Cluster programming for reverse time migration

Sang Yong Suh, Alex Yeh, Bin Wang, Jun Cai, Kwangjin Yoon, and Zhiming Li, TGS-Nopec Geophysical Company

Reverse time migration (RTM) is well suited for imaging steep dips in areas with high velocity contrast. In order to image steep dips at the correct positions, anisotropy has to be taken into account. In most cases, we can assume the symmetry axis is normal to the bedding, and model is tilted transverse isotropy or TTI. Figure 1 compares TTI Kirchhoff migration and TTI RTM images. We can see that the turning wave helps RTM image the steep salt flank and the steep dip truncations against the base of the salt.

In addition to its use as a final imaging tool, RTM is being used as a salt model building tool as well; this involves many RTM iterations. Figure 2 shows different possible salt geometries and their corresponding RTM images. In order to reduce the computation time, the redatuming concept was created. By using the redatuming technique, we can save the wavefield at a particular depth above the salt geometry that we would like to modify, and only remigrate the deeper portion with different salt models. We have also developed a delay image time (DIT) technique in our RTM algorithm. Conventional RTM calculates only one image at zero delay time between source wavefield and receiver wavefield. The DIT RTM will compute a sweep of images by applying the imaging conditions at different delay times. The DIT images can be used to obtain an optimal composite image and to update the velocity model, after analyzing and picking on a delay time field. Figure 3 shows the original RTM image, and the new RTM image by using the DIT updated velocity.

Cluster computing is widely used to satisfy the requirements of computationally intensive applications. In a cluster environment, message passing interface (MPI) is one of the most well known programming tools, providing a set of well defined library functions for multinode parallel computing. However, MPI has one critical drawback for big production jobs. If the job runs for days or weeks using hundreds of CPUs, there is a great chance of a program crash due to a hardware malfunction. If MPI is used, one node crash results in the crash of all other nodes. Because of this, people often avoid using MPI in their RTM programs. This article discusses an alternative solution using remote shell-based (rsh) multinode programming techniques for RTM.

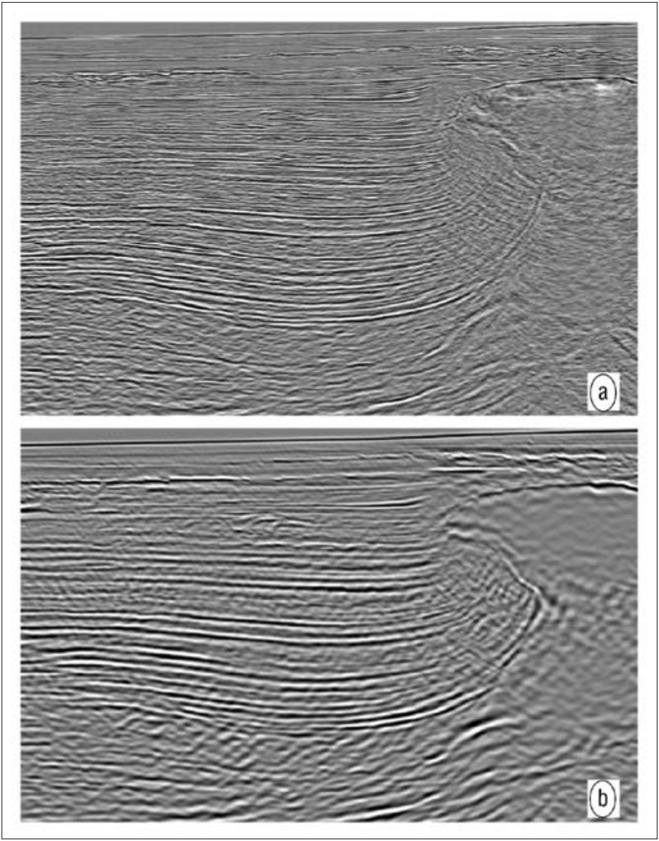

Figure 1. *(a) TTI Kirchhoff migration image. (b) TTI RTM image.*

Running many remote programs

To run a program on a remote host, we use the rsh command that uses two ports per program. The first port is used by the remote program for stdin and stdout while the second port is used by the remote program for stderr and as the signal delivery channel. The ports are continuously used even if the local processes are terminated, in which case the local rsh client becomes a zombie and stays that way until the remote program exits.

