Computational Geosciences (2018) 22:775–787
https://doi.org/10.1007/s10596-018-9725-9

ORIGINAL PAPER

Cache-efficient parallel eikonal solver for multicore CPUs

Alexandr A. Nikitin[1,2] · Alexandr S. Serdyukov[1,2] · Anton A. Duchkov[1,2]

Received: 15 February 2017 / Accepted: 19 January 2018 / Published online: 29 January 2018

Abstract
Numerical solution of the eikonal equation is frequently used to compute first-arrival travel times for a given velocity model in seismic applications. Computations for large three-dimensional models become expensive requiring the use of efficient parallel solvers. We present new parallel implementations of the fast sweeping and locking sweeping methods optimized for shared memory systems such as multicore CPUs; we call them block fast sweeping method (BFSM) and block locking sweeping method (BLSM). Proposed methods are based on the domain decomposition approach with a special attention paid to high efficiency of the cache utilization and task execution synchronization. Performance tests on realistic models show high parallel efficiency of 85–95% on modern multicore CPUs and require the same number of iterations to converge as do the serial sweeping methods. We also highlight the importance of properly selecting the stopping criterion in the iterative sweeping methods aiming for a balance between computational time and accuracy of the result required by an application. In particular, we show that in seismic applications one can reach reasonable accuracy of computed travel times while dramatically reducing the number of iterations compared to the case of using the full convergence stopping criterion.

Keywords Eikonal equation · Fast sweeping method · Parallel algorithm · Shared memory · Seismic

Mathematics Subject Classification (2010) 35Q86 · 86A15 · 35L60 · 65Y05 · 65Y10

1 Introduction

Computation of travel times of seismic waves is widely used in seismic exploration and earthquake seismology. It forms the basis of seismic data processing and inversion [2] aiming at reconstruction of the distribution of seismic velocities in the subsurface.

Seismic travel times are computed by solving the eikonal equation; the most popular method for that is the ray tracing [5]. For complicated heterogeneous velocity models, it is increasingly difficult to use ray tracing for computing travel times. In this case, it is better to use the so-called eikonal solvers: to solve the eikonal equation numerically using the viscosity solution [8] which accounts only for the fastest arrivals. The necessary components of any eikonal solver are local approximation of the PDE and global algorithm of time field numerical propagation, that assures causality and minimality of obtained travel times. One of the first methods of finite difference solution of the eikonal equation was proposed in [29, 30]. However, this method does not always guarantee computation of correct first-arrival travel times depending on the velocity model complexity.

Note that the eikonal equation is a particular case of static Hamilton-Jacobi equations. More universal numerical techniques were developed for solving this class of equations [18, 24]. Such equations also appear in seismic processing: seismic data continuation with offset [14], velocity continuation of seismic data and images [12, 15], and data continuation in depth [13, 25].

Here, we mention some of the numerical eikonal solver algorithms in more details with a special attention paid to their parallel implementations. One of the most frequently used eikonal solver algorithms is the fast marching method (FMM) [22, 23, 28]. It is closely related to Dijkstra's method for finding the shortest path on a graph [11]. FMM

This research was partially supported by RFBR, Project No. 15-05-06752A, and Russian President Grant No. MK-7778.2016.5.

✉ Alexandr A. Nikitin
NikitinAA@ipgg.sbras.ru

[1] Trofimuk Institute of Petroleum Geology and Geophysics SB RAS (IPGG SB RAS), Pr. Ak. Koptuga 3, Novosibirsk, Russia, 630090

[2] Novosibirsk State University (NSU), St. Pirogova 2, Novosibirsk, Russia, 630090

is unconditionally stable—it always finds the viscosity solution by using upwind finite difference stencil and dynamically determining the order in which to process grid points according to the causality property of the eikonal equation. FMM does this by starting from the initial front position (the source point) and marching the front outwards one grid point at a time by using the heapsort algorithm to find the correct grid point to update. The complexity of this method is $O(N \log N)$ where N is the total number of grid points and $\log N$ comes from the heapsort algorithm. Due to the strict causal ordering of computations in grid points, FMM is an inherently serial method which complicates its efficient parallel implementation. A recent parallel version of the FMM [3] is based on domain decomposition of the grid, where serial FMM is used to compute solutions inside subdomains assigned to different execution threads. Causality between subdomains is enforced by re-running serial FMM whenever the boundary values change.

Another popular method is the fast sweeping method (FSM) [31]. FSM uses upwind finite difference stencil and Gauss-Seidel iterations with alternating sweeping orderings to converge to the viscosity solution. Modification of the FSM, called locking sweeping method (LSM) [1], reduces computational time of FSM by eliminating unnecessary computations. Theoretically, it takes only $O(N)$ operations to compute travel times using FSM for N grid points. However, the number of iterations required to converge depends upon the complexity of the velocity model and generally cannot be estimated beforehand. That is why FMM can be faster than FSM in some situations. In fact, it is hard to compare these approaches, since there is a variety of FMM and FSM modifications and the efficiency is also very sensitive to program implementation (see [4] for one of the recent studies on the matter). There have been several parallel implementations proposed for the fast sweeping method [9, 10, 32].

More recent methods of the eikonal equation solution include fast iterative method [17], heap-cell method [6], and its parallel implementation [7]. Fast iterative method is well designed for parallel implementation on SIMD architectures, such as GPUs. It uses an unsorted list of active points where the solution is updated in parallel. In its block-based version, groups of points (blocks) are maintained in this list and blocks are updated in parallel, with new blocks added to the list if any of their grid points have received updates. The heap-cell method combines the approaches of FMM and FSM on different scales by using the domain decomposition. Fast-marching type approach is used to order the subdomains for processing, and the locking sweeping method is used inside these subdomains.

In this paper, we introduce new parallel implementations of FSM and LSM methods called block fast sweeping method (BFSM) and block locking sweeping method (BLSM). These algorithms are specifically designed for running on shared memory architectures, and they are optimized to efficiently use CPU caches. Detailed performance testing showed high efficiency of parallel implementation. Source code for proposed methods is available under open source 3-Clause BSD license [19].

In Section 2, we will give an overview of the serial sweeping methods. In Sections 3 and 4, we will analyse in detail prior parallel implementations [10, 32] of the sweeping methods, highlighting performance difficulties that we attempt to solve in our methods. In Section 5, we introduce proposed block sweeping methods with different approaches to task execution synchronization, and in Section 6, we present and discuss performance tests results and selection of the stopping criterion parameter in sweeping methods.

2 Serial sweeping methods

Here, we will give a brief overview of serial sweeping methods for reader's convenience since it is directly relevant for analysis and discussion of prior and proposed parallel sweeping methods that will follow (for detailed information about these methods, see original papers [1, 31]).

Fast sweeping method is used for computing the viscosity solution $t(\mathbf{x}) \geq 0$ to the boundary value problem for the eikonal equation given in the following form:

$$|\nabla t(\mathbf{x})| = \frac{1}{v(\mathbf{x})}, \quad \mathbf{x} \in \Omega \subset R^n, \tag{1}$$

$$t(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \Gamma \subset \Omega, \tag{2}$$

where, in our case of seismic applications, $t(\mathbf{x})$ is the unknown function describing first-arrival travel time in point $\mathbf{x}$, $v(\mathbf{x})$ is a given velocity in $\mathbf{x}$, Ω is the computational domain where the solution will be computed, and Γ is a subset of Ω (e.g. a point source or an area around it) where boundary values $f(\mathbf{x})$ of the $t(\mathbf{x})$ solution are given.

