DNA Sequence Reads Compression

User Guide

Release 2.0

March 31, 2014

Contents

1

Introduction

1.1 What is DSRC?

DNA Sequence Reads Compression is an application designed for loosless and lossy compression of DNA sequencing reads stored in FASTQ format. The first release (0.x) of DSRC was accompanied by the research paper:

- S. Deorowicz and S. Grabowski: *Compression of DNA sequence reads in FASTQ format*, Bioinformatics (2011).

The newest release is described in:

- Ł. Roguski and S. Deorowicz: *DSRC 2—Industry-oriented compression of FASTQ files* (under review).

1.2 Main features

The main features of DSRC are:

- Effective multithreaded compression of DNA sequencing data stored in FASTQ format.
- Full support for Illumina, ABI SOLiD, and 454/Ion Torrent dataset formats with non-standard (AGCTN) IUPAC base values.
- Support for lossy quality values compression using Illumina binning scheme.
- Support for lossy IDs compression keeping only key fields selected by user.
- Pipes support for easy integration with current pipelines.
- Python and C++ libraries allowing to integrate DSRC archives in own applications.
- Availability for Linux, Windows, and MacOS 64-bit operating systems.
- Open C++ source code under GNU GPL 2 license.

1.3 Compression factor and speed

In terms of lossless compression factor (the ability to reduce the file size), DSRC in is usually 35–60% better than gzip and 15–30% better than bzip2, which are currently the most common tools for storing FASTQ files in a compressed format. DSRC is a multithreaded software and the speed in the fast mode usually reaches the I/O limits (e.g., about 500 MB/s for 8 threads). The achieved compression ratio and speeds are described in the mentioned research papers.

1.4 CONTACT AND SUPPORT

- Official Web site: http://sun.aei.polsl.pl/dsrc/
- Contact and bug reports: sebastian.deorowicz [at] polsl.pl , lucas.roguski [at] gmail.com

2

Quickstart

2.1 Download

The compiled binaries, C++ and Python libraries with example of usage can be downloaded from the official Web site: `http://sun.aei.polsl.pl/dsrc/`

2.2 Building

DSRC compiles on both Windows and Linux platforms.

2.2.1 Linux

To compile DSRC on Linux platform do use provided makefile files in the main directory. The default `Makefile` will compile DSRC in a static-linking variant using `boost::thread`[1] library for multithreading functionality, so `boost` development libraries are required to be present. The `Makefile.g++11` will use g++ (≥ 4.8) with implementation of C++11 standard, which provides multithreading functionality without using Boost libraries.
To build DSRC binary in the main directory:

```
make bin
```

where the resulting `dsrc` binary file will be placed in `bin` directory.

To build DSRC C++ library use:

```
make lib
```

where the resulting `dsrc` binary file will be placed in `lib` directory.

2.2.2 Windows

To compile on Windows platform there are provided project solution files `dsrc-vs2k10.sln` and `dsrc-vs2k12.sln` for respectively Microsoft Visual Studio 2010 and 2012 with configurations for building DSRC binary and C++ library. When building DSRC using Microsoft Visual Studio 2010 `boost::thread` library will be required to provide multithreading functionality, while by using Microsoft Visual Studio 2012 C++11 threads implementation will be used. DSRC can be also compiled using MinGW-W64 (64-bit) using linux makefile files.

[1] Boost libraries can be downloaded from `http://www.boost.org/`

2.2.3 Mac OSX

To compile DSRC on Mac OSX use the provided `Makefile.osx` file in the main directory. The makefile uses clang compiler with C++11 standard support for multithreading.
To build DSRC binary in the main directory:

```
make -f Makefile.osx bin
```

where the resulting `dsrc` binary file will be placed in `bin` directory.

To build DSRC C++ library use:

```
make -f Makefile.osx lib
```

where the resulting `dsrc` binary file will be placed in `lib` directory.

2.2.4 Python module

To build the DSRC Python module the `boost::python`, `boost::thread` and `boost::system` libraries in development versions are required. The module will be build using the Boost Build `bjam` tool using `Jamroot` file located in py folder.