An experiment on a modern Linux (kernel 2.6.9) shows that the port number used by the rsh client starts from 1023 and decreases to 824. Therefore, the total number of available ports is 200. If the connection between the local rsh client and the remote program does not die quickly, the number of remote programs that can be run is limited to 100. In other words, the maximum number of computing nodes that can be started by a master node would be limited to 100.

The solution to this problem is making the remote program like a daemon running background. This can be achieved from the remote program side by closing all open files such as stdin, stdout, and stderr, forking a child process, and exiting. The child process uses the setsid() system call to create a session and releases the existing signal delivery channel and takes care of the actual computation. Because the remote program has closed all files and the signal delivery channel, and has terminated itself, the connection between the local rsh client and the remote program is closed immediately. The ports used in the rsh connection will be available to subsequent connections after some TCP close wait times.

Dynamic shot allocation

RTM is well suited for cluster computation. Suppose we have a cluster of N nodes, each having equal performance. If we were to process M shots, each node would be assigned M/N shots.

There are two types of allocation, static and dynamic.

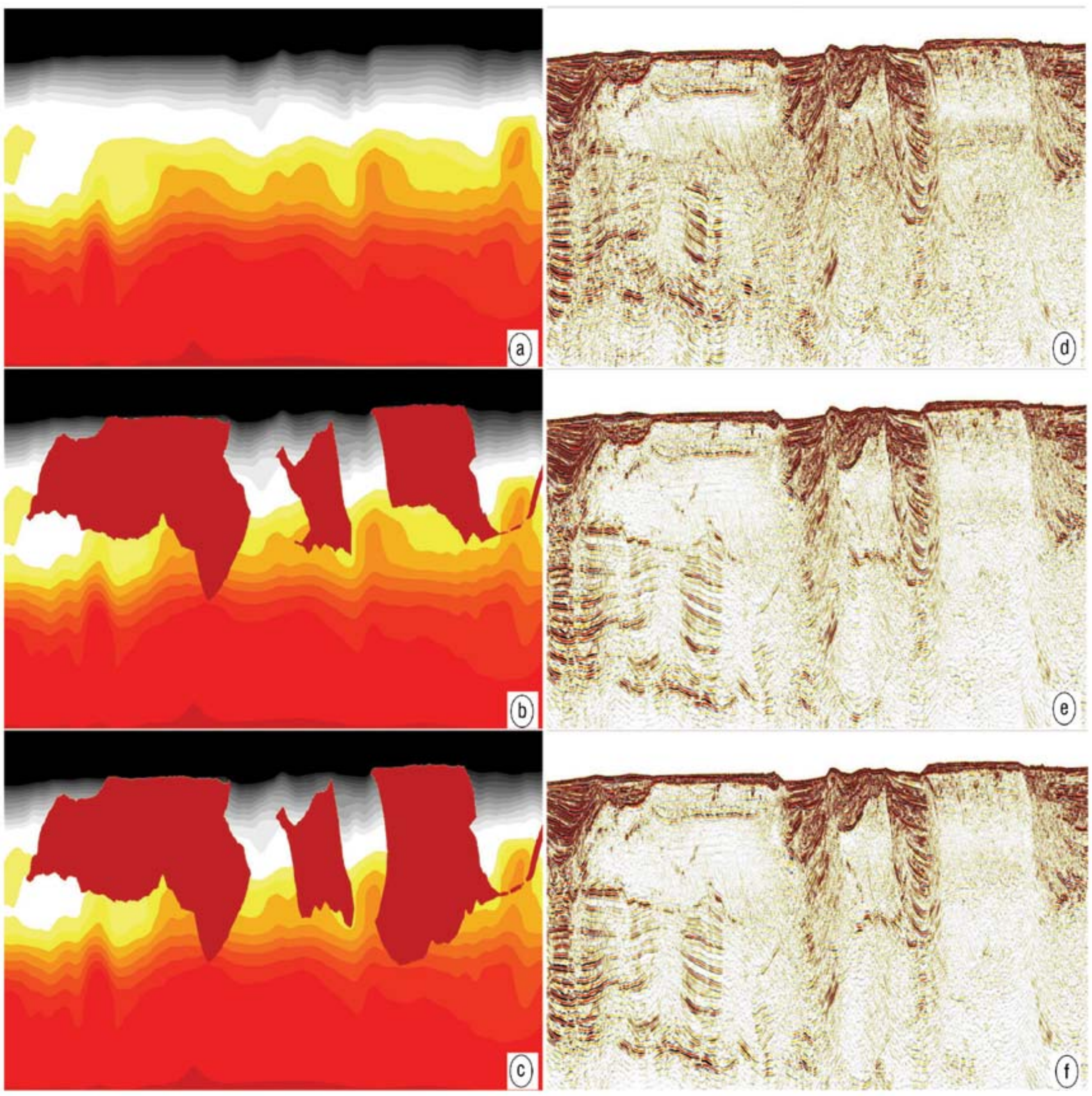

Figure 2. *(a), (b), and (c): Different possible salt geometries. (d), (e), and (f): Corresponding RTM images.*

In the static allocation, a node is assigned with a predefined number of shots. For example, the shots assigned on the first node would be 1, N+1, 2N+1 .., and the shots assigned on the second node would be 2, N+2, 2N+2 ..., and so on. As an aside, if the cluster is composed of different CPU models, the above scheme would not be acceptable. We would expect permanent failure of nodes during the computation. On the other hand, we might have a generous friend who would donate some idle nodes which could be added to the computation.