FSM uses the following Godunov upwind finite difference scheme [20] to discretize the partial differential equation (1) on a regular grid. In three-dimensional case with $S_i \times S_j \times S_k$ grid dimensions, it can be written as follows:

$$\begin{aligned}&[(t_{i,j,k} - t_{i\,\min})^+]^2 + [(t_{i,j,k} - t_{j\,\min})^+]^2 \\ &+[(t_{i,j,k} - t_{k\,\min})^+]^2 = \tfrac{h^2}{v^2},\end{aligned} \tag{3}$$

where

$$(\eta)^+ = \begin{cases} \eta, & \eta > 0, \\ 0, & \eta \leq 0, \end{cases} \tag{4}$$

and $t_{i\,\min}$, $t_{j\,\min}$, and $t_{k\,\min}$ are the minimum values of left and right neighbours of $t_{i,j,k}$ in the computational stencil along i, j, and k dimension respectively: $t_{i\,\min} = \min(t_{i-1,j,k}, t_{i+1,j,k})$, etc. This upwind nature of the discretization enforces the causality of the eikonal equation, i.e. the solution is only determined by neighbouring values that are smaller. One-sided difference is used on the grid boundaries to ensure flow of information from inside the computational domain where Γ is contained to the outside.

The first step of the fast sweeping method is the initialization. Exact or interpolated values must be assigned at grid points in or near Γ to enforce boundary condition (2). These values are fixed during the rest of calculations. In all other grid points, sufficiently large positive values must be assigned, i.e. larger than the maximum possible value in the computational domain for the given velocity model.

Next, Gauss-Seidel iterations are performed with alternating sweeping orderings as a way to enforce causality in the entire computational domain. There are 2^N sweep directions in N-dimensional space, so, for example, in 2D case, these are (not necessarily in that particular order) as follows:

$$\begin{aligned}
1: &\quad i=\{1,\dots,S_i\},\ j=\{1,\dots,S_j\},\\
2: &\quad i=\{S_i,\dots,1\},\ j=\{1,\dots,S_j\},\\
3: &\quad i=\{S_i,\dots,1\},\ j=\{S_j,\dots,1\},\\
4: &\quad i=\{1,\dots,S_i\},\ j=\{S_j,\dots,1\}.
\end{aligned} \tag{5}$$

During each sweep, we compute the solution $t_{i,j,k}$ to the quadratic equation (3) at each grid point (i, j, k). To do that, we must account for the case when some of the summands on the left side of Eq. 3 are zero, so we order $t_{i\,\min}$, $t_{j\,\min}$, and $t_{k\,\min}$ in increasing order as $t_{a_1\,\min} \le t_{a_2\,\min} \le t_{a_3\,\min} \le t_{a_4\,\min} = \infty$. We look for integer $1 \le p \le 3$ and the corresponding unique solution $\overline{x}$ that satisfies

$$\begin{aligned}
&\sum_{i=1}^{p}(x - t_{a_i\,\min})^2 = \frac{h^2}{v^2},\\
&t_{a_p\,\min} < x \le t_{a_{p+1}\,\min}.
\end{aligned} \tag{6}$$

Then, we select the smaller value between the old value $t^{\text{old}}_{i,j,k}$ and the computed value $\overline{x}$ as the new value at that grid point:

$$t^{\text{new}}_{i,j,k} = \min(t^{\text{old}}_{i,j,k}, \overline{x}). \tag{7}$$

In summary, during each sweep, we update the solution in accordance with the causality along continuous parts of its characteristics with the direction of information propagation corresponding to that of the sweep. Due to the updating rule (7), the value at each grid point is non-increasing; once it reaches its minimum value, it is the correct one and will not be changed. The method converges when all grid points are assigned their correct values and cannot be changed anymore. The number of sweeps needed for convergence largely depends upon how often characteristics change their direction from one octant to another in three-dimensional case, or from one quadrant to another in two-dimensions. So for homogeneous N-dimensional velocity model, the number of sweeps needed for full convergence will be 2^N, since characteristic curves are straight lines. This is illustrated in Fig. 1 for the two-dimensional case with Eq. 5 ordering of sweeps. In the case of more complex models, the fast sweeping method can require higher number of iterations until full convergence. Note, however, that a more effective stopping criterion can be selected that ensures the information from Γ has reached all grid points:

$$\| t^{\text{new}}_{i,j,k} - t^{\text{old}}_{i,j,k} \|_\infty \le \epsilon. \tag{8}$$

The idea behind the locking sweeping method is to eliminate the unnecessary computation of the solution to Eq. 6 when we know in advance that the value $t^{\text{new}}_{i,j,k}$ in updating rule (7) will not be changed, i.e. when the neighbouring values in the computational stencil were not changed in the last iteration. To do that, LSM uses "locks" (boolean flags) assigned to each grid point. In the initialization step, all grid points are locked, except for the stencil neighbours of any fixed grid points with initial values (sources). While sweeping, the lock of the current grid point is checked. If the point is locked, then the solution is not computed and we move on to the next grid point. If it is unlocked, we compute the solution as usual, and if it is less than the old value, we update the time as per (7) and unlock all neighbours with times greater than the updated time at the current grid point. This modification can significantly reduce computational time compared to the original fast sweeping method for a lot of velocity models; however, it imposes slightly increased costs in terms of required memory to store locks.

It is important to note here that the way $\overline{x}$ value in Eq. 6 is calculated (using either one, two, or three neighbouring grid points) and whether we update the current value using updating rule (7) or not can lead to potential load imbalance between computations of solution at different grid points depending on the velocity model. In locking sweeping method, load imbalance due to some of the points being locked is of even greater concern. We have taken this into

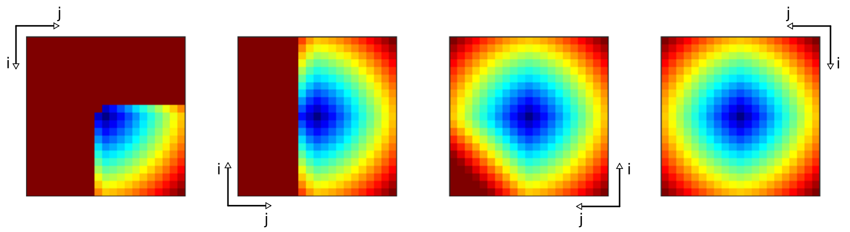


Fig. 1 FSM 2D example for homogeneous velocity model. Only four iterations are required for full convergence. Solution is shown after each iteration. Arrows denote sweeping direction for each iteration

consideration during development of our proposed parallel sweeping methods, discussed in Section 5.

3 Prior parallel sweeping methods

There have been several parallelization approaches proposed for the fast sweeping method. In [32], two different parallel implementations were introduced. The first one is based on the idea that sweeps with different directions can be performed simultaneously. After each 2^N sweeps, there are 2^N solutions, where N is space dimensionality. The minimum value of these solutions is taken at each grid point as the initial value for the next iteration. Therefore, it is guaranteed that the solution will get better after each iteration and eventually converge when using first-order finite difference scheme employed in the FSM. Note, however, that this implementation does not scale above 2^N execution threads and also requires more memory to store 2^N numerical grids.

The second approach proposed in [32] is the domain decomposition. Computations in each subdomain can be performed using fast sweeping method either in serial manner as in the original FSM or in parallel as in the above-mentioned approach. Overlapping grid points between neighbouring subdomains are updated after each 2^N sweeps to minimize the number of communications required. Minimum value of each group of overlapping points is selected as the updated solution for the group. As with the previous implementation, this one will also converge in a finite number of iterations. However, as testing in [32] shows, both approaches can require more sweeps to converge than the original serial FSM. This can complicate efficient application of these implementations in cases where velocity model is complex and therefore requires many sweeps to converge.

