Loop Level Parallelization of a Seismic Inversion Code

William W. Symes
Michel Kern

CRPC-TR93304
March 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Loop Level Parallelization of a Seismic Inversion Code

William W. Symes
Michel Kern

March, 1993

TR93-06

Loop level Parallelization of a seismic inversion code *

M. Kern† W. W. Symes†

Abstract

We present a parallel implementation of a seismic inversion code. Parallelism is exploited at the loop level within the finite difference modeling, as this is the most time consuming part of the code. We give details of our implementation, and present numerical results. We have reached a performance of 250 Mflops on one processor, and speedups of 6 on 8 processors, on a Cray Y-MP.

1 Introduction

This paper presents experience with trying to exploit loop-level parallelism in DSO, a seismic inversion under development in the Rice Inversion Project.

Seismic problems are characterized by the huge amount of data they generate. A typical seismic survey would generate several hundreds of Mbytes of data! Then, as will be explained below, the inversion procedure requires repeated solution of the wave equation (if we can be satisfied with acoustics !). It is thus of paramount importance that each of these solutions be done as fast as possible. Obviously, it is also desirable to solve as few of them as possible, an issue we are currently addressing.

This paper focuses on one aspect: trying to get the code to run efficiently on vector-parallel machines, a prime example of which is the Cray Y-MP.

An outline of the rest of the paper is as follows: in the next section we give some background on the algorithm, and the way the code is written. We detail the guidelines in obtaining good performance on a Cray Y-MP in section 3, and show a few representative examples in section 4. We conclude with an assessment of these results, and discuss directions for further work.

2 DSO, a framework for inverse problems

2.1 Motivation

It is not the purpose of this paper to describe the algorithm in detail. The reader is referred to [16], [18] for a detailed explanation of the algorithm, and to [17] for examples.

*This work was partially supported by the Office of Naval Research (N00014-89-J-111), the Texas Geophysical Parallel Computation Project and the Rice Inversion Project. TRIP sponsors for 1992 were Amoco Research, Conoco Inc., Cray Research, Earth Modeling Systems, Exxon Production Research Co., and Mobil Research and Development Corp.

†The Rice Inversion Project, Department of Computational and Applied Mathematics, Rice University, Houston TX 77251-1892

Inverse problems have traditionally been formulated as least-squares. This has several advantages: it makes fitting inexact or noisy data possible, and it makes the function to be minimized differentiable, thus amenable to classical optimization methods. Attempts along these lines have met with limited success (see [19] for an extensive study of least-squares methods in the geophysical context, and [10] for a successful application of the method, and what is needed to make it successful). It is now realized that pure least-squares is flawed, and *cannot* work (see [14] for an explanation). Still, the desirable properties quoted above (and the experience accumulated with least-squares) makes it desirable to look for modifications instead of a radical change of method (this is also being pursued, see [15] for example).

The main difficulty in using least squares to fit reflection data is that a good knowledge of the low frequency trend of the velocity is needed in order to find the high frequency component. Geophysicists all know that "once you know the velocity, it is comparatively simpler to determine the reflectivity". As will be explained below, "the reflectivity" is the rapidly oscillating part of the velocity field, and is responsible for reflected waves, whereas "the velocity" is the slowly varying component, mainly responsible for the kinematics.

Seismic campaigns are based on a large number of different experiments (corresponding to different source positions, or shot points), so each shot point generates its own "view of the earth", and unless the velocity is already quite accurate, it is difficult to reconcile these views, that is to fit the reflectivity to different shots. On the other hand, for any given velocity, it is possible to fit the reflectivity for one shot, but different shots will give different reflectivities.

Accordingly, DSO is based on two modifications to simple least squares:

- first, one separates the different scales in reflectivity and velocity;
- the model is then *enlarged* to allow the reflectivity to depend on the position of the shot point. This this does not make sense ("there is only one earth"), a penalty term is applied to impose that *neighboring* shots look alike.

The actual objective function is thus the sum of two terms:

- a least-squares misfit term, to still try and fit the data,
- a differential-semblance term, to force the reflectivity to be independent on the location of the source.

Analysis shows that if this function is minimized first over the reflectivity (which is feasible, according to the above discussion), the resulting cost function, which only depends on the velocity, is smooth and convex and thus can be minimized effectively by gradient-based methods.

