Journal of Mathematics in Industry
a SpringerOpen Journal

RESEARCH Open Access

Parallel solutions of static Hamilton-Jacobi equations for simulations of geological folds

Tor Gillberg[1,2*], Are Magnus Bruaset[1], Øyvind Hjelle[2] and Mohammed Sourouri[1]

*Correspondence: torgi@simula.no
[1]Simula Research Laboratory, Martin Linges vei, Oslo, Norway
[2]Kalkulo AS, Martin Linges vei, Oslo, Norway

Abstract

Two new algorithms for numerical solution of static Hamilton-Jacobi equations are presented. These algorithms are designed to work efficiently on different parallel computing architectures, and numerical results for multicore CPU and GPU implementations are reported and discussed. The numerical experiments show that the proposed solution strategies scale well with the computational power of the hardware. The performance of the new methods are investigate for tow types of static Hamilton-Jacobi formulations are investigated, the isotropic eikonal equation and an anisotropic formulation used to simulate different types of geological folding. Simulations of geological folding is a key component in an industrial software used in oil and gas exploration. In particular, our experiments indicate that the new algorithms would be capable of accelerating an existing industrial simulator substantially. Direct comparison with the current industry code shows that computing times can be reduced from several minutes to a few seconds. The new methods are now being migrated to the industrial software.

Keywords: parallel computing; general purpose GPU computing (GPGPU); multicore CPU; eikonal equation; static Hamilton-Jacobi equations; front propagation; geological folding; generalised distance computing; CUDA; OpenMP

1 Static Hamilton-Jacobi equations

This paper concerns methods for simulating monotonically propagating fronts modelled by static Hamilton-Jacobi equations. A general formulation of such equations is

$$\begin{aligned} H\big(\mathbf{x}, \nabla T(\mathbf{x})\big) &= 1, \\ T(\mathbf{x}) &= g(\mathbf{x}) \quad \forall \mathbf{x} \in \Gamma, \end{aligned} \tag{1}$$

where $g(\mathbf{x})$ represents initially given values at points in the set Γ, and the Hamiltonian H is here assumed Lipschitz continuous and convex in ∇T. The unknown entity, T, can be thought of as a generalised distance field, as well as the time of arrival of a monotonically expanding front. Usually, $g(\mathbf{x}) = 0$ which means that Γ represents the object from which the generalised distance should be computed. Static Hamilton-Jacobi equations are known to often have multiple-valued solutions, in that a point in the domain may have several values simultaneously. The viscosity solution of the static Hamilton-Jacobi equations is also the minimal distance value, that is the first time of arrival [1]. In some applications, the multiple-valued solutions can be important [2]. However, in many applications it is

Springer

©2014 Gillberg et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/2.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

enough to solve for the viscosity solution of the equation. In this paper we solve solely for the viscosity solution.

1.1 A mathematical framework for propagating fronts

The solution T of (1) can be thought of as the time of arrival of a wavefront that propagates through a domain. More generally, the t-isosurface of T shows the position of the front at time t. The t-isosurface can also be regarded as the generalised t-distance from Γ, that is, solution values are increasing away from Γ. In a front propagation setting, equation (1) is often formulated as $V\|\nabla T\| = 1$, where the scalar V is the speed at which the front is moving in the direction normal to the front. At every point, V is greater or equal to 0, with equality implying that the front never reaches the point. If the speed is independent of the direction the front is moving in, the front propagation is said to be isotropic. The isotropic eikonal equation is important in many applications and is formulated as

$$F(\mathbf{x})\|\nabla T(\mathbf{x})\| = 1. \tag{2}$$

For the special case of $F \equiv 1$ and $g(x) \equiv 0$, the viscosity solution of the eikonal equation is the minimal Euclidean distance to Γ.

The problem is anisotropic if the rate of increase of T values changes with the direction it is measured in. There is a clear causality interpretation of a propagating front since the current motion does not depend on future events. In other words, smaller values of T are independent of larger T values, which imply that the discrete solution representing T should be computed in increasing order. In the isotropic eikonal case this causality observation can be directly translated to discrete nodal values. However, for the more general anisotropic case a similar discrete interpretation is not valid [3], and anisotropic problems are therefore more complicated to solve. The algorithms developed in this paper have been applied to both isotropic eikonal formulations and to anisotropic problems of the form (3).

Most formulations of static Hamilton-Jacobi equations are nonlinear, and the computations needed for updating the solution in a grid node are expensive in terms of both floating-point operations and logical branching. The amount of computations needed can be decreased by updating nodes in different orderings.

In the following section we give a brief overview of solution algorithms. For a more complete algorithmic overview the reader is to [4] and references therein.

1.2 Applications

Many physical phenomena can be described by a propagating front. A front can for example describe an interface between different objects or fluids (multi-phase flow) [5], a shock wave [6], or the arrival of a wave [7]. In this work, the front is assumed monotonically expanding, like a wildfire spreading only to unburned grounds. Monotonically expanding fronts can be entirely described by the time of arrival of the front to all points in the domain. Applications requiring fast simulations of monotonic front propagations include medical tomography [8], simulation of cardiac activation times [9], segmentation of images [10], and seismic wave propagation [11]. Shorter computing times may result in faster and more accurate models since one can afford to better explore parameter ranges, and thereby the model uncertainty. As a result, faster methods can provide further insight into the underlying physical problems. In this paper, we focus on the development of new

efficient algorithms. For a proper introduction to front propagation and a range of applications, see [1]. The application initiating the research presented in this paper is from a software project developed by Statoil ASA and Kalkulo, and is presented in the following paragraph.