A more recent approach to FSM parallelization was introduced in [10], which we will refer to as DFSM for short. Unlike in the previously discussed approaches, in DFSM, grid point updates in each sweep are parallelized due to the fact that there exist sets of independent points where grid point values can be computed simultaneously. To illustrate that fact, let us perform dependency analysis of the original FSM for the case of three-dimensional space.

To simplify the exposition, we will fix sweep direction to $i = \{1, \ldots, S_i\}$, $j = \{1, \ldots, S_j\}$, $k = \{1, \ldots, S_k\}$, since other sweep directions can be given by inverting some or all of the indices, i.e. rotating the axes. As was noted in Section 2, when using discretization scheme (3), each grid point update procedure updates value of current (i, j, k) point and reads current values of neighbouring points in the computational stencil: $(i-1, j, k)$, $(i+1, j, k)$, $(i, j-1, k)$, $(i, j+1, k)$, $(i, j, k-1)$, and $(i, j, k+1)$. At first glance FSM and LSM are strictly serial methods, since each step of the sweeping cycle depends upon the previous steps. However, since only the neighbouring points from the computational stencil are required, update of the (i, j, k) point can actually be performed immediately after all points with lower indices were updated, i.e. points from the set:

$$D_{(i,j,k)} = \{(p, q, r) \mid p \leq i, q \leq j, r \leq k\} \setminus (i, j, k). \quad (9)$$

Therefore, sweeping process can in fact be parallelized, since many of the points can be computed independently from each other.

In DFSM, grid points along the so-called levels, that are diagonals in two-dimensional case and diagonal planar slices in three-dimensional case (Fig. 2), are updated simultaneously, since there are no dependencies between any of them. Global synchronization of execution threads is used between computations of different levels. Therefore, since the information flow of this algorithm is essentially the same as in serial FSM, the same number of sweeps is needed for it to converge. And it does not require additional memory.

In [9], authors of DFSM proposed a new hybrid massively parallel fast sweeping method (HMP-FSM) for distributed memory architectures. It uses coarse-grained domain decomposition to partition the domain among available compute nodes, while DFSM is used as a fine-grained shared memory method to compute the solution within each subdomain.

In this paper, we will focus on parallel implementation of the sweeping methods for shared memory architectures, specifically, on multicore CPUs only.

4 DFSM performance analysis

We have tested the performance of our implementations of the original serial FSM and parallel DFSM, written using C language with OpenMP used for parallelization on shared memory architectures. We emphasise that we have tested our own implementation of DFSM; therefore,

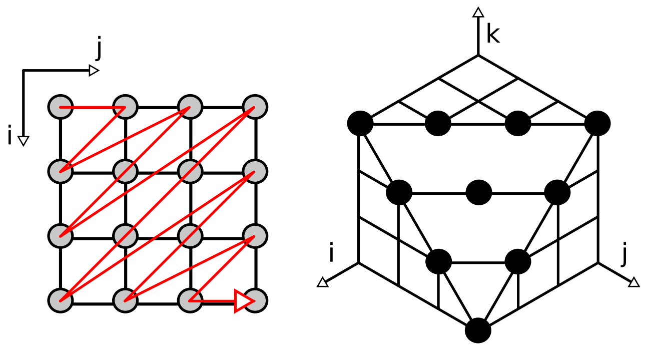


Fig. 2 Examples of DFSM levels in 2D and 3D for given sweeping directions

Table 1 Performance test results for FSM and DFSM (1 and 4 threads) on homogeneous velocity model, 601 × 601 × 601 grid points, nine iterations

Performance metric	FSM	DFSM(1)	DFSM(4)	LSM	DLSM(1)	DLSM(4)
Time (s)	50.15	241.32	71.39	23.94	103.94	40.79
Cycles	1.58E+11	5.21E+11	6.13E+11	8.21E+10	2.31E+11	3.30E+11
Instructions	2.73E+11	3.51E+11	3.91E+11	1.38E+11	1.68E+11	2.16E+11
IPC	1.73	0.67	0.64	1.68	0.73	0.65
L1-dcache-loads	8.74E+10	9.79E+10	1.09E+11	5.00E+10	5.20E+10	6.13E+10
L1-dcache-load-misses	1.22E+09	1.13E+10	9.18E+09	4.62E+08	6.77E+09	6.58E+09
L1 load miss rate (%)	1.39	11.58	8.45	0.92	13.02	10.73
L1-dcache-stores	2.81E+10	5.16E+10	5.56E+10	2.31E+10	2.19E+10	2.50E+10
L1-dcache-store-misses	1.60E+08	5.29E+08	5.21E+08	1.78E+08	1.06E+09	1.05E+09
L1 store miss rate (%)	0.57	1.03	0.94	0.77	4.85	4.19
L1-dcache-prefetch-misses	1.16E+09	2.66E+09	2.18E+09	4.16E+08	2.07E+09	2.24E+09
LLC-loads	1.31E+08	7.09E+09	5.80E+09	3.86E+07	4.64E+09	4.56E+09
LLC-stores	1.09E+08	1.12E+08	1.11E+08	1.12E+08	1.11E+08	1.22E+08
dTLB-loads	8.75E+10	9.78E+10	1.09E+11	5.01E+10	5.20E+10	6.14E+10
dTLB-load-misses	5.12E+04	4.02E+09	4.08E+09	3.74E+04	1.25E+09	9.72E+08
dTLB load miss rate (%)	0.00	4.11	3.76	0.00	2.41	1.58
dTLB-stores	2.81E+10	5.16E+10	5.57E+10	2.30E+10	2.20E+10	2.49E+10
dTLB-store-misses	1.44E+04	3.73E+04	4.44E+04	1.80E+04	4.19E+04	7.29E+04
dTLB store miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00

Test performed on Intel Core i7-2630QM quad core CPU

our performance testing results may differ from the results presented in [10] due to differences in implementations. As can be seen from the results of testing presented in Table 1, performance of DFSM scales well when compared to its single-threaded performance. However, DFSM on single thread is actually slower than serial FSM, which leads to low speed-up for parallel DFSM when compared with serial FSM. Since computational kernel and number of iterations of the sweeping algorithm are the same for both implementations, this can be explained by different memory access patterns in these methods. In FSM, memory is accessed sequentially during the whole sweep. In DFSM, memory is accessed non-sequentially by iterating across different levels, which can potentially lead to problems with efficiency of cache utilization.

To confirm this hypothesis, we have used Linux perf[1] program. Perf is a powerful tool for performance profiling that uses PMUs (performance monitoring units) in CPUs to collect data on hardware events, including the number of instructions executed, cache references and cache-misses, and branch mispredictions. As can be seen from the results, our DFSM implementation indeed suffers from poor CPU cache utilization, with high rate of cache-misses. Therefore, we conclude that FSM with employed first-order scheme is memory-bound, and we should optimize our implementation primarily with the efficient use of CPU cache in mind.

