Overcoming computational cost problems of reverse-time migration

Zaiming Jiang, Kayla Bonham, John C. Bancroft, and Laurence R. Lines

ABSTRACT

Prestack reverse-time migration is computationally expensive. Program run times are long, in terms of the total number of CPU cycles, and it requires large amounts of hard disk free space. To accelerate computing, we do parallel processing using Intel Threading Building Blocks (TBB) and multi-core computers, for both the forward-time modelling and reverse-time migration phases of the computation. To solve the problem of limited free disk space, we use a technique that may seem counter-intuitive: the forward modelling phase is done twice instead of once. Two other enduring problems are described at the end of the paper: the requirement for large working memory, and limited access speeds of mass storage (hard disk) relative to the speed of computation.

INTRODUCTION

Elastic wave modelling based on finite-difference methods is time consuming. For example, it took Martin (2004) a total of 70,000 hours, or approximately 8 CPU years to do elastic wave modelling using the Marmousi2 model.

Prestack reverse-time migration needs even more computational time. Gavrilov at al. (2000) have already modified parallel computing to accelerate reverse-time migration. They used the message-passing interface (MPI) to develop a “distributed parallel implementation” and carried out the computing on a cluster computer with many processors.

This report describes another method of parallel computing, developing software using Intel TBB, a C++ template library introduced in 2006 for writing software programs that take advantage of multi-core processors, and carrying out computation on multi-core processors, which have been widely used both in cluster computers and personal computers ever since Intel developed its first dual-core processor in 2005. As pointed out by Lines, Castagna, and Treitel (2001), the “advances in computer hardware have had a big impact on how we operate.” Multi-core processor parallel computing will become more and more common.

Besides CPU time, disk space is another important computational resource when doing reverse-time migration. To solve the problem of limited free disk space, we use a technique that may seem counter-intuitive: the forward modelling phase is done twice instead of once.

This report also addresses two other computational problems: the requirement for large working memory and limited access speeds of mass storage (hard disk) relative to the speed of computation.

COMPUTATIONAL TIME

Prestack reverse-time migration is more computationally intensive than seismic modelling since it involves not only forward modelling, but also reverse-time extrapolation and imaging. If we do reverse-time migration on the Marmousi2 model, for which the number of numerical nodes in the x direction is 13601 and the number in the z direction is 2801, the total time will be at least doubled, i.e., up to 16 CPU years.

Our solution is to use parallel processing to accelerate the computation. The implementation of parallelization was first done on a dual-core PC, and then the parallelized code was ported onto an 8-CPU shared memory computer, available to us as a single node of Gilgamesh, CREWES' new cluster computer.

Hardware for parallel computing: multi-core computers

Our first parallel computing experiment was carried out on a dual-core PC. This is a Lenovo R60e notebook computer, which has an Intel core 2 CPU (1.83 GHz) and 3 GB memory. The Intel Core 2 CPU has two CPU cores, which can be deployed to do parallel computing. The logical architecture of the computer is shown schematically in Figure 1.

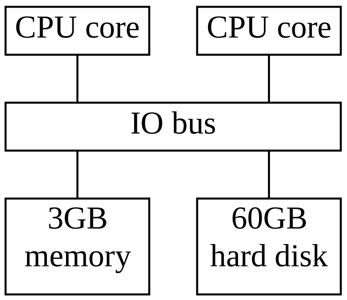

FIG. 1. The logical architecture of the dual-core Lenovo R60e notebook computer.

One of Gilgamesh's 19 nodes is used to perform our second set of experiments in parallel processing. Each node of Gilgamesh is based on the Super Micro X7DVL-E system, with two Intel Harpertown 2.66 GHz quad-core processors. The logical architecture of the Gilgamesh node is similar to that of the dual-core PC, except that the number of cores is 8 instead of 2, the RAM memory size is 16 GB instead of 3 GB, and the total space of 2 hard disks is 320 GB (Bonham et al. 2008). The logical architecture of Gilgamesh is shown schematically in Figure 2.

Software: Intel TBB

Intel® Threading Building Blocks (TBB) is used to parallelize the modelling and reverse-time migration application. Intel TBB is a C++ template library for writing software programs that take advantage of multi-core processors. There are two builds of it: commercial build and open source build. According to Intel's website, "these are built from the same source code, the only real difference is the license and support offering". What we use is the open source build. The most recent version is 2.2; what we use is version 2.1, which was released in June 2008.

