User Guide

ExSeisPIOL Documentation

Cathal Ó Broin, Michael Lysaght

1st September 2016

Table of Contents

Introduction

The ExSeisPIOL is a parallel library for enabling the processing of seismology traces and related parameters such as the coordinates of the source and receiver etc., from SEG-Y files and other file formats which are interconvertible with SEG-Y.

The API which is exposed at the lowest level to the end-user is known as the *File API*. The File API is intended to be simple in terms abstraction and usage, The end-user employs the API to directly extract traces and parameters.

In keeping with the goal of simplicity for this API, the handling of memory limitations is left to the end user. The File API also assumes that the end user has decided on their own decomposition strategy.

At present, no MPI-IO collective action[1] is taken in the File Layer and no inter-processor communication are performed. This will change in the future, although the external function calls are expected to remain static or be modified in a minimal way.[2]

The *Set API* will manage parallelisation and decomposition and is orientated around the operations the end user requires to be performed on traces in the file rather than on accessing of the data. This will allow ExSeisPIOL developers to tune the PIOL for specific operations by being able to fully determine access patterns, caching and memory management.

The purpose of this document is to describe the general approach for using the File API. The code documentation should be checked for the appropriate values of specific parameters.

This document will now describe the initialisation of the PIOL and ancillary functionality accessed through the PIOL before discussing the File API in more detail. Then size related functions for correctly allocating memory are also discussed. Following this, two examples are shown, one for writing a SEG-Y file using the library and another which reads a file and writes out a new file.

[1] i.e multiple processes using communication to optimise the I/O pattern
[2] No guarantees are made for backwards compatibility for pre-release APIs.

PIOL API

A list of operations is given in Table 1. The PIOL must first be initialised in an initial call to initMPIOL() before any calls are made to open files. While the handle exists, MPI will be active. The PIOL can be set to close with a call to closePIOL(). If any file is open with the PIOL, de-initialisation will automatically happen when the last file is closed rather than with the closePIOL() call.

Operation	Function Example
Initialise the API using MPI for communication	C99 ExSeisHandle piol = initMPIOL();
	C++14 auto piol = std::make_shared<ExSeisPIOL>();
De-initialise the API	C99 closePIOL(piol);
	C++14 (automatic)
Get the process MPI rank	C99 getRank(piol);
	C++14 piol->comm->getRank();
Get the number of MPI processes	C99 getNumRank(piol);
	C++14 piol->comm->getNumRank();
Invoke a barrier across all processes involved in I/O	C99 barrier(piol);
	C++14 piol->comm->barrier();
Check the log (terminate on error)	C99 isErr(piol);
	C++14 piol->isErr();

Table 1. A table of PIOL related system calls

File API

A file is initially opened with a call to openReadFile() or openWriteFile(). All subsequent file operations rely on the handle returned. A file is closed by passing the handle to closeFile().

All subsequent operations are listed in Table 2. Three types of operations are listed:

1. **File Parameters -** File parameters are single numerical values or strings which characterise a file: Read/write the number of traces in the file; the number of samples per trace; the increment between samples; and the text header (converted to ASCII where applicable).
2. **Individual Trace Parameters -** Read/write associated parameters; coordinates and the inline/crossline grid of a trace. A single call can be made for parameters in consecutive traces.
3. **Traces -** Read/write traces. A consecutive set of traces can be read in a single call.

Operation	Function Example
Open a SEG-Y file as read-only.	C99 ExSeisFile in = openReadFile(piol, “name.segy”);
	C++14 auto file = std::make_unique<File::SEGY>(piol, outname, FileMode::Read);
Open a SEG-Y file as write-only.	C99 ExSeisFile fh = openWriteFile(piol, “name.segy”);
	C++14 File::SEGY file(piol, outname, FileMode::Write);
Close a file	C99 closeFile(fh);
	C++14 (automatic)
Read text header	C99 const char * str = readText(fh);
	C++14 std::string text = file.readText();
Read number of samples per trace	C99 size_t ns = readNs(fh);
	C++14 size_t ns file.readNs();
Read number of traces	C99 size_t nt = readNt(fh);
	C++14 size_t nt = file.readNt();
Read increment	C99 float inc = readInc(fh);
	C++14 float inc = file.readinc();
Write text header	C99 writeText(fh, str);
	C++14 file.writeText(text);
Write number of samples per trace	C99 writeNs(fh, ns);
	C++14 file.writeNs(ns);
Write number of traces	C99 writeNt(fh, nt);
	C++14 file.writeNt(nt);
Write increment	C99 writeInc(fh, inc);
	C++14 file.writeInc(inc);

Table 2. A table of File API calls.

Operation	Function Example
Read *num* coordinate points (E.g source) from the offset.	C99 readCoordPoint(fh, Src, offset, num, array);
	C++14 file.readCoordPoint(Coord::Src, offset, num, array);
Read *num* grid points (i.e inline, xline) from the offset.	C99 readGridPoint(fh, Line, offset, num, array);
	C++14 file.readGridPoint(Grid::Line, offset, num, array);
Read *num* structs of trace parameters from the offset	C99 readTraceParam(fh, offset, num, prmarray);
	C++14 file.readTraceParam(offset, num, prmarray);
Write *num* structs of trace parameters from the offset	C99 writeTraceParam(fh, offset, num, prmarray);
	C++14 file.writeTraceParam(offset, num, prmarray);
Read *num* traces from the offset	C99 readTrace(fh, offset, num, array);
	C++14 file.readTrace(offset, num, array);
Write *num* traces.	C99 writeTrace(fh, offset, num, array);
	C++14 file.writeTrace(offset, num, array);

Table 2. A table of File API calls (contd.).