The compound earth simulator

Structural restoration and reconstruction is a methodology for building and improving structural models of subsurface geology. The Compound Earth Simulator (CES) is a software in which large three dimensional models can be transformed between different structural states [12]. Generalised distance fields are needed in the transformations of the geology and to deposit the layers. The generalised distance fields are given from the following static Hamilton-Jacobi equation,

$$\begin{aligned} F\left\|\nabla T(\mathbf{x})\right\| + \psi\left(\mathbf{a}\cdot\nabla T(\mathbf{x})\right) = 1, \quad \mathbf{x}\in\Omega, \\ T(\mathbf{x}) = t_0 \quad \forall\mathbf{x}\in\Gamma. \end{aligned} \tag{3}$$

Here, Ω is the computational domain, Γ_0 is the surface of a geological layer (horizon), and $\mathbf{a}$ is a unit vector. Geological volumes are transformed repeatedly in the workflow [13]. For the software to be user friendly, the distance fields must be computed rapidly while still being accurate.

In this paper, software is said to be interactive if the delay due to computations is of the order of a few seconds at the most. Some of the available algorithms are fast enough for interactive use in two spatial dimensions. This is not the case in three dimensions, since the number of nodes and the nodal update cost increase significantly. A case study of geological folding based on structures extracted from offshore seismic data, is included in the numerical experiments reported in this paper. The methods presented in this study enables interactive high-resolution restorations in three dimensions.

The ray equations

Characteristic curves are paths along which 'particles' on the front are pushed forward, as well as lines along which distances are measured. Let $\mathbf{x}(s) = (x,y,z)(s)$ denote a characteristic curve parameterised by the scalar s. Characteristic curves are often referred to simply as characteristics. The solution T is strictly increasing along characteristics that are perpendicular to the moving front. The characteristic equations can be formulated on basis of (1) as

$$\frac{d\mathbf{x}}{ds} = \nabla_{\mathbf{p}} H(\mathbf{p},\mathbf{x}), \tag{4}$$

$$\frac{d\mathbf{p}}{ds} = \nabla_{\mathbf{x}} H(\mathbf{p},\mathbf{x}), \tag{5}$$

where $\mathbf{p} = \nabla T$, $\nabla_{\mathbf{x}} = (\partial/\partial x, \partial/\partial y, \partial/\partial z)^T$, and $\nabla_{\mathbf{p}} = (\partial/\partial T_x, \partial/\partial T_y, \partial/\partial T_z)^T$ [14, 15]. For the Hamiltonians (2) and (3), the respective characteristic curves $\mathbf{x}_e(s)$ and $\mathbf{x}_f(s)$ are given by

$$\mathbf{x}_e(s) = \mathbf{x}_e(0) + sF\mathbf{n}, \tag{6}$$

$$\mathbf{x}_f(s) = \mathbf{x}_f(0) + s(F\mathbf{n} + \psi\mathbf{a}), \tag{7}$$

where $\mathbf{n} = \nabla T/\|\nabla T\|$ is the unit normal vector to the propagating front, and $x(0)$ marks the starting location of the curve. The velocity parameters F and $\psi\mathbf{a}$ are assumed to be constant in a small neighbourhood, which results in locally linear characteristic curves along which $\mathbf{p}$ is constant. On a discrete grid, the parameters are commonly assumed to be constant within the stencil area. An investigation of the solution value along a characteristic curve shows that

$$\frac{dT}{ds} = \mathbf{p} \cdot \frac{d\mathbf{x}}{ds} = \mathbf{p} \cdot \nabla_{\mathbf{p}} H = 1 \quad \text{for (2) and (3)} \tag{8}$$

$$\implies \quad T\big(\mathbf{x}(s)\big) = T\big(\mathbf{x}(0)\big) + s. \tag{9}$$

That is, T is increasing linearly in s along characteristics. Because of this one-dimensional dependency, solution values can be extrapolated with high accuracy along characteristic curves. An understanding of characteristics is therefore important when creating numerical stencils, as well as designing new algorithms for front propagation problems. The solutions T_e and T_f for equations (2) and (3) change explicitly according to

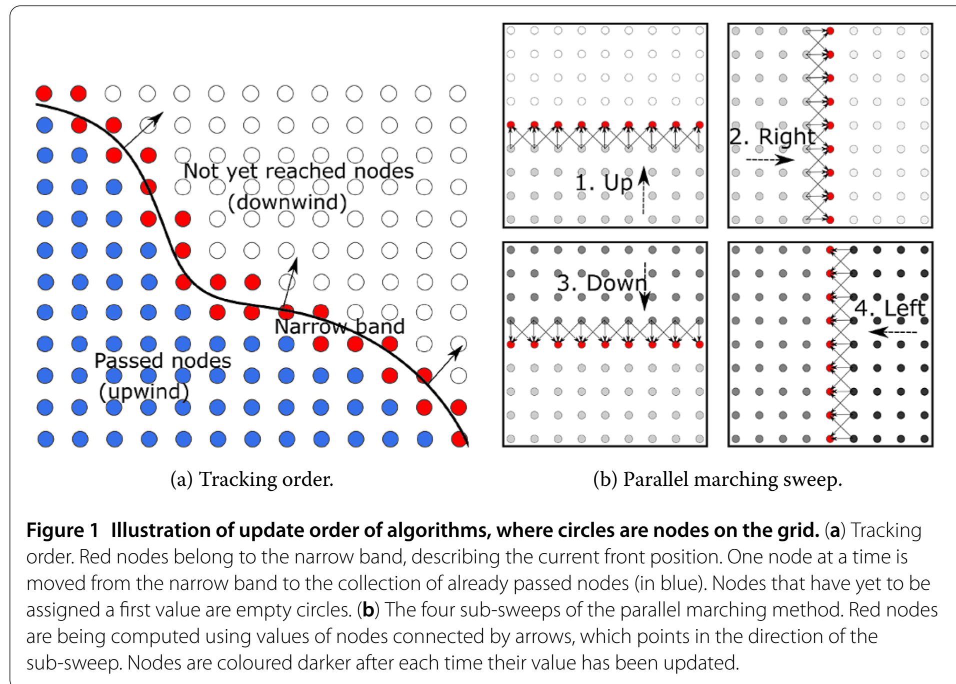
$$T_e(s\mathbf{n}) = T_e(\mathbf{0}) + \frac{s}{F}, \tag{10}$$

