On improving performance of migration algorithms using MPI and MPI-IO

Dheeraj Bhardwaj, Suhas Phadke and Sudhakar Yerneni*
Centre for Development of Advanced Computing, Pune University Campus, GaneshKhind, Pune 411007, India

SUMMARY

High level of I/O performance is necessary in making use of parallel machines for many scientific applications. In this paper we discuss the I/O requirements of one such application: 3D seismic migration. I/O performance is the bottleneck rather than the computational or communication performance in 3D seismic imaging. For large 3D data volume it is not possible to read and keep all the required data and information in computer memory. Therefore the data are partially read and intermediate results are written out at various stages during the execution of the code. In this article we have discussed an approach to handle the massive I/O requirements of seismic migration using MPI-IO. We have also optimised some data communication using MPI calls. The performance of parallel seismic migration code on PARAM 10000, which is a cluster of SUN workstations, is discussed for two data sets.

INTRODUCTION

The processor and communication speeds of parallel computers have steadily increased, but the technology for improving the I/O sub systems has not progressed at the same pace. I/O subsystems for distributed memory parallel computers are often not designed to handle efficiently an application with massive I/O requirements, such as seismic data processing. Most of the parallel computers work well with computationally intensive applications, but they are inefficient when it comes to satisfying the needs of applications that are also I/O intensive.

Recently, cluster of workstations or network of workstations has gained popularity as they provide a very cost-effective parallel-computing environment. Most of these clusters use Network File System (NFS) and use MPI (Message Passing Interface) for parallel programming. One limitation of NFS is that the I/O nodes are driven by standard UNIX read and write calls, which are blocking requests. This is not a problem for applications with small volume of I/O, but as the volume increases, it is necessary to be able to overlap computations with the I/O to maintain efficient operation (Olfield et al., 1998, Poole, 1994).

In this paper we first give a description of MPI-IO and the migration technique. Next we show a comparison of the conventional UNIX I/O and MPI-IO. We also talk about some code optimization using MPI. Then we discuss the implementation of MPI-IO in migration code and show its performance for two data sets.

MPI-IO

Parallel programming has long been hampered by the lack of a standard, portable Application Programming Interface (API) for parallel I/O. Unix API is not appropriate for parallel I/O as it lacks some of the common features observed in parallel programs, such as noncontiguous accesses and collective I/O. This results in poor performance. MPI-IO is a comprehensive API, which includes features for I/O parallelism, portability and high performance.

ROMIO (A high-performance, Portable MPI-IO) is a very well suited software solution to a cluster environment where each machine has its own disk and processor. It provides a solution such that instead of a single processor reading the entire file and then scattering it to other processors, each processor does a local read or write. Most of the functions in MPI-2 I/O standard have been implemented in ROMIO (Thakur et al. 1998, 1999).

A COMPARISON OF UNIX I/O AND MPI-IO

An increasing number of applications, such as migration, need to access large files (greater than or equal to 2GB). In order to improve the access performance for large files, the file system interface and internal data structures must use 64 bit integers to represent file offsets. ROMIO defines MPI_Offset as an 8-byte integer and uses the corresponding file system functions for large files.

Most file-systems, support only the regular Unix *open* and do not have collective open functions. MPI-IO provides collective IO functions which must be called by the processors that together open the file. Collective I/O has been shown to be a very important optimization in parallel I/O and can improve performance significantly (Thakur and Choudhary, 1996). This is because the most file systems do not support shared file pointers.

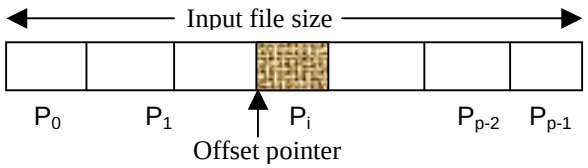

Figure 1: File view and its distribution across the *p* processors.

For the purpose of implementing an application on parallel machine, we need to partition the input file across the processors. Figure 1, depicts the partitioning of the entire input file across the *p* processors. We studied the distribution of the input file using UNIX I/O and MPI-IO. In case of UNIX I/O, process with rank zero reads the whole input file, partitions it and distributes it to other processors. Distribution is done using *MPI_Send* and processors with rank greater than zero receive their corresponding blocks by *MPI_Recv*. In MPI-IO, all the processors open the file using *MPI_File_open* and read their required data blocks by moving offset pointer

to the beginning of their corresponding data block in the input file. This is carried out by using *MPI_File_read_at*. In order to compare both these approaches, we fixed the size of the block needed by each processor. Figure 2, shows the time required to read a data size of 20.48 MB on each processor versus number of processors. It is evident from the graphs that the difference in the I/O time between both the approaches increases as we increase the number of processors.

