as Bismark to index the reference genome for computing BS-seq alignments, but it still emits XG and XR fields for each mapped read that are compatible with those generated by Bismark. This maintains compatibility with the `bismark_methylation_extractor` tool so that it can report per-context methylation rates.

Troubleshooting

We test each Arioc release against full-size human WGS and WGBS data samples to validate its functionality and measure its performance, but we cannot guarantee that the software is entirely free from bugs and other problems with functionality. Here are some suggestions for troubleshooting problems you encounter with the Arioc software:

- Ensure that Arioc operates correctly with the sample reference and read data that we provide. These data are downloadable in the `Arioc.RQA.140.zip` archive. You should be able to complete an end-to-end test of the software (AriocE, AriocU, and AriocP) in 10 to 15 minutes.
- Use the `verboseMask=` parameter to generate trace and debugging information. A setting of `verboseMask="0xE0000007"` is a good place to start.
- Do a debug build of the Arioc software and add additional `verboseMask=` flags to obtain more detailed trace information (see Tuning for speed and sensitivity, page 28).
- Use the public source-code repository (`https://github.com/RWilton/Arioc`) to describe the problem publicly. You may also contact us directly (`richard.wilton@jhu.edu`).

References

Altschul SF et al. (1990) Basic Local Alignment Search Tool. J Mol Biol **215**, 403-410.

Anderson M et al. (2011) Considerations when evaluating microprocessor platforms. Proceedings of the 3rd USENIX conference on hot topics in parallelism.

Cooper A. (2004) The inmates are running the asylum. Sams Publishing, Indianapolis, IN. ISBN 0-672-32614-0.

Illumina Corporation. (2015) CASAVA v1.8 User Guide. Available at http://support.illumina.com/help/SequencingAnalysisWorkflow/Content/Vault/Informatics/Sequencing_Analysis/CASAVA/swSEQ_mCA_FASTQFiles.htm.

Krueger F, Andrews SR. (2011) Bismark: a flexible aligner and methylation caller for Bisulfite-Seq applications. Bioinformatics **27**:11.

Langmead B, Salzberg S. (2012) Fast gapped-read alignment with Bowtie 2. Nature Methods **9**:357-359.

Li H. (2013) Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. arXiv 1303.3997v1.

Liu Y et al. (2013) CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC Bioinformatics **14**:117.

Morgulis et al. (2006) A fast and symmetric DUST implementation to mask low-complexity DNA sequences. J Comp Biol **13**, 1028-1040

SAM/BAM Format Specification Working Group. (2020) Sequence Alignment/Map Format Specification. Available at https://samtools.github.io/hts-specs/SAMv1.pdf.

SAM/BAM Format Specification Working Group. (2020) Sequence Alignment/Map Optional Fields Specification. Available at https://samtools.github.io/hts-specs/SAMtags.pdf.

Smith TF, Waterman MS. (1981) Identification of common molecular subsequences. J Mol Biol **147**, 195-197.

Ukkonen E. (1983) On approximate string matching. Springer: Foundations of Computer Theory, Lecture Notes on Computer Science **158**, 487–495.

Wilton R et al. (2015) Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space. PeerJ **3**:e808; DOI 10.7717/peerj.808

Wilton R et al. (2018) Arioc: GPU-accelerated alignment of bisulfite-treated short-read sequences. Bioinformatics **34**:1-3; DOI 10.1093/bioinformatics/bty167