$$T_f\left(s\frac{F\mathbf{n} + \mathbf{a}}{\|F\mathbf{n} + \mathbf{a}\|}\right) = T_f(\mathbf{0}) + \frac{s}{\|F\mathbf{n} + \mathbf{a}\|} \tag{11}$$

along characteristics. In particular, T_e increases at a constant rate since the front moves with a speed that is independent of the normal and its direction. This is not the case for T_f, which increases with a rate that depends on the direction in which the front moves. From these formulations the increased complexity for the anisotropic formulation is obvious. The anisotropic dependencies in the fold equation complicate the computations, and the computational cost is significantly higher than for the isotropic problem.

1.3 Numerical solution methods

The solution can be built by computing the local solution along a set of characteristic curves, and thereafter interpolating the travel time between the curves. This approach is traditionally referred to as the method of characteristics, or ray tracing. In seismology, the term 'rays' is often used in lieu of characteristics. The methods developed in this paper assumes a fixed grid of nodal values. Grid-based methods use stencils that are strongly related to ray-tracing methods. Instead of tracing values along specific characteristics, the grid-based tracking methods update nodal values by interpolating solution values between neighbouring nodes organised in stencils. Commonly, solution values are interpolated linearly along edge-connected nodes in the grid [16], implying a planar modelling of the front. In Appendix 1, we present an efficient stencil formulation for the eikonal equation. A smaller value corresponds to an earlier arrival time, and thus a better estimate of the sought viscosity solution. The solution at every point is therefore strictly decreasing. A too small value will not be enlarged, but instead it generates an error that will spread throughout the domain. It is therefore important that new estimates are based on upwind information, that is, the characteristic originates from values that are upwind of the node being updated and therefore already passed by the front.



(a) Tracking order.

(b) Parallel marching sweep.

Figure 1 Illustration of update order of algorithms, where circles are nodes on the grid. (**a**) Tracking order. Red nodes belong to the narrow band, describing the current front position. One node at a time is moved from the narrow band to the collection of already passed nodes (in blue). Nodes that have yet to be assigned a first value are empty circles. (**b**) The four sub-sweeps of the parallel marching method. Red nodes are being computed using values of nodes connected by arrows, which points in the direction of the sub-sweep. Nodes are coloured darker after each time their value has been updated.

Tracking methods

Methods that compute nodes in the same order as they are reached by the front, are referred to as front *tracking* methods. Tracking algorithms classify a node as passed by the front only once, and are often referred to as one-pass algorithms. Therefore, these algorithms compute solution estimates only a few times per unknown node. The current position of the front is described by a set of nodes known as the *narrow band*. Figure 1(a) is an illustration of the classification of nodes in tracking methods. To know which value is the to be considered passed by front, the narrow band nodes are stored in a min-heap data structure that is kept sorted [17, 18]. With N being the total number of nodes, the worst case computational cost of the algorithm is $O(N \log(N))$, where the $\log(N)$ factor is due to the min-heap sorting. It is important to note that a direct application of the causality principle to discrete nodal values does not hold for anisotropic problems [3, 19], and the original tracking method sometimes fails to converge [20]. An algorithmic extension for tracking solvers, is to use adaptive (also called dynamic) stencil shapes [20, 21]. When updating a node, the stencil shape then varies depending on the dynamics of the front, and nodes covering a wider area must contribute to the update step. The footprint of the stencil is often determined on basis of the anisotropy coefficient [20], which is a measure of the degree of anisotropy in the chosen problem.

The need for a sorted data structure makes tracking methods conceptually sequential. However, there are some parallel implementations of tracking-like methods. The group marching method [22] classifies several nodes as passed by the front in every iteration. Since several nodes are passed simultaneously, the method allows a certain degree of parallel processing. However, a parallel implementation is not straightforward since all neighbours should be updated according to two different orderings to ensure convergence. The entire domain can also be decomposed into subdomains in which separate processors update nodes with tracking methods [23, 24]. Subdomains must then be padded with extra

layers of nodes, referred to as *ghost nodes* or *ghost points*. In order for rays to travel between subdomains, the resulting algorithms are not strictly of tracking type, since already computed subdomains can be recomputed several times.

Iterative methods

Sweeping methods are iterative methods based on the observation that the minimal distance follows along a unique direction. By sweeping through the grid in one direction at a time, the minimal distances in that direction is computed for all nodes [25]. Computed nodes are used shortly after they have been assigned new values. The characteristic curves are extended in the same direction as the sweep moves through the domain. Moreover, sweeping methods are almost directly applicable to anisotropic problems [26], and can be adapted to parallel architectures with good performance on multicore systems with a domain-division approach [8, 27]. Still, the parallelism of traditional sweeping methods is coarse grained. This is caused by neighbouring nodes in a sweep depending on each other's values. Nodes on 'slices' of the domain do not share dependencies, and can therefore be computed in parallel [28]. Slices are small in corners of the domain, and parallel processing is then not beneficial. Another disadvantage is that memory accesses of nodes on a slice are incoherent, which impedes the performance of GPU implementations [29].