Dynamic allocation assigns a node one shot at a time. If a node finishes the shot, it contacts the master server to do the following four tasks: (1) read from the shot list; (2) mark the current shot as completed; (3) find the next shot to process and flag it as assigned; and (4) write to the shot list.

The tasks must occur sequentially, i.e., if a process is executing the tasks, no other processes are allowed to do the job. Otherwise a disaster occurs known as racing condition.

To protect against racing, a server could be run on the master node and manage the shot list exclusively. If a remote node connects, the server assigns, one by one, a shot to the connecting node. This is a network program involving a server and multiple clients. However, network programming is error-prone and it is very difficult to write a reliable program, especially one operating in a heavily congested network environment, which is often the case with large clusters.

One simple method is to split the seismic data into many small files, each containing just one shot gather, and put them in a common directory residing on an NFS-mounted partition. A computing node tries to move a file from the common directory to its own unique directory. If the operation fails (i.e., if the file is not found from the destination directory), we know that another process has taken the file. Note that the destination directory must reside on the same partition as the source directory because it exploits the atomic behavior of the Unix rename system call.

A cleaner method would be to use a shot database of multiple records each having six fields: shot number; seismic data

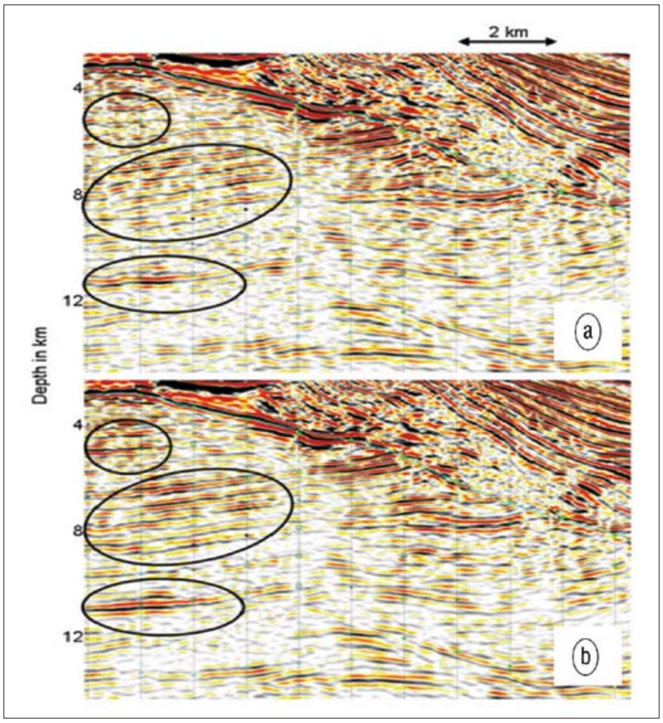

Figure 3. *RTM images before (a) and after (b) DIT velocity update.*

file number; byte offset of the first trace of the shot gather within the data file; migration start time; migration end time; and the node number of the working node.