In order to compile Python module, the location of the local boost path needs to be edited in `Jamroot` file:

```
# User-specified path to the Boost project
use-project boost
   : /path/to/main/boost/directory/ ;
```

After setting the local boost path to compile the `pydsrc` module just type in the main directory:

```
make pylib
```

which will launch Boost Build tool and place the library in the output `pylib` directory tree.

2.3 PROGRAM USAGE

DSRC can be run from the command prompt:

```
dsrc <c|d> [options] <input_file_name> <output_file_name>
```

in one of two modes:

`c` — compression,

`d` — decompression.

Available compression options are:

`-d<`*n*`>` — DNA compression mode: 0–3, default: 0

`-q<`*n*`>` — Quality compression mode: 0–2, default: 0

`-f<1,...>` — keep only those fields no. in ID field string, default: keep all

`-b<`*n*`>` — FASTQ input buffer size in MB, default: 8

`-m<n>` — Automated compression mode (one of the three preset combination of other parameters): 0–2

`-o<n>` — Quality offset, 0 for auto selection, default: 0

`-l` — use Quality lossy mode (Illumina binning scheme), default: false

`-c` — calculate and check CRC32 checksum calculation per block (slows the compression about twice), default: false

Available 'automated' compression modes:

`-m0` — *fast* mode, equivalent to: `-d0 -q0 -b8`

`-m1` — *medium* mode, equivalent to: `-d2 -q2 -b64`

`-m2` — *best* mode, equivalent to: `-d3 -q2 -b256`

Options for both compression and decompression modes:

`-t<n>` — processing threads number, default: max available hardware threads

`-s` — use stdin/stdout for reading/writing FASTQ data (stderr is used for info/warning messages)

Usage examples:

- Compress SRR001471.fastq file saving DSRC archive to SRR001471.dsrc:
  `dsrc c SRR001471.fastq SRR001471.dsrc`
- Compress file in the *fast* mode with CRC32 checking and using 4 threads:
  `dsrc c -m0 -c -t4 SRR001471.fastq SRR001471.dsrc`
- Compress file using DNA and Quality compression level 2 and using 512 MB buffer:
  `dsrc c -d2 -q2 -b512 SRR001471.fastq SRR001471.dsrc`
- Compress file in the *best* mode with lossy Quality mode and preserving only 1–4 fields from record IDs:
  `dsrc c -m2 -l -f1,2,3,4 SRR001471.fastq SRR001471.dsrc`
- Compress in the *best* mode reading FASTQ file read from stdin:
  `cat SRR001471.fastq | dsrc c -m2 -s SRR001471.dsrc`
- Decompress SRR001471.dsrc archive saving output FASTQ file to SRR001471.out.fastq:
  `dsrc d SRR001471.dsrc SRR001471.out.fastq`
- Decompress archive using 4 threads and streaming FASTQ data to stdout:
  `dsrc d -t4 -s SRR001471.dsrc » SRR001471.out.fastq`

3

DSRC integration

DSRC can be easily integrated with applications written in C++ or Python. We provide C++ and Python libraries with very similar interfaces. Although the methods and members names are almost identical in both cases, for clarity the C++ and Python descriptions are divided into two separate sections.

3.1 PYTHON API

To start using the compressor functionality in Python it's only needed to import `pydsrc` module (`pydsrc.so` file) in the project. In order to provide high performance of compression routines the core operations were written in C++ and exported to Python.

3.1.1 FastqRecord

`FastqRecord` represents a single DNA sequencing read. The ID, sequence, plus, and quality fields are accessible as Python `string` type.

3.1.2 FastqFile

`FastqFile` is used for reading and writing FASTQ files in a sequential manner, read by read, where each is of type `FastqRecord`. Figure 3.1 shows `FastqFile` available public methods—all methods throw exception on error or failure.