The parallel marching methods also sweep through the domain, but in slightly modified directions and with alternative stencil shapes [29, 30]. The alternative stencil shapes remove dependencies between neighbouring nodes, so that they can be updated simultaneously. Figure 1(b) give an illustration of the sweeping process of the two dimensional parallel marching method. A full sweep of the grid consists of four sub-sweeps, during which the front is propagated up, right, down and in the left direction. In two spatial dimensions, lines of nodes can be updated simultaneously, and in three dimensions an entire layer of nodes can be updated in parallel [30, 31]. The method scales well when implemented on GPUs. Sweeping methods are faster than tracking methods when the domain structure and the velocity formulation are simple. For complicated problems, many sweeps are needed for convergence and the methods become slow [32, 33]. Similarly, when anisotropy is dominant, more iterations are often needed for convergence [30]. Traditional sweeping methods need to sweep the domain in 2^k directions in k dimensions [34] to span the solution space. Interestingly, only $2k$ sweeps are needed for extensions of the parallel marching method to higher dimensions. However, in practice more full sweeps of the domain might be needed by the parallel marching method than by the traditional sweeping methods [29].

Label correcting methods

There are algorithms that combine iterative and tracking methods by tracing a front with a less strict ordering. A common name for such methods is *label correcting*, emanating from shortest path methods in graph networks [35, 36]. Tracking methods are referred to as *label setting*, since nodes are labelled as 'passed' only once. Since label correcting methods use a less strict ordering than tracking methods, they can sometimes be implemented on parallel architectures. For instance, the massive marching method [37] updates an entire list of nodes in parallel on a multicore CPU. If a node receives a new value during the update, all direct neighbours with larger solution values are collected in a new list. The procedure is repeated until there are no nodes to be added to a list. Massive marching

is very similar to the fast iterative method [38] (FIM), which has also been used to solve problems with anisotropy [10]. GPU FIM have been used in several applications [8, 10], and can be extended to compute geodesic distances on triangulated surfaces [39]. Large velocity contrasts may cause the list to grow long and nodes to be recomputed many times, thereby increasing the computing time significantly [38]. The amount of needed computations is reduced if nodes are updated in an order more similar to the actual position of the expanding front. The two-queue method [40] and semi-ordered FIM (SOFI) [3] are more uniform in their performance since they enforce a stricter ordering of the updates. In these methods, only selected neighbours are directly added to the new list, leaving for later computations the ones too far ahead of the front. As a result, the lists in SOFI and the two-queue method follow the isocurves of the solution more closely than FIM.

2 Developing fast parallel solvers

For a front propagation method to perform efficiently, the algorithm must make use of the causality information embedded in the problem. Brute force iterative and sweeping approaches are too costly in terms of computations, and methods that are strictly tracking the front are too sequential. For instance, the parallel marching algorithms work well on GPUs, but their performance on multi-core devices is not impressive when compared to the algorithms presented in this paper. Since the entire grid is updated repeatedly, such sweeping-like methods cannot perform well enough due to the high amount of computations. Tracking methods are stable, and each node is recomputed only a few times [33]. Since tracking methods are conceptually sequential, the computing time increases noticeably when the grid size increases. Extensions for problems with anisotropy are complicated and sometimes not applicable [19]. In contrast, the more general class of label correcting methods can solve anisotropic problems [3] and can sometimes be implemented on parallel computing architectures [37, 41]. Such methods are therefore particularly interesting to investigate further.

2.1 Algorithmic inspiration

Several recent contributions to the field have inspired the design of the algorithms presented in this paper. In this section we discuss the relation between our algorithms and other existing methods.

Two-scale methods

For problems where the characteristics are straight lines, sweeping methods converge quickly and are often faster than tracking methods. However, problems with highly nonlinear characteristic curves often require many sweeps to reach convergence, and ordered methods then perform better. For most problem formulations, the characteristics bend smoothly and can be approximated well as piecewise linear. With these properties in mind, Chacon et al. [35] design several two-scale algorithms for isotropic problems. In the two-scale methods, the domain is split into subdomains and an ordering of the subdomains is created, for instance with a FMM solver acting at the subdomain level. Every subdomain is thereafter computed with a sweeping method, one subdomain at a time in the determined order. Alternatively, subdomains can enter a heap repeatedly in the *heap-cell method* (HCM). The local sweeping method converges fast since characteristic curves do not bend much within a neighbourhood, such as a subdomain. Recently, the same authors

introduced a parallel two-scale method with the *parallel Heap-Cell Method* (pHCM) for eikonal equations [34]. The pHCM targets specifically multicore architectures, and show impressive speedups when compared with other algorithms. Every thread has its own heap with non-overlapping subdomains awaiting computations. In this paper we consider more general anisotropic front propagations.

Domain decomposition
Two subdomains can be updated independently of each other as long as they do not share a boundary. On shared boundaries there is a risk of read-write memory interferences [42]. To avoid any collision between read and write operations, all subdomains are padded with a layer of ghost nodes [43]. Ghost nodes are local copies of nodes belonging to subdomains that are immediate neighbours. After a subdomain has been computed, ghost nodes in surrounding subdomains must be updated with the most recently computed values. This is done in a *synchronisation* step that follows the computation of a subdomain. The boundary condition for a subdomain is then based on the values that are synchronised from the neighbouring subdomains. Subdomains surrounded by ghost nodes are further discussed in Appendix 2. Any algorithm can be used to compute new values within a subdomain.

