# A Data Parallel Algorithm for Seismic Raytracing

Allen D. Malony, Stephanie McCumsey, Joseph Byrnes, Craig Rasmusen,
Soren Rasmusen, Erik Keever, and Doug Toomey

University of Oregon

**Abstract.** Dijkstra's single-source shortest path algorithm has been applied in seismic tomography to determine paths of minimum travel time from all locations in a 3D earth model to sensors used in seismic experiments. An iterative data parallel algorithm is formulated for seismic tomography based on the Bellman-Ford-Moore (BFM) algorithm. Performance is demonstrated for OpenMP and OpenCL.

**Keywords:** seismic tomography, shortest path, data parallel

## 1 Introduction

A common problem in scientific computing is finding the shortest path from a point to all other points in a given dataset. One of the most commonly used algorithms was first described by Dijkstra in his famous 1959 paper, "A Note on Two Problems in Connexion with Graphs" [5]. Given a graph of $n$ nodes each with cost $u$ and a starting node $s$, Dijkstra's "single-source shortest path" (SSSP) algorithm finds the path from $s$ to all other nodes with minimum cost. Unfortunately, Dijkstra's algorithm is difficult to implement in parallel. In this study, we describe a data parallel algorithm for finding shortest paths on a regularly-spaced grid of points that can compete with Dijkstra's in seismic raytracing.

Just as doctors use x-ray tomography to image the internal structure of the human body, scientists studying the Earth use seismic tomography to image its interior. Seismic waves propagate through the Earth at a velocity that varies with local temperature, composition, and the presence of magma. Understanding how these factors vary within the Earth is crucial to understanding the dynamic processes that shape the planet. The method works by measuring the arrival times of seismic waves from a source of seismic radiation, often an earthquake or an explosion, and comparing the observed arrival times to the arrival times predicted with a starting model. Perturbations to the starting model are then solved by minimizing the misfit between the predicted and measured arrivals times, generally with one of several variations on a least-squares approach [8].

In many cases, perturbations to the starting model are large enough to change the geometric ray paths of the first-arriving waves [13]. Ray paths for the new model must be calculated before new perturbations to the model are calculated. This process increases the computation time of the algorithm from minutes to
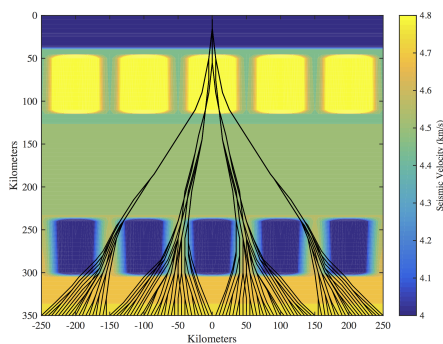
many hours. For example, Bezada et al. [3] implemented Dijkstra's algorithm for a tomographic study of the upper mantle beneath Spain and Morocco. While the improved results settled a long-standing controversy regarding the tectonic history of the Western Mediterranean Ocean, Dijkstra's algorithm had to be run for 322 starting points over 8 iterations. A significant amount of computing time was used for this purpose.

In general, reducing computation time is necessary to improve seismic tomography for several reasons. Seismic tomography is an ill-posed inverse problem, and several subjective parameters must be chosen to define the inverse problem. Since these parameters influence the final model, many inversions employing different parameter values must be analyzed before reliable inferences about structures inside the Earth can be made. The resolving power of the available data must also be explored, often through the inversion of many synthetic data sets. Finally, the computation time taken by Dijkstra's algorithm limits the scale of the problems that can be addressed. Efficiency of the shortest path problem must be improved for application to modern tomographic algorithms.

In this study, we present an alternative algorithm to that of Dijkstra [5] that is more amenable to parallelization. The algorithm's origin goes back to the famous Bellman-Ford-Moore (BFM) [2] algorithms which iterates to determine shortest paths. In contrast to Dijkstra's direct solution, ours then is an indirect computation that must iterate to reach a converged state. Nevertheless, the significant increase in parallelism enabled by the algorithm translates to overall reductions in execution times, as demonstrated for applications written with OpenMP and OpenCL. More importantly, the algorithm will hopefully expand the scale of the problems that can be addressed with seismic tomography and aid in the rapid development of high quality models of seismic velocity.

