Three strategies for parallelizing a 3D seismic migration algorithm on distributed memory environments

Alejandro E. Murillo
Center for Wave Phenomena

Omar Uzcategui
Intevep SA, Caracas 1070A, Venezuela

ABSTRACT

In this paper we describe the design and implementation of parallel algorithms for a post-stack depth migration with explicit operators program. We analyze three different strategies to parallelize this numerical and I/O intensive application. Even though performance efficiency is a useful factor that can be applied objectively, we do not focus the analysis of these three strategies on it. We consider other factors like memory management and feasibility of the implementation. We shall see that sacrificing some levels of parallelism we can come out with a parallel algorithm implementable on cluster of medium capacity processing units, like a network of Pentiums running LINUX. We also illustrate a technique to incorporate restart capabilities in the final implementation to avoid spending costly CPU cycles redoing computational steps after unexpected system crashes.

Introduction

In this paper we describe our experiences implementing three strategies to network-parallelize a 3D depth migration with explicit operators algorithm (Uzcategui, 1994), (Hale, 1992). The major contributions of this paper include:

(i) Illustrating how we can tackle down a numeric and I/O intensive algorithm on a cluster of low capacity workstations by sacrificing some level of parallelism

(ii) Illustrating how the design of costly parallel algorithms can incorporate useful and inexpensive restart capabilities

Some of the strategies discussed here are similar to the ones discussed in papers by (Black & Su, 1992) and (Almasi *et al.*, 1993), nevertheless, we show them along with implementation details. We shall provide the reader with the advantages and disadvantages of using each one of the them and how they adapt to various network configurations. The first strategy presented can be expected to achieve very good speedup and efficiency, but its memory requirements make it kind of expensive to implement. The second proposed strategy, intended to target low end cluster of workstations, sacrifices some level of parallelism to highly reduce the need for processing units with high memory capacity. It exploits the pipelining concept. The third strategy proposed, which arises from the need of incorporating restart capabilities to the final implemented program is a minor modification of the first scheme that reaches similar level of performance.

The organization of this paper is as follows. First we present a description of the sequential algorithm. Next, we describe and analyze each one of the three alternatives.

Sequential Algorithm

The migration program starts by performing a 1D Fast Fourier Transform (FFT) on each of the *(nx * ny)* input traces and produces another *(nx * ny)* complex traces of *nw* elements each, where *nw* depends on *nt* . The output of this transformation is a data cube of frequency values as shown in Figure 2. This cube can be visualized as *nw* horizontal slices of *(nx * ny)* elements each, every one associated with a wave number.

Once this transformation is completed, the next step of the algorithm is to calculate the desired depth steps (*nz*). Each depth step, $P(z_i)$, is a slice of *(nx * ny)* floating-

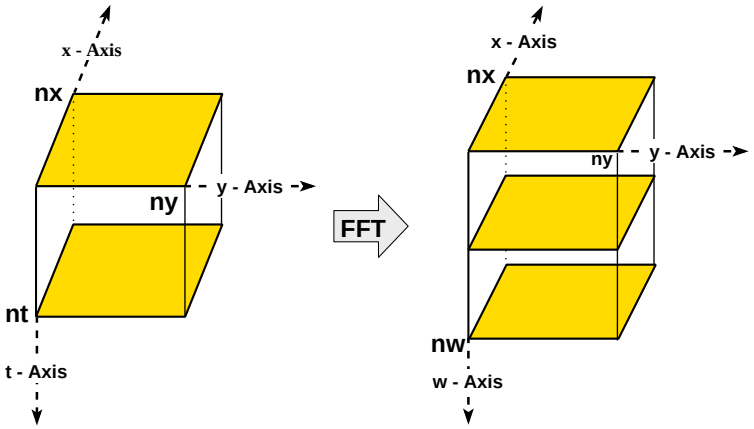

Figure 2. Example of a 3D seismic data FFT transformed to *(x,y,omega)*-space. Each slice of the right hand side cube is denoted by g_i and corresponds to wave number i.