Locks
The concept of locking is a methodology for reducing the number of unnecessary computations, which was introduced at the nodal level in the locked-sweeping method [40]. The solution value of a node needs to be updated only if values on neighbouring nodes have changed since the last update. Therefore, a node can be locked for computation right after an update, and be kept locked until relevant changes are observed in its vicinity. A lock can be considered a label that can be reset several times. Consequently, algorithms using locks are of the label correcting type. We have extended the idea of locks to the subdomain level. In our algorithms, we use a compute lock (`CL`) to prevent unnecessary computations of subdomains. After computing all values in a subdomain, it is locked for computation. This compute lock is unlocked whenever the subdomain receives a new value in the synchronisation step. Adopting the terminology used in FIM, we refer to a subdomain with an open compute lock as *active*.

2.2 Two new algorithms

In this paper, we propose two new algorithms that process sets of subdomains in parallel. We refer to the sets of subdomains as *schedules*. The order in which subdomains are computed depends on which subdomains are put into schedules. A schedule is represented as a list of indices to subdomains. The new algorithms differ mainly in the way they build their respective schedules. All subdomains in the schedule are computed and synchronised repeatedly until most subdomains have been locked for computations, and a new schedule is built. In the development of these algorithms also a third algorithm was investigated, called the *two-scale parallel marching method* [4]. Since this method is outperformed by the list-based methods, a presentation is omitted from this paper.

In the reported implementations, we compute new nodal values in a subdomain with the *three-dimensional parallel marching method* (3D PMM) [30], of which details are given in Appendix B.2. Since 3D PMM is a parallel method, the resulting algorithm has two levels of parallelism. Hardware support for multiple layers of parallelism exists for instance

in GPUs. More advanced subdomain solvers (FMM, FIM or SOFI) may increase performance, especially when implemented on devices with lesser processing capabilities. However, such investigations are beyond the scope of this paper. The remainder of this section presents the two new algorithms in detail.

Algorithmic framework

The computed values $T_{i,j,k}$ approximating T in the nodal positions (i,j,k) are stored in an array corresponding to a rectangular grid in three spatial dimensions. The domain is decomposed in (s_x, s_y, s_z) subdomains in the x, y, and z, directions, respectively. Every subdomain consists of (b_x, b_y, b_z) nodes, making the number of nodes in the x direction of the rectangular domain $b_x s_x$. All subdomains are surrounded by a layer of ghost nodes on either sides. The ghost nodes are stored physically in memory, resulting in a total storage of $(b_x + 2)s_x$ nodal values along the x-axis. Similarly there are $(b_y + 2)s_y$ and $(b_z + 2)s_z$ nodes along the two other axes when ghost nodes are included. The grid may have a non-uniform spacing defined by the parameters dx, dy, and dz, but the same spacing values apply to all subdomains. When algorithmic details are discussed on a nodal basis, nodes are indexed with (i,j,k) while subdomains are indexed with (ii,jj,kk). For increased readability, subdomains are sometimes indexed by the triple `sd` = (ii,jj,kk). The compute lock `CL` is implemented as an array with one binary state value for each subdomain, `Open` or `Locked`.

Similar to most front propagation methods, the proposed algorithms assume monotone convergence of the solution. All nodes are first assumed never to be reached by the front, $T_{i,j,k} = \infty$. A set of nodes with initial values is assumed known initially. These values constitute the discretised version of the boundary condition in (3). The subdomains that receive at least one initial value are unlocked for computations during the initialisation phase. As new T values are computed, they are only accepted if they are smaller than the previous value, thus assuring monotone convergence from above. The solution is bounded from below by the smallest initialised value since the solution T is increasing away from Γ_0. Note that solution values will not be increased during the iterations once accepted, and too small values should therefore never be accepted. Too small approximations can be avoided in upwind stencils if the used solution values lie upwind of the node being updated. Further details on the construction of stencils for front propagation problems are discussed in Appendix 1, where an efficient eikonal stencil is presented.

Generic solver

The proposed algorithms have a top level structure given by the template in Algorithm 2. As discussed below, each algorithm have individual specifications of the conditions governing the loops marked $\textcircled{\mathcal{A}}$ and $\textcircled{\mathcal{B}}$. After initiating the problem, a schedule of computing tasks is created. In this context, a task corresponds to the computations needed for a single subdomain. The scheduled tasks are computed and the ghost nodes updated repeatedly through synchronisations with neighbouring subdomains, before a new schedule is created.

The compute and synchronisation steps are similar for both algorithms while the actual construction of the schedule differs. After a subdomain `sd` has been computed, the compute lock is closed, `CL(sd)` $\leftarrow$ `Locked`. In ComputeSchedule(), all active scheduled subdomains are computed in parallel as shown in Algorithm 3.

```
Algorithm 2 SOLVER()
comment: Generic driver for the proposed algorithms.

INITIATEPROBLEM()
BUILDSCHEDULE(Schedule)
while Schedule is not empty (A)
  do { while Repeat condition fulfilled (B)
     {   do { COMPUTESCHEDULE(Schedule)
     {        SYNCFROMSCHEDULE(Schedule)
     { BUILDSCHEDULE(Schedule)
```

```
Algorithm 3 COMPUTESCHEDULE(Schedule)
comment: Parallel execution of computations for all
         scheduled subdomains sd.

for each sd in Schedule in parallel
  do { if CL(sd) is Open
     {   then { COMPUTESUBDOMAIN(sd)
     {          CL(sd) ← Locked
```

```
Algorithm 4 SYNCFROMSCHEDULE(Schedule)
comment: Communication between subdomains updating the
         values of locally stored ghost nodes.

for dir ∈ 1, −1
  do { for each sd in Schedule in parallel
     {   do SYNCiVALUES(sd, dir)
     { for each sd in Schedule in parallel
     {   do SYNCjVALUES(sd, dir)
     { for each sd in Schedule in parallel
     {   do SYNCkVALUES(sd, dir)
```

The ComputeSubdomain() function updates the solution for all nodes and ghost nodes in the subdomain sd. Because of the presence of the ghost nodes, the computations rely only on nodal values stored locally for the subdomain, thus eliminating the need for subdomain communication during the computations. After computing new values for the active subdomains in the schedule, nodal values must be communicated between adjacent subdomains to assure that all ghost nodes have correct values. This is done in SyncFromSchedule(), shown in Algorithm 4.