Let us perform more detailed analysis of the test results before moving on to optimization goals and strategies. First, we will give a brief overview to the reader of the memory hierarchy in modern computer architectures, focusing on CPU caches. Readers familiar with this topic can skip on to the next paragraph. Generally speaking, to avoid problems related to its efficient use, one must follow the principle of locality, also known as locality of reference [26]. There are two types of locality of reference that are discussed in computer architecture—spacial and temporal. Spacial locality means that elements that are close to each other in memory are more likely to be accessed, such as the case with sequential processing of elements in a one-dimensional array, accessing local variables on the call stack, reading program instructions from memory, etc. Temporal locality means that if the element was accessed recently, it will likely be accessed again soon, obvious example being again local variables such as cycle counters or temporal values in calculations, instructions from bodies of cycles, and elements in the array that are repeatedly accessed. In modern computers, the speed at which CPUs can process data is much faster than the speed at which

[1]See https://perf.wiki.kernel.org

data can be fetched from RAM, and to compensate for that difference, relatively small but fast caches are used between CPU and RAM as a temporary storage for data being processed. Modern multicore Intel/AMD CPUs commonly used in workstations and servers usually include three levels of caches: per-core L1 (commonly 32–64 KB), per-core/per-module L2 (commonly 256–512 KB), and shared L3 (several megabytes), also known as the last level cache, or LLC. Data is fetched to cache in cache lines, usually 64 B in size. Each cache line can be placed in a specific location in cache, depending on cache associativity. For example, in an eight-way associative cache, there are eight banks. Each bank works as a direct-mapped cache, i.e. any cache line from memory can be placed at a single specific location in the bank, determined by the cache line address in memory. Therefore, in eight-way associative cache, there are eight possible locations in the cache where a cache line can be placed that form what is called a cache set. When a line is found in a cache, it is called a cache-hit, and data is retrieved quickly. Otherwise, if it is not found, then it is called a cache-miss, and cache line has to be retrieved from slower levels of memory hierarchy. Since caches are considerably smaller than RAM, some cache lines may have to be evicted from cache to make room for the cache line required at the moment. Which line is evicted is determined by the replacement policy, such as the least recently used (LRU). To improve performance, CPUs can also use a form of speculative execution called cache prefetching, i.e. they load cache lines to cache before they are actually requested. Prefetching can help a lot when memory access pattern is easily identifiable, such as when sequential elements of the array are being accessed. Another type of cache is translation lookaside buffer, which is used to improve translation speed of virtual addresses to physical ones in architectures with paged or segmented memory. If the required address is found in the TLB, then matching physical address is retrieved quickly; otherwise, page table must be accessed in RAM in a process called page walk which takes considerably more time. Virtual-to-physical mapping of the requested address is then entered into TLB, evicting one of the other cached mappings. To summarize, memory hierarchy is built on and works well in the general case because of the principle of locality. Particularly, it is important to maintain spacial locality of reference to efficiently reuse data from a single cache line and to take advantage of hardware prefetching done by the CPU, and to maintain temporal locality to minimize the chance of cache lines getting evicted and reloaded more times than necessary.

Now, let us analyse performance metrics from Table 1. Testing was done on an Intel Core i7-2630QM quad core CPU based on the Sandy Bridge microarchitecture. Firstly, as can be seen from the L1-dcache-load-misses and L1-dcache-store-misses events, there are several times more cache-misses in DFSM than there are in serial FSM. Cache-miss rates, that are calculated as ratios of load-misses and store-misses to loads and stores respectively, are significantly higher in DFSM. This can be attributed to DFSM's memory access pattern having poor spacial as well as temporal locality. In one cache line, there are elements from multiple levels of grid points in DFSM due to the way that array elements are stored in memory. Therefore, cache lines can get evicted while processing grids sufficiently larger than cache size, since it is highly likely that while updating points from the current level, there will be much more cache lines accessed from the same cache set than there are banks.

Secondly, LLC-loads are an order of magnitude higher in DFSM which means that cache lines are frequently evicted and are reloaded later from slower memory. Latency of data access differs significantly between different levels of memory hierarchy—for Sandy Bridge CPUs, L1 data cache has a best case latency of 4 cycles, L2—12 cycles, and L3—26–31 cycles (see [16]), and RAM latency is usually in the order of at least 100 cycles and more, so small changes in cache-miss rates can have potentially huge performance difference, as can be seen from the results. Also notice higher number of L1-dcache-prefetch-misses in DFSM compared to FSM, which indicates that CPU is having more problems with efficiently prefetching data with DFSM's memory access pattern than in sequential access pattern in FSM.

Another potential problem for parallel implementation is the loss of temporal locality due to false sharing of cache lines between cores, i.e. lines that are written to in one core are simultaneously read in another. This means that due to cache coherency protocol that enforces correctness in cases where the same cache lines can reside in several coherent caches on different CPU cores, the next time that another core will try to read a modified cache line it will have to reload it completely.

Lastly, higher number of dTLB load-misses and store-misses and corresponding high dTLB miss rates indicate that TLB is as well not used efficiently, which can be explained by elements from the same level having poor spacial locality—too many memory pages are accessed during processing of one level, which leads to more frequent page walks. In all metrics, FSM performs much better due to having higher spacial and temporal locality of references as can be summarized by a higher instruction/cycle (IPC) metric for FSM. All of these points are also valid for LSM/DLSM comparison.

Problems with efficient use of CPU caches in DFSM can potentially reduce the efficiency of HMP-FSM [9] implementation for distributed memory since it uses DFSM at the shared memory level.

5 Proposed block sweeping methods

In order to solve problems with efficient use of memory hierarchy outlined in the previous section, we should ensure high locality of reference in a parallel implementation of the sweeping methods. As we saw earlier, FSM and LSM already utilize CPU cache quite efficiently due to the sequential nature of sweep orderings. So a good idea would be to maintain this sequential memory access pattern in each thread of the parallel sweeping algorithms, which is exactly what we do in our proposed block fast sweeping method (BFSM) and block locking sweeping method (BLSM) for numerical solution of the three-dimensional eikonal equation.

The key idea of the block sweeping methods is to perform decomposition of the computational grid into blocks of grid points as shown in Fig. 3. During each sweep of the algorithm, we distribute computations of different blocks among the computational threads. The solution inside each block is computed using standard sequential fast or locking sweeping method respectively, which allows us to maintain good locality of memory accesses. Data dependencies between computations of different blocks are the same as those for grid points (9): block (i, j, k) can be computed after blocks $(i + c_i, j, k)$, $(i, j + c_j, k)$, $(i, j, k + c_k)$, where $c_i, c_j, c_k \in \{-1, 1\}$ are determined by the current sweep direction. There are two important questions involved in the implementation of the block sweeping methods: the choice of block size and synchronization of computations of different blocks according to the data dependencies. We have developed several implementation approaches for these problems.

The first implementation approach, which we will call BFSMv1 and BLSMv1 respectively, is using the same idea as in DFSM, only on the block scale. We divide our set of blocks into levels, where blocks from the same level are distributed to threads and are processed simultaneously.

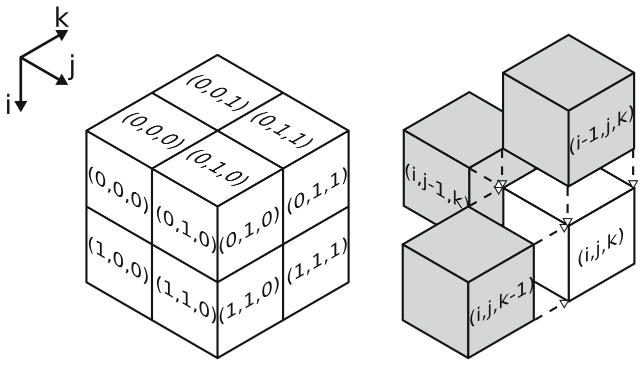


Fig. 3 Example $2 \times 2 \times 2$ decomposition of the grid in BFSM/BLSM, with block indices specified as (i, j, k). Dependencies of (i,j,k) block for the given sweep direction

Global synchronization of threads is used between different levels of blocks (see Fig. 4). This approach alleviates problems with efficient use of the memory hierarchy encountered in DFSM; however, it may not be the most efficient approach to synchronization when we consider the fact that the amount of computations in each grid point, and, therefore, block, may vary. As noted in Section 2, the first obvious source of imbalance is the way that the solution is computed for Eq. 6. Another is the updating rule (7), although it will have a less profound effect since the possible difference is only one memory store operation. When using LSM to compute solutions in blocks, the imbalance problem can become even more damaging to performance since it is possible that in some blocks, there will be almost no computations performed compared to other blocks due to the locking mechanism. All of these factors can potentially, depending on the velocity model and block assignments to threads, lead to situations when there is significantly less work available to some of the threads during processing of certain levels, which will negatively impact efficiency of the parallel algorithm compared to the serial method.

