ENABLING LARGE DATA PROCESSING WITH THE 3D ZO CRS STACK SOFTWARE

E. Borin, A. Novo, I. Rodrigues, J. Sacramento, J. H. Faccipieri, and M. Tygel

email: *edson@ic.unicamp.br*

keywords: *3D ZO CRS Stack software, large data processing, HPC*

ABSTRACT

The 3D ZO CRS Stack software, provided by the WIT consortium, leverages the CRS method to stack seismic data. In this work, we provide an overview of the 3D ZO CRS Stack software, including its parallel execution model, and an analysis of the performance of the software when executing large data sets (1TB). We show that the current implementation of a key procedure, `makeGeometry`*, executed during the CRS processing, uses an inefficient algorithm that hinders the processing of large data sets. We present a new algorithm for the* `makeGeometry` *procedure that reduces its execution time from 20 000 seconds to just 30 seconds in our infrastructure.*

INTRODUCTION

Stacking may be considered one of the most important steps in seismic processing providing an approximate zero-offset (ZO) section with an enhanced signal-to-noise ratio. Conventionally, these sections are obtained with the Common Midpoint (CMP) method, which depends only on Normal moveout (NMO) velocity. The Common Reflection Surface (CRS) method represents a natural generalization of the CMP method, where an ensemble of CMPs is considered for stacking. CRS employs a moveout equation in which source-receiver pairs are allowed to be arbitrarily located around the CMP position.

The CRS moveout is a second-order approximation of the squared travel time in the vicinity of the normal ray. For the 2D situation, it is given by

$$t^2(m,h) = [t_0 + A(m-m_0)]^2 + B(m-m_0)^2 + Ch^2\,, \tag{1}$$

where m_0 is the midpoint position, m is the midpoint of a source-receiver pair in the vicinity of m_0, h is the half-offset, and A, B and C are stacking (scalar) parameters that define the stacking surface to be estimated. In the present 2D case, three stacking parameters are to be estimated. In 3D, eight parameters are required to be estimated.

The search for the stacking parameters for each midpoint and time samples can be performed in parallel, since there are no dependencies between the searches. Besides allowing the acceleration of the search through parallel computing, this property enables the execution time to scale linearly with respect to the number of midpoints. In fact, we conducted several experiments increasing the number of midpoints to be processed and showed that, up to 22GB of pre-stacked input data, the execution time increased linearly with respect to the input file size. However, we noticed a super linear execution time increase when experimenting with a 44GB data set, which took four times longer to load than the 22GB data set.

Our preliminary analysis shows that the current implementation of the `makeGeometry` procedure has quadratic execution time growth, and we estimate that the software would take about 115 days just to execute the `makeGeometry` procedure when processing a 1TB input data file in our infrastructure. The main contributions of this work are:

- We provide an overview of the 3D ZO CRS Stack software, including its source code organization and its parallel execution model.

- We analyze the performance of the software when executing large data sets and show that the current implementation of the `makeGeometry` procedure has quadratic execution time growth, which hinders the processing of large (1TB) data sets.

- We designed and implemented new algorithms to improve the time complexity of the `makeGeometry` procedure, which reduced its execution time from 20 000 to 30 seconds when processing a 44GB input dataset.

This report is organized as follows. First we provide an overview of the 3D ZO CRS Stack software, including its organization and the parallel execution model. Then, we discuss the performance issues and solutions for the `makeGeometry` procedure.

3D ZO CRS Stack SOFTWARE

The 3D ZO CRS Stack software leverages the CRS method to stack seismic data. This software, provided by the WIT consortium, is implemented in C++, using the object-oriented programming paradigm, and the MPI library (Gropp et al., 1994), which allows it to be executed with multiple processes on a single host or on a distributed memory system. The next sections provide an overview of the source code organization and the parallel execution model.