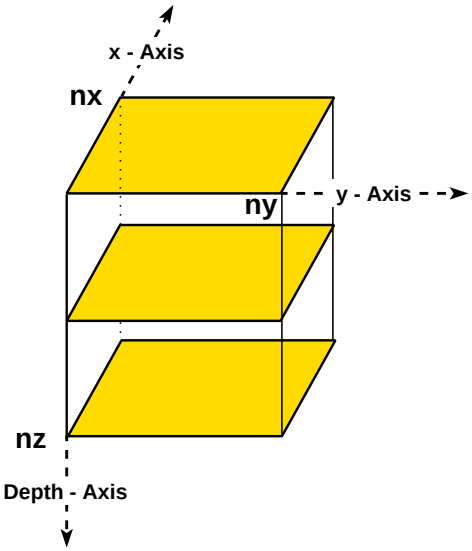

Figure 1. Example of a 3D seismic data in depth. Each horizontal slice represents the subsurface at depth z_i.

point numbers describing the subsurface at depth z_i of the area under study. This part of the algorithm is a nested loop that consumes more than 90% of the total execution time. This loop can be expressed mathematically as:

$$P(z_i) = \sum_{j=1}^{nw} Filter(g_j, v_i), \text{ for } i = 1, ..., nz$$

where g_j is the frequency data (slice) associated with the wave number j of the FFT data cube (the one shown in Figure 2) and v_i the velocity data at depth z_i. In summary, the wave field at each depth step is obtained by summing all the frequency slices g_i, after having applied the depth-dependent McClellan filter (Uzcategui, 1994) to each one of them.

The operation $Filter(g_j, v_i)$ corresponds to applying a depth-dependent McClellan filter to every one of the elements of the slice g_j, After applying a filter to one slice, each element of that slice becomes the sum of the convolution of the 20 elements surrounding it with the specific filter. An important point here is that the filter values depend on the depth being calculated and the velocity at that depth. A critical issue to pinpoint here is that to apply this filter to a slice of frequency g_j to obtain the wave field at depth step (slice) $P(z_i)$, that frequency slice must have been previously processed with the corresponding filter for depth step $P(z_{i-1})$.

Once all this processing has been done, the output will be a cube of *(nx * ny * nz)* floating-point numbers describing the structure of the subsurface under study (see Figure 1). The pseudo code for this algorithm is the following:

```
1.- FFT transformations to obtain cube Q.

      -- Q cube of frequency values (FFT output)
      -- Z cube with the wavefield in depth
      -- V cube with the velocity model

2.- For (IZ = 0; IZ < NZ; ++IZ) {

        2.1 Z[IZ] = 0.0 /* Clear depth step IZ */

        2.2 For (IW = 0; IW < NW; ++IW) {

                2.2.1 Q[IW] = filter (Q[IW], V[IZ])


          2.2.2 Z[IZ] = Z[IZ] + real_part(Q[IW])

            } /* End of loop for IW */

        2.3 Write or store Z[IZ]
```

```
    } /* End of loop for IZ */

3.- End
```

Simplest Strategy

The first alternative to network-parallelize this algorithm is to extend to the 3D case, the algorithm proposed by Almasi et. al. (1993) for the 2D case. This particular scheme follows the master-slave paradigm; a master program spawns p slaves processes, each one is in charge of computing the contribution of specific slices of frequencies to each one of the depth steps to be calculated. In other words, each slave process is provided with a copy of the whole velocity field (this is a cube of NX x NY x NZ floating values) and whenever it is idle, it will receive a frequency slice from the master and compute the contribution of that slice to all the depth steps. When requested by the master, all the slaves send their cubes representing the wave field in depth to the master, which will then sum them to obtain the final image. The next listing shows the algorithm followed by the master program:

```
1.- Send (or have slaves read) the velocity cube

2.- For (IW = 0; IW < NW; ++IW) {

        2.1 IP = next ready slave

        2.1 Send frequency slice IW to process IP

    } /* End of loop for IW */

3.- For (IZ = 0; IZ < NZ; ++IZ) {

        3.1 For (IP = 0; IP < P; ++IP) {


          3.1.1 Recv partial Z-step IZ from Proc IP


          3.1.2 Add partial result to depth step IZ

        } /* End of loop for IP */

        3.2 Write depth step IZ to disk

    } /* End of loop for IZ */
```