2.2 An actual example

As introduced, DSO is applicable to almost any physical model of propagation. Indeed, the code is built in such a way as to be independent of any particular model (we return to that point in section 2.3). Nevertheless, the results described in this paper pertain to the simplest such model: 2D, constant density, linearized acoustics.

We assume that the earth can be described by just one parameter: its velocity distribution (density is constant). The seismic experiment consists in setting off a source

at different points (x_s, z_s) in the subsurface, and recording the excess pressure at several receivers (x_r, z_r). The time dependence of the source, denoted by $f(t)$, is assumed known. We distinguish between the smooth and rough component of the velocity by linearizing the wave equation around a reference, smooth velocity $c(x,z)$, and we denote by $r(x,z)$ the relative perturbation. First order perturbation theory easily shows that p, the scattered field is solution of the following coupled wave equations, where p_0 is the direct field:

$$(1) \qquad \begin{cases} \frac{1}{c^2}\frac{\partial^2 p_0}{\partial t^2} - \Delta p_0 = f(t)\delta(x - x_s, z - z_s) & \text{for } z > 0 \\ \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} - \Delta p = 2r\Delta p_0 & \text{for } z > 0 \end{cases}$$

with zero initial conditions:

$$(2) \qquad \begin{cases} p_0(x,z,0) = \frac{\partial p_0}{\partial t}(x,z,0) = 0 \\ p(x,z,0) = \frac{\partial p}{\partial t}(x,z,0) = 0 \end{cases}$$

and zero pressure on the surface $z = 0$:

$$(3) \qquad \begin{cases} p_0(x,0,t) = 0 & \text{on } z = 0 \\ p(x,0,t) = 0 & \text{on } z = 0 \end{cases}$$

The measured quantity is $p(x_r, z_r, x_s, t)$, and this defines the *forward* map $F[c,r]$, since this quantity depends functionally on c and r.

We can now state the DSO optimization problem. The usual misfit function is (p_{data} being the observed data):

$$J_{\text{LS}}[c,r] = \frac{1}{2}\|F[c,r] - p_{\text{data}}\|^2 = \frac{1}{2}\sum_{x_r, x_s}\int_0^T |p(x_r, x_s, t) - p_{\text{data}}|^2 dt$$

For DSO, we first let r be a function of shot position $r = r(x, z; x_s)$, then penalize for this dependence:

$$J_{\text{DS}}[c,r] = \frac{1}{2}\{\|F[c,r] - f_{\text{data}}\|^2 + \sigma^2\|\frac{\partial r}{\partial x_s}\|^2\},$$

This function is of course no more convex convex than J_{LS}. According to the discussion in section 2.1, we minimize it with respect to r to obtain our cost function:

$$J[c] = \min_r J_{\text{DS}}[c,r].$$

Notice that the definition of J already requires the solution of a quadratic minimization problem in infinite dimension. Hence, in practice we will only be able to produce an approximation to its solution:

$$\tilde{J}[c] = J_{\text{DS}}[c, \tilde{r}[c]]$$

where $\tilde{r}[c]$ is the result of applying a finite number of iterations of some quadratic optimization algorithm to J_{DS}.

Thus we have a two step algorithm:

- An outer iteration to minimize J over velocities;
- An inner iteration to minimize J_{DS} over reflectivities.

Details on how the two steps are implemented can be found in [18]. Let us just say here that the inner iterations involve computing the normal operator associated with J_{DS}, and that the outer iteration will require the gradient of J.

The evaluation of the normal operator uses the now classical adjoint-state technique [11]. To compute $D_r F[c,r]^* \varphi$, where φ is a "seismogram-like" quantity, we first compute p_0 as in equation 1 above. Then we compute the adjoint field w by solving the following problem:

$$\begin{cases} \frac{1}{c^2}\frac{\partial^2 w}{\partial t^2} - \Delta w = \sum_{x_r} \varphi(t)\delta(x - x_r) & \text{for } z > 0 \\ w(x,z,T) = \frac{\partial w}{\partial t}(x,z,T) = 0. \end{cases} \tag{4}$$

Notice that this is solved backwards in time. Then,

$$D_r F[c,r]^* \varphi = -\frac{2}{c}\int w \Delta p_0 dt$$

In practice, this is very similar to solving the wave equation, and just means solving more of them (backwards in time).