Source code organization

The 3D ZO CRS Stack software has 68 126 lines of code and it is organized in four modules, as shown in Figure 1: mpi3dcrs.v2, libfparse, libsio e libcrs. Follow a brief description of each module.

- mpi3dcrs.v2: The mpi3dcrs.v2 module has 52 888 lines of code, approximately 78% of the total, containing the main structure of the software. It is responsible for initializing the MPI library; parsing command line parameters; coordinating execution of master and slaves processes and writing results back to the disk.

- libfparse: The libfparse module has 938 lines of code, approximately 1% of total. It contains generic classes for describing and parsing command line parameters.

- libsio: The libsio module has 3248 code lines, approximately 4% of total. It contains classes to read and write seismic data in various formats. The supported standard formats are SEG-Y REV-0 (Barry et al., 1975) and SU (Seismic Unix Data Format) (Stockwell, 1999).

- libcrs: The libcrs module is the core of the application and has 11052 lines of code, approximately 17% of total. It contains classes that model an abstract CRS method that is agnostic with respect to the search method, coherence method and interpolation method. It also contains concrete implementations of the CRS: a sequential search that determines the velocity first, emergence angle second and curvature third; a CRS Stack; and Simulated Annealing search/stack.

In order to execute the software on a distributed memory system, with multiple computers, the implementation requires the existence of a distributed file system, e.g. Network File System (NFS), to allow the input and output data files to be shared among the MPI process. The input data file, read by all the processes, may be replicated in each cluster node in order to avoid the need for a distributed file system. However, the partial output data files, produced by all the slave processes and consumed by the master process, need to be shared between these processes.

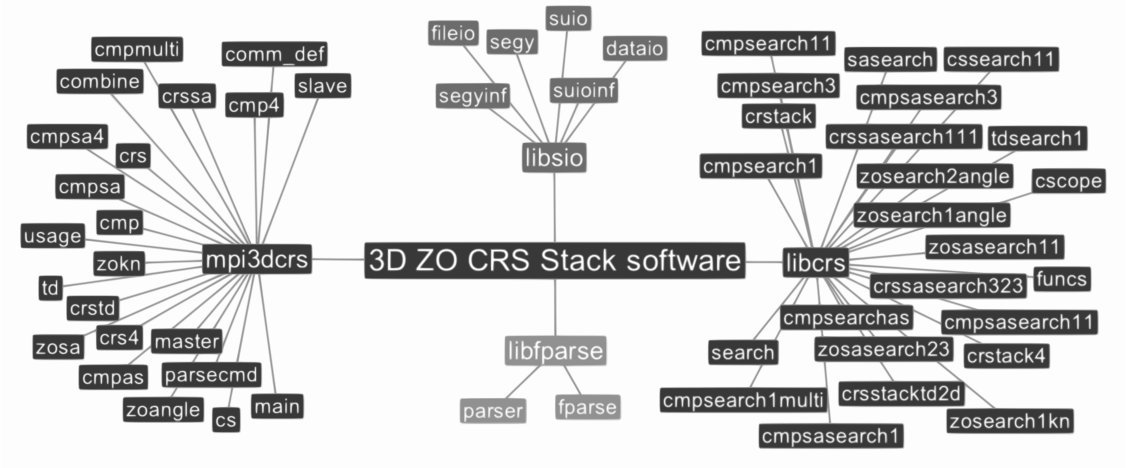

Figure 1: 3D ZO CRS Stack software files organization.

Parallel execution model

The 3D ZO CRS Stack software uses the *Bag of Tasks* parallel execution model to search for the stacking parameters. In this model, a master worker distributes tasks to slave workers. Whenever a slave concludes the task, it informs the master, which in turn sends another task to the slave. This process continues until all the tasks are finished.

A task consists in searching the best stacking parameters for a given midpoint (CMP gather). Each midpoint is associated with a set of seismic data traces that are read from the input files. The results of the search are written to a partial results file (one for each slave) and, after all the tasks are finished, the master worker concatenates all partial results files in a single result file. The overall computation is performed in three stages: **setup**, **search**, and **combine**.