## 2   Stingray

Moser's formulation of the Dijkstra's shortest-path (DSP) method for seismic tomography [7] as implemented by Toomey et al. [13] is referred to in this paper as *Stingray-DSP*. Moser's approach represents the seismic velocity of a region of the Earth by a 3D grid of $n_x$ X $n_y$ X $n_z$ points. The objective is to find the shortest path from a starting point $s$ to all other points $p$ in the grid. This is done by initializing the travel time to $s$ at all points $p$ to $\infty$, and the travel time at point $s$ to zero. The travel time to all the points



**Fig. 1.** Visualization of ray paths in a 2D velocity model.

in a neighborhood near $s$ are then calculated along straight line paths. (The travel time between any 2 points $p$ and $q$ is a function of the velocity values at $p$ and $q$ and the distance between them.) The neighborhood of points used is

called the "forward star" for $s$, $FS(s)$. In this first step, the infinite travel times for all the points in $FS(s)$ are replaced by the travel time from $s$.

Once the travel time to all of the points within $FS(s)$ are found, the point $p$ with the minimum travel time from $s$ is identified. The algorithm moves to $p$ and then the travel times for $FS(p)$ are found. This is the standard DSP step. Again, only the minimum travel time for each point in $FS(p)$ is kept. Travel times along paths that are not minimum time are discarded. The algorithm continues with the next point $q$ with minimum travel time which has not yet had a forward star centered on it. The process is repeated until a forward star had been centered at every point in the velocity model.

The accuracy of the Moser method primarily depends on how finely the model is discretized and how many points are included in the forward star. The error varies with both these quantities because both control how the angles of the ray paths leaving the center of the forward star are discretized. In pratice, we generally use a forward star of radius 7, with certain points removed that do not effect the final accuracy of the solution [6]. In practice, a total of 818 points are included in the reduced forward star of size 7. Figure 1 shows an example of the fidelity of ray paths that can be obtained with Stingray-DSP with an 818-point forward star resolution.

## 3 Stingray Iterative Constraint Convergence Algorithm

The Stingray-DSP algorithm gives seismic modeling scientists high-quality raytracing results compared to other methods. However, there are inherent limitations on parallelism in the algorithm that prevent high-performance computing (HPC) implementations. At each step of the algorithm, it is necessary to find the leaf point on the unfolding spanning tree that has the minimum travel time. This point must be the next one to expand, effectively sequentializing the control path. It is possible to execute the Stingray-DSP algorithm on multiple starting points at the same time, thus taking advantage of multiple computing resources. Over a hundred starting points for a single velocity model are used routinely in our work. Replicated parallelism is beneficial to geological scientists for throughput purposes, but it does not produce a faster single starting point solution. Also, very large velocity models could exceed the memory bounds of a single processing node, requiring a splitting of the Stingray-DSP computation across nodes. In general, DSP algorithms face fundamental performance inefficiencies when executing in distributed memory systems.

It is possible to reformulate seismic raytracing as an *iterative constraint convergence* (ICC) problem, where the constraint is the minimization of a travel time metric. Let $V$ be the velocity field defined on a 3D grid of points and $T$ the travel times from each grid point to a starting point $s$. Assuming the final travel times are known for each model point $p$, $T(p)$, they must satisfy the constraint:

$$T(p) \; = \; min( \; T(q) \; + \; Delay(p, q)) \; , \; \forall \; q \; \in \; FS(p) \tag{1}$$

where $FS(p)$ is the "forward star" set of points of $p$ and $Delay(p, q)$ is the seismic time delay at $p$ and $q$. Based on this final constraint, an interative procedure to

update the travel times can be specified as follows:

$$T_0(p) \; = \; \infty \; \; \forall \; p \; \neq \; s \; \; , \; \; T_0(s) \; = \; 0 \tag{2}$$

$$T_{i+1}(p) \; = \; min( \; T_i(q) \; + \; Delay(p,q)) \; \; \forall \; q \; \in \; FS(p) \tag{3}$$

where $T_i(p)$ and $T_{i+1}(p)$ are the travel times of $p$ to $s$ at steps $i$ and $i+1$, respectively. The procedure continues until $T_{i+1}(p) \; = \; T_i(p) \; \; \forall \; p$. Note, convergence is guaranteed because the travel times at each point are monotonically decreasing.