To avoid any memory interference, the nodal values of neighbouring subdomains are updated in parallel in one direction at a time. First, the ghost nodes belonging to subdomains with larger i values are updated in parallel, and thereafter ghost nodes in subdomains with larger j, k, and smaller i, j, k indices are updated. Note that no computations are performed during the synchronisation, rather the nodal values are set to match their copies in the other subdomain. A nodal value is only changed if the value of its counter-node is smaller.

Algorithm 5 BuildScheduleLAS(L,CL)

```
comment: Construction of the computational schedule,
         tailored for the LAS method.

noSched ← 0
for sd ← index to all subdomains
  do { if CL(sd) is Open
     {   then { L[noSched++] ← (sd)
return (noSched)
```

If a node in subdomain sd receives a new value, the subdomain is activated by opening the compute lock, CL(sd) ← Open. Further details of the Sync□Values() functions are given in Appendix B.1. The algorithms finish when all subdomains are locked for computations and the schedule is empty.

2.2.1 The list of active subdomains method

The list of active subdomains (LAS) algorithm schedules all active subdomains for computation by storing their indices in a list. As the algorithm iterates through the grid, the list follows a front of solution estimates as it travels through the domain, similarly to the lists in the massive marching and FIM methods [37, 38]. Subdomains that have already been computed may at a later stage be added to a new version of the list. Pseudocode for building the schedule (list) for the LAS method is shown in Algorithm 5. The returned variable noSched is the number of subdomains in the list L.

After computing and synchronising the list of subdomains, the number of active subdomains left in the list are computed, here referred to as noActive. The schedule is computed again if the density of active subdomains is higher than a given fraction. Let noSched denote the length of the list. After some experimentation with Ⓑ in Algorithm 2, we have chosen to repeat the computations while $\frac{1}{64} < \frac{\texttt{noActive}}{\texttt{noSched}}$. This threshold gives good performance for most examples. Moreover, let noC denote the number of subdomains that can be computed simultaneously by the current computational platform. If there are fewer active subdomains than what can be computed simultaneously, i.e. noActive < noC, a new schedule is created. This optional condition to improve load balancing is included in the LAS pseudocode given in Algorithm 6, which replaces the generic Solver() in Algorithm 2.

The LAS method has some similarities with FIM, but there are differences to notice: For FIM, only the GPU implementation computes subdomains, and no compute locks are used. In FIM, the scheduled tasks are computed only once before a new schedule is built, whereas in LAS the schedule is reused until at most (around 98%) scheduled subdomains are locked for computation. Any method can be used to compute a subdomain in LAS, whereas in FIM all nodes of a subdomain are computed iteratively until convergence. This brute force approach of FIM would likely reduce the performance on multicore systems.

2.2.2 The semi-ordered list of active subdomains method

In the LAS method, previously active subdomains may be reactivated at later stages of the algorithm. In some front propagation problems, this reactivation can force the list to grow long. A similar observation holds for FIM and SOFI, although SOFI performs

Algorithm 6 SolverLAS()

```
comment: Specific driver for LAS, variant of algorithm 2.1.

InitiateProblem()
noSched ← BuildScheduleLAS(L, CL)
noActive ← noSched
while L is not empty
  do { while 1/64 < noActive/noSched
         do { ComputeSchedule(L)
              SyncFromSchedule(L)
              noActive ← Number of active subdomains in L
              if noActive < noC
                then noActive ← 0
       noSched ← BuildScheduleLAS(L, CL)
       noActive ← noSched
```

more uniformly thanks to the enforced heuristic ordering of updates [3], which makes the behaviour of the list more similar to the narrow band of tracking methods. The semi-ordered list of active subdomains (SOLAS) method similarly schedules subdomains in lists that in a sense stay close in shape to isosurfaces of the solution. If a subdomain is too far ahead of the others, it is not added to the schedule.

In order to measure the approximate location of a subdomain, each subdomain has an associated scalar value stored in a global array structure, here referred to as `SD`. The `SD` value for a subdomain is the smallest new T value that activates the subdomain, and is set during the synchronisation procedure. This minimal T value is similar to the 'rollback state' of the domain decomposition FMM [23]. Details on handling the `SD` values in the synchronisation is given in Appendix B.1.

The `SD` value is an approximate position of the current front as it reaches a subdomain. Let `Av` denote the current average `SD` value of all active subdomains, while `oldAv` denotes the same average of the previous iteration. All subdomains with a `SD` value smaller than a cutoff value `cutT` are scheduled for computation. In our implementation the `cutT` value is set to be $\texttt{Av} + 0.4 \max(0, \texttt{Av} - \texttt{oldAv})$. Several methods of creating this cut-off value was tested, but the chosen 0.4 relaxation from the average value gives good performance for many problems. With the enforced ordering of subdomains, the list will mimic the behavior of the solution's isosurfaces.

