样式定义: TOC 3: 制表位: 39.5 字符, 右对齐,前导符: …

样式定义: TOC 2: 制表位: 39.5 字符, 右对齐,前导符: …

FM3VCF Tool Documentation

Zhen Zuo

2024-4-3

Contents

FM3VCF Tool Documentation

1. Compilation and Generation

1）Required Libraries: gcc, zlib, pthread, openmp

2）On Linux Platforms:

After entering the project folder, navigate to the Linux version directory:

```
cd zM3vcf
```

Compile the vcflib library and generate the vcflib library files to the specified directory:

```
make vcflib
```

Compile and generate the zM3vcf tool and the m3vcf interface testing program m3vcfTest:

```
make
```

3）On Windows Platforms:

First, you need to install the tdm64-gcc compiler (download URL: https://jmeubank.github.io/tdm-gcc/, the win_supportPackage contains the downloaded installer package tdm64-gcc-10.3.0-2.exe). When installing this compiler, be sure to select the option to install openMP, otherwise, the project cannot be compiled correctly (see Appendix 1 for detailed installation instructions).

Additionally, download and compile the zlib library with tdm64-gcc (download URL: https://www.zlib.net/, win_supportPackage contains the downloaded source package zlib-1.2.13.tar.gz). Copy the successfully compiled zconf.h and zlib.h to the include folder in the tdm64-gcc installation directory, and copy the successfully compiled libz.a to the lib folder in the tdm64-gcc installation directory (see Appendix 1 for detailed installation instructions).

Enter the cmd command window, after entering the project folder, navigate to the Windows version directory:

```
cd zM3vcf_win
```

Compile the vcflib library and generate the vclib library files to the specified directory:

```
mingw32-make vcflib
```

Compile and generate the zM3vcf.exe command-line tool and the m3vcf interface testing program m3vcfTest.exe:

```
mingw32-make
```

2. Command Line Tool Functions

1）On Linux Platforms:
compress: vcf->m3vcf
-b Number of records per compression block [1000]
-o Output filename
-O Output file type: M for uncompressed, m for compressed
-t Maximum number of pthreads for reading, writing, and compressing specified by the user, this value must not be less than 3, the default is 8 if not specified (1 reading thread, 1 writing thread, 6 compression threads). **Note: This thread count does not include the number of openMP threads used by vcflib.**
-m Maximum memory amount specified by the user in GB (it is recommended not to specify)
convert：m3vcf->vcf
-o Output filename
-O Output file type: M for uncompressed, m for compressed
2）On Windows Platforms:
The functions and parameters are the same as on Linux platforms.

3. Examples

1）In the project folder, use the zM3vcf command line tool to compress the vcf file in the testFile folder (this file is generated from the first 20 markers of the first 10 samples extracted from the ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20110521/ALL.chr22.phase1_release_v3.20101123.snps_indels_svs.genotypes.vcf.gz file):
ALL.chr22.20Markers.10Samples.vcf.gz
into an m3vcf file under the testFile directory:
ALL.chr22.20Markers.10Samples.m3vcf.gz
Execute the following command on Linux:

```
./zM3vcf compress testFile/ALL.chr22.20Markers.10Samples.vcf.gz -O m -o testFile/ALL.chr22.20Markers.10Samples.m3vcf.gz
```

Execute the following command on Windows:

```
zM3vcf.exe compress testFile\ALL.chr22.20Markers.10Samples.vcf.gz -O m -o testFile\ALL.chr22.20Markers.10Samples.m3vcf.gz
```

2）In the project folder, use the zM3vcf command line tool to convert the m3vcf file created above in the testFile folder (this file is the m3vcf file compressed from the ALL.chr22.20Markers.10Samples.vcf.gz file through the correct operation mentioned above):
ALL.chr22.20Markers.10Samples.m3vcf.gz
back into a vcf file in the current working directory:
ALL.chr22.20Markers.10Samples.vcf.gz
Execute the following command on Linux:

```
./zM3vcf convert testFile/ALL.chr22.20Markers.10Samples.m3vcf.gz -O m -o ALL.chr22.20Markers.10Samples.vcf.gz
```

Execute the following command on Windows:

```
zM3vcf.exe convert testFile\ALL.chr22.20Markers.10Samples.m3vcf.gz -O m -o ALL.chr22.20Markers.10Samples.vcf.gz
```

4. Interface Testing Program

The m3vcfTest program (m3vcfTest.exe on Windows platforms) in the project folder demonstrates the entire process of compressing the vcf file ALL.chr22.20Markers.10Samples.vcf.gz from the testFile folder into the m3vcf file ALL.chr22.20Markers.10Samples.m3vcf.gz in the testFile folder, and the entire process of decompressing and restoring the generated m3vcf file ALL.chr22.20Markers.10Samples.m3vcf.gz from the testFile folder back into the vcf file ALL.chr22.20Markers.10Samples.IF.vcf.gz in the testFile folder, using the interface functions in the m3vcf functionality library. Users can learn how to conveniently and efficiently convert between vcf and m3vcf files in their own code by using the interface functions of the m3vcf functionality library through this testing program.

5. Software Acceleration Principle

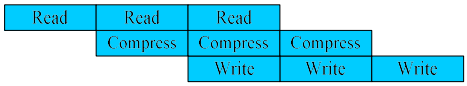
1）The main steps for completing a compression task with the original M3vcftools are:

- Read：Read data from the VCF file and parse it into the corresponding memory structure.
- Compress：Compress VCF file records into M3VCF file records.
- Write：Write the generated data into the M3VCF file.