Table 3.1: `FastqFile` public methods

Method	Returns	Parameters	Description
`Open`	—	`string` filename	Opens specified file.
`Create`	—	`string` filename	Creates a new file or overwrites one if exists.
`Close`	—	—	Closes the file.
`ReadNextRecord`	`boolean`	`FastqRecord` record	Gets the next record. Returns `True` on successful read and `False` on reaching the end of file.
`WriteRecord`	—	`FastqRecord` record	Writes the next record.

3.1.3 DsrcArchive

`DsrcArchive` represents DSRC archive file and provides compression and decompression routines. Figure 3.2 shows `DsrcArchive` available public methods—all methods throw exception on error or failure. Figure 3.3 shows public properties. It's important to note, that properties cannot be set, when `DsrcArchive` is already in processing mode (compression or decompression routines have been stared).

Table 3.2: `DsrcArchive` methods

Method	Returns	Parameters	Description
`StartCompress`	—	`string` filename	Creates a new DSRC archive and prepares for compression.
`WriteNextRecord`	—	`FastqRecord` rec	Writes a new record to file.
`FinishCompress`	—	—	Finalizes the compression of the archive and performs cleanup.
`StartDecompress`	—	`string` filename	Opens a DSRC archive and prepares for decompression.
`ReadNextRecord`	`boolean`	`FastqRecord` record	Reads the next decompressed FASTQ record from archive. Returns `True` on success and `False` on reaching end of archive.
`FinishDecompress`	—	—	Finalizes the decompression of the archive and performs cleanup.

Table 3.3: `DsrcArchive` properties

Property	Type	Description
`DnaCompressionLevel`	`int`	DNA compression level: 0–3. Default: 0.
`QualityCompressionLevel`	`int`	Quality compression level: 0–2. Default: 0.
`FastqBufferSize`	`int`	Size of input FASTQ buffer specified in MB. Default: 8.
`LossyCompression`	`boolean`	Lossy quality compression mode indicator. Default: `False`.
`TagFieldFilterMask`	`int`	Mask defining ID fields to keep. Default: 0 (keep all).
`QualityOffset`	`int`	Value of Quality offset—typically 33 or 64. Default: auto determined.
`Crc32Checking`	`boolean`	CRC32 checksum check during compression indicator. Default: `False`.
`ColorSpace`	`boo lean`	Color space archive type (e.g., ABI SOLiD) indicator. Default: `False`.
`PlusRepetition`	`boolean`	Flag indicating keeping additional information in plus lines. Default: `False`.

3.1.4 DsrcModule

`DsrcModule` provides automated and parallel compression routines, working on whole files instead of single records. Figures 3.4 and 3.5 show available public methods and public properties. All methods throw exception on error or failure.

Table 3.4: `DsrcModule` methods

Method	Returns	Parameters	Description
`Compress`	—	`string fastqFilename`, `string dsrcFilename`	Compresses the FASTQ file to DSRC archive.
`Decompress`	—	`string dsrcFilename`, `string fastqFilename`	Decompresses the DSRC archive to file in FASTQ format.

Table 3.5: `DsrcModule` properties

Property	Type	Description
`DnaCompressionLevel`	`int`	DNA compression level: 0–3. Default: 0.
`QualityCompressionLevel`	`int`	Quality compression level: 0–2. Default: 0.
`FastqBufferSize`	`int`	Size of input FASTQ buffer in MB. Default: 8.
`LossyCompression`	`boolean`	Lossy Quality compression mode indicator. Default: `False`.
`TagFieldFilterMask`	`int`	Mask defining ID fields to keep. Default: 0 (keep all).
`QualityOffset`	`int`	Value of Quality offset, typically 33 or 64. Default: auto determined.
`Crc32Checking`	`boolean`	Flag indicating if CRC32 checking is enabled during compression. Default: `False`.