SEG-Y Size queries

To deal with the allocation of memory and the management of memory, some function calls are required by the end-user. These calls are listed in Table 3.

Operation	Function Example
Get the size of the text field.	C99 size_t sz = getSEGYTextSz();
	C++14 size_t sz = SEGSz::getTextSz();

Get the size of the traces (bytes).	C99 size_t ns = getSEGYTraceLen(ns)
	C++14 size_t ns = SEGSZ::getTraceLen(ns);
Get a file size given the *ns* and *nt* values.	C99 size_t fsz = getSEGYFileSz();
	C++14 size_t fsz = SEGSz::getFileSz();
Get the memory needed to read a parameter.	C99 size_t sz = getSEGYParamSz()
	C++14 size_t sz = sizeof(TraceParam) + SEGSz::getMDSz();

Table 3. A table of SEG-Y Size API calls.

Example 1

In this example we generate a small SEG-Y file called *test.segy* based on simple synthetic data.

```
//Each PIOL function call is highlighted in green.
#include <stdlib.h>
#include "cfileapi.h"
int main(void)
{
    //Initialise the PIOL by creating an ExSeisPIOL object
    ExSeisHandle piol = initMPIOL();

    size_t rank = getRank(piol);
    size_t numRank = getNumRank(piol);

    //Create a SEGY file object
    ExSeisFile fh = openWriteFile(piol, "test.segy");

    size_t lnt = 40;    //number of traces
    size_t ns = 300;    //samples per trace
    double inc = 0.04;  //increment between samples

    //Write some header parameters
    writeNs(fh, ns);
    writeNt(fh, lnt*numRank);
    writeInc(fh, inc);
    writeText(fh, "Test file\n");

    //Set and write some trace parameters
    TraceParam * prm = calloc(lnt, sizeof(TraceParam));
    for (size_t j = 0; j < lnt; j++)
    {
        float k = lnt*rank+j;
        prm[j].src.x = 1600.0 + k;
        prm[j].src.y = 2400.0 + k;
        prm[j].rcv.x = 100000.0 + k;
```

```
        prm[j].rcv.y = 3000000.0 + k;
        prm[j].cmp.x = 10000.0 + k;
        prm[j].cmp.y = 4000.0 + k;
        prm[j].line.il = 2400 + k;
        prm[j].line.xl = 1600 + k;
        prm[j].tn = lnt*rank+j;
    }
    writeTraceParam(fh, lnt*rank, lnt, prm);
    free(prm);

    //Set and write some traces
    float * trc = calloc(lnt*ns, sizeof(float));
    for (size_t j = 0; j < lnt*ns; j++)
        trc[j] = (float)(lnt*rank+j);
    writeTrace(fh, lnt*rank, lnt, trc);
    free(trc);

    //Close the file handle and close the piol
    closeFile(fh);
    closePIOL(piol);

    return 0;
}
```

Example 2

In this example we read in a small SEG-Y file called *test.segy* and create a new SEG-Y file called *test1.segy*.

```
#include "cfileapi.h"
#include <stdlib.h>

int main(void)
{
    //Initialise the PIOL by creating an ExSeisPIOL object
    ExSeisHandle piol = initMPIOL();
    size_t rank = getRank(piol);

    //Create a SEGY file object for input
    ExSeisFile ifh = openReadFile(piol, "test.segy");
    isErr(piol);

    //Create some local variables based on the input file
    size_t nt = readNt(ifh);
    size_t ns = readNs(ifh);

    //lnt is a local subset of the number of traces
    size_t lnt = nt / getNumRank(piol);

    //Alloc the required memory for the data we want
    float * trace = malloc(lnt * getSEGYTraceLen(ns));
    TraceParam * trhdr = malloc(lnt * sizeof(TraceParam));

    //Create a SEGY file object for output
    ExSeisFile ofh = openWriteFile(piol, "test1.segy");
    isErr(piol);

    //Write the headers based on the input file.
    writeText(ofh, readText(ifh));
    writeNs(ofh, readNs(ifh));
    writeNt(ofh, readNt(ifh));
    writeInc(ofh, readInc(ifh));

    //Read the trace parameters from the input file and to the output
    readTraceParam(ifh, lnt * rank, lnt, trhdr);
    writeTraceParam(ofh, lnt * rank, lnt, trhdr);
```

```
    //Read the traces from the input file and to the output
    readTrace(ifh, lnt * rank, lnt, trace);
    writeTrace(ofh, lnt * rank, lnt, trace);

    free(trace);
    free(trhdr);

    //Close the file handles and close the piol
    closeFile(ifh);
    closeFile(ofh);
    closePIOL(piol);
    return 0;
}
```