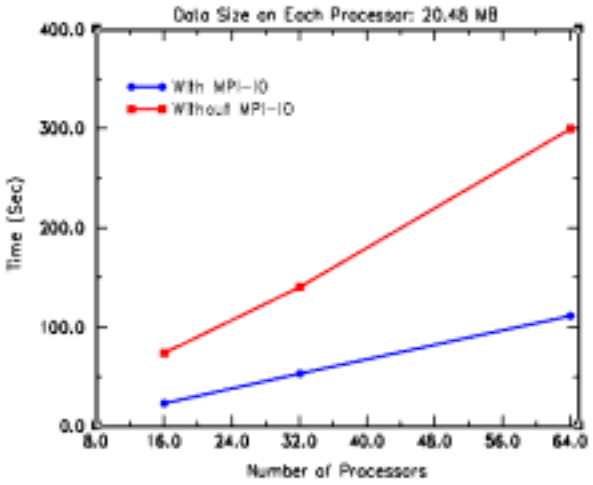

Figure 2: Graphs showing the time required to read a data size of 20.48 MB on each processor against the number of processors.

The time required to read and distribute the input data (without MPI-IO) and the time required to read the data blocks by the processors concurrently (with MPI-IO) using shared file pointers, are functions of data size. Figure 3, shows the comparison of these two approaches for a fixed number of processors. It is clear from this figure that for large data blocks MPI-IO approach gives better performance.

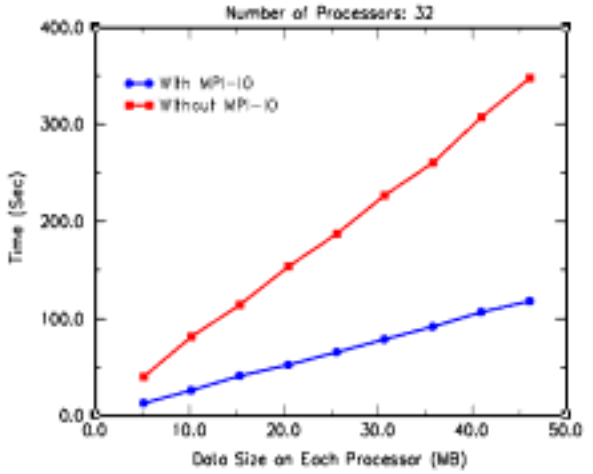

Figure 3: Graphs showing the time required to read the data (with and without MPI-IO) against the data size on each processor for 32 processors.

Next, we studied the I/O for a data block that is required by all the processors. Figure 4 illustrates the comparison between the read and broadcast and concurrent read. From this figure we infer that read and broadcast is faster. Therefore, we decided to use this for reading velocity slices for migration.

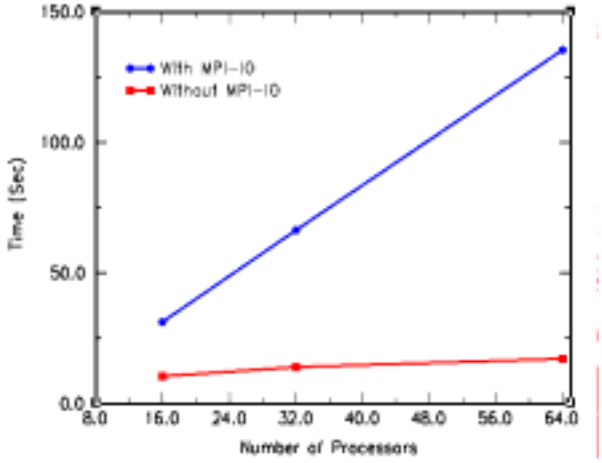

Figure 4: Graphs showing the time required to read and send a data block to a large number of processors. The blue line indicates the time when each processor opens the file and read concurrently. The red line indicates the time when the processor with rank 0 (master) reads the file and broadcasts to all the processors.

PARALLEL 3D POST-STACK DEPTH MIGRATION

For imaging complex geological structures, where the velocities vary in all directions, depth migration is a necessity. Depth migration for laterally varying velocity structures is a compute intensive wave equation based method. The depth migration method comprises of two steps, extrapolation and imaging. In this paper we have used a finite difference formulation of the 65 degree parabolic approximation in ω-x domain (Claerbout, 1985), for extrapolating the wavefield. The imaging is a summation of all the frequencies at each depth for t=0.

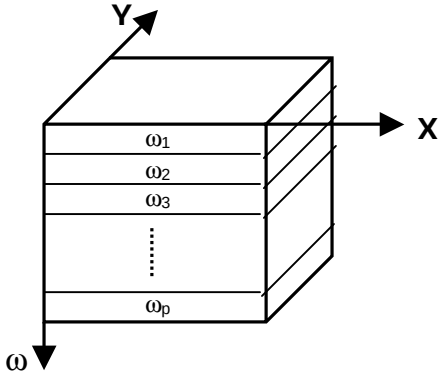

Figure 5: Distribution of frequency sequential data on *p* processors.