Since the original program is single-threaded, the process of completing three compression tasks within a single core of the CPU is as follows:

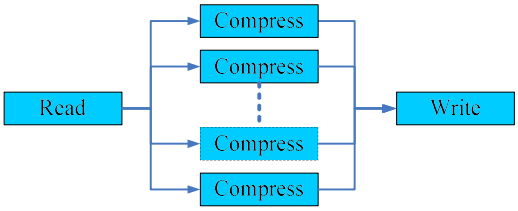
2）zM3vcf firstly separates the Read, Compress, and Write steps, placing them into different threads for execution.

Therefore, completing the same three compression tasks is done in parallel across multiple cores of the CPU, with the process as follows:



Compared vertically with the diagram above, it can be seen that the original program requires nine steps to complete the same three compression tasks, while this program only needs five steps, resulting in a significant increase in speed. Moreover, the more compression tasks, the more pronounced the effect. This method can partially accelerate the process.

3）Through experimentation, it was found that among the Read, Compress, and Write steps, Compress takes up the most time. Therefore, the pthread multi-threading method is used to run the Compress step in parallel across multiple CPU cores. This approach makes the duration of the Read, Compress, and Write steps approximately equal and matched.

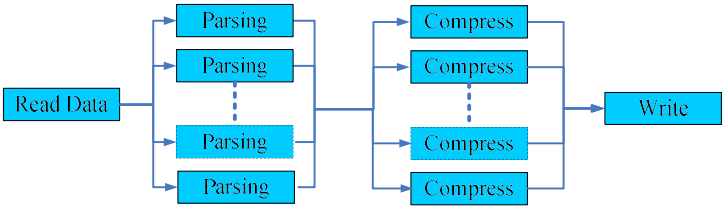


This method can further accelerate the process.

4）The Read step can be subdivided into:

- Read Data： Read data from the hard disk into memory.
- Parsing：Parse the raw data in memory into the corresponding structured memory.

The Parsing part occupies the majority of the time in the Read step, so the Parsing part uses the multi-threading form of OpenMP for acceleration.



The vcflib part of the software completes this functionality. Through this method, further acceleration of the Read step can be achieved.

vcflib Library Documentation

1. vcflib Features

The primary function of this library is to rapidly read VCF files into memory. It supports uncompressed VCF files and zlib-compressed gz format files.

2. vcflib File Structure

Makefile： The compilation rule file for the library, which compiles the library through the make command.

bin： Stores the related files generated after successful compilation of the library **(empty before compilation).**

vcflib.h:The header file of the library, containing all the function interface declarations provided by the library and related macro parameters that users can adjust.

libvcf.so:The dynamic library file of the library.

libvcf.a: The static library file of the library.

inc： Stores the header files of the library.

baseTool.h: Declaration of internal basic functionality functions used in the library.

vcflib.h:The header file of the library, containing all the function interface declarations provided by the library and related macro parameters that users can adjust.

obj： Stores the intermediate files generated during the compilation of the library **(empty before compilation).**

src： Stores the source code files of the library.

baseTools.c:Internal basic functionality functions used in the library.

vcfFile.c： The main functional interface functions of the library.

tst： Stores sample code for the library, including a simpleRead (simple read operation) example program.

3. vcflib Interface

1) File Open/Close Interface

```
VCF_STATUS vcfFileOpen(VCF_FILE *fp,const char *fileName, FILE_MODE fileMode, unsigned int parseItem);
```

Function:
Opens an existing VCF file for reading, supports uncompressed text file formats, and zlib-compressed gz formats. Enables vcflib logging.
Parameters:
fp：A pointer to the VCF_FILE structure. The file handle in the vcflib library, structure definition as follows:

```
typedef struct
{
    union
    {
        FILE * fp;              //Pointer for uncompressed format files
        gzFile gfp;             //Pointer for zlib-compressed format files
    }fp;
    FILE_MODE mode;             //Stores the content of the third parameter fileMode
                                //Indicates the file's compression format:
                                //FILE_MODE_NORMAL :Uncompressed.
                                //FILE_MODE_GZ: zlib-compressed gz format.
    FILE_HEAD head;             //Used to store the file header data of the VCF file
                                //Before any file header operations are performed, this content is empty.
    int numSamples;             //The number of samples in the VCF file
                                //Before any file header operations are performed, this value is 0.
                                //After reading the file header, this value becomes the number of samples in the file.
    unsigned int parsingItems;  //Bit option for user-set parsing fields in the marker
                                //P_GT is for parsing the GT field (currently the default parsing option).
                                //P_DS is for parsing the DS field.
}VCF_FILE;
```

fileName：The path and name of the VCF file to be opened.
fileMode：Specifies the compression format of the file to be opened, FILE_MODE_NORMAL for uncompressed; FILE_MODE_GZ for zlib-compressed gz format.
parseItem：Specifies which fields to parse during the read operation. P_DS|P_GT means parsing DS and GT fields during read operations; P_GT means only parsing GT fields during read operations.
Return Value:
VCF_ERROR：Open failed.
VCF_OK：Open succeeded.

VCF_STATUS vcfFileCreate(VCF_FILE *fp,const char *fileName,FILE_MODE fileMode);

Function:
Creates a VCF file for writing. If the file already exists, its contents will be cleared. Supports uncompressed text file formats and zlib-compressed gz formats. Enables vcflib logging.
Parameters:
fp：A pointer to the VCF_FILE structure; see above **(vcfFileOpen)** for a detailed explanation.
fileName：The path and name of the VCF file to be opened.
fileMode：Specifies the compression format of the file to be opened, FILE_MODE_NORMAL for uncompressed; FILE_MODE_GZ for zlib-compressed gz format.
Return Value:
VCF_ERROR：Creation/opening failed.
VCF_OK：Creation/opening succeeded.