3.2 PYTHON EXAMPLES

3.2.1 Compressing FASTQ file using automated DsrcModule module

```
import pydsrc

filename_fq = "SRR001471.fastq"
filename_dsrc = "SRR001471.dsrc"

# create and configure DSRC module
module = pydsrc.DsrcModule()
module.LossyCompression = True
module.DnaCompressionLevel = 2
module.QualityCompressionLevel = 2
module.FastqBufferSizeMB = 512
module.ThreadsNumber = 2

module.Compress(filename_fq, filename_dsrc)
```

3.2.2 Decompressing DSRC archive using automated DsrcModule module

```
import pydsrc

filename_dsrc = "SRR001471.dsrc"
filename_fq = "SRR001471.dsrc.fastq"

# create and configure DSRC module
module = pydsrc.DsrcModule()
module.ThreadsNumber = 2

module.Decompress(filename_dsrc, filename_fq)
```

3.2.3 Manual compression using DsrcArchive and FastqFile

```
import pydsrc

filename_fq = "SRR001471.fastq"
filename_dsrc = "SRR001471.dsrc"

fqfile = pydsrc.FastqFile()
fqfile.Open(filename_fq)

# create and configure DSRC archive file
archive = pydsrc.DsrcArchive()
archive.DnaCompressionLevel = 2

# lossy compress quality values using use Illumina bining scheme
archive.LossyCompression = True
archive.QualityCompressionLevel = 2

archive.FastqBufferSizeMB = 512

# do not keep extra information in 'plus' field
archive.PlusRepetition = False

archive.StartCompress(filename_dsrc)


# read all records from FASTQ file and write to DSRC archive
rc = 0
rec = pydsrc.FastqRecord()
while fqfile.ReadNextRecord(rec):
    archive.WriteNextRecord(rec)
    rc += 1

# save archive and close FASTQ file
archive.FinishCompress()
fqfile.Close()

print "Success!\nRecords written: %d" % rc
```

3.2.4 Manual decompression using DsrcArchive and FastqFile

```
import pydsrc

filename_dsrc = "SRR001471.dsrc"
filename_fq = "SRR001471.dsrc.fastq"


# open existing DSRC archive file
archive = pydsrc.DsrcArchive()
archive.StartDecompress(filename_dsrc)

# create results FASTQ file
fqfile = pydsrc.FastqFile()
fqfile.Open(filename_fq)

# read all records from DSRC archive and write to FASTQ file
rc = 0
rec = pydsrc.FastqRecord()
while archive.ReadNextRecord(rec):
    fqfile.WriteNextRecord(rec)
    rc += 1

# close (and save) FASTQ file and close DSRC archive
fqfile.Close()
archive.FinishDecompress()

print "Success!\nRecords written: %d" % rc
```

3.3 C++ API

To start using DSRC C++ library it's only needed to include `Dsrc.h` header file and to link application with `libdsrc` (`libdsrc.a` file under Linux or `libdsrc.lib` file under Windows) library.

3.3.1 FastqRecord

`FastqRecord` stores a single DNA sequencing read information. The IDs, sequence, plus, and quality fields are represented using `std::string` type.

3.3.2 FastqFile

`FastqFile` is used to read and write FASTQ records, where each record is of `FastqRecord` type. Figure 3.6 shows `FasqtFile` public methods—methods throw `std::runtime_error` exception on error.

Table 3.6: `FastqFile` public methods

Method	Returns	Parameters	Description
`Open`	—	`std::string` filename	Opens specified file for reading.
`Create`	—	`std::string` filename	Creates a new file or overwrites if one exists for writing.
`Close`	—	—	Closes the file.
`ReadNextRecord`	`bool`	`FastqRecord&` record	Gets the next buffered record. Returns false when reaching end of file.
`WriteNextRecord`	—	`FastqRecord` record	Writes the next record.

3.3.3 DsrcArchive

`DsrcArchve` provides methods to read from and write to DSRC archive file in a continuous way, read by read, where record is of `FastqRecord` type. Figure 3.7 shows `DsrcArchive` public methods, while Figure 3.8 shows public accessors.

3.3.4 DsrcModule

`DsrcModule` provides automated parallel compression and decompression routines operating on whole files instead of single records. Figure 3.9 shows available public methods, Figure 3.10 shows public accessors.