The depth migration algorithm in ω-x domain is inherently parallel in terms of frequencies. The frequencies are evenly distributed to the available number of processors as shown in Figure 5. The ultimate goal is to have as many processors as frequencies. The parabolic approximation of the wave equation in frequency-space domain is then used for downward propagation of each monochromatic wave component (Phadke and Bhardwaj, 1997). Therefore, each frequency harmonic can be extrapolated in depth independently and there is no need of intertask communication.

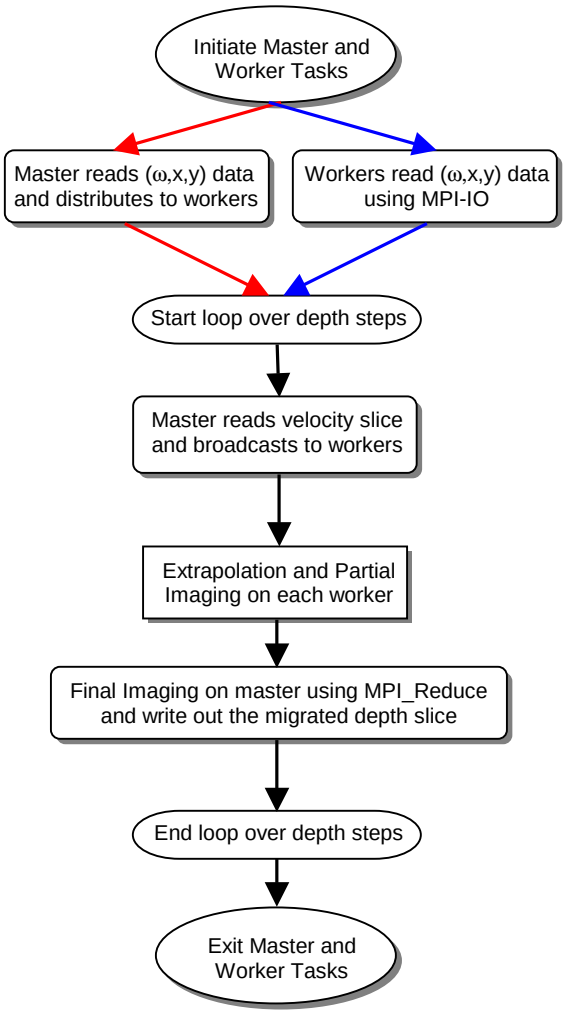

Figure 6: The flow diagram for migration algorithm with and without MPI-IO. The red arrows indicate the flow direction with MPI-IO and the blue arrows indicates the flow direction without MPI-IO. The black arrows are common to both.

Before we migrate a given data set we apply some pre-processing. (1) The stacked data is first Fourier transformed with respect to time and stored in frequency sequential format. (2) The input velocity model is stored in sequential depth slices. Only the required number of frequencies are stored after Fourier transformation. The frequency bandwidth to be used for migration is determined from spectral analysis of the input traces. This forms the input data to the depth migration code.

Parallel Implementation without parallel I/O

The parallel implementation is analogous to Master-Worker system. After reading all the required parameters, the Master (processor with rank 0) determines the number of frequencies and frequency bandwidth to be assigned to each Worker. Then it reads and sends the frequency data to the designated Worker in a sequential manner. Then the migration algorithm runs through the depth steps. The required velocity data for each depth step is sent to the Workers in the depth loop. Also the migrated data from all the Workers for that depth is collected by master, imaged and stored on the disk (Phadke et al., 1998). A flow chart of this algorithm is shown in figure 6 (red and black arrows).

Parallel Implementation with parallel I/O

In MPI-IO implementation all the processors read their respective frequency data in parallel. Then the migration algorithm runs through the depth steps. Master reads velocity depth slice and broadcasts to all the workers. The processor with rank zero, collects the migrated data from all the processors for that depth, images it and then stores it on the disk. A flow chart of this algorithm is shown in figure 6 (blue and black arrows).

PERFORMANCE ANALYSIS

Even though the developed codes for ω-x depth migration (with and without MPI-IO) are portable across various platforms, most of the development was done on PARAM 10000. The PARAM 10000 system has 40 SUN E450 compute nodes, each with 4 processors @300MHz. Out of 40 nodes 4 nodes are network file servers with 1GB RAM and 512K cache. High-speed network such as fast Ethernet with peak bandwidth 100MB/s connects the nodes.