VCF_STATUS vcfFileAppend(VCF_FILE *fp,const char *fileName,FILE_MODE fileMode);

Function:
Creates a VCF file for writing. If the file already exists, its contents will be preserved, and the file write pointer will be positioned at the end of the file. Supports uncompressed text file formats and zlib-compressed gz formats. Enables vcflib logging.
Parameters:
fp：A pointer to the VCF_FILE structure; see above **(vcfFileOpen)** for a detailed explanation.
fileName：The path and name of the VCF file to be opened.
fileMode：Specifies the compression format of the file to be opened, FILE_MODE_NORMAL for uncompressed; FILE_MODE_GZ for zlib-compressed gz format.
Return Value:
VCF_ERROR：Append/opening failed.
VCF_OK： Append/opening succeeded.

```
VCF_STATUS vcfFileClose(VCF_FILE *fp);
```

Function:
Closes the already opened file, clears, and releases the memory space occupied by the file header. And closes vcflib logging.
Parameters:
fp：A pointer to the VCF_FILE structure; see above **(vcfFileOpen)** for a detailed explanation.
Return Value:
VCF_ERROR：Closing failed.
VCF_OK：Closing succeeded.

2) File Read Interfaces

```
VCF_STATUS vcfFileReadLine(VCF_FILE *fp,char *lineStr,int lineSize);
```

Function:
Reads the current line from the vcf file pointed to by the opened fp, and stores the content in lineStr.
Parameters:
fp：A pointer to the VCF_FILE structure.
lineStr：A pointer to the memory that stores the file content.
lineSize：The length of the memory pointed to by lineStr.
Return Value:
VCF_ERROR：Read failed.
VCF_OK：Read succeeded.

```
VCF_STATUS vcfFileReadHead(VCF_FILE *fp);
```

Function:
Reads the vcf file header part from the vcf file pointed to by the opened fp, and stores the content in the structure pointed to by fp -> head.
Parameters:
fp：A pointer to the VCF_FILE structure, where its member variable head is a FILE_HEAD type variable used to store the vcf file header. The structure definition is as follows:

```
typedef struct
{
    char **metaInfoLines;        //For storing vcf file header's meta information lines
    int numMetaInfoLines;        //Number of meta information lines in the vcf file header
    char *headerLine;            // For storing the vcf file header's header line
}FILE_HEAD;
```

Return Value:
VCF_ERROR： Read failed.
VCF_OK：Read succeeded.

VCF_STATUS getNumMetaInfoLines(FILE_HEAD *fhp,int *NumMetaInfoLines);

Function:
Gets the number of meta information lines from the vcf file header pointed to by fhp.
Parameters:
fhp： A pointer to the FILE_HEAD structure. See above **(vcfFileReadHead)** for a detailed explanation.
NumMetaInfoLines：The number of meta information lines in the vcf file header pointed to by fhp.
Return Value:
VCF_ERROR：Read failed.
VCF_OK：Read succeeded.

VCF_STATUS getNumSamples(VCF_FILE *fp,int *NumSamples);

Function:
Gets the number of samples from the vcf file pointed to by the opened fp.
Parameters:
fp：A pointer to the VCF_FILE structure.
NumSamples：The obtained number of samples.
Return Value:
VCF_ERROR：Read failed.
VCF_OK：Read succeeded.

```
char* vcfFileParseDataLineInfo(char *lineStr,DATA_INFO *dataInfo);
```

Function:
Parses the first nine fields of a vcf file data line contained in lineStr into the DATA_INFO structure pointed to by dataInfo. After the operation, the first nine fields are removed from lineStr.
Parameters:
lineStr：A line of data from a vcf file.
dataInfo：A pointer to the DATA_INFO structure. The vcflib library uses it to store the first nine fields of each data line in a vcf file. The structure definition is as follows:

```
typedef struct
{
    char *chrom;
    char *pos;
    char *ID;
    char *ref;
    char *alt;
    char *qual;
    char *filter;
    char *info;
    char *format;
}DATA_INFO;
```

Return Value:
The vcf file data line with the first nine fields removed.

```
VCF_STATUS vcfFileParseDataLine(VCF_FILE *fp,char *lineStr,DATA_LINE *dlp);
```

Function:
Parses a line of vcf file data contained in lineStr into the DATA_LINE structure pointed to by dlp.
Parameters:
fp： A pointer to the VCF_FILE structure.
lineStr：A line of data from a vcf file.
dlp：A pointer to the DATA_LINE structure. The vcflib library uses it to store each line of data from a vcf file. The structure definition is as follows:

```
typedef struct
{
    char *rawDataLine;          //The original string of this data line, not the entire string
                                //The tabs (\t) in the first nine items have already been replaced with null terminators (\0).
    DATA_INFO dataInfo;         //The first nine fields of this data line
    char *samplesRawString;     //The original string of data content after removing the first nine fields
    char *gtData;               //The GT part data of this line
    float *dsData;              //The DS part data of this line
    int numSamples;             //The number of samples in this data line
}DATA_LINE;
```

Return Value:
VCF_ERROR：Parsing failed.
VCF_OK：Parsing succeeded.

VCF_STATUS vcfFileReadDataLine(VCF_FILE *fp,DATA_LINE *dlp);

Function:
Reads the current data line from the vcf file pointed to by the opened fp, and stores the content in the DATA_LINE structure pointed to by dlp.
Parameters:
fp：A pointer to the VCF_FILE structure.
dlp：A pointer to the DATA_LINE structure. See above **(vcfFileParseDataLine)** for a detailed explanation.
Return Value:
VCF_ERROR：Read failed.
VCF_OK：Read succeeded.