The second approach, BFSMv2 and BLSMv2, was developed primarily to see if any performance gains can be achieved by reducing the time it takes for all threads to begin execution due to data dependencies between blocks by reducing their size, possibly down to one or two cache lines, while still maintaining good spacial and temporal locality of memory accesses. We do this by statically assigning i-planar slices of blocks (i being the slowest changing grid dimension in regards to memory addresses) to different threads in round-robin fashion. One or two cache line block size lower limit and data alignment on cache line-sized boundaries are used to avoid false sharing of cache lines between L1D-caches of different CPU cores. Synchronization of block computations is done using semaphores similar to classic producer-consumer problem. This approach is illustrated in Fig. 5. Blocks computed by the tid thread, where tid is the thread number, or thread ID, are depended upon blocks computed by the $(tid - 1)$ thread. Each tid thread is assigned its own semaphore $S[tid]$. Once tid thread has completed computation of another one of

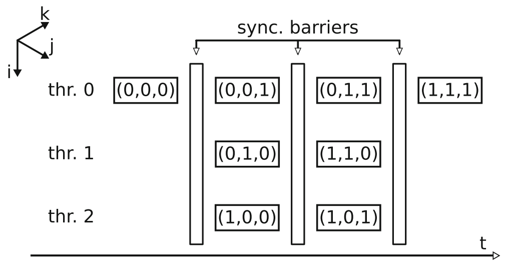
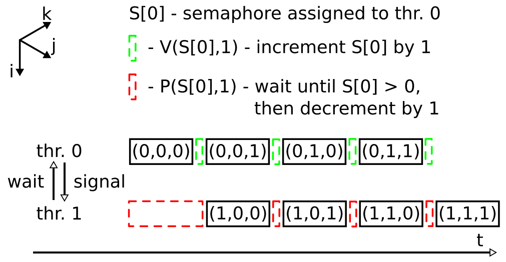


Fig. 4 BFSMv1 example for $2 \times 2 \times 2$ decomposition on three threads



k
j
i
S[0] - semaphore assigned to thr. 0
- V(S[0],1) - increment S[0] by 1
- P(S[0],1) - wait until S[0] > 0, then decrement by 1
thr. 0
(0,0,0)
(0,0,1)
(0,1,0)
(0,1,1)
wait
signal
thr. 1
(1,0,0)
(1,0,1)
(1,1,0)
(1,1,1)
t

Fig. 5 BFSMv2 example for $2 \times 2 \times 2$ decomposition on two threads

the blocks assigned to it, it increments $S[tid]$ by one. After processing each block, $(tid + 1)$ thread waits until $S[tid] > 0$, then decrements $S[tid]$ by one and begins processing its next block. Therefore, we avoid global synchronization of threads. Synchronization delays due to load imbalance in tid thread can only be caused by a delay in $(tid - 1)$ thread. This approach can be thought of as pipelining of the block computations and can be advantageous on systems with a high number of CPUs and CPU cores. Note, however, that such static assignment of block computations can still suffer from the same limitations as the previous implementation method in the case of highly unbalanced computational load between blocks as delays can still propagate. Higher load in tid thread can potentially cause all threads with higher thread ID to idle.

The third approach, BFSMv3 and BLSMv3, was developed specifically to avoid imbalance problems by assigning blocks to threads in a completely dynamic, dataflow fashion (Fig. 6). The key idea is to avoid global synchronization between threads by allowing blocks to be executed as soon as their data dependencies are satisfied. Each thread must pick up a new ready for execution block as soon as it completes any previous work. To implement this approach, we are using OpenMP 4.0 standard which introduced the ability to directly specify data dependencies between tasks using the `task depend` construct. That way, we have reduced synchronization to the bare minimum possibility in accordance with data dependencies in the sweeping methods. However, we also lose increased data locality of the previous strategy. Performance of this approach will depend largely upon the efficiency of dynamic task scheduling in the OpenMP runtime.

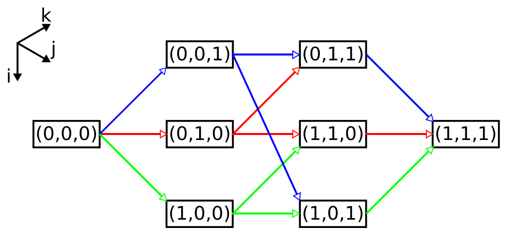


Fig. 6 Dataflow in BFSMv3 in the case of $2 \times 2 \times 2$ block decomposition

All of the three proposed methods of task synchronization preserve the dataflow of the serial sweeping methods; therefore, they will require the same number of iterations (sweeps) as the serial methods to converge.

6 Testing results

The following performance tests were done on Novosibirsk State University cluster[2] on one of HP BL2x220c G7 nodes with two six-core Intel Xeon X5670 CPUs, with a maximum of 12 execution threads, one thread-per-core.

Velocity models We have tested performance of our proposed block sweeping methods using three velocity models. The first one is a $601 \times 601 \times 601$ grid points homogeneous velocity model, with $V = 1$ in the whole computational domain. Point-source was placed in the centre of domain, i.e. in $(300, 300, 300)$ grid point, and initialized with $t = 0$ initial travel time. Since characteristics of this model's solution are straight lines, sweeping methods fully converge in nine iterations (eight sweeps with different directions and one more sweep when solution does not change at all). It is the same model that was used in FSM/LSM vs DFSM/DLSM performance analysis in Section 4.

To test performance of our methods on datasets more typical to seismic problems, we have used Overthrust and Salt velocity models (Fig. 7), produced by the Society of Exploration Geophysicists (SEG) and the European Association of Geoscientists and Engineers (EAGE). Salt model dimensions are $676 \times 676 \times 210$ grid points; source was placed in $(338, 338, 105)$ grid point. Overthrust model dimensions are $801 \times 801 \times 161$ grid points; source was placed in $(400, 400, 80)$ grid point.

Stopping criterion parameter selection Overthrust and Salt models are more complicated than the homogeneous model and require many more iterations to fully converge. However, as noted in [31], we do not need full convergence to obtain sufficiently accurate solution in practice and can use more efficient stopping criterion (8) with ϵ as $O(h)$, where h is a grid step, therefore, terminating sweeping methods early.

To select the best balance between solution accuracy and performance we logged maximum relative error

$$\mathrm{MRE} = \max_{i,j,k}(|(t^{(Q)}_{i,j,k} - t^{(q)}_{i,j,k})/t^{(Q)}_{i,j,k}|) \times 100, \qquad (10)$$

between fully converged solution $t^{(Q)}$, obtained on Q iteration, and current solution $t^{(q)}$ at each $q \le Q$ iteration, and the time it took to compute current solution using BLSMv3 method. The results are presented in Fig. 8. A dozen or so

[2]See http://nusc.nsu.ru

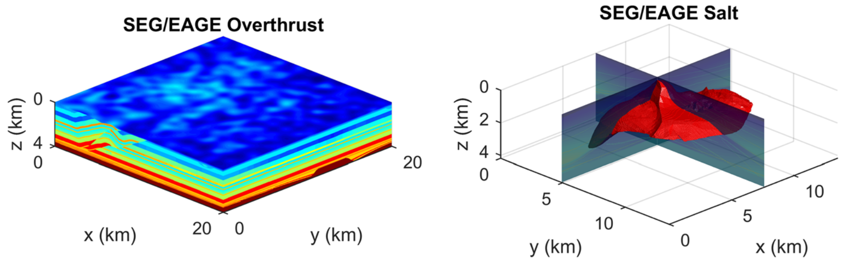


Fig. 7 SEG/EAGE Overthrust and Salt velocity models

iterations is enough for both models to obtain a solution with MRE close to one percent, and performing more than 50 iterations is impractical due to the fact that the MRE drops to less than $10^{-2}\%$ in both cases. In our subsequent performance tests, we use $\epsilon = h_{\min}/v_{\max}$ as the stopping criterion parameter, where $h_{\min}$ is the minimal grid step and $v_{\max}$ is the maximum velocity in the computational domain. With it, early terminated solutions for Overthrust and Salt models take 17 and 24 iterations to compute, with resulting MREs 0.59 and 0.18%, while fully converged solutions were obtained only after 346 and 339 iterations respectively.