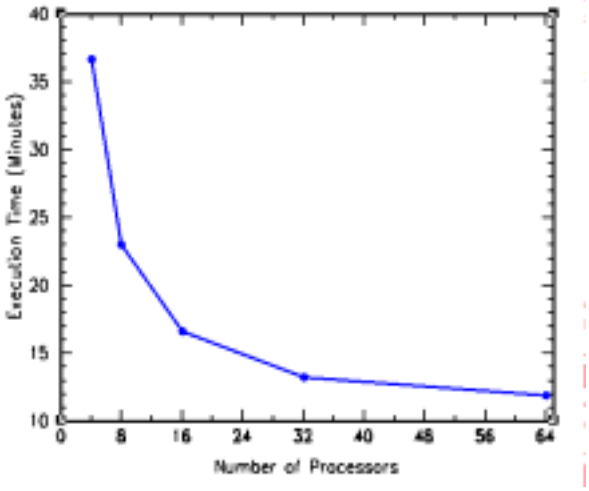

Figure 7: Number of processors versus execution time chart for SEG/EAGE Overthrust model.

We first tested the migration algorithm for the data set of SEG/EAGE (1997) Overthrust model. The original data had 101×25 CDP traces with inline spacing of 100m and crossline spacing of 100m. We interpolated this data to 401×97 CDP traces to make both inline and crossline spacing 25m for avoiding spatial aliasing. The input Fourier Transformed data size was of the order of 46MB. This data set was migrated with a depth step of 25m for 161 depth steps. Figure 7 illustrates the execution time as a function of number of processors. Since the problem size is small the speedup is not linear.

The second data set used for testing comprised of 950×665 CDP's. The inline spacing, crossline spacing, and depth step size were 25m. The data was migrated for 480 depth steps. Table 1 shows all the other parameters and the time required to migrate this data set with 64 processors.

Size of FFT data	1.3 GB
Size of Velocity model	1.2 GB
Total number of frequencies migrated	256
Number of Processors	64
Total Execution time with MPI-IO	7 hrs 44 mins

Table 1: Problem size for the second data set and the execution time on 64 processors.

DISCUSSION AND CONCLUSIONS

In this paper we first studied the difference between Unix I/O and MPI-IO and then looked at some of the data distribution strategies from the point of view of migration codes. If every processor has to read different blocks of memory from a file, then every processor can open and read the required data set using a collective call of MPI-IO. This methodology is good for reading Fourier Transformed seismic data. But if every processor needs the same data then one processor should read it and broadcast it to other processors. This method is good for reading velocity depth slices. Final imaging, where all the partially imaged data have to be summed, should be done using MPI_Reduce operation. The migration of two test data sets clearly shows the performance of our algorithm. This programming paradigm will be very helpful for prestack migration.

In the seismic industry, where the amount of data that needs to be processed is often measured by the number of tapes, which amount to hundreds of gigabytes or even terabytes, the improvement in execution time by making efficient use of the I/O subsystem, and overlapping I/O with communications and computations, becomes increasingly apparent. A 10% to 20% improvement in runtime, especially for prestack migration, would amount to savings of millions of dollars of processing time. The above-mentioned results are a step in that direction

ACKNOWLEDGEMENTS

Authors wish to express their gratitude to the Department of Science and Technology (DST), Government of India, for funding the seismic data processing project under DCS. Authors also wish to thank the Centre for Development of Advanced Computing (C-DAC), Pune for providing the computational facilities on PARAM 10000 and permission to publish this work.

REFERENCES

Claerbout, J. F., 1985, Imaging the Earth's interior, Blackwell Scientific Publications

Oldfield, R. A., Womble, D. E., and Ober, C. C., 1998, Efficient Parallel I/O in Seismic Imaging, The Int. J. of High Performance Computing Applications, Vol. 12, No. 3, P. 333-344.

Phadke, S. & Bhardwaj, D., 1997, Depth extrapolation of seismsic wavefields using cubic spline approximation, Expanded Abstracts, SEG 67th Annual International Meeting, P. 1658-1661.

Phadke, S., Bhardwaj, D. & Yerneni, S., 1998, Wave equation based migration and modelling algorithms on parallel computers, In Proc. of second conference of SPG (Society of Petroleum Geophysicists), P. 55 - 59.

Poole, J., 1994, Preliminary survey of I/O intensive applications, Technical Report CCSF-38, Scalable I/O initiative, Caltech Concurrent supercomputing facilities, California Institute of Technology, Pasadena.

SEG/EAGE 3-D Modeling series No. 1, 1997, 3-D Salt and Over thrust models. SEG publications.

Thakur, R., and Choudhary, A., 1996, An extended two-phase method for accessing sections of out-of-core arrays, Scientific Programming, 5, P.301-317.

Thakur, R., Lusk, E., and Gropp, W., 1998, User Guide for ROMIO: A High Performance, Portable MPI-IO Implementation, TM No. 234, ANL, IL 60439(USA).

Thakur, R., Gropp, W., Lusk, E., 1999, On implementing MPI-IO portably and with high performance, Proc. Of the Sixth Workshop on I/O in Parallel and Distributed Systems. P. 23-32.