The first three fields enable direct access of the first trace of a shot. The remaining three fields record the processing history.

During the setup, an initialization program examines the seismic data files and creates the shot database. For each available node, the program allocates a shot, runs the migration program on the node with that shot, records the start time, and updates the database.

Upon completion of the shot, a computing node contacts the master node and starts a program that reads the shot database, marks the current shot as completed by entering the migration end time, fetches the next shot to be processed, and updates the database. To prevent racing condition, the program accesses the shot database exclusively using lockf(), a C library function, which is an interface to the fcntl() system call.

There are other Unix-locking techniques that exploit the atomic nature of some system calls on file operation. These techniques create and use an ancillary file, generally known as a dotfile or lockfile, as an indicator that the process has locked the resource. If the ancillary file exists, the database is locked by some other processes; otherwise the calling process must create the ancillary file to lock the database.

One frequently used technique exploits the fact that the link() system call fails if the name of the new link to the file already exists. The actual implementation first creates a unique temporary file whose name is generally derived from the process ID. Then an attempt is made to create a hard link having the new name of common ancillary lockfile against the already created temporary file. The link system call fails if the name of the new link already exists, and we know that another process has locked the database already.

Traditionally file locking over NFS especially using flock(), a BSD style system call, was unreliable. Therefore, the computing node would contact the master node to create a remote process and would perform the lock operation remotely either using lockf() or an ancillary lockfile. Running a remote shell process with some arguments can be accomplished using the popen() C library call. Also, receiving the result can be accomplished by the fread() function using the file pointer returned by the popen() function.

Recovering from network congestion

A problem occurs if a large number of computing nodes try to contact the master node in a relatively short time window. If this happens, the cluster network would be congested. Also, there will be no more network ports available for a new remote shell connection. The situation is similar to an Internet server which is under a distributed denial of service (DDoS) attack.

Unix serves remote shells with the help of an Internet super server such as inetd or xinetd, and many Linux distributions including Red Hat use xinetd as the front-end server to rshd, the remote shell server program. If the rate of incoming connections to a specific port exceeds a certain rate, the program xinetd disables service for a certain amount of time. This is controlled by the cps attribute of xinetd.conf(5), the Extended Internet Service Daemon configuration file. The cps attribute takes two arguments. The first argument is the number of maximum allowable incoming connections per second, and the second argument is the number of seconds to wait before re-enabling the service. The default for this setting is 50 incoming connections and a 10-s wait time, although the actual setting varies from Linux vendor to vendor.

Therefore, the client program must anticipate the network congestion, or the attempted connection to the master server could result in failure. If failure occurs, the client should retry after waiting a sufficiently long time.

Stacking the result

Each computing node maintains its own migration image on a local disk. Therefore the sum of these images yields the total migration image. The MPI provides a library function, MPI_Reduce(), exactly for this purpose. However, experience shows that IP-based MPI is almost useless on a large and busy cluster environment.

A practical solution to the busy and congested cluster environment would be using a multithreaded program. The program allocates an accumulating image buffer and creates a certain number of threads, each reading a partial migration image on the remote node and accumulating these data in the buffer. The racing condition is easily avoided using a set of mutexes. The stacking performance is relatively good because the relatively time consuming disk read operation is running in parallel. Also, the network transmission is more reliable than NFS because it is using TCP rather than UDP.

Node performance

The last three fields of the shot database record contain the processing history (i.e., start time, end time, and the working node ID). Therefore, node-by-node performance can be easily produced. Table 1 shows the run time summary of an RTM job run on a small experimental Linux cluster. The input data used were from the BP 2004 2D model seismic data. The first column shows node ID (i.e., sequential node number). The second column shows the number of shots processed by the node. The third column shows elapsed time in seconds. The last column shows the node performance (i.e., the average elapsed time in seconds per shot).