All of the tested implementations (serial sweeping methods, DFSM/DLSM and all versions of block sweeping methods) required the same number of iterations to converge on all tested models when using the same stopping criterion parameter.

Block size selection We have also tested computational time versus chosen block size for each of our proposed methods. The results are presented in Table 2. All proposed block sweeping implementation approaches show similar performance on tested velocity models and hardware for practical purposes. BFSMv2 and BLSMv2 (with static assignment of i-planes of blocks) appears to be the slowest performing implementation approach; however, it is also the least sensitive to block dimensions selection and shows consistent performance on a wider range of block sizes. BFSMv1 and BLSMv1 (modified DFSM/DLSM) is the second best approach; however, more care should be taken when selecting block size. BFSMv3 and BLSMv3 with dynamic scheduling based on data dependencies between blocks is the best of all three approaches when if comes to computational time; however, it also has the smallest range of optimal block dimensions. For all approaches, selecting a too fine-grained block size leads to bad performance due to significant synchronization overhead, while selecting large block sizes leads to increased waiting times of threads for data dependencies. Small block sizes are extremely detrimental to BFSMv3/BLSMv3 performance also due to increased potential for reduction of data reuse in L1/L2 caches, since threads can be assigned any block awaiting execution, not necessarily the one adjacent to the block that the thread just finished processing. Therefore, same cache lines would have to be reloaded more frequently.

The best choice of block dimensions would depend upon performance of employed software, such as compiler optimizations, OpenMP runtime and synchronization primitives implementations, and CPU processing power, so our recommendation is to perform benchmark of the block sweeping methods on the system where they will be run for data processing on simple homogeneous model with dimensions common to those of planned datasets, in order to select optimal block size. First, increase block size along the fastest growing index, then along the second-fastest. In our testing across different systems, block dimensions with a small number of (i, j)-rows, for example, $1 \times 8 \times S_k$, worked best. This approach does not require knowledge of specific details of CPU cache implementation such as cache size and cache associativity. However, on sufficiently large models, where S_k grid dimension along fastest changing index

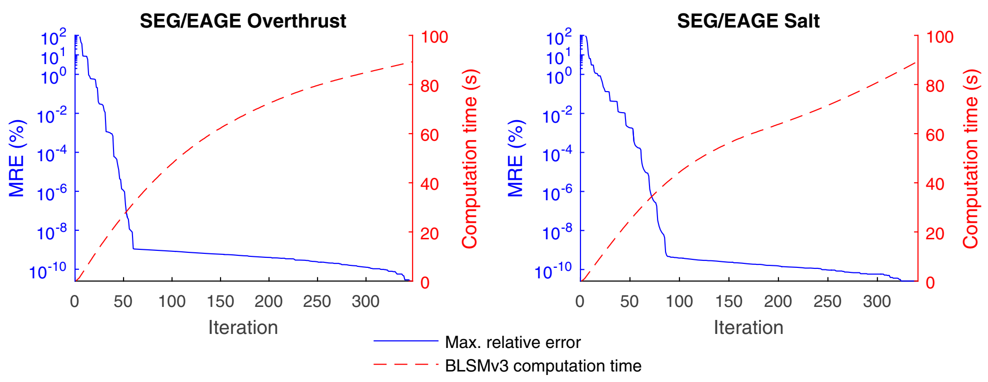


Fig. 8 SEG/EAGE Overthrust and Salt velocity models BLSMv3 computation time and maximum relative error versus the number of iterations completed

Table 2 Block dimensions versus computation time for block sweeping methods

Block dims	Time (s)					
	BFSMv1	BFSMv2	BFSMv3	BLSMv1	BLSMv2	BLSMv3
Homogeneous model:						
1 × 1 × 16	11.63	10.94	136.86	12.51	9.91	146.74
1 × 1 × 64	8.47	8.35	40.78	7.96	6.33	39.29
1 × 1 × 301	7.78	7.81	11.81	6.05	5.65	10.96
1 × 1 × 601	7.52	7.77	8.04	5.83	4.54	4.84
1 × 4 × 601	7.60	7.77	7.46	3.48	3.51	3.56
1 × 8 × 601	7.49	7.76	7.38	3.60	3.56	3.47
1 × 12 × 601	8.52	7.72	7.42	4.01	3.53	3.92
1 × 24 × 601	10.06	7.77	11.14	4.51	3.60	6.80
Overthrust model:						
1 × 1 × 16	12.81	11.43	162.35	17.71	11.15	165.45
1 × 1 × 64	9.58	9.07	47.33	14.16	8.52	49.82
1 × 1 × 161	8.55	8.69	11.64	12.40	7.75	11.45
1 × 4 × 161	8.20	8.50	8.37	6.67	6.71	6.83
1 × 8 × 161	8.45	8.46	8.02	6.59	6.61	6.45
1 × 12 × 161	8.50	8.44	8.30	6.68	6.64	6.52
1 × 24 × 161	8.60	8.48	9.26	7.13	6.65	9.99
1 × 36 × 161	8.68	8.50	13.67	7.30	6.72	14.44
Salt model:						
1 × 1 × 16	16.59	15.22	192.87	26.34	17.00	196.96
1 × 1 × 64	13.19	12.08	58.85	21.01	12.65	60.84
1 × 1 × 210	11.32	11.56	13.16	18.00	12.15	13.48
1 × 4 × 210	11.02	11.36	11.00	10.25	10.32	10.23
1 × 8 × 210	11.05	11.34	10.69	10.33	10.38	9.94
1 × 12 × 210	11.27	11.31	10.87	10.39	10.35	10.27
1 × 24 × 210	13.58	11.39	14.42	11.73	10.46	16.69
1 × 36 × 210	13.47	11.43	21.29	12.16	10.62	24.34

k is large, about at least two times greater than L1 cache bank size, reducing block size along k-dimension to at least this bank size might be warranted to increase data reuse in caches.

Parallel efficiency We have tested scalability of our proposed methods' efficiency. We calculate efficiency as

$$\text{Efficiency (\%)} = \frac{t_{\text{serial}}}{t_{\text{parallel}} \times nth} \times 100, \quad (11)$$

where t_{serial} is computation time for the original serial method, i.e. FSM for DFSM/BFSM methods and LSM for DLSM/BLSM methods; t_{parallel} is computational time of the parallel method, and nth is the number of computational threads. As can be seen from Fig. 9, all block sweeping methods scale well, with efficiency in range of 85–95%, while DFSM/DLSM methods show efficiency of 15–45% compared to FSM/LSM due to problems with inefficient use of memory hierarchy, discussed in Section 4.