If the list is made too short, some processing units may be idling since there are not enough computations to perform. Since the list is computed repeatedly, neighbouring subdomains may be reactivated repeatedly, and if the neighbouring subdomain is not in the list it will not be computed until a new list is created. It is therefore not beneficial to restrict the list when the number of active subdomains is low. A cut-off is therefore only created if there are more active subdomains than an empirically chosen `MinAct` value. With the variable `noC` representing the total number of subdomains that the targeted platform can compute simultaneously, the choice $\texttt{MinAct} = \max(2\texttt{noC}, \frac{3}{2}\max(s_x s_y, \texttt{noSched}))$ seems to give a good overall performance, and has been used in all reported results. The `noSched` bound assures that the list is not limited if there are much fewer active subdo-

Algorithm 7 BuildScheduleSOLAS(L,CL)

```
comment: Construction of the computational schedule,
         tailored for the SOLAS method.

noActive ← 0
sumSD ← 0
cutT ← ∞
for sd ← index to all subdomains
   do { if CL(sd) is Open
        then { noActive++
               sumSD ← sumSD + SD(sd)
Av ← ∞
if noActive > MinAct
   then { Av ← sumSD / noActive
          cutT ← Av + 0.4 max(0, Av − oldAv)
          oldAv ← Av
noSched ← 0
for sd ← index to all subdomains
   do { if CL(sd) is Open and SD(sd) < cutT
          then { L[noSched++] ← sd
        MinAct = max(2noC, 2/3 max(s_x s_y, noSched))
return (noSched)
```

mains than was scheduled in the previous list. The $s_x s_y$ bound is included so that the list is shortened only if the current front is rather large (or thick).

Pseudocode for creating a task schedule in the SOLAS method is given in Algorithm 7. The returned variable `noSched` is the number of scheduled tasks that have been added to the list L.

The pseudocode for the SOLAS driver is identical to the driver SolverLAS() in Algorithm 6, except that calls to BuildScheduleLAS() are replaced by calls to the SOLAS-specific procedure for constructing the subdomain schedule, that is, BuildScheduleSOLAS() defined in Algorithm 7.

2.3 Algorithmic observations and implementation details

In this section we discuss implementation specific details, and some of the optimisations that were tested. The gain of a particular optimisation depends on its interplay with other imposed optimisations. In general, if an optimisation is said to give a 'significant' performance boost, the reduction in computational time was lowered with more than 10%. From an implementation perspective, LAS is easier to implement. However, SOLAS requires only small modifications of the LAS code. The workload and number of iterations of LAS and SOLAS differ significantly with the underlying physical problem and boundary conditions. A performance modelling analysis is therefore difficult to perform.

Initial computations

An expanding front passes a node only once and reaches nodes further away at later times. Because of this causal dependency, the parts of the solution holding larger values depend

on nodes with smaller values. Numerical errors introduced close to the origin will propagate to the whole domain. It is therefore important that the computed solution is particularly accurate close to the initial condition. To ensure that all initiated subdomains fully converge, they are computed with one sweep of 3D PMM in the beginning of the algorithms.

Blocking for cache
In our initial implementation, the numerical grid was stored in row-major order. Nodes within one subdomain are stored in several rows, and every row of data continues into other subdomains. Therefore, some of the data read into cache will belong to other subdomains, and will not be used in computations. When data that are reused often are stored physically close in memory, more relevant data are loaded at once to the faster cache memory. Nodes within a subdomain were therefore ordered contiguously in memory. This optimisation improves the data locality, and is sometimes referred to as *blocking for cache* [44]. The GPU performance was improved significantly with this optimisation, since the access time between global memory and shared memory differ more on GPUs than on CPUs. On GPUs, rows of two and three dimensional data structures are stored in memory allocations of 'pitch' bytes. The pitch is chosen by CUDA to ensue best performance when accessing the row addresses [45]. In the GPU implementation, all nodes in one subdomain lie within one pitch so that the memory accesses are aligned for both the synchronisation and the computation kernels. The CPU performance was also improved, but only by a couple of per cent.

Sliding window
If the entire subdomain is read into the shared memory on the GPU, the performance is limited by the high use of shared memory. This was implemented as a first attempt, but led to unacceptably low speedup. Instead, we use a *sliding window* approach [46]. When two adjacent nodes are updated in 3D PMM, they share several values in the bottom layer of the pyramid stencils, see Appendix B.2. To reduce the number of reads from global memory, the entire bottom layer of nodes is first loaded to a shared memory structure, from where cache loads are faster. The GPU performance was improved significantly.

A similar sliding window optimisation is used in the CPU implementation, since the entire bottom layer of nodes is loaded to a cached data structure. This increases the performance slightly, and has been used in the numerical examples in Section 3.

List building in GPU implementations
Both the SOLAS and the LAS methods create lists. This is difficult to do efficiently on a GPU. Instead, a boolean structure indicating whether or not a subdomain should be present in the list is created on the GPU. This structure is transferred to the CPU where the list creation can be performed fast. Unfortunately, this requires the GPU to wait while the list is being built on the CPU. Stalling communications between the CPU and the GPU is known to be a bottleneck of GPU algorithms [47].

Kernel timeout on GPUs
In our application, we sometimes experienced difficulties launching kernels for the anisotropic experiments in double precision on laptop GPUs. These devices were simultaneously used by the operating system to display the graphical desktop environment. It

appeared that when launching a kernel on many subdomains, we received the error message CUDA_ERROR_LAUNCH_TIMEOUT. According to the CUDA Driver API [48], this error is returned when a kernel takes too long to execute. To avoid the device making the operating system halt, Nvidia has implemented a specific time limit on the execution time of kernels. If this time limit is exceeded by the kernel, the kernel will simply not launch or in some cases be terminated once being launched.