During the **setup** stage, the master and slave workers read the input data file in order to construct a data structure that associates seismic data traces with midpoint identifiers. The master uses this information to define the set of tasks to be computed (one for each midpoint), while the slaves use it to identify the seismic data traces associated with a given midpoint. All slaves and master workers perform this stage in parallel. At the end of the stage, the slave workers send messages to the master informing that they are ready to work.

The **search** stage starts whenever the master worker finishes the setup stage. In this stage, the master distributes tasks to the slave workers. Whenever a slave receives a task, it reads the seismic data traces associated with the task midpoint and performs the search. At the end of the search, the slave writes the result to a partial results file and informs the master that it is ready to work on another task. The search stage finishes whenever all tasks are concluded.

At the **combine** stage, the master worker combines all the partial results files into a single results file. The slaves do not perform any work during this stage. Figure 2 illustrates the parallel execution of the master and slave workers during the three stages.

The master and slave workers are implemented as processes and the communication between them is performed through the MPI library and a shared file system. The MPI library is used by the master to distribute tasks to the slave workers and by the slaves to inform the master that they are ready for a new task. The shared file system is used to store partial results and to store the input files, which are read by all workers.

The task is identified by the midpoint identifier, which in turn is an unsigned number defined during the setup stage. This means that very little information is transmitted between the master and the slave processes when communicating via the MPI library. But this also means that every slave process *must* have access to the input file in order to retrieve the seismic traces associated with the midpoint.

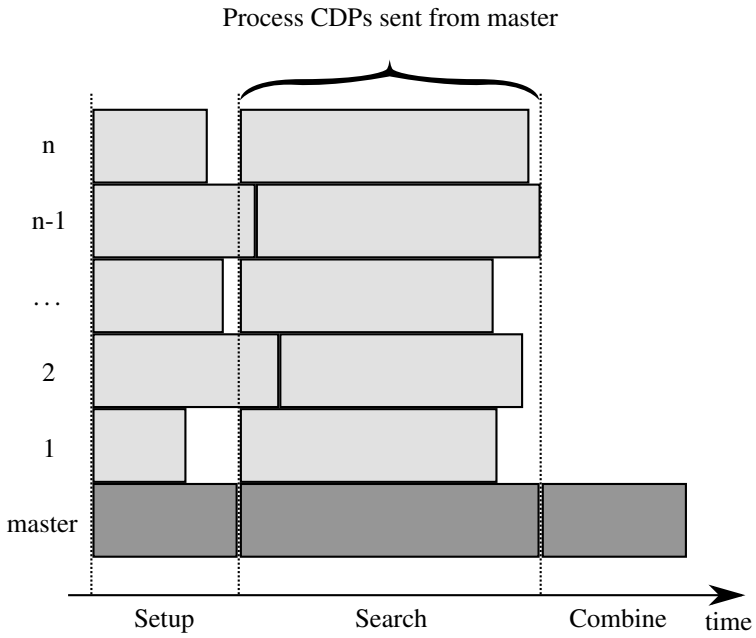

Figure 2: Parallel execution of the master and slave workers during the three stages. Gray areas indicate active time and white areas indicate idle time.

SCALABILITY ANALYSIS

In order to evaluate the scalability of the 3D ZO CRS Stack software, we carried out several experiments varying the size of the input seismic data. Our experiments were conducted in a cluster with 15 nodes, each one featuring two AMD Opteron 2376 processor with 4 cores and 16 GB of RAM. The network interconnection is a gigabit star topology model with a central Extreme Networks Summit X44-48p switch.

Our preliminary results indicated that, for input files larger than 22GB, the execution time grows in a faster rate than the input data size. In fact, the software took 20 000 seconds just to read the 44GB data file, as opposed to 5 000 seconds when reading the 22GB data file.