Load imbalance We have also analysed potential sources of load imbalance that were discussed in Section 5. In Table 3, we present method event counts for processing of all three models by the LSM method, namely: `update` calls—the number of times grid point value update procedure was called to perform computations according to Eq. 6 and (7), unlocked points—percentage of calls to process unlocked point that was not skipped over immediately, 1D, 2D, 3D computed—percentage of calls when new time value was computed using only one, two or three neighbouring grid points when solving (6), actually updated—percentage of calls when new value was assigned to the grid point according to updating rule (7). As can be seen from the table, LSM reduces the number of computations the most on simple homogeneous model, and the least on Salt model. This explains low performance advantage of LSM for this model, observed in Table 2 and Fig. 9. Potential imbalance due to different new value computation paths is on the

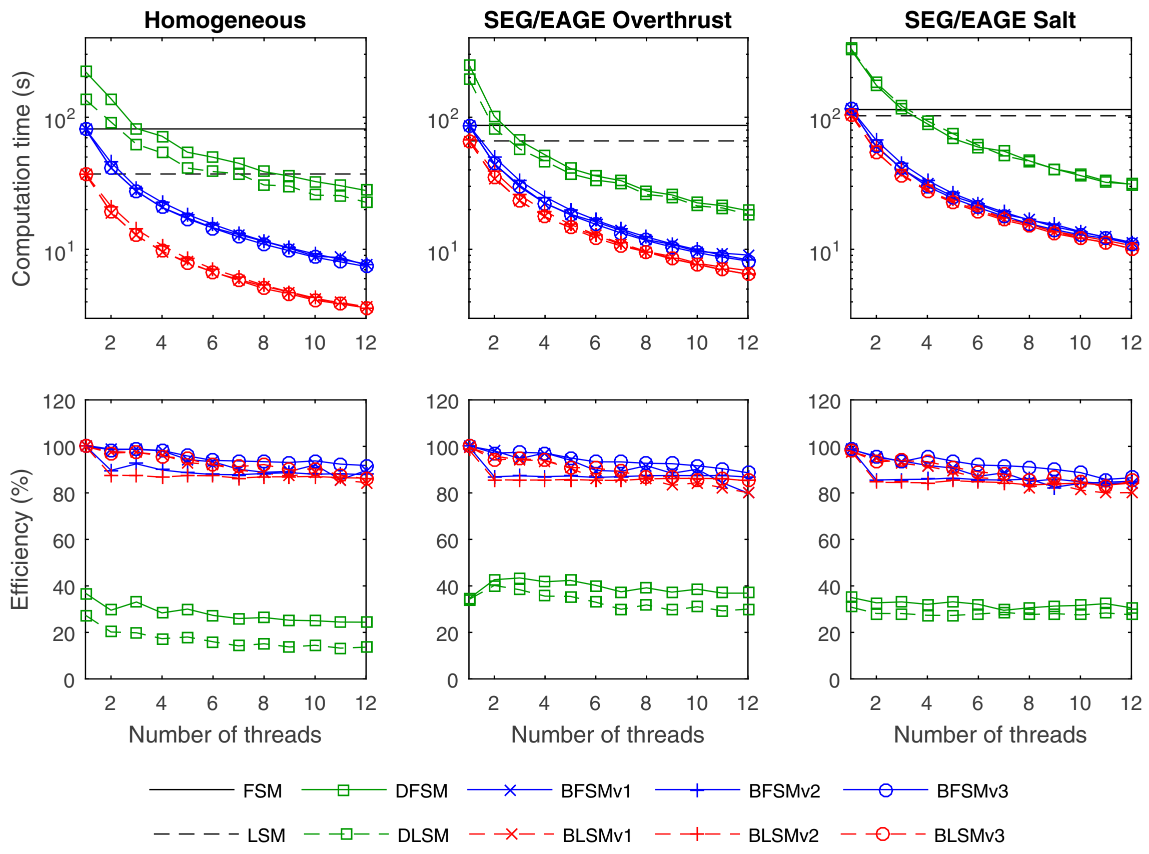


Fig. 9 BFSM and BLSM computation time and efficiency

contrary highest for homogeneous model and lowest for Salt model. This data further supports the notion that dataflow synchronization of tasks as in BFSMv3 and BLSMv3 should be more preferable since it does not enforce any global synchronization points that can lead to some of the threads finishing work early and idling.

Perf metrics Lastly, we have tested performance of our proposed methods using Linux perf on the same Intel Core i7-2630QM system as in Section 4. We have not used cluster in either of perf tests because it does not have perf installed and this test required root access. Results are presented in Table 4. Test were done with 1 × 10 × 600 block size which was the best choice in our test for this system and homogeneous 601 × 601 × 601 model. As can be seen from the results and Table 1, our parallel implementations show the same high efficiency of data caches and TLB utilization as FSM and LSM serial methods. Miss rates and other metrics are close to those of FSM/LSM. All methods again show the same level of performance in a practical sense. BLSMv3 turned out to be a little slower that BLSMv2 probably due to higher dTLB load miss rate which can be attributed to less restrictive assignment of blocks to threads in the dataflow synchronization scheme. Lower efficiency than during testing on clusters can be explained by higher system load, since this is a workstation system. There are more active processes competing for CPU time than on a cluster node, where only essential processes are usually run. Some of the threads can be frequently pre-empted by the operating system, increasing the time it takes to process some of the blocks, potentially causing other threads to idle. Therefore, dataflow is again the recommended approach for use on workstations due

Table 3 Sources of computational load imbalance in grid points for tested velocity models for LSM

Events	Homogen.	Overthrust	Salt
`update` calls	1.95E+09	1.76E+09	2.30E+09
of those:			
unlocked points	20.98%	46.21%	59.09%
1D computed	1.40%	0.96%	0.66%
2D computed	7.40%	5.60%	5.47%
3D computed	12.18%	39.65%	52.96%
actually updated	20.98%	41.30%	53.31%

Total number of grid point value `update` procedure calls and percentages for different computation paths taken during those calls

Table 4 Performance test results for BFSM and BLSM (4 threads) homogeneous velocity model, 601 × 601 × 601 grid points, nine iterations

Performance metric	BFSMv1	BFSMv2	BFSMv3	BLSMv1	BLSMv2	BLSMv3
Time	14.60	14.35	14.24	6.93	6.82	6.89
Efficiency (%)	85.87	87.34	88.07	86.43	87.81	86.84
Cycles	1.63E+11	1.60E+11	1.60E+11	8.42E+10	8.28E+10	8.45E+10
Instructions	2.75E+11	2.75E+11	2.74E+11	1.39E+11	1.38E+11	1.39E+11
IPC	1.68	1.71	1.71	1.65	1.67	1.64
L1-dcache-loads	8.74E+10	8.74E+10	8.77E+10	4.99E+10	4.99E+10	5.02E+10
L1-dcache-load-misses	1.26E+09	1.25E+09	1.31E+09	4.85E+08	4.92E+08	5.13E+08
L1 load miss rate (%)	1.45	1.43	1.49	0.97	0.99	1.02
L1-dcache-stores	2.81E+10	2.81E+10	2.82E+10	2.31E+10	2.32E+10	2.32E+10
L1-dcache-store-misses	1.61E+08	1.73E+08	1.79E+08	1.83E+08	1.74E+08	1.93E+08
L1 store miss rate (%)	0.57	0.62	0.64	0.79	0.75	0.83
L1-dcache-prefetch-misses	9.12E+08	9.11E+08	1.22E+09	3.05E+08	2.72E+08	4.14E+08
LLC-loads	1.32E+08	1.03E+08	1.61E+08	4.16E+07	3.38E+07	6.89E+07
LLC-stores	1.07E+08	1.17E+08	1.23E+08	1.23E+08	1.20E+12	1.30E+08
dTLB-loads	8.74E+10	8.73E+10	8.77E+10	5.02E+10	5.00E+10	5.03E+10
dTLB-load-misses	1.70E+06	9.44E+04	3.05E+06	5.79E+05	7.23E+04	3.09E+06
dTLB load miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.01
dTLB-stores	2.82E+10	2.82E+10	2.82E+10	2.31E+10	2.31E+10	2.32E+10
dTLB-store-misses	1.83E+04	1.40E+04	9.08E+05	1.65E+04	1.49E+04	1.13E+06
dTLB store miss rate (%)	0.00	0.00	0.00	0.00	0.00	0.00