The algorithm corresponding to each slave program is described in the following listing:

```
1.- Receive (or read from a file) velocity INFO

2.- While ((IW = get_a_slice_of_freq()) != NULL)
    {

        For (IZ = 0; IZ < NZ; ++IZ) {

                Compute contribution of this wave
                number (IW) to the depth step IZ

            } /* End of loop for IZ */

        } /* End of While */

3.- For (IZ = 0; IZ < NZ; ++IZ) {

        Send depth step IZ to the MASTER

    } /* End of loop for IZ */
```

This scheme provides a very efficient parallel implementation. All the subprocesses are busy all the time (high efficiency) and in the event of having nw processing units available, we might accomplish the complete migration in a time very close to that taken by a single processor to complete the contribution of a single slice of frequencies to all the depth steps; this means a speedup of nw with respect to the sequential algorithm.

However, the memory needs associated with this algorithm are extremely high. Each processing unit needs enough space to store the cube of velocity and the cube corresponding to the wave field in depth. For an input dataset of size 300 x 300 x 700, we are talking of cubes of 252 MB each. A communication bottleneck associated with this scheme arises when the all the slaves, simultaneously, send the partial wave field in depth they calculated to the master. This is required to sum them up and obtain the final image. This is an expensive communication step (many-to-one) in which all the slaves simultaneously send a substantial amount of data to the master program.

This algorithm can be safely implemented on clusters of high capacity workstations like those composed by IBM RS6000 (SP2) or SGI power challenge machines.

Pipelining Strategy

As we mentioned above, the time consuming portion of this algorithm is a nested loop that we can summarize as:

For i = 1 to nz
  For j = 1 to nw
    $g_j = Filter(g_j, v_i)$
    $z_i = z_i + g_j$
  EndFor
EndFor

where g_j is the slice of frequencies corresponding to wavenumber j, z_i is the ith depth step, v_i the velocities at that depth step and $Filter(g_j, v_i)$ corresponds to applying the McClellan filter to the slice of frequency g_j.

An opportunity to parallelize this loop under a message-passing environment comes from the fact that iteration $i + 1$ of the outer loop can be started right after

the first iteration of the inner loop has been completed for iteration i. As the inner loop code takes substantial time to complete, we have a chance of overlapping computation with communication by breaking the outer loop into nz processes, each one in charge of executing the inner loop code for a specific iteration of the outer loop. Since each iteration of the outer loop can be started just after the previous one has completed the first iteration of the inner loop, we get a concurrent execution of the outer-loop iterations similar to that available in a pipeline processing unit.

Essentially, what we are doing is using pipelining techniques to achieve a certain level of parallelism when executing the outer-loop iterations. If we implement this technique over a networked group of machines, then we will be using the network as a coarse grain pipeline unit (Murillo, 1995).

The technique can be summarized in the following way. We assume that the nested loop has to perform n outer iterations and n inner iterations, and we can think that the data processed at iteration i are the input to the iteration $i + 1$. The idea consists of creating n or fewer virtual processes. Process P_i will be in charge of performing the inner-loop code for one or more iterations of the outer loop; it will receive its input from virtual process P_{i-1} and will send its output to process P_{i+1}. The input for process P_0 is the input of the loop, and the output of process P_{n-1} will be the output of the loop.

Each application node (virtual process) can be considered an accumulator. It receives frequency vectors, processes them, accumulates their contribution to each of the depth steps it has been assigned to calculate and sends the modified frequency vectors to the next process in the pipeline.

If we neglect the communication time and assume that each process convolves a slice of frequencies in τ units of time, and let $p = nz$ be the number of processors available, then the total time spent in this algorithm to compute nz depth steps is given by $2 * nw * \tau$ time units, as opposed to the $nz * nw * \tau$ consumed by the sequential algorithm. Thus computation time is reduced from $O(n^2\tau)$ to linear order.