It turns out that the computation of the gradient uses an extension of this method. The upshot of these consideration is that the basic computational block of the method is the repeated solution there are several shot points) of the wave equation.

2.3 Structure of the code

The DSO principle is independent of any particular model of the earth. This is also true of our implementation. Procedures for linear and non-linear optimization, Fourier transform or linear algebra, are implemented in a model-independent fashion. Obviously, some description of the particular model used should eventually appear. Since we solve the inner optimization problem iteratively, we need procedure to compute, in addition to F itself, its derivatives, and their adjoints. In principle, these can be obtained in a systematic manner once F is known. In our current implementation they are still hand-coded, which gives us a (hopefully) optimal implementation. We should, however, mention efforts to automate this step: Liu [13] generates Fortran code for the adjoint state given specifications for the forward map. In a more general direction, ADIFOR [2] directly differentiates a Fortran code.

The basic principle guiding the design of DSO can be restated as: *Generic tasks should be coded in a generic way.* An immediate payoff of this approach is that the code is being used with several different models: 2D acoustic is reported here, but we also work on a plane wave, layered model [17], and a visco-elastic model is in progress [3].

The application-dependent part of the code is where all the "action" is, as far ad performance is concerned. Fortunately, in most cases this part will be small and relatively easy to tune, as is certainly the case with our present application. We now present present the implementation for 2D acoustics in the next section section.

3 Implementation of the finite difference code on a multi-vector computer

In this section, we detail our implementation of the finite difference discretization of equations(1) and(4), with a view of obtaining optimal single processor performance on one processor of a Cray Y-MP, and good parallel speedup on several processors.

3.1 The finite difference code

We concentrate on equations (1), as computing adjoints just means solving more wave equations. The first step is to restrict computations to a bounded domain, which we take to be the rectangle $R = \{x_{\min} < x < x_{\max},\ 0 < z < z_{\max}\}$. As shown in equation (3), the fields are 0 on the surface of the earth $z = 0$.

It is commonplace to employ so-called absorbing boundary conditions on the other sides of the rectangle to simulate wave propagation in an infinite domain (see eg. [7]). However such conditions implicitly assume that no reflections occur outside R; if this assumption is incorrect, some part of the data may not be explicable by the model. Instead we prefer to bear the extra computational expense of putting the boundaries of R far enough away from the source that no reflection can arrive in the receiver array from the boundary in the recording interval $\{0 < t < T\}$. That is, we assume that p_0 and p are required to vanish on *all* sides of R for all t;

In practice it is easy to check that R is sufficiently large, using an average value of the slowness $1/c$.

We solve the wave equations (1) numerically using a finite difference method of fourth order in space and second order in time ([5], [12]). Since the boundary conditions specify the vanishing of the fields p_0 and p on the boundary of R, the method of images gives numerical boundary conditions (i.e. one-sided difference stencils near the boundary) of the same accuracy as the interior scheme.

3.2 The parallel implementation

Vectorization is the simplest form of parallelism. Conceptually, the same operation is applied simultaneously to several data. This is similar to data-parallelism, used on the Connection Machine [9]. Compilers now are very successful at automatically recognizing vectorizable code.

We only look at the simplest form of multiprocessing possible on Cray computers, namely Autotasking. It enables loop level parallelism with a minimum of code modifications, and little overhead. Contrary to multitasking, which requires calls to special library functions, autotasking will need, at most, the insertion of directives. Thus the code stays portable.

An explicit finite difference code is inherently parallel. Actually, an early implementation of DSO ran on the CM-2, by exploiting this data parallelism. On a vector-parallel architecture, parallelism can be found on two level: vectorization, and multiprocessing.

The first thing usually taught in Cray parallelization classes is: "Never sacrifice vectorization to parallelism". Indeed, the gains coming from vectorization are much larger than those coming from parallelism. Good vectorization can give speedups of up to 20,

parallelization is limited to the number of processors (4 to 8 in most shared memory vector computers). Accordingly, our first step in optimizing the code is vectorizing, then exploit any remaining parallelism.