The Stingray-ICC algorithm formulated in this manner is highly data-parallel, in that all points can be updated simultaneously. However, time to solution will depend on how long the algorithm takes to converge. There are three issues to consider. First, at step 0, all points have a time time of $\infty$, except for the starting point. Thus, much of the early computation will be irrelevant and wasted until valid travel times radiate from the starting point. Second, the propagation of valid travel times is directly correlated with the radius of the forward star. Geological scientists prefer forward stars with larger radii for better accuracy, which will radiate travel times faster and hopefully result in fewer steps for convergence, but will also increase the computational work at each step. Third, as the iterative algorithm gets closer to convergence, fewer travel times will be adjusted, meaning more points will be already at their final travel times and the computation will be redundant.
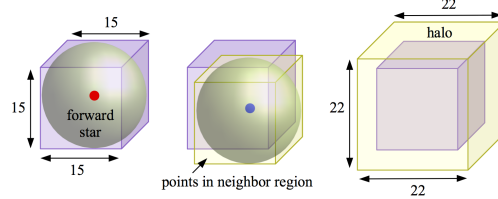
## 4  Parallelization Design Strategy

The highly-parallel nature of the Stingray-ICC algorithm provides an excellent opportunity for parallelization on both multicore CPUs and manycore coprocessors. Ideally, we would like to articulate a parallelization design model that could map to different execution targets. The idea is to decompose the model domain into rectangular regions that can be worked on in parallel at each iteration step. The regions will be defined such that they are non-overlapping, in order to eliminate dependencies between regions during the step-wise parallel computation. However, between steps, exchanges between neighbor regions will be required to update the travel times for points on the region boundaries. This is a standard domain decomposition approach with halos used for exchanging boundary data. Typically, applications using domain decomposition will apply stencils in updating values within a region. The forward star in Stingray-ICC is effectively a stencil. The problem is that the 818-point forward star is a very large stencil. This makes it more challenging.

In order to update the travel time of a single point, a region in the Stingray-ICC domain decomposition must be at least of size 15 $x$ 15 $x$ 15 in order to hold all of the points in the 818-point forward star. For a 150 $x$ 150 $x$ 150 velocity model, this partitioning would generate 1000 regions. Once the partitioning is done, the objective is to update every point in the region in parallel across all regions, at each step. However, to do so, we would need to access the forward star around every point in the region. That requires information to be exchanged

with our region neighbors to get those forward star points that are outside the region boundary. (Only, the point in the center of the region has it entire forward star set of points contained in the region.)

Deciding on halo size is essentially a tradeoff of extra buffer space versus when the exchanges with neighbors must be made. To accommodate all points in neighboring regions needed to update all points in a 15 x 15 x 15 region with a 818-point forward



**Fig. 2.** Illustration of 15 x 15 x 15 region, forward star, and 22 x 22 x 22 region with halo.

star, a *region + halo* dimension of 22 x 22 x 22 is necessary. Figure 2 illustrates the decomposition approach. It shows how the forward star defines the boundary overlap and the resulting halo surrounding the region.

The general parallelization design strategy above provides a basis for translation to target environments. In doing so, there are some additional strategies we can apply. For instance, in multi-core shared memory systems, where multithreading is used to process regions (1 thread per region), it is possible to
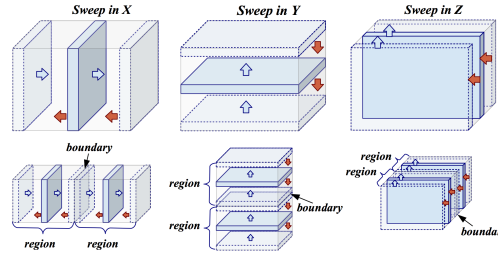


**Fig. 3.** Illustration of sweep methods for coordinated scheduling in and between regions.

avoid the allocation of halos altogether by scheduling which region points are updated when in a cooperative manner with neighbor regions. The basic idea is illustrated in Figure 3. Inspired by alternating direction implicit methods [1], the top row shows how points in a region could be processed in sweeps across the $X$ (left), $Y$ (middle), and $Z$ (right) directions. (Reverse sweeps are also shown.) By coordinating neighbor regions in synchronous sweeps, forward star points in neighbor regions can be accessed directly without memory races. This is shown in the bottom row for two neighbor regions in the $X, Y$, and $Z$ orientations. The strategy above could also have benefit in translation to manycore coprocessors, but more specialization will likely be required, especially for GPU accelerators.

A strategy to improve convergence is made possible by a slight addition in the Stingray-ICC algorithm. At every step, the algorithm updates the travel time of a point $p$ by checking the travel times and delays of the points in its forward star. In doing so, the following condition might occur:

$$T_i(q) \; < \; T_i(p) \; + \; Delay(p,q) \qquad (4)$$

This means that we have discovered a better travel time for $q$. The strategy then is to update $q$'s travel time opportunistically:

$$T_i^{'}(q) \; = \; T_i(p) \; + \; Delay(p,q)) \;\; if \; T_i(q) \; < \; T_i(p) \; + \; Delay(p,q) \qquad (5)$$

The notation $T_i^{'}(q)$ is used to indicate that the update occurs in step $i$. The intuition is that any travel time updates carry new information, potentially improving convergence rate. However, care must be taken with this strategy to ensure that new memory race conditions are not introduced. Combining it with the "sweeping" strategy above will help.