Restart Capabilities

As this algorithm handles a large amount of data and performs a large amount of numerical computations, it is natural to expect wall clock execution times in order of hours and some time days, when applied to real data input. Introducing restart options in the implementation of this algorithm involves the execution of some extra computing steps, usually I/O related, that might have some impact in the total computing time, but it will facilitate recovering from machine crashes and thus avoid the re-execution of expensive computations.

For instance, the output of this algorithm are nz depth steps. We can arrange the processing so that at any moment in time all the slave processes work toward completion of a single depth step. When a depth step has been fully calculated it can be saved on disk, along with its depth step number, so that in the event of a restart, the computation will be resumed at that point.

The algorithm is as follows. Let us suppose that we wish to calculate nz depth steps and after the FF Transformation we ended up with nw slices of frequency. Also, let us assume that we have p processing units available. The algorithm for the master program is shown below:

```
1.- Read and distribute NW frequency slices
    among the slaves. Each one will receive
    SW slices, where

        SW = NW / P
        NW = Total number of freq. slices
        P  = Number of slaves

2.- For (IZ = 0; IZ < NZ; ++IZ) {

        2.1 Read and broadcast velocity for Z-step IZ

        2.2 For (IP = 0; IP < P; ++IP) {

                2.2.1 Receive partial Z-step IZ from
                      process IP


          2.2.2 Add that partial result to Z-step IZ

            } /* End of loop for IP */

        2.3 Write depth step IZ in disk

        2.4 Save restart information (if required)

    } /* End of loop for IZ */
```

The algorithm for the slaves processes is as follows:

```
1.- Read or receive SW frequency slices

        SW = NW / P
        NW = Total number of freq. slices
        P - Number of slaves

2.- For (IZ = 0; IZ < NZ; ++IZ) {

        2.1 Get velocity for depth step IZ

        2.2 For (IW = 0; IW < SW; ++IW) {

                2.2.1 Compute IW wave number

                2.2.2 Compute contribution of IW
                      wave number to the Z-step IZ

            } /* End of loop for IW */

        2.3 Send depth step IZ to the MASTER
```

```
        2.4 Save restart information (if required)

    } /* End of loop for IZ */
```

We can quickly summarize some advantages of this scheme in comparison with the previous two. First it has the restart capabilities. Second, the slaves processes do not need to store the whole velocity field. The slaves receive the velocity information step-by-step, they use it at once and then is not needed any more, so it can be overwritten by the velocity information for the next depth step. Similarly, the slaves don't need to allocate space to store the whole wave field in depth. The depth steps are obtained sequentially, so once a process has computed its corresponding portion for a depth step, it sends it to the master and that space is reused for the computations of following depth steps.

Summary

We have presented three different strategies to parallelize a depth migration algorithm in a distributed memory memory environment. The first strategy, a 3D generalization of Almasi, et. al. algorithm, can be expected to achieve very good speedup and efficiency, but its memory requirements make its implementation infeasible in low-end workstation clusters. It is recommended for implementation on networks with processing units having high memory capacity like a farm of SGI Power Challenges or similar. The second proposed strategy, intended to target low end clusters of workstations, sacrifices some level of parallelism to reduce memory requirements. It exploits the pipelining concept. The third strategy proposed, is a minor modification of the first scheme. It can reach similar level of performance and beside that incorporates restart capabilities to the final program which is extremely convenient given the expected execution time when this algorithm is applied to field data.

References

Almasi, G., Hale, D., Bell, J., & Gordon, A. 1993. Parallel Distributed Seismic Migration. *Concurrency: Practice and Experience*, **5**, 105–131.

Black, J., & Su, C. 1992. Performance of Parallel Downward Continuation. *62th Ann. Mtg., SEG, Expanded Abstracts*, **1**, 326–329.

Hale, Dave. 1992. Computational aspects of Gaussian bean migration. *CWP annual report*, **1**.

Murillo, Alejandro E. 1995. Parallelizing loops over a network of workstations. *CWP annual report*, **15**, 249–263.

Uzcategui, Omar. 1994. Anisotropic depth Migration . *CWP annual report*, **14**, 305–315.