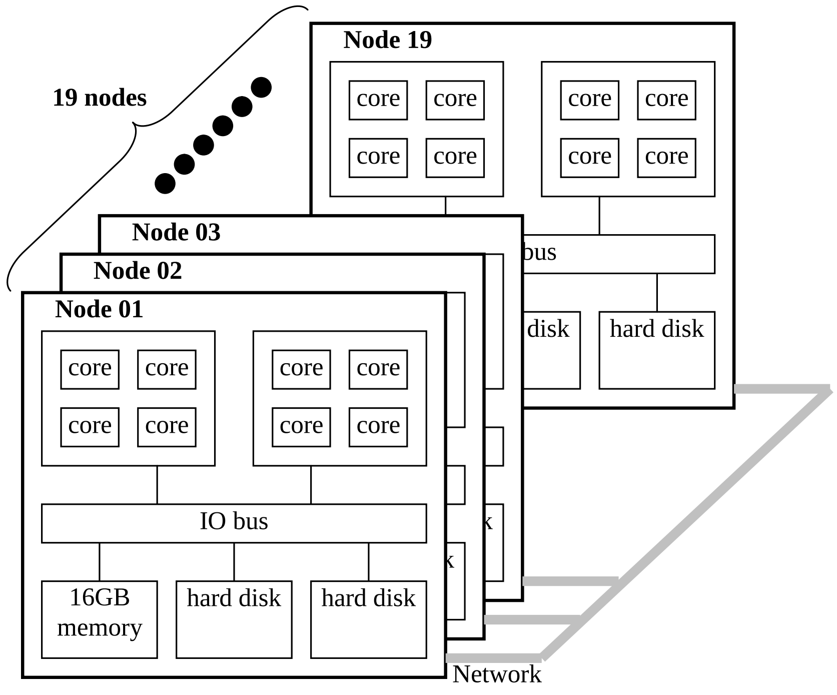

FIG. 2. The logical architecture of Gilgamesh. There are 19 computer nodes connected by a high speed network. There are 8 CPU cores for each node. The total hard disk space is 320 GB for each node except for node 01.

Environment variables need to be registered before Intel TBB can be used. Practically, the Intel TBB documentation does not cover all the installation cases over different operation systems, and for completeness we list our practices in an appendix.

Software design using Intel TBB

Once the Intel TBB package is installed, the modelling and migration programs, which are written in C++, are parallelized using the header files and library provided by the TBB package.

The simplest form of parallelization is a loop of the C++ template function tbb::parallel_for. It looks like a "for" loop in C/C++, but involves the definition of an iteration space. The tbb::parallel_for compares the iteration space size and the available CPU cores, and then breaks down the iteration space into chunks. Then the function tbb::parallel_for runs each chunk on a separate thread.

For the prestack reverse-time migration, the intuitive choice of iteration space is by shot numbers. In the "main" function of our C++ code, the parallel_for is called as follows:

```
parallel_for(

        blocked_range<int>(0,nSrcXNumb),

        CModelingAndRTM(param),
```

auto_partitioner());

where 'blocked_range' indicates what the operation object or iteration space is; 'CModelingAndRTM' is a C++ class which contains the code for modelling and reverse-time migration and specifies how the operation is done.

Efficiency of parallelization

The efficiency of parallelization was first tested running the modelling and reverse-time migration program on the dual-core PC using a simplified model which contains 872 nodes in the x direction and 366 nodes in the z direction. For 10 shots, the parallelized program employs the dual-core CPU, and the total computation time is reduced by 44.7% (Figure 3).

We then tested a modelling program on a Gilgamesh node with eight CPU cores. The subsurface model contains 3000 nodes in the x direction and 800 nodes in the z direction. For 16 shots, the parallelized program employs the eight CPU cores and the total computation time is reduced by 75.3% (Figure 3).

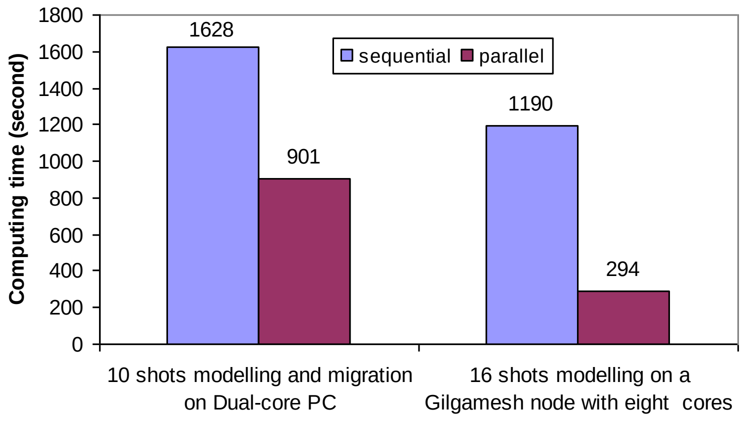

FIG. 3. Computational costs of sequential and parallel programs on a Dual-core PC, and a Gilgamesh node with eight CPU cores.