As mentioned in the previous section, the part of our code that needs to be tuned is fairly small. As a first approximation, the most important module is the one that implements one step in time for the homogeneous wave equation:

$$p_{i,j}^{n+1} = -p_{i,j}^{n-1} + 2p_{i,j}^{n} + c^2\Delta t^2\left(\frac{4}{3}(\Delta_h p^n)_{i,j} - \frac{1}{3}(\Delta_{2h} p^n)_{i,j}\right) \tag{5}$$

with

$$(\Delta_h p)_{i,j} = \frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{h^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{h^2} \tag{6}$$

This is most naturally implemented via nested loops. Than, we could expect to vectorize the inner loop, and parallelize the outer loop. But by doing this, the inner loop will quite short, only the size of one grid dimension, typically a few hundreds. Even though Cray computers perform well on short vectors, this is not the best approach. The key to obtaining good vector performance is to rewrite these two loops with one single loop over the whole grid, correcting for the wrong boundary values in a separate step. This had already been observed by Clement [4], and indeed leads to a decrease in CPU time by more than a factor of 2.

This is not a transformation that can be expected from a compiler, as an additional corrective step is needed to obtain correct boundary values, and anti-symmetrize the field.

With this transformation, the code performs well on one processor. Because of the rule quoted above, autotasking works by default only on outer-loops, the rationale being that inner loops will be vectorized. But we had to get rid of outer loops to be able to get good vectorization. However, we can still use parallelism, because now the loop is very long: In a typical example, the grid will be 512×128, and this is the length of the loop. we can split this in 8 (if we have 8 processors), and still retain a sufficiently large vector length.

As we show in the next section, small inefficiencies that are innocuous on one processor are noticeable were running in parallel. For example, the adjoint map requires computing the adjoint to the interpolated seismogram. Doing this one time step at a time is inefficient, because only two grid lines take part in the computation. It is better to do it for all time steps at once, as a preprocessing step, than to simply fetch the right values at each time step.

4 Numerical results

4.1 Description of the experiments

We have ran experiments using two different sets of data. The first case, "small grid", corresponds to the typical size of grid we use in inversion experiments. The space grid is 512 by 128. The experiment simulates 20 shot points, and we "record" during 2 s, using 500 time steps.

To discuss scalability issues, we also used a "large grid": 1024 by 256 space nodes. To actually use all of the grid, we had to let the recording last 4 s, (because the code internally selects the smallest computational domain that produces no reflections), that is 1000 time steps, and we extended the line to 40 shot points, as would be done on this larger domain.

In both cases, the velocity was constant, equal to 1500 m/s. The results reported below are for the computation of an adjoint map, for a given seismogram. This was a good compromise between a short execution time, and still exercising a significant part of the program. We have run a full step of non-linear conjugate gradient iteration to check that the huge majority of the time is spent in the finite difference code. Thus, even though our results are not strictly speaking for the whole application, they still pertain to a complete code, with a significant amount of I/O.

4.2 Uniprocessor performance

On a single processor, DSO runs at speeds from 250 Mflops for the small example, to 285 Mflops for the large example. This corresponds to times form 1 minute to 13 minutes.

These numbers were obtained using Cray's performance tools. It is worth mentioning, at this point, how valuable such tools are. They allow the programmer access to such information as the percentage of time spent in a routine (profiling), the performance of any give routine, and the global performance of the program.

For example, it is by using `perfview` that we could check that, on one processor, the solution of the wave equation accounts for more than 95 % of the total computation time.

The numbers above were obtained with the Hardware Performance Monitor. We did not have to insert any flops counting code. A combination of HPM and `perfview` gives, for each routine, the number of floating points operations performed. broken down by type of operations. Looking at this information helps understand why the "single-step laplacian" performs more efficiently than the routine that simply accumulates the gradient at each time step. The first one has a balanced number of additions and multiplications, whereas the second one does more multiplications than additions, leaving one of the pipes empty most of the time. Hence, the first routine achieves 280 Mflops, and the second one only 186 Mflops.

This helps understand the code, and of course this is the key to better performance.

4.3 Multiprocessor performance

The story is more complicated here, because of the way autotasking works. Designed to be used in a production environment, autotasking will only use a processor if it free. This makes it difficult to obtain speedup information on a loaded machine. The `atexpert` tool shows dedicated speedup curves, i.e. what would be obtained on a dedicated machine.

Figure 1 summarizes our results, by showing two curves for each problem size: the one labeled "Amdahl" is what Amdahl's law predicts is the inherent speedup in the program. The one labeled "Dedicated" is what we should obtain by running on a dedicated machine.