VCF_STATUS vcfFileReadDataBlock(VCF_FILE *fp,DATA_BLOCK *dbp,int numLines);

Function:
Reads numLines data lines from the vcf file pointed to by the opened fp, and stores the content in the DATA_BLOCK structure pointed to by dbp.

Parameters:
fp：A pointer to the VCF_FILE structure.
dbp：A pointer to the DATA_BLOCK structure. The vcflib library uses it to store each block of data from a vcf file. The structure definition is as follows:

```
typedef struct
{
    DATA_LINE *dataLines;          //Pointer to an array storing multiple lines of data in this block
    int numDataLines;              //The number of data lines in this block
}DATA_BLOCK;
```

numLines：The number of lines to be read from the file at once.
Return Value:
VCF_ERROR：Read failed.
VCF_OK：Read succeeded.

VCF_STATUS vcfFileReadDataBlockOverlap1Line(VCF_FILE *fp,DATA_BLOCK *dbp,int numLines);

Function:
Reads numLines data lines from the vcf file pointed to by the opened fp, and stores the content in the DATA_BLOCK structure pointed to by dbp. Except for the first time, the first line of each subsequent data block read overlaps with the last line of the previous data block read.
Parameters:：
fp：A pointer to the VCF_FILE structure.
dbp：A pointer to the DATA_BLOCK structure. For a detailed explanation, see above **(vcfFileReadDataBlock)**.
numLines：The number of lines to read from the file in one go.
Return Value:
VCF_ERROR：Reading failed.
VCF_OK：Reading succeeded.

3) File Write Interfaces

VCF_STATUS vcfFileWriteLine(VCF_FILE *fp,char *lineStr);

Function:
Writes the content of lineStr as a line into the vcf file pointed to by the opened fp.
Parameters:
fp：A pointer to the VCF_FILE structure.
lineStr： A pointer to the memory storing the content to be written into the file.
Return Value:
VCF_ERROR：Writing failed.
VCF_OK：Writing succeeded.

VCF_STATUS vcfFileWriteHead(VCF_FILE *fp,FILE_HEAD *fhp);

Function:
Writes the vcf file header information contained in the structure pointed to by fhp into the vcf file pointed to by the opened fp.
Parameters:
fp：A pointer to the VCF_FILE structure.
fhp：A pointer to the FILE_HEAD structure.
Return Value:
VCF_ERROR：Writing failed.
VCF_OK：Writing succeeded.

VCF_STATUS vcfFileAddMetaInfoLine(FILE_HEAD *fhp,int posIndex,char *MetaInfoLine);

Function:
Adds the meta information line contained in MetaInfoLine to the vcf file header structure pointed to by fhp, specifying the position of the newly added meta information line in the meta information as the posIndexth line.
Parameters:
fhp：A pointer to the FILE_HEAD structure.
posIndex：The position to insert the new meta information line in the meta information.
MetaInfoLine：The content of the meta information to be added.
Return Value:
VCF_ERROR：Writing failed.
VCF_OK：Writing succeeded.

```
VCF_STATUS vcfFileRemoveMetaInfoLine(FILE_HEAD *fhp,int posIndex);
```

Function:
Removes the posIndexth meta information line from the vcf file header structure pointed to by fhp.
Parameters:
fhp：A pointer to the FILE_HEAD structure.
posIndex：The position of the meta information line to be removed.
Return Value:
VCF_ERROR：Removal failed.
VCF_OK：Removal succeeded.

4) Other Interfaces

```
void clearFileHead(FILE_HEAD *fhp);
```

Function:
Clears the data in the FILE_HEAD structure pointed to by fhp and releases its memory space.
Parameters:
fhp：A pointer to the FILE_HEAD structure.
Return Value:
None.

```
void clearDataLine(DATA_LINE *dlp);
```

Function:
Clears the data in the DATA_LINE structure pointed to by dlp and releases its memory space. (Note: If the vcfFileReadDataLine interface is called and then called again with the same dlp parameter without first calling this interface to clear and release the content in the DATA_LINE structure, it will result in a memory leak.)
Parameters:
dlp：A pointer to the DATA_LINE structure.
Return Value:
None.

void clearDataBlock(DATA_BLOCK *dbp);

Function:
Clears the data in the DATA_BLOCK structure pointed to by dbp and releases its memory space.
Parameters:
dbp： A pointer to the DATA_BLOCK structure.
Return Value:
None.

VCF_STATUS vcfPopSubString(char **lineStr,char *subStr);

Function:
Extracts the first substring separated by spaces, tabs ‘\t’, or newline characters ‘\n’ from the string pointed to by lineStr, and stores it in subStr. After the operation, the extracted substring is removed from lineStr.
Parameters:
lineStr：The long string to be truncated.
subStr: The extracted substring.
Return Value:
VCF_ERROR： Extraction failed; or the long string to be truncated is empty (the entire long string has been extracted).
VCF_OK：Extraction succeeded.

void printDataLine(DATA_LINE *dlp);

Function:
Prints and displays the values of all member variables in the DATA_LINE structure pointed to by dlp, used to show the content stored in dlp.
Parameters:
dlp：A pointer to the DATA_LINE structure.
Return Value:
None.