HARD DISK FREE SPACE

The problem of limited hard disk free space arises when we try to calculate the source-normalized crosscorrelation imaging condition. The imaging condition is

$$image(x,z) = \frac{\sum_{time} S(x,z,t)R(x,z,t)}{\sum_{time} S^2(x,z,t)}, \quad (1)$$

where $S(x,z,t)$ and $R(x,z,t)$ are, respectively, the source wavefield produced by modelling and the receiver wavefield produced by reverse-time extrapolation. Imagine that we have decided that we should compute the forward modelling by time steps t = [1, 2, 3, …, T]. When we reach the last time T, we begin the reverse-time migration phase by time steps t = [T, T-1, T-2, …, 1]. At each step t in the reverse-time calculation, the imaging condition requires crosscorrelation with the corresponding t in the forward time calculation. This means stepping backward through the snapshots of the wavefield representation, which had been previously computed in the forward direction. Unfortunately, the disk space required to store every step in the forward calculation would be prohibitive.

Take the size-shrunk Marmousi, which has a size of 2453×798 nodes, for example. When it is necessarily padded on both sides and on the bottom, the size is 4452×1048. To store the vertical component of one snapshot, we need at least 18,644,976 bytes, i.e., approximately 18 MB. For the 9599 time steps to model one shot, we need free disk space of about 170,682 MB. To fully use the 8 cores of a Gilgamesh node, we need free disk space of 1,365,456 MB, which is far larger than the local hard disk total space. If both vertical and horizontal components of the wavefield are needed, the free disk space needed is doubled.

One solution to this problem can be to use CPU time, doing modelling twice instead of once, to keep the disk space requirements within available limits. During the first forward modelling phase, instead of saving all the wavefield snapshots (subsurface particle horizontal and vertical velocities) for each time t = [0, 1, 2, 3, …, 9599], we save the wavefield state (subsurface particle velocities and stresses) for only every 1000th one, i.e., for t = [1000, 2000, 3000, …, 9000]. When we work backwards in the reverse-time migration for t = [9599, 9598, 9597, …, 1, 0], we can re-model each block of 1000 from the stored wavefield state at the time it is needed for the crosscorrelation. For example, we would re-compute snapshots for time t = [3001, 3002, 3003, …, 3999] from the stored wavefield state at time t = 3000. Thus, without storing all the model snapshots at every time t onto disk, the imaging condition can be implemented, although the modelling has to be done twice (Figure 4).

Even though the modelling is done twice, the computation is still accelerated. Without this re-modelling strategy, the modelling and migration experiment over the size-shrunk Marmousi model, which is padded to a 4452×1048 grid, could only be done serially on the Gilgamesh node storing only one component of the snapshots because of the limited local disk space. The computation time was 12048 seconds, i.e., approximately 3.35 hours for one shot. To compute 32 shots shot-by-shot, it would take 385536 seconds, i.e., 107.09 hours. Now, with this re-modelling strategy, all 8 CPU cores are made use of at the same time, and although the modelling of each shot is done twice, the computation time is still reduced. The parallel computation time is 143757 seconds, i.e., approximately 40 hours for 32 shots. Thus the computation time is reduced by more than 62.7%.

OTHER COMPUTATIONAL COST PROBLEMS

There also exist some other problems in addition to the computing time and free disk space challenges. Here we describe two of them: memory requirements and the bottleneck of hard disk drive I/O speed.