Amdahl's law [1], [6] helps us understand the performance we can expect form the code. It states that if only a fraction f of the operations in a program can be carried out in parallel, the speedup on p processors is:

$$S = \frac{p}{f + p(1-f)}.$$

This is bounded by $1/1-f$, regardless of the number of processors.

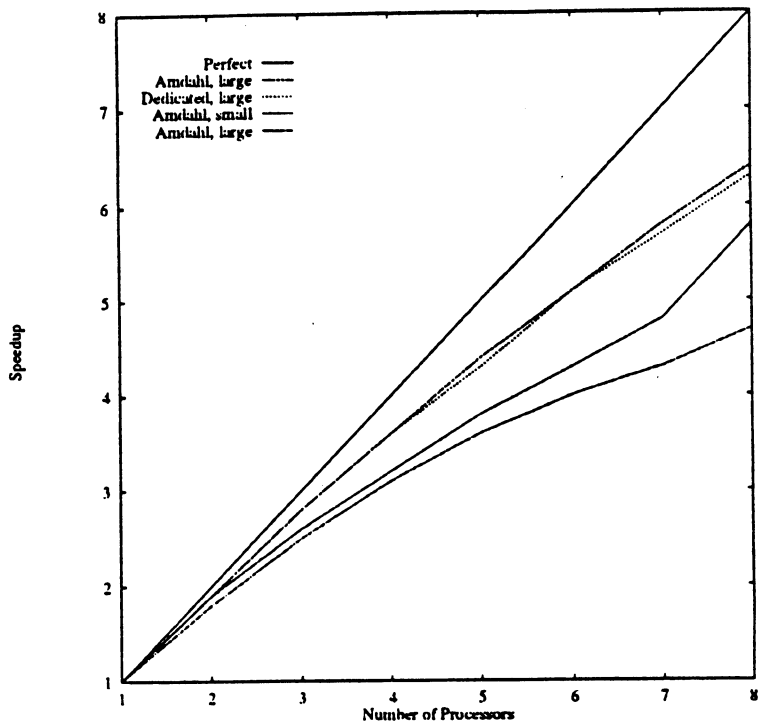

FIG. 1. *Parallel speedups predicted by* **atexpert**

Using this relationship, **atexpert** tells us how much parallelism is in a code. In our case, the small grid is 94.6 % parallel, and the large grid is 96.4 % parallel. This illustrates the main difficulty in obtaining good parallel performance, as opposed to vector performance: in vector mode, one could concentrate on the main modules, neglect small inefficiencies, and still obtain very good performance. This is not true in parallel. Amdahl's law tells us that any serial part in the code will be felt. Actually, Amdahl's law applies to vectorization as well, with the number of processors replaced by the ratio of vector to scalar speeds. This is much larger, so the speedup is still good, even for moderately vectorized codes.

We have also ran the code on a dedicated 4 processor Y-MP. The results of running the adjoint map on the small grid example are shown on table 1. The results are not as good

Nb CPUS	1	2	3	4
CPU time	45.4	49.3	53.0	57.8
Elapsed time	55.5	33.9	25.6	22.5
Mflops	209	342	453	516
Speedup	1	1.64	2.18	2.48

TABLE 1

Performance for small problem on a dedicated computer

as predicted by **atexpert**.

We also ran the "large grid" example, on 1 and 4 processors, and show the results on table 2. We are now closer to what `atexpert` predicts (3.3 compared to 3.6).

Nb CPUS	1	4
CPU time	682	772
Elapsed time	700	214
Mflops	262	857
Speedup	1	3.27

TABLE 2

Performance for large problem on a dedicated computer

4.4 Discussion

The last example is interesting, as it illustrates the widely debated issue of scalability. The concept of scaled speed-up has been introduced in [8] to illustrate the fact that "with larger computers we want to solve larger problems". Thus, Amdahl's law doesn't apply, because when going to larger problem, the sequential bottleneck grows more slowly than the parallel (useful) part of the code.

Our two grids example shows the validity of this argument: the parallel fraction of the code did increase when going from the "small" to the "large" problem; but it also shows an often overlooked consequence of this fact. Computation time increases faster than problem size. As we explained in section 4.1, it practice, the number of time steps, and the number of sources and receivers will increase with the grid size. This results in an increase in problem size by a factor of 16 if we double the grid size.