Based on the performance, the nodes can be grouped as in Table 2: group A = nodes 0–24; group B = nodes 25–31, and group C = nodes 32–37. Group A consists of two-socket single core Intel Nocona and Irwindale. Group B consists of two-socket dual-core Intel Woodcrest and Opteron 280/2216 (four cores per node). Group C consists of two-socket quad-core Intel Clovertown (eight cores per node).

Rank	Shots	Elapse time (s)	Performance (s/shot)
0	15	11,803	786.9
1	15	11,585	772.3
2	15	11,948	796.5
3	15	11,808	787.2
4	15	11,950	796.7
…	…	…	…
20	15	11,540	769.3
21	15	11,758	783.9
22	15	11,648	776.5
23	15	11,715	781.0
24	15	11,614	774.3
25	52	11,397	219.2
26	53	11,427	215.6
27	52	11,345	218.2
28	53	11,543	217.8
29	49	11,455	233.8
30	51	11,496	225.4
31	51	11,495	225.4
32	102	11,367	111.4
33	102	11,350	111.3
34	103	11,433	111.0
35	103	11,418	110.9
36	102	11,390	111.7
37	100	11,379	113.8
total	1348	443,117	328.7

Table 1. *BP 2004 RTM run time summary.*

It is interesting that the performance gain of 3.5 from Group A to Group B exceeds the core ratio of 2.0. This means that the hardware performance has improved significantly between the models. The performance gain from Group B to Group C is approximately equal to the ratio of the number of cores. This suggests that this particular RTM software is still CPU hungry rather than being limited by input or output. Figure 4 shows the migration image.

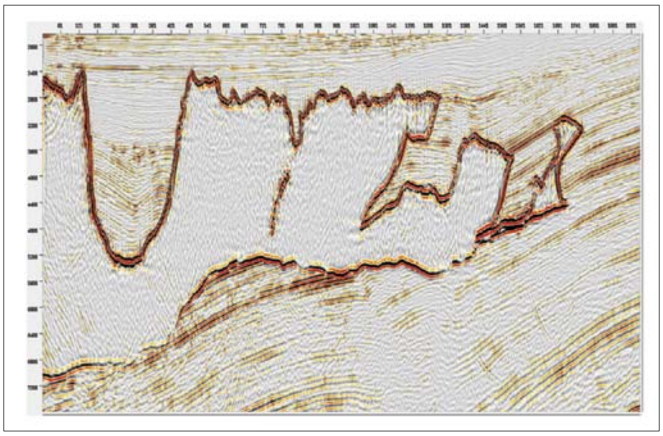

Figure 4. *RTM image of BP 2004 model.*

Group	nodes	cores	perf	model
A	0–24	2	787.5	Nocona/Irwindale
B	25–31	4	222.0	Woodcrest/Opteron
C	32–37	8	111.7	Colvertown

Table 2. *RTM run time summary by group.*

Summary and discussion

The production applications of RTM raise great challenges for cluster based computation. Remote shell-based multi-node programming techniques have been discussed in detail for various stages of RTM programming. Significant improvement in performance has resulted from the application of these techniques. It makes production-scale RTM runs possible. TLE

References

Wang, B., C. Mason, K. Yoon, J. Ji, J. Cai, and S. Suh, 2009, Complex salt model building using combination of interactive beam migration and localized RTM. SEG *Expanded Abstracts*, 28, 3680–3684.

Suh, S., and J. Cai, 2009. Reverse-time migration by fan filtering plus wavefield decomposition. SEG *Expanded Abstracts*, 28, 2804–2808.

Acknowledgments: We Thank TGS for permission to publish this work. We also thank Will Whiteside, Chuck Mason, Yang He, Xuening Ma, and Jean Ji for their valuable contributions toward making this publication possible. Finally, we thank BP and Frederic Billette for providing the BP 2004 model seismic data.

Corresponding author: sang.suh@tgsnopec.com