By profiling the application, we were able to identify one of the main sources of overhead, the `makeGeometry` procedure. Figure 3 indicates that the execution time of the `makeGeometry` procedure has quadratic growth with the size of the input file.

We had fit a parabola with a very good approximation on the performance data and estimated that the procedure would take more than 115 days just to process a 1TB input data file. The next sections present an analysis of the `makeGeometry` procedure and our solution to the problem.

The `makeGeometry` procedure

The `makeGeometry` procedure organizes the input seismic traces into gathers. Each gather contains a set of traces with similar properties. First, it gathers traces with common shots. Then, the procedure gathers traces with common midpoints, and, finally, it gathers the traces with similar zero offsets. Our profiling analysis indicated that the main cause of overhead is the gathering process.

The algorithm for the gathering process works as follows: for each trace, the algorithm scans a list of gathers searching for one that contains the same properties of the trace (e.g., a common shot). If a gather is found, the trace is inserted into the gather, otherwise, a new gather is created, featuring the properties of the trace, and the trace is inserted into this gather. The pseudocode for the common-shot gathering process is provided in Algorithm 1. The pseudocodes for the other two gathering processes are similar.

Let n be the number of seismic traces at the input data file and m be the number of distinct common shot, the time complexity of the algorithm is given by $O(nm)$.

The number of distinct common shots (m) is typically proportional to n, since the acquisition geometry

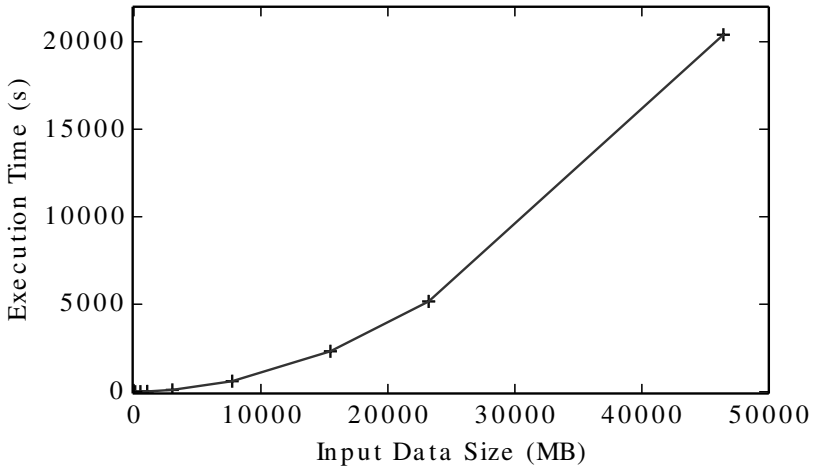

Figure 3: Execution time of `makeGeometry` procedure for different input data sizes.

Algorithm 1 Common Shot Gathering

```
for t in traces; do
    gather_found = false;
    /* Search for a Common Shot Gather to insert trace t */
    for csg in CSGs; do
        if csg.sx == t.sx and csg.sy == t.sy then
            csg.insert(t)
            gather_found = true;
        end
    end
    /* If not found, create a new Common Shot Gather and insert trace t */
    if gather_found == false then
        csg = createNewCommonShotGather(t.sx,t.sy)
        CSGs.append(csg)
        csg.insert(t)
    end
end
```

(number of receivers) can be considered constant for a given data set. Hence, the time complexity of the algorithm is $O(n^2)$. In order to improve the execution time, we modified the algorithm to sort the traces before organizing them. By doing so, we are able to perform the search for the correct gather in constant time, which reduces the loop time complexity to $O(n)$. The overall time complexity is reduced from $O(n^2)$ to $O(n\ log\ n)$, since the sorting procedure is $O(n\ log\ n)$. Algorithm 2 shows the pseudocode for the optimized algorithm.