Table 3.7: `DsrcArchive` public methods

Method	Returns	Parameters	Description
`StartCompress`	—	`std::string`	Creates a new DSRC archive and prepares for compression.
`WriteNextRecord`	—	`FastqRecord` record	Writes a new record to file.
`FinishCompress`	—	—	Finalizes the compression of the archive and performs cleanup.
`StartDecompress`	—	`std::string`	Opens a DSRC archive and prepares for decompression.
`ReadNextRecord`	`bool`	`FastqRecord&` record	Reads the next decompressed FASTQ record from archive. Returns false on reaching end-of-file.
`FinishDecompress`	—	—	Finalizes the decompression of the archive and performs cleanup.

Table 3.8: `DsrcArchive` public accessors

Methods	Value type	Description
`Get/Set DnaCompressionLevel`	`int`	Gets/Sets the level of DNA compression: 0–3. Default: 0
`Get/Set QualityCompressionLevel`	`int`	Gets/Sets the level of quality compression: 0–3. Default: 0.
`Get/Set FastqBufferSize`	`int`	Gets/Sets the size of input FASTQ buffer in MB. Default: 8.
`Is/Set LossyCompression`	`bool`	Gets/Sets the state of lossy quality mode enabled. Default: `false`.
`Get/Set TagFieldFilterMask`	`unsigned long`	Gets/Sets the mask describing ID fields to keep in FASTQ record. Default: 0 (keep all fields).
`Get/Set QualityOffset`	`int`	Gets/sets the quality values offset—typically 33 or 64. Default: automatic selection.
`Is/Set Crc32Checking`	`bool`	Gets/sets CRC32 hash calculations. Default: `false`.
`Is/Set PlusRepetition`	`bool`	Gets/sets repetition of ID fields on 'plus' field in FASTQ record.
`Is/Set ColorSpace`	`bool`	Gets/sets indicator whether file is encoded in color space (e.g., ABI SOLiD type).

Table 3.9: `DsrcModule` public methods

Method	Returns	Parameters	Description
`Compress`	`bool`	`std::string fastqFilename`, `std::string dsrcFilename`	Compresses the FASTQ file to DSRC archive.
`Decompress`	`bool`	`std::string dsrcFilename`, `std::string fastqFilename`	Decompresses the DSRC archive to file.

Table 3.10: `DsrcModule` public accessors

Methods	Value type	Description
`Get/Set DnaCompressionLevel`	`int`	Gets/Sets the level of DNA compression: 0–3. Default: 0
`Get/Set QualityCompressionLevel`	`int`	Gets/Sets the level of quality compression: 0–2. Default: 0.
`Get/Set FastqBufferSize`	`int`	Gets/Sets the size of input FASTQ buffer in MB. Default: 8.
`Is/Set LossyCompression`	`bool`	Gets/Sets the state of lossy quality mode enabled. Default: `false`.
`Get/Set TagFieldFilterMask`	`unsigned long`	Gets/Sets the mask describing ID fields to keep in FASTQ record. Default: 0 (keep all fields).
`Get/Set QualityOffset`	`int`	Gets/sets the quality values offset—typically 33 or 64. Default: automatic selection.
`Is/Set Crc32Checking`	`bool`	Gets/sets CRC32 hash calculations. Default: `false`.

3.4 C++ EXAMPLES

3.4.1 Compressing FASTQ file using automated DsrcModule module

```cpp
#include <string>
#include <iostream>

#include "Dsrc.h"
using namespace dsrc::lib;

int main(int argc_, char* argv_[])
{
    if (argc_ != 3)
        return -1;

    const std::string inFastqFile = argv_[0];
    const std::string outDsrcFile = argv_[1];

    try
    {
        DsrcModule dsrc;

        // compress archive in MED mode (flags: -d2 -q2 -b64)
        dsrc.SetQualityCompressionLevel(2);
        dsrc.SetDnaCompressionLevel(2);
        dsrc.SetFastqBufferSizeMB(64);

        // additionally perform CRC32 checking when compressing
        dsrc.SetCrc32Checking(true);

        // set maximum compressing worker threads number
        dsrc.SetThreadsNumber(4);

        dsrc.Compress(inFastqFile, outDsrcFile);
    }
    catch (const DsrcException& e)
    {
        std::cerr << "Error!" << std::endl;
        std::cerr << e.what() << std::endl;
        return -1;
    }

    std::cout << "Success!" << std::endl;
    return 0;
}
```