5) Full Format VCF Interface

```
VCF_STATUS vcfFileParseDataLine_allFormat(VCF_FILE *fp,char *lineStr,DATA_LINE_ALL_FORMAT * dlafp);
```

Function:
Parses a line of vcf file data contained in lineStr into the DATA_LINE_ALL_FORMAT structure pointed to by dlafp.
Parameters:
fp： A pointer to the VCF_FILE structure.
lineStr：A line of data from a vcf file.
dlafp：A pointer to the DATA_LINE_ALL_FORMAT structure. The vcflib library uses this to store each line of data from a vcf file that supports all formats, defined as follows:

```
    typedef struct
    {
        char *rawDataLine;          //The original string address of this data line, not the whole string.
                                    //The tabs (\t) in the first nine items have already been replaced with null terminators (\0).
        DATA_INFO dataInfo;         //The first nine items of this data line.
        int numFormats;             //The number of format items in this data line.
        DATA_FORMAT_STR *dataFormatStr;//Pointer to DATA_FORMAT_STR structure.
                                    //space allocated based on numFormats.
                                    //storing all format data contents of this data line.
        int numSamples;             //The number of samples in this data line.
    }DATA_LINE;
    typedef struct
    {
        char *format;               //Format name.
        char **dataStr;             //The specific value of each sample for this format in string form.
}DATA_FORMAT_STR;
```

Return Value:
VCF_ERROR：Parsing failed.
VCF_OK：Parsing succeeded.

VCF_STATUS vcfFileReadDataLine_allFormat(VCF_FILE *fp,DATA_LINE_ALL_FORMAT *dlafp);

Function:
Reads the current data line from the vcf file pointed to by the opened fp, and stores the content in the DATA_LINE_ALL_FORMAT structure pointed to by dlafp.
Parameters:
fp：A pointer to the VCF_FILE structure.
dlafp： A pointer to the DATA_LINE_ALL_FORMAT structure. For a detailed explanation, see above **(vcfFileParseDataLine_allFormat)**.
Return Value:
VCF_ERROR：Reading failed.
VCF_OK：Reading succeeded.

VCF_STATUS vcfFileReadDataBlock_allFormat(VCF_FILE *fp,DATA_BLOCK_ALL_FORMAT *dbafp,int numLines);

Function:
Reads numLines data lines from the vcf file pointed to by the opened fp, and stores the content in the DATA_BLOCK_ALL_FORMAT structure pointed to by dbafp.
Parameters:
fp：A pointer to the VCF_FILE structure.
dbafp：A pointer to the DATA_BLOCK_ALL_FORMAT structure. The vcflib library uses this to store blocks of data from a vcf file, defined as follows:

```
typedef struct
{
    DATA_LINE_ALL_FORMAT *dataLines;   //Pointer to an array storing multiple lines of data in this block.
    int numDataLines;                  // The number of data lines in this block.
}DATA_BLOCK_ALL_FORMAT;
```

numLines：The number of lines to be read from the file at once.
Return Value:
VCF_ERROR：Reading failed.
VCF_OK：Reading succeeded.

void clearDataLine_allFormat (DATA_LINE_ALL_FORMAT *dlafp);

Function:
Clears the data in the DATA_LINE_ALL_FORMAT structure pointed to by dlafp and releases its memory space. (Note: If the vcfFileReadDataLine_allFormat interface is called and then called again with the same dlafp parameter without first calling this interface to clear and release the content in the DATA_LINE_ALL_FORMAT structure, it will result in a memory leak.)
Parameters:
dlafp：A pointer to the DATA_LINE_ALL_FORMAT structure.
Return Value:
None.

void clearDataBlock_allFormat (DATA_BLOCK_ALL_FORMAT *dbafp);

Function:
Clears the data in the DATA_BLOCK_ALL_FORMAT structure pointed to by dbafp and releases its memory space.
Parameters:
dbafp：A pointer to the DATA_BLOCK_ALL_FORMAT structure.
Return Value:
None.

void printDataLine_allFormat (DATA_LINE_ALL_FORMAT *dlafp);

Function:
Prints and displays the values of all member variables in the DATA_LINE_ALL_FORMAT structure pointed to by dlafp, used to show the content stored in dlafp.
Parameters:
dlafp：A pointer to the DATA_LINE_ALL_FORMAT structure.
Return Value:
None.

4. vcflib User-Adjustable Macro Parameters

Within the inc/vcflib.h file, there are several macro definitions that users can dynamically adjust according to different needs.

BT_MAX_LINE_SIZE： The maximum number of characters in a line within a vcf file that the library currently supports. The current value is set to 6M. If an error occurs due to the need to process vcf files with particularly large numbers of characters per line, you can increase this value and recompile.

OPEN_MP_THREAD_NUM： The number of OpenMP threads opened when parsing data from vcf files. The current value is set to 10. Users can increase this value to speed up the library's reading of vcf files, but the overall speed is limited by disk I/O and CPU core numbers.

Tools Description

The zM3vcf folder provides some small utility programs that are used for testing the speed and efficiency of the project.

1. Compilation and Generation

The compilation method is as follows:

1）On Linux Platforms:

After entering the project folder, navigate to the Linux version directory:

```
cd zM3vcf
```

Compile the vcflib library and generate the vcflib library files to the specified directory:

```
make vcflib
```

Enter the tools folder:

```
cd tools
```

Compile and generate the tools:

```
make
```

2）On Windows Platforms:

Open the cmd command window, after entering the project folder, navigate to the Windows version directory:

```
cd zM3vcf_win
```

Compile the vcflib library and generate the vcflib library files to the specified directory:

```
mingw32-make vcflib
```

Enter the tools folder:

```
cd tools
```

Compile and generate the tools:

```
mingw32-make
```

2. Tool Descriptions

1）pVcf

- Source code located in the tools folder as copyCreateVcf.c.。
- Functionality: Uses a real VCF file to fill and create a new VCF test file with n times more samples than the original VCF file. It expands each line of the real VCF by n times (with the order of expansion randomly shuffled) to synthesize a line in the target VCF. Requires a real VCF file.
- Usage:

```
pVcf sourceVCFFile sampleTimes targetVCFFile
```

sourceVCFFile: Path to the original VCF file.
sampleTimes: Multiplier for how many times the original VCF file's sample count is to be expanded.
targetVCFFile: Path for the newly generated VCF file.