The execution times for the optimized `makeGeometry` procedure are shown at Figure 4. Notice that the execution time was reduced from more than 20 000 seconds to just 30 seconds when processing the 44GB seismic data. In order to estimate the time the `makeGeometry` procedure would take to process a

Algorithm 2 Optimized Common Shot Gathering

```
/* Sort traces by common shot source */
sort(traces)
for t in traces; do
    /* Search for a Common Shot Gather to insert trace t */
    csg = CSGs.last()
    if csg.sx == t.sx and csg.sy == t.sy then
        csg.insert(t)
    else
        csg = createNewCommonShotGather(t.sx,t.sy)
        CSGs.append(csg)
        csg.insert(t)
    end
end
```

1TB input file we fit a $n\ log\ n$ curve to the performance results. The result indicates that the time to process a 1TB input data file would be reduced from 115 days to just 15 minutes.

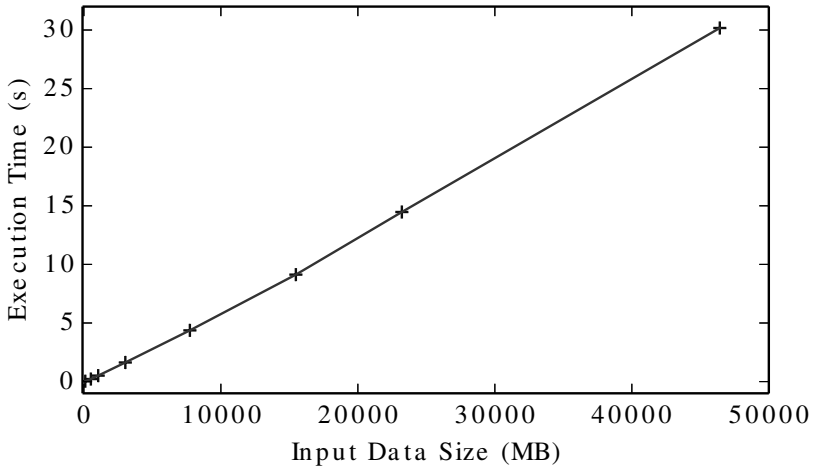

Figure 4: Execution times for the optimized `makeGeometry` procedure.

CONCLUSIONS

Stacking is an important step in seismic processing that provides an approximate zero-offset section with an enhanced signal-to-noise ratio. This process can be performed by the 3D ZO CRS Stack software, a tool provided by the WIT consortium. In this work, we provided an overview of the 3D ZO CRS Stack software, including its parallel execution model and its source code organization, and we analyzed the performance of the software when executing large data sets (1TB).

Our results indicate that the current implementation of the `makeGeometry` procedure, which is executed during the CRS processing, has a quadratic execution time growth that hinders the processing of

large data sets. In fact, our estimates indicate that the software would take about 115 days just to execute the `makeGeometry` procedure when processing a 1TB input data file in our infrastructure.

Finally, we proposed a new $O(n\ log\ n)$ algorithm for the `makeGeometry` procedure and showed that it reduces the procedure execution time from 20 000 seconds to just 30 seconds when processing a 44GB data set in our infrastructure. Moreover, our estimates indicate that the new procedure would take only 15 minutes to process a 1TB input data file.

All modifications and codes for this experiment were uploaded in time of subscription of this report and should be available in the WIT repository.

ACKNOWLEDGMENTS

This work was kindly supported by the sponsors of the *Wave Inversion Technology (WIT) Consortium*.

REFERENCES

Barry, K. M., Cavers, D. A., and Kneale, C. W. (1975). Recommended standards for digital tape formats. *Geophysics*, 40(2):344–352.

Gropp, W., Lusk, E., and Skjellum, A. (1994). *Using MPI: portable parallel programming with the message-passing interface*. MIT Press Scientific And Engineering Computation Series.

Stockwell, J. W. (1999). The cwp/su: Seismic unix package. *Computers & Geosciences*, 25(4):415–419.