## 5　Implementation Approach

Our objective was to compare the Stingray-DSP implementation of Moser's method with different implementations of the Stingray-ICC algorithm. The Fortran Stingray-DSP code runs sequentially for a velocity model and single starting point. Travel times for multiple starting points can be solved by replicating the Stingray-DSP execution across computing threads.

　　The Stingray-ICC algorithm was implemented for both a CPU and GPU. The CPU code was written in C with OpenMP for parallelization. The *Stingray-ICC-multistart* version will execute the algorithm sequentially, but for multiple starting points. This provides a close approximation to how the Stingray-DSP program is used in practice. The *Stingray-ICC-parsingle* parallelizes the algorithm for a single starting point. The *Stingray-ICC-gpu* program was adapted from the original Fortran source using CoArray Fortran extensions (CAFe) to communicate with and run OpenCL kernels on the GPU. CAFe allows the programmer to explicitly allocate memory on the GPU, transfer memory between the CPU and the GPU, and execute OpenCL kernels using coarray Fortran [10, 12] syntax. CAFe is implemented as an embedded Domain Specific Language (DSL) and CAFe source is transformed automatically to standard Fortran [9], with wrappers [11] implementing the OpenCL C library interfaces. The OpenCL kernels implementing the *Stingray-ICC-gpu* algorithm were coded by hand.

## 6　Experimental Results

To evaluate the performance and scaling behavior of the Stingray-DSP and Stingray-ICC codes, we ran a series of experiments on different velocity models and sizes. These are described in Table 1. The *v100*, *v150*, *v200*, and *v300* models are synthetically generated. The *v241* model

| Model | X Dim | X Dim | X Dim |
|-------|-------|-------|-------|
| v100  | 100   | 100   | 100   |
| v150  | 150   | 150   | 150   |
| v200  | 200   | 200   | 200   |
| v300  | 300   | 300   | 300   |
| v241  | 241   | 241   | 51    |

**Table 1.** Velocity model descriptions.

is taken from a real-world example. Each model is run with 12 starting points. This is done in Stingray-DSP by replicating the code as a separate process on each core of the CPU server. This is done in the Stingray-ICC-multistart code with OpenMP. An additional set of experiments using the *v241* model and a single starting point were conducted with the Stingray-ICC-parsingle for 1, 2, 4, 8, and 12 threads.

　　The shared memory machine used for our study was a HP ProLiant SL390 G7 server with two Intel X5650 2.66 GHz 6-core CPUs (12 cores total) and 72
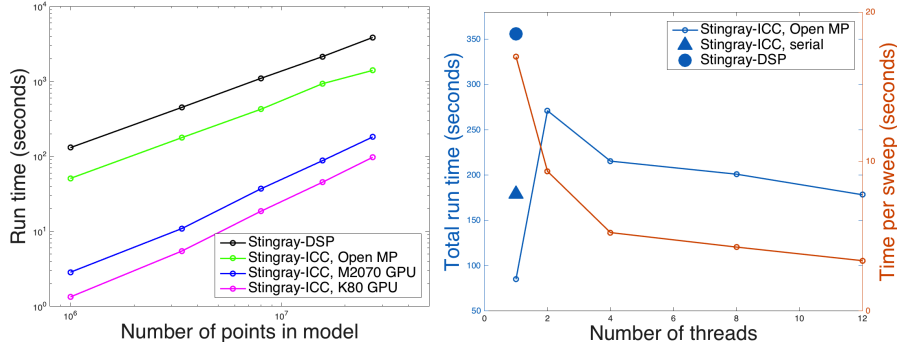
**Fig. 4.** Performance with synthetic and real velocity models.

GB DDR3 memory. Two GPUs were used: a NVIDIA M2070 (448 CUDA cores, 6GB) and NVIDIA K80 (2496 CUDA cores, 12GB).

Figure 4 (left) shows how the performance scales for the synthetic models and different codes. The Stingray-ICC versions perform significantly better than Stingray-DSP. Both Stingray-DSP and Stingray-ICC-multistart solve for 12 starting points, where each is run sequentially on 1 of 12 cores. Thus, these times reflect how long a serial execution for 1 starting point would take. In contrast, the Stingray-ICC-gpu results also solve for 12 starting points, but one after the other. We plot the average execution time for a single starting point for each GPU. Note, the average number of steps to reach convergence are constant at 6 steps for Stingray-ICC-multistart, but increase from 21 (v100) to 61 (v300) for Stingray-ICC-gpu.