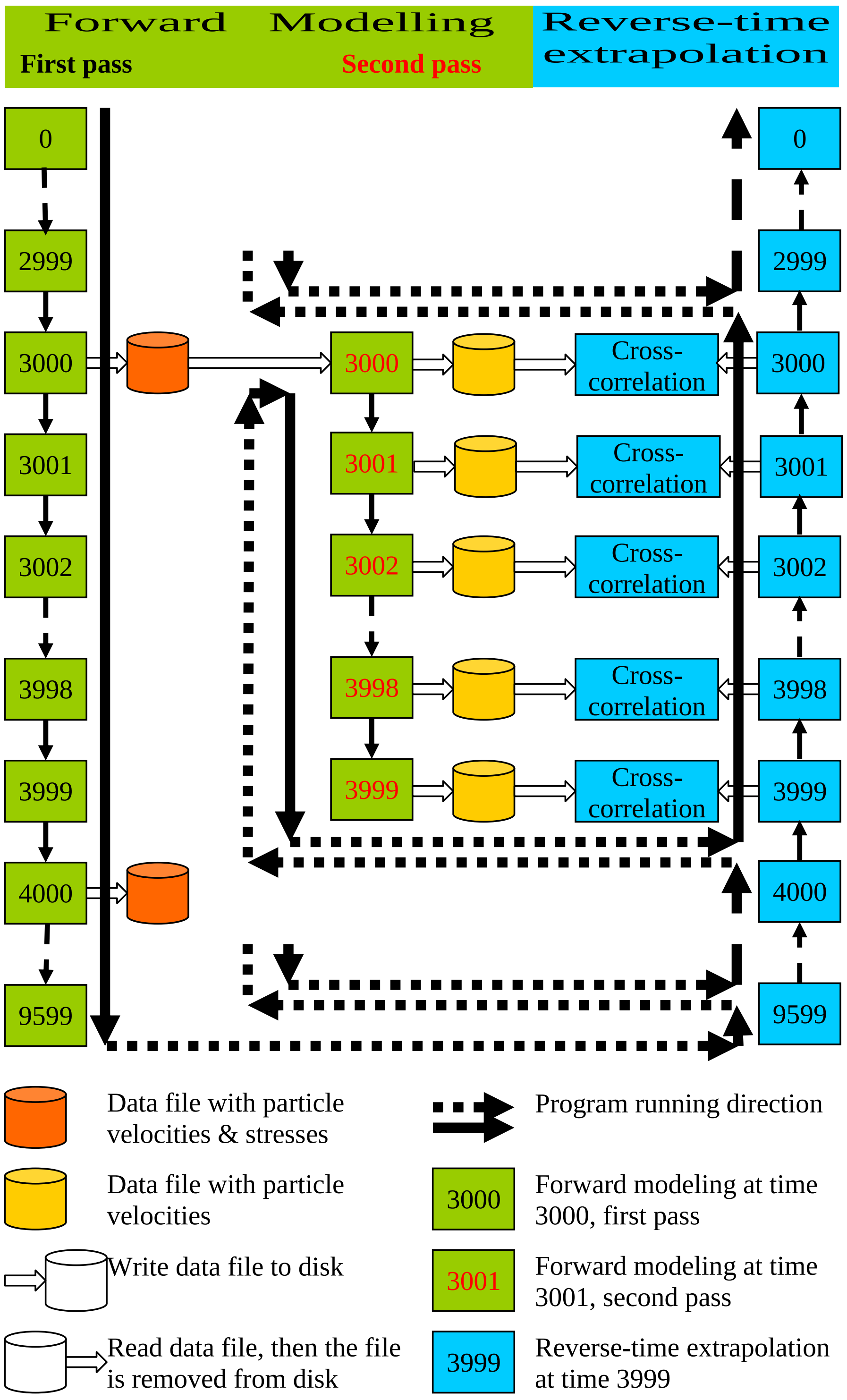
Forward Modelling
First pass
Second pass
Reverse-time extrapolation
0
2999
3000
3001
3002
3998
3999
4000
9599
Cross-correlation
Data file with particle velocities & stresses
Data file with particle velocities
Write data file to disk
Read data file, then the file is removed from disk
Program running direction
Forward modeling at time 3000, first pass
Forward modeling at time 3001, second pass
Reverse-time extrapolation at time 3999

FIG. 4. Do modelling twice instead of once, to keep the disk space requirements within available limits.

Reverse-time migration needs a large amount of memory. Take for example the well-known elastic Marmousi2 model. The model has a 13601×2801 grid. Thus, there are 38,096,401 nodes in all. For each node, 4 bytes are needed to store data of type 'float', which means 152,385,604 bytes for the matrix. The elastic model has 3 parameter matrices: densities, P-wave velocities, and S-wave velocities, which are used in a non-staggered grid finite-difference method, or densities and Lamé coefficients, which are used in a staggered-grid scheme. So, to load the model, 457,156,812 bytes or approximately 436 MB of memory are needed. To perform finite-differencing, 5 more parameter matrices (2 particle velocities and 3 stresses in the case of a staggered-grid scheme) of the same size as the grid need to be loaded in memory for each of 2 successive time steps. So the memory needed for finite-differencing is 1,523,856,040 bytes, i.e., more than 1,453 MB. Thus, the total memory requirement for forward modelling or reverse-time extrapolation is at least 1,981,012,852 bytes, i.e., more than 1,889 MB.