3.4.2 Decompressing DSRC archive using automated DsrcModule module

```
#include <string>
#include <iostream>

#include "Dsrc.h"
using namespace dsrc::lib;

int main(int argc_, char* argv_[])
{
    if (argc_ != 3)
        return -1;

    const std::string inDsrcFile = argv_[0];
    const std::string outFastqFile = argv_[1];

    try
    {
        DsrcModule dsrc;

        // set maximum decompressing worker threads number
        dsrc.SetThreadsNumber(4);

        dsrc.Decompress(inDsrcFile, outFastqFile);
    }
    catch (const DsrcException& e)
    {
        std::cerr << "Error!" << std::endl;
        std::cerr << e.what() << std::endl;
        return -1;
    }

    std::cerr << "Success!" << std::endl;
    return 0;
}
```

3.4.3 Manual compression using DsrcArchive and FastqFile

```
#include <string>
#include <iostream>

#include "Dsrc.h"
using namespace dsrc::lib;

int main(int argc_, char* argv_[])
{
    if (argc_ != 3)
        return -1;

    const std::string inFastqFile = argv_[0];
    const std::string outDsrcFile = argv_[1];

    // 1. Open FASTQ file
    FastqFile fastqFile;
    try
    {
        fastqFile.Open(inFastqFile);
    }
    catch (const DsrcException& e)
    {
        std::cerr << "Error!\n" << e.what() << std::endl;
        return -1;
    }

    // 2. Create and configure DSRC archive
    DsrcArchive archive;
    try
    {
        // compress quality values using Illumina bining scheme
        archive.SetLossyCompression(true);
        archive.SetQualityCompressionLevel(2);

        archive.SetDnaCompressionLevel(2);
        archive.SetFastqBufferSizeMB(512);

        // discard repeated TAG information in "+" lines
        archive.SetPlusRepetition(false);

        archive.StartCompress(outDsrcFile);
    }
    catch (const DsrcException& e)
    {
        fastqFile.Close();

        std::cerr << "Error!\n" << e.what() << std::endl;
        return -1;
    }

    // 3. When configured, start compression
    FastqRecord rec;
    long int recCount = 0;
    while (fastqFile.ReadNextRecord(rec))
    {
        archive.WriteNextRecord(rec);
        recCount++;
```

```
    }

    // 4. Finish compression and close FASTQ file
    archive.FinishCompress();
    fastqFile.Close();

    std::cout << "Sucess!\nCompressed␣records:␣" << recCount << std::endl;
    return 0;
}
```

3.4.4 Manual decompression using DsrcArchive and FastqFile

```
#include <string>
#include <iostream>

#include "Dsrc.h"
using namespace dsrc::lib;

int main(int argc_, char* argv_[])
{
    if (argc_ != 3)
        return -1;

    const std::string inDsrcFile = argv_[0];
    const std::string outFastqFile = argv_[1];

    // 1. Open DSRC archive
    DsrcArchive archive;
    try
    {
        archive.StartDecompress(inDsrcFile);
    }
    catch (const DsrcException& e)
    {
        std::cerr << "Error!\n" << e.what() << std::endl;
        return -1;
    }

    // 2. Create output FASTQ file
    FastqFile fastqFile;
    try
    {
        fastqFile.Create(outFastqFile);
    }
    catch (const DsrcException& e)
    {
        archive.FinishDecompress();

        std::cerr << "Error!\n" << e.what() << std::endl;
        return -1;
    }

    // 3. Start decompression
    FastqRecord rec;
    long int recCount = 0;
    while (archive.ReadNextRecord(rec))
    {
        fastqFile.WriteNextRecord(rec);
        recCount++;
    }

    // 4. Finish decompression and close FASTQ file
    archive.FinishDecompress();
    fastqFile.Close();

    std::cout << "Success!\nDecompressed␣records:␣" << recCount << std::endl;
    return 0;
}
```