The story gets more intriguing moving to the *v241* model experiments. Figure 4 (right) shows results from running Stingray-DSP and Stingray-ICC-multistart on 12 starting points. Again, Stingray-ICC-multistart is faster and it takes 7 steps to reach convergence for all 12 starting points. Figure 4 (right) also plots Stingray-ICC-parsingle results for 1, 2, 4, 8, and 12 threads, run with a single starting point. In this case, only 5 steps are needed to converge for 1 thread. However, the convergence steps increase from 2 to 12 threads (29 to 53 steps), though the time per sweep improves from 17.02 (1 thread) to 3.37 (12 threads). The increase in convergence steps nullifies the parallel performance gains (12 threads take 178.4 seconds).

## 7  Related Work

Methods for parallelizing Dijkstra's SSSP have been developed and recent work targets GPU implementations [4]. However, these have not been used the field of seismic tomography to solve the problems we consider here. Recasting the DSP approach to seismic raytracing as an iterative constraint convergence algorithm for parallelization purposes is similar to what is being done in calculating accumulated cost surfaces (ACS) [14] in spatial modeling. The BFM algorithm is the fundamental basis for both, except ACS applications are typically in 2D, such

as in spatial analysis of raster images to determine route travel times. Speedup on ACS problems has been demonstrated with the BFM-inspired data parallel algorithm when targeting GPU.

## 8 Conclusion

We have described and demonstrated an alternative algorithm for seismic ray-tracing by reformulating the problem as an iterative constraint convergence algorithm. The Stingray-ICC approach is more amenable to parallelization and hence significantly reduces the computation time needed to calculate high quality seismic velocity models. We have demonstrated the application of the algorithm with OpenMP and OpenCL for GPUs. The use of this algorithm in the future will aid seismologists in enhancing our understanding the internal structure and dynamic behavior of our ever mysterious planet.

## References

1. Alternating Direction Implicit. *https://en.wikipedia.org/wiki/Alternating_direction_implicit_method.*
2. Bellman-Ford-Moore. *https://en.wikipedia.org/wiki/BellmanFord_algorithm.*
3. M. Bezada, E. Humphreys, D. Toomey, M. Harnafi, J. Dvila, and Gallart J. Evidence for slab rollback in westernmost Mediterranean from improved upper mantle imaging. *Earth and Planetary Science Letters*, 368:51–60, 2013.
4. A. Davidson, S. Baxter, M. Garland, and J. Owens. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE, May 2014.
5. E. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1:269–271, 1959.
6. L. Klimes and M. Kvasnicka. 3-D network ray tracing. *Geophysical Journal International*, 116(3):726–738, 1994.
7. T. Moser. Shortest path calculation of seismic rays. *Geophysics*, 56(1):59–67, 1991.
8. G. Nolet. *A breviary of Seismic Tomography: Imaging the Interior of the Earth and Sun.* Cambridge University Press, New York, 2008.
9. C. Rasmussen, M. Sottile, S. Rasmussen, D. Nagle, and W. Dumars. Cafe: Coarray fortran extensions for heterogeneous computing. *High-Level Parallel Programming Models and Supportive Environments, 21st International Workshop, HIPS 2016 Chicago, IL, USA, May 23, 2016, Proceedings*, 2016.
10. J. Reid. The new features of fortran 2008. *SIGPLAN Fortran Forum*, 27(2):8–21, August 2008.
11. M. Sottile, C. Rasmussen, W. Weseloh, R. Robey, D. Quinlan, and J. Overbey. Foropencl: Transformations exploiting array syntax in fortran for accelerator programming. *Int. J. Comput. Sci. Eng.*, 8(1):47–57, February 2013.
12. The Fortran Committee. TS 18508 Additional parallel features in Fortran. ISO/IEC JTC1/SC22/WG5 N2007, March 2014.
13. D. Toomey, Solomon S., and Purdy G. Tomographic imaging of the shallow crustal structure of the east pacific rise at 9 30'n. *J. Geophys. Res.*, 99:24–24, 1994.
14. G. Trunfio and G. Sirakoulis. Computing Multiple Accumulated Cost Surfaces with Graphics Processing Units. In *International Conference on Parallel, Distributed, and Network-based Processing (PDP)*. Euromicro, 2016.