Hard disk drive input / output speed is sometimes the performance bottleneck of parallel programs which are based on multi-core processing. There are two reasons why hard disks can be the bottleneck. First, hard disk I/O speed is much slower than memory I/O. Secondly, the eight CPU cores in one Gilgamesh node compete with each other for writing to and reading from the two local disks. When there is a lot of disk I/O, the whole node will be slowed down. In fact, we actually observed this phenomenon: when all the eight CPU cores are computing without disk I/O, the percentage of CPU usage shown by the utility ‘top’, is usually 800 percent or close to this number, i.e., all the eight cores are fully made use of; when the eight CPU cores need to do disk I/O, the percentage sometimes can be as low as 200, i.e., six of the eight CPU cores are waiting for the disks at that moment.

If we must process large-size seismic data, or we must do a lot of disk I/O for some other reasons, we would have to learn to play some new tricks to overcome the challenges.

CONCLUSIONS

Modelling and reverse-time migration based on finite-difference methods are compute-intensive. The challenges are the long computational time and the need of large hard disk free space. To accelerate computation, parallel computing is implemented using Intel TBB and multi-core computers; to overcome disk space limitations, the modelling part of the reverse-time migration is done twice, instead of once, in “chunks” whose size is optimized to avoid exceeding the available disk space.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of CREWES sponsor companies and various CREWES staff and students.

REFERENCES

Bonham, K., Hall, K.W., and Ferguson R.J., 2008, The epic of Gilgamesh: CREWES' new cluster computer, CREWES research report, **20**, 70.1-70.12.

Gavrilov, D., Lines, L., Bland, H., Kocurko, A., 2000, 3-D depth migration: parallel processing and migration movies, The Leading Edge, **19**, 1282-1284.

Lines, L.R., Castagna, J.P., and Treitel, S., 2001, Geophysics in the new millennium, Geophysics, **66**, 14.

Martin, G.S., 2004, The marmousi2 model, elastic synthetic data, and an analysis of imaging and AVO in a structurally complex environment, MSc thesis, University of Houston.

APPENDIX

Before Intel TBB can be used, environment variables need to be registered to the operating system. Unfortunately, the Intel TBB documentation does not cover all the environment variable registration cases over different operating systems. We list our practices as below.

On our PC, which has Windows XP and Microsoft visual C++ 2008 Express Edition installed, environment variables are manually set, since, unfortunately, the Intel TBB plug-in for Microsoft Studio did not work. The manual way of registering environment variables is: right click 'My Computer' --> Properties --> Advanced --> Environment variables. On the dialog 'Environment variables', set values for the variables 'include', 'lib', and 'path'. Suppose that the Intel TBB directory 'tbb21_20080605oss' is copied to 'C:\Program Files\Intel\', and suppose that the CPU has a 32-bit Intel Architecture, then add 'C:\Program Files\Intel\tbb21_20080605oss\include' to the 'include' variable, 'C:\Program Files\Intel\tbb21_20080605oss\ia32\vc9\lib' to the 'lib' variable, and 'C:\Program Files\Intel\tbb21_20080605oss\ia32\vc9\bin' to the 'path' variable.

When Fedora 9 is used on the PC, the following lines are added to the bash shell start-up file:

```
TBB21_INSTALL_DIR=$HOME/tbb21_20080605oss

TBB_ARCH_PLATFORM=ia32/cc4.1.0_libc2.4_kernel2.6.16.21

LD_LIBRARY_PATH="${TBB21_INSTALL_DIR}/${TBB_ARCH_PLATFORM}/lib"

export LD_LIBRARY_PATH

LIBRARY_PATH="${TBB21_INSTALL_DIR}/${TBB_ARCH_PLATFORM}/lib"

export LIBRARY_PATH

CPATH=$HOME/tbb21_20080605oss/include

export CPATH
```

where ‘$HOME/tbb21_20080605oss’ is the directory of the Intel TBB package.

To use Intel TBB on Gilgamesh, on which the operation system is CentOS, a ‘.cshrc’ start-up file is created, which contains:

```
setenv TBB21_INSTALL_DIR        $HOME/tbb21_20080605oss

setenv TBB_ARCH_PLATFORM        em64t/cc4.1.0_libc2.4_kernel2.6.16.21

setenv LD_LIBRARY_PATH          $TBB21_INSTALL_DIR/$TBB_ARCH_PLATFORM/lib

setenv LIBRARY_PATH             $TBB21_INSTALL_DIR/$TBB_ARCH_PLATFORM/lib

setenv CPATH                    $HOME/tbb21_20080605oss/include
```

where ‘$HOME/tbb21_20080605oss’ is the directory of the Intel TBB package.