- Example: To use this tool to expand the sample count by 2 times for the file ALL.chr22.20Markers.10Samples.vcf.gz in the testFile folder and generate a new file ALL.chr22.20Markers.20Samples.vcf.gz in the testFile folder, use the following command (assuming the current working directory is the tools folder):

```
./pVcf ../testFile/ ALL.chr22.20Markers.10Samples.vcf.gz 2 ../
ALL.chr22.20Markers.20Samples.vcf.gz
```

（For Windows, refer to the Linux operation described above.）

2）gVMH

- Source code located in the tools folder as getVcfMH.c.
- Functionality: Prints the number of markers and samples in a specified VCF file, and tests if every marker in the VCF file conforms to the sample count.
- Usage:

```
gVMH VCFFile
```

VCFFile: Path to the VCF file to be examined.

- Example: To print the number of samples and markers in ALL.chr22.20Markers.10Samples.vcf.gz from the testFile folder and check if every marker's sample count is consistent, use the following command (assuming the current working directory is the tools folder):

```
./gVMH ../testFile/ ALL.chr22.20Markers.10Samples.vcf.gz
```

（For Windows, refer to the Linux operation described above.）

3）vCM

- Source code located in the tools folder as vcfCutMarkers.c.
- Functionality: Cuts the first n markers and the first m samples of each marker from a specified large file to generate a new smaller VCF file.
- Usage:

```
vCM sourceVCFFile targetVCFFile markersNumber samplesNumber
```

sourceVCFFile: Path to the large VCF file that serves as the data source.
targetVCFFile: Path for the newly generated smaller VCF file.
markersNumber: The number of markers to cut from the large file. If this value is 0, retain all markers.
samplesNumber: The number of samples to cut from the large file. If this value is 0, retain all samples.

- Example: To cut 10 markers and 5 samples from ALL.chr22.20Markers.10Samples.vcf.gz in the testFile folder and generate ALL.chr22.10Markers.5Samples.vcf.gz in the testFile, use the following command (assuming the current working directory is the tools folder):

```
./vCM ../testFile/
ALL.chr22.20Markers.10Samples.vcf.gz ../testFile/ALL.chr22.10Markers.5Samples.vcf.gz
10 5
```

（For Windows, refer to the Linux operation described above.）

4）readVcfTest

- Source code located in the tools folder as readVcfTest.c.
- Functionality: Tests the speed and resource usage of reading a VCF file with vcflib. It is read-only and performs no operations.

5）readvcf.c

- Used to test the speed and resource usage of reading a VCF file with htslib. It is read-only and performs no operations.

6）cmWatcher.sh

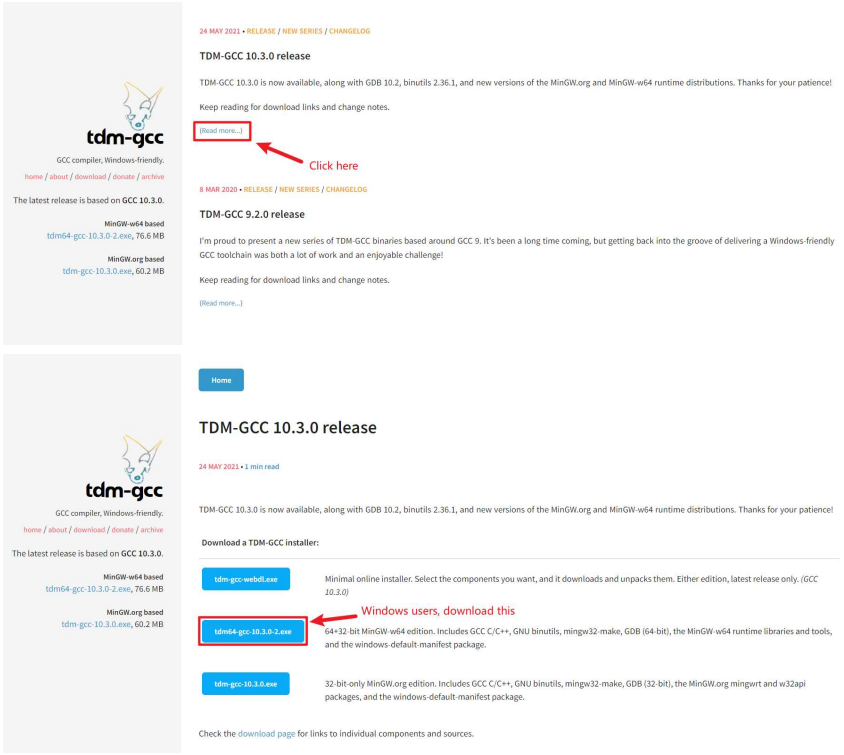
- Used to monitor the peak CPU and memory usage of a specified process and output the results to a specified text document.