Test performed on Intel Core i7-2630QM quad core CPU

to its most relaxed synchronization scheme in relation to data dependencies, so the impact of load imbalance and context switches on performance should be the lowest possible among the three schemes in most cases. We also note that we do not use software prefetch optimization in our implementations so as to not increase its complexity and introduce additional hardware/compiler dependencies (aside from OpenMP 4.0 support requirement). In fact, since miss rates are already low, we do not expect high-performance gains from further cache use optimization. We consider vectorization of computations a more promising direction of further studies, since sweeping methods due to their sequential nature of computations of neighbouring grid points do not take advantage of SIMD capabilities in modern CPUs. We plan to investigate the possibilities of efficient vectorization of sweeping methods in the future. Another interesting direction of studies would be comparison of proposed methods and other parallel eikonal equation solvers, such as parallel implementations of fast marching, heap-cell, and fast iterative methods.

As noted in [9], any eikonal equation solver can be used at the shared memory level in HMP-FSM distributed algorithm. Therefore, proposed block sweeping methods can also be used in HMP-FSM instead of DFSM to increase the efficiency of computations performed on multicore CPUs inside compute nodes.

7 Conclusion

In the present paper, we have analysed existing and proposed new parallel sweeping methods for numerical solution of the eikonal equation. These solutions can be used in different applications: computing first-arrival travel times as well as first-arrival waveforms [21], image processing [27] etc.

Proposed methods are optimized for efficient use of CPU caches and show high parallel efficiency of 85–95% on modern multicore CPUs. Basic idea of the proposed methods is to decompose numerical grid into blocks of tasks to be processed by different threads using serial fast or locking sweeping methods. Proposed methods do not require additional memory and converge in the same number of iterations when compared to the original serial sweeping methods. Several approaches were proposed for synchronization of block computations; all have shown similar performance, with the third approach based on dataflow synchronization being the fastest in the majority of tests.

We have also performed convergence tests of the locking sweeping method on SEG/EAGE Overthrust and Salt models in order to determine suitable stopping criterion parameter selection for practical applications. By using the ratio of minimal grid step to maximum velocity in the computational domain as the stopping criterion parameter we have greatly reduced computational times for these models keeping maximum relative error of the early terminated solutions being less than 1% compared to the fully converged ones.

References

1. Bak, S., McLaughlin, J., Renzi, D.: Some improvements for the fast sweeping method. SIAM J. Sci. Comput. **32**(5), 2853–2874 (2010)
2. Bleistein, N., Cohen, J.K., John, W. Jr., et al.: Mathematics of Multidimensional Seismic Imaging, Migration, and Inversion, vol. 13. Springer Science & Business Media, Berlin (2013)
3. Breuß, M., Cristiani, E., Gwosdek, P., Vogel, O.: An adaptive domain-decomposition technique for parallelization of the fast marching method. Appl. Math. Comput. **218**(1), 32–44 (2011)
4. Capozzoli, A., Curcio, C., Liseno, A., Savarese, S.: A comparison of fast marching, fast sweeping and fast iterative methods for the solution of the Eikonal equation. In: 21st Telecommunications Forum, pp. 685–688 (2013)
5. Cerveny, V.: Seismic Ray Theory. Cambridge University Press, Cambridge (2005)
6. Chacon, A., Vladimirsky, A.: Fast two-scale methods for eikonal equations. SIAM J. Sci. Comput. **34**(2), A547–A578 (2012)
7. Chacon, A., Vladimirsky, A.: A parallel two-scale method for Eikonal equations. SIAM J. Sci. Comput. **37**(1), A156–A180 (2015)
8. Crandall, M.G., Lions, P.L.: Viscosity solutions of Hamilton-Jacobi equations. Trans. Am. Math. Soc. **277**(1), 1–42 (1983)
9. Detrixhe, M., Gibou, F.: Hybrid massively parallel fast sweeping method for static Hamilton-Jacobi equations. J. Comput. Phys. **322**, 199–223 (2016)
10. Detrixhe, M., Gibou, F., Min, C.: A parallel fast sweeping method for the Eikonal equation. J. Comput. Phys. **237**, 46–55 (2013)
11. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
12. Duchkov, A., de Hoop, M.: Velocity continuation in the downward continuation approach to seismic imaging. Geophys. J. Int. **176**, 909–924 (2009)
13. Duchkov, A.A., De Hoop, M.V.: Extended isochron rays in prestack depth (map) migration. Geophysics **75**(4), S139–S150 (2010)
14. Fomel, S.: Theory of differential offset continuation. Geophysics **68**(2), 718–732 (2003)
15. Hubral, P., Tygel, M., Schleicher, J.: Seismic image waves. Geophys. J. Int. **125**(2), 431–442 (1996)
16. Intel Coorporation: Intel 64 and IA-32 architectures optimization reference manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. Accessed 2017-01-16 (2016)
17. Jeong, W.K., Whitaker, R.T.: A fast iterative method for eikonal equations. SIAM J. Sci. Comput. **30**(5), 2512–2534 (2008)
18. Kao, C.Y., Osher, S., Qian, J.: Lax–friedrichs sweeping scheme for static hamilton–jacobi equations. J. Comput. Phys. **196**(1), 367–391 (2004)
19. Nikitin, A.: Block sweeping methods (source code). https://doi.org/10.5281/zenodo.269001 (2017)
20. Rouy, E., Tourin, A.: A viscosity solutions approach to shape-from-shading. SIAM J. Numer. Anal. **29**(3), 867–884 (1992)
21. Serdyukov, A., Duchkov, A.: Hybrid kinematic-dynamic approach to seismic wave-equation modeling, imaging, and tomography. Math. Probl. Eng. **2015**, 543,540 (2015)
22. Sethian, J.A.: A fast marching level set method for monotonically advancing fronts. Proc. Natl. Acad. Sci. **93**(4), 1591–1595 (1996)
23. Sethian, J.A.: Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science, vol. 3. Cambridge University Press, Cambridge (1999)
24. Sethian, J.A., Vladimirsky, A.: Ordered upwind methods for static Hamilton-Jacobi equations: theory and algorithms. SIAM J. Numer. Anal. **41**(1), 325–363 (2003)
25. Stolk, C.C., de Hoop, M.V., Symes, W.W.: Kinematics of shot-geophone migration. Geophysics **74**(6), WCA19–WCA34 (2009)
26. Tanenbaum, A.S.: Structured computer organization Pearson (2006)
27. Tsai, R., Osher, S., et al.: Review article: level set methods and their applications in image science. Commun. Math. Sci. **1**(4), 1–20 (2003)
28. Tsitsiklis, J.N.: Efficient algorithms for globally optimal trajectories. IEEE Trans. Autom. Control **40**(9), 1528–1538 (1995)
29. Vidale, J.: Finite-difference calculation of travel times. Bull. Seismol. Soc. Am. **78**(6), 2062–2076 (1988)
30. Vidale, J.E.: Finite-difference calculation of traveltimes in three dimensions. Geophysics **55**(5), 521–526 (1990)
31. Zhao, H.: A fast sweeping method for Eikonal equations. Mathematics of computation **74**(250), 603–627 (2005)
32. Zhao, H.: Parallel implementations of the fast sweeping method. J. Comput. Math. **25**, 421–429 (2007)