Our small problem took roughly a minute to solve. The "large" one took almost 4 minutes of wall-clock time on 4 processors. Remember that we want to solve the inverse problem, that is a series of wave equation solutions. So, if computing a function value took 10 minutes for the small problem, this translates to 40 minutes for the larger one, for just one function evaluation.

Yes, problem size increases with computing power, but so does computation time.

5 Conclusion

We have presented what we believe is an efficient implementation of an inverse problem solver on a parallel vector machine. We have shown actual performance measurements, which lead us to believe that the code would perform at 4 to 5 Gflops on a C90. But the limits of this approach were also shown. This clearly does not scale to a much larger number of processor, as there will be too little work for processor.

For the next generation of massively parallel processors, we pursue a different approach. The problem has an obvious level of coarse grain parallelism since the solution of the wave equations for each shot point are independent of one another. For inversion computations based on two-dimensional computations, this seems to be well suited to both distributed memory machines with relatively fast nodes such as the Intel Hypercube, and to networks

of workstations. For calculations based on 3D simulations, we also intend to exploit an intermediate level of parallelism through domain decomposition. We are currently implementing shot level parallelism by using PVM, a parallel message library that runs on heterogeneous networks. We hope to present preliminary results at the conference.

Acknowledgments

The authors thank Frank Kampe and Victor Parr for sharing their expertise on getting the most out of a Cray computer. Part of this work was carried out while one of the authors (MK) attended the Workshop on Scientific Computing at Los Alamos National Laboratory.

References

[1] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing*, AFIPS Proc. of the SJCC, 31 (1967), pp. 483–485.

[2] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *Generating derivative codes from fortran programs*, Scientific Programming, 1 (1992), pp. 11–30.

[3] J. Blanch, J. O. A. Robertsson, and W. W. Symes, *Viscoelastic finite difference modelling*, Tech. Rep. TR93-04, Rice University, 1993.

[4] F. Clement and G. Chavent, *Waveform inversion through MBTT formulation*, Tech. Rep. 1839, INRIA, 1992. Submitted to Geophysics.

[5] M. Dablain, *The application of high-order differencing to the scalar wave equation*, Geophysics, 51 (1986), pp. 54–66.

[6] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving Linear systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1991.

[7] B. Engquist and A. Majda, *Absorbing boundary conditions for the numerical simulation of waves*, Math. Comp., 31 (1977), pp. 629–651.

[8] J. L. Gustafson, G. R. Montry, and R. E. Benner, *Development of parallel methods for a 1024-processor hypercube*, SIAM Journal on Scientific and Statistical Computing, 9 (1988).

[9] W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge MA, 1985.

[10] P. Kolb, F. Collino, and P. Lailly, *Prestack inversion of a 1d medium*, Proceedings of IEEE, 74 (1986), pp. 498–506.

[11] P. Lailly, *The seismic inverse problem as a sequence of before-stack migrations*, in Conference on Inverse Scattering: Theory and Applications, J. Bednar et al., eds., Philadelphia, 1983, SIAM, pp. 206–220.

[12] A. Levander, *Fourth order finite difference P-SV seismograms*, Geophysics, 53 (1988), pp. 1425–1434.

[13] J. Liu, *Gradpack: a symbolic system for automatic generation of numerical programs in parameter estimation*, in Proceedings of the 5th IFAC Symposium on Control of Distributed Parameter Systems, 1989.

[14] F. Santosa and W. W. Symes, *An Analysis of Least-Squares Velocity Inversion*, vol. 4 of Geophysical Monographs, Soc. of Expl. Geophys., Tulsa, 1989.

[15] M. Sen and P. Stoffa, *Nonlinear one-dimensional seismic waveform inversion using simulated annealing*, Geophysics, 56 (1991), pp. 1624–1636.

[16] W. W. Symes, *Dso user's manual*, tech. rep., The Rice Inversion Project, Rice University, 1992.

[17] W. W. Symes and J. J. Carazzone, *Velocity inversion by differential semblance optimization*, Geophysics, 56 (1991), pp. 654–663.

[18] W. W. Symes and M. Kern, *Inversion of reflection seismograms by differential semblance analysis: Algorithm structure and synthetic examples*, Tech. Rep. TR92-03, Rice University, July 1992.

[19] A. Tarantola, *Inverse Problem Theory*, Elsevier, 1987.