Appendix 1: Setting Up a Windows Compilation Environment

1. Installing tdm64-gcc on Windows:

1) tdm64-gcc Installer Package

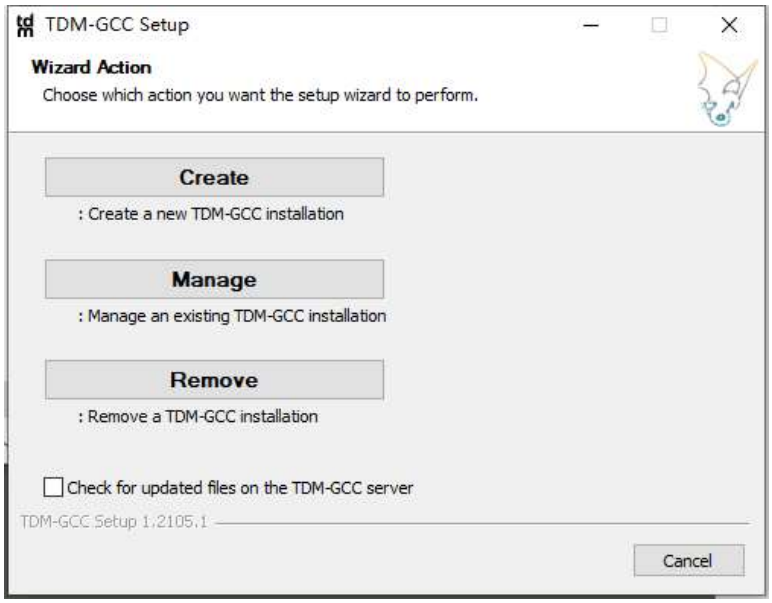
Visit the official website at https://jmeubank.github.io/tdm-gcc/ and directly download the installer package.



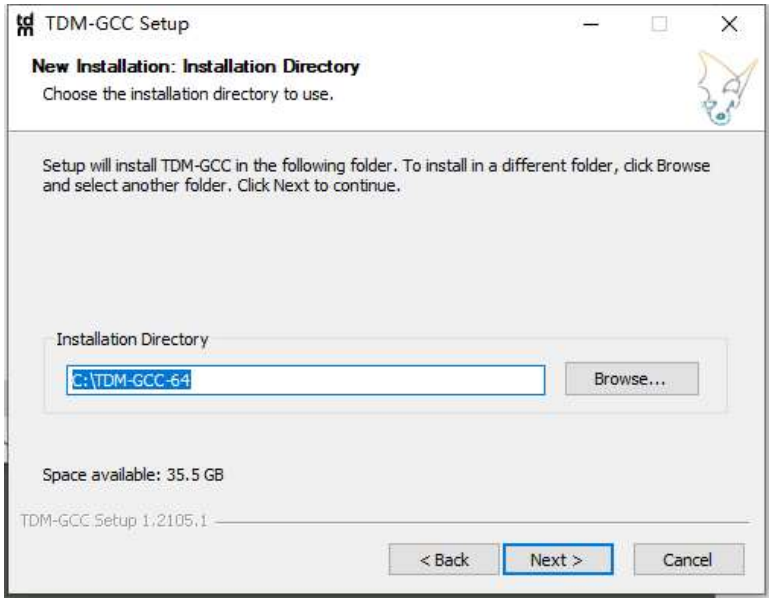
Or it's available in the win_supportPackage I provided, with the downloaded installer package tdm64-gcc-10.3.0-2.exe.

2) Installing tdm64-gcc

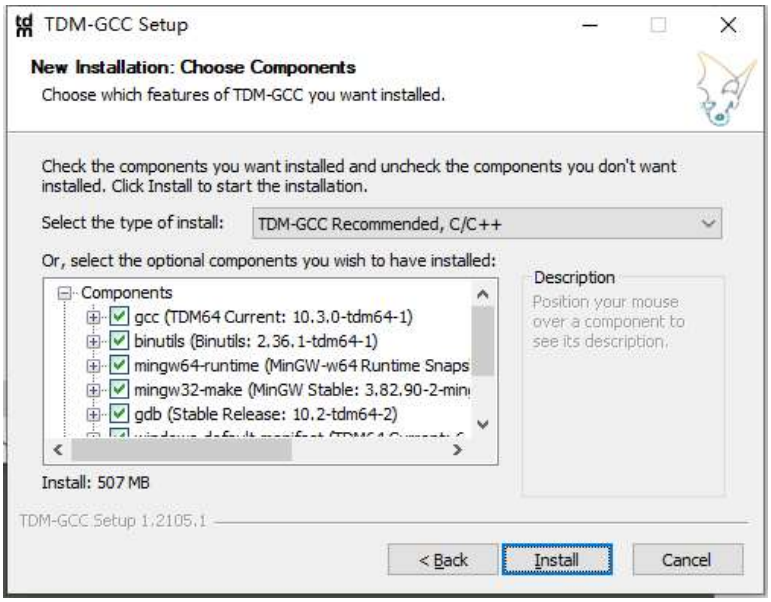
Below is the pop-up installation wizard:



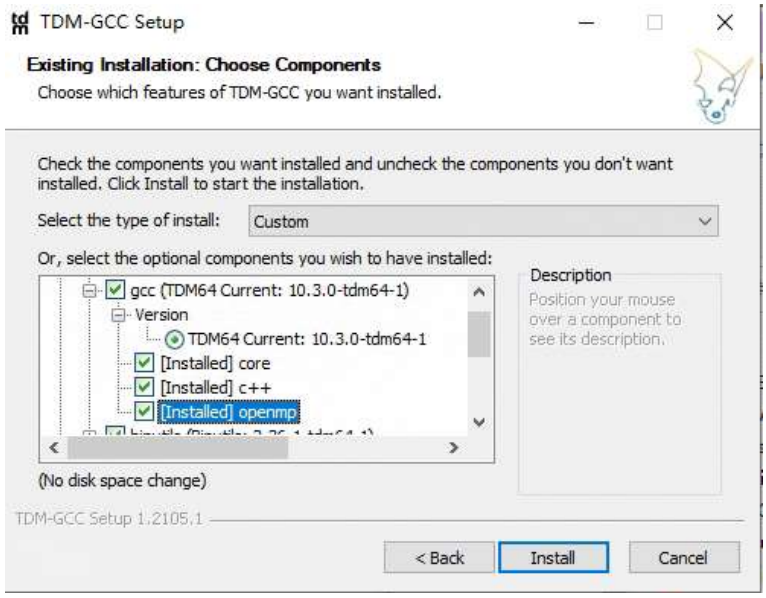
Uncheck the "check for updated files on the TDM-GCC server" option, then click Create.



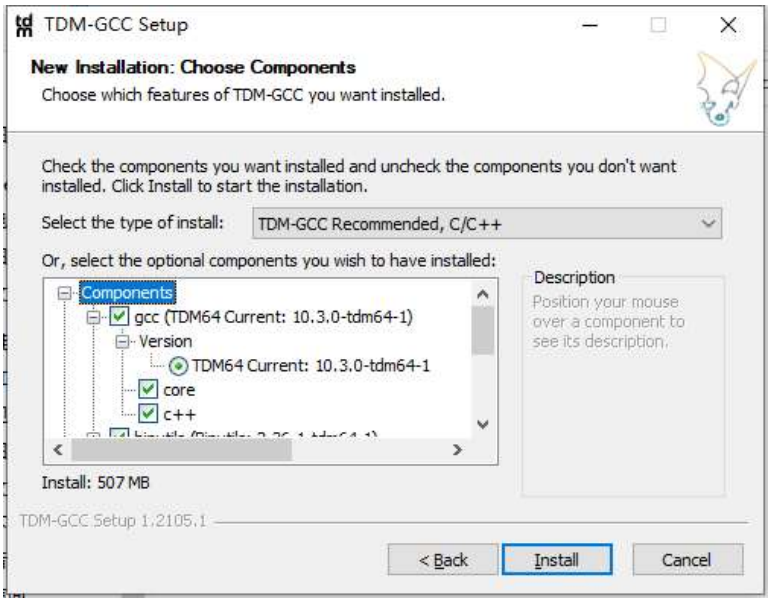
Choose an installation path, avoiding Chinese characters and special symbols, then click Next.



Select all options. Pay special attention to ensure that "gcc" includes openmp; it must be selected. As shown in the following image:



If openmp is not included in the first item, as shown in the image below:



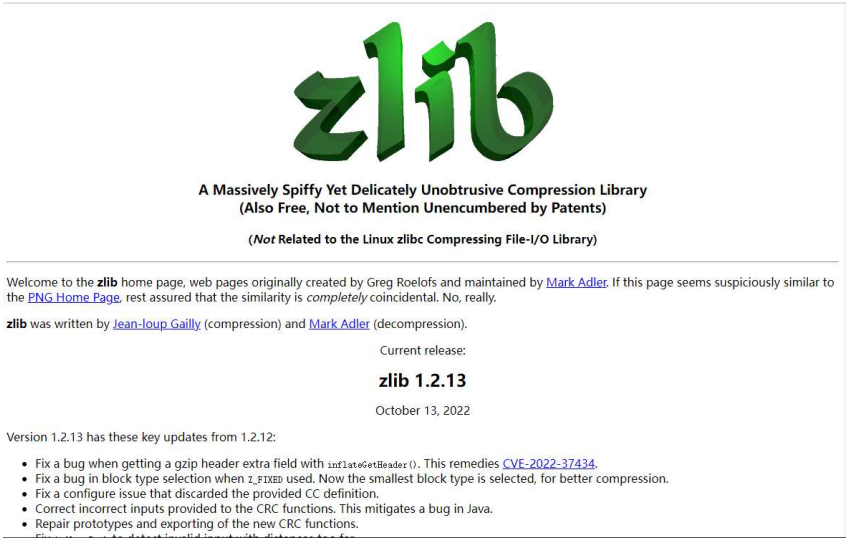
Then, after the installation is complete, you must double-click the installer package again to enter the installation wizard, manage the installed tdm-gcc, and add openmp. Otherwise, vcflib cannot be compiled properly.

Finally, click Install to complete the installation.

2. Compilation and Porting of the Zlib Library

1) Zlib Installation Package

Visit the official website https://www.zlib.net/ to directly download the package.



A Massively Spiffy Yet Delicately Unobtrusive Compression Library
(Also Free, Not to Mention Unencumbered by Patents)

(*Not* Related to the Linux zlibc Compressing File-I/O Library)

Welcome to the **zlib** home page, web pages originally created by Greg Roelofs and maintained by Mark Adler. If this page seems suspiciously similar to the PNG Home Page, rest assured that the similarity is *completely* coincidental. No, really.

zlib was written by Jean-loup Gailly (compression) and Mark Adler (decompression).

Current release:

zlib 1.2.13

October 13, 2022

Version 1.2.13 has these key updates from 1.2.12:

- Fix a bug when getting a gzip header extra field with `inflateGetHeader()`. This remedies CVE-2022-37434.
- Fix a bug in block type selection when `Z_FIXED` used. Now the smallest block type is selected, for better compression.
- Fix a configure issue that discarded the provided CC definition.
- Correct incorrect inputs provided to the CRC functions. This mitigates a bug in Java.
- Repair prototypes and exporting of the new CRC functions.

Or use the downloaded tar package zlib-1.2.13.tar.gz available in the win_supportPackage I provided.

2) Compiling and Porting Zlib

On Windows, open the CMD command line.
Extract the zlib package to any directory. In this example, the zlib package is placed in d:\tmp and extracted locally with the following commands:

```
d:
cd tmp
tar -xzvf zlib-1.2.13.tar.gz
```

Go into the extracted zlib folder, copy the win32\Makefile.gcc to the zlib folder with the following commands:

```
cd zlib-1.2.13
copy win32\Makefile.gcc makefile.gcc
```

Compile zlib with the command:

```
Mingw32-make -f makefile.gcc
```

After compilation is completed, copy the successfully compiled zconf.h and zlib.h to the include folder in the tdm64-gcc installation directory; copy the successfully compiled libz.a to the lib folder in the tdm64-gcc installation directory to finish.