Our solution is to limit the number of launched kernels. If there are too many active subdomains in the list, the list is sliced into chunks on which kernels are launched. The size of each chunk is chosen so that the device can execute the kernel with maximum occupancy. In cases where the number of active subdomains will not lead to a kernel timeout, the chunking is simply omitted.

3 Numerical verification

In this section, we review a series of numerical experiments conducted in order to study the performance of the new solvers proposed in this paper. These experiments include isotropic and anisotropic problems for which analytic solutions can be derived, and isotropic problems with changing velocity and obstacles that are impermeable or very slowly permeable. Our final case is that of simulating a geological fold, based on seismic field data.

Definitions

For any scalar field ξ represented by values in every grid node (i,j,k), we will apply the discrete L_2 norm

$$\|\xi\|_{L_2} = \sqrt{\frac{\sum_{i,j,k} \xi_{i,j,k}^2}{N}}, \tag{12}$$

where the summation is performed over all N nodes in the domain. We refer to the grid as uniform only when $dx = dy = dz$.

Computing platforms

The numerical experiments reported herein have been conducted on a 16 core CPU referred to as *SandyBridge*, and a Nvidia Tesla K20 GPU, hereafter called *K20*. The capabilities of these platforms are summarised in Tables 1 and 2, respectively.

All computing times presented in this section are from implementations of the algorithms, one multicore CPU variant using OpenMP [49], and one GPU variant using CUDA [50]. For all test cases, we report on the fastest solution extracted from at least three executions of the chosen algorithm on the chosen platform. In general, we have observed that the computing times have only insignificant variations from one run to another.

Sizing subdomains

In general, 3D PMM is an efficient algorithm with significant parallel capabilities. For each subdomain, one needs to store both the internal nodes and the ghost nodes to the subdomain. The domain decomposition is slightly overlapping, considering that the ghost nodes are inner nodes of other subdomains. Therefore, the smaller the subdomains get, the larger will the fraction of ghost nodes relative to inner nodes become. From a global

Table 1 Specifications for the Intel Xeon E5-2650 2 × 8-core CPU (one of two identical NUMA domains).

SandyBridge: Intel Xeon E5-2650	
CPU frequency	2.0 GHz
Number of cores	2×8
Single precision	256 GFLOP/s
Double precision	512 GFLOP/s
Shared L3	2×20 MB
Memory	32 GB DDR3
Memory bandwidth	2×51.2 GB/s

Table 2 Specifications for the NVIDIA Tesla K20 GPU.

K20: NVIDIA Tesla K20	
CUDA cores	2496
Single precision	3520 GFLOP/s
Double precision	1170 GFLOP/s
Memory	5120 MB GDDR5
Memory bandwidth	208.0 GB/s

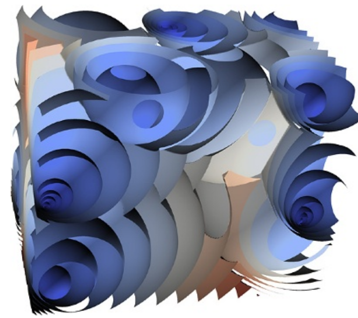
perspective, a reduction of the subdomain size will increase the memory footprint and the communication overhead due to storing and handling a growing number of ghost nodes. Still, a relatively small subdomain size may offer an advantage in that the list can better mimic the front propagation, thus leading to a more efficient exploitation of the underlying causality. Moreover, nodal values of smaller subdomains are more likely to fit in fast memory locations, thereby increasing the computational speed per subdomain.

We experimented with different cubic sizes of subdomains for a range of problems. For the CPU implementation, the optimal subdomain size depends on the exact problem formulation and the total number of nodes. A change in subdomain size have only small impact on the performance of the CPU implementation. We have therefore focused on the GPU implementation when choosing the subdomain size for our computations. Subdomains with 14^3 nodes lead to the best GPU performance in terms of wall clock time used for the solution. Since the computations include updates of ghost nodes, a total of $(14+2)^2$ threads are used to compute a subdomain, see Appendix B.2 for details. The GPUs used for the analysis all have a warp size of 32, that is, a minimum of 32 identical operations per instruction is performed in a Single Instruction Multiple Data (SIMD) fashion [47]. If the number of threads is not a multiple of the warp size, some GPU threads will be idling while the others performs computations. Having analysed our implementation with the CUDA Device Occupancy calculator [51], we have concluded that the subdomain size of 14^3 allows for both the best occupancy and the shortest computing time. This subdomain size is kept constant throughout all experiments reported in this paper.

3.1 Synthetic examples

The LAS and SOLAS algorithms proposed in Section 2.2 have been tested on a range of problems that expose the solvers to several challenges. Here, we briefly summarise our findings by presenting five of those examples. For all these examples, we have computed solutions for grids with 84, 168, 336 and 504 nodes along each axis, grouped accordingly in 6^3, 12^3, 24^3 and 36^3 subdomains. That is, these grids represent from 592, 704 to 128, 024, 064 unknowns. The computations have been conducted on the SandyBridge (CPU)

Figure 8 Isosurfaces of the anisotropic solution with 13 point sources located irregularly in the rectangular domain as boundary condition (Ex_A).



Ex_A: 13 point sources.

and the K20 (GPU) platforms. In all experiments, the solvers stop when all subdomains are locked for computations, and the schedules are empty.

The reported performance numbers bellow are based on computations in double precision. Due to specific hardware features of GPUs, a transition from CPU to GPU is far more attractive for computations in single precision. For the type of calculations done in this paper, single precision arithmetics can be used without noticeable loss of accuracy. This topic is further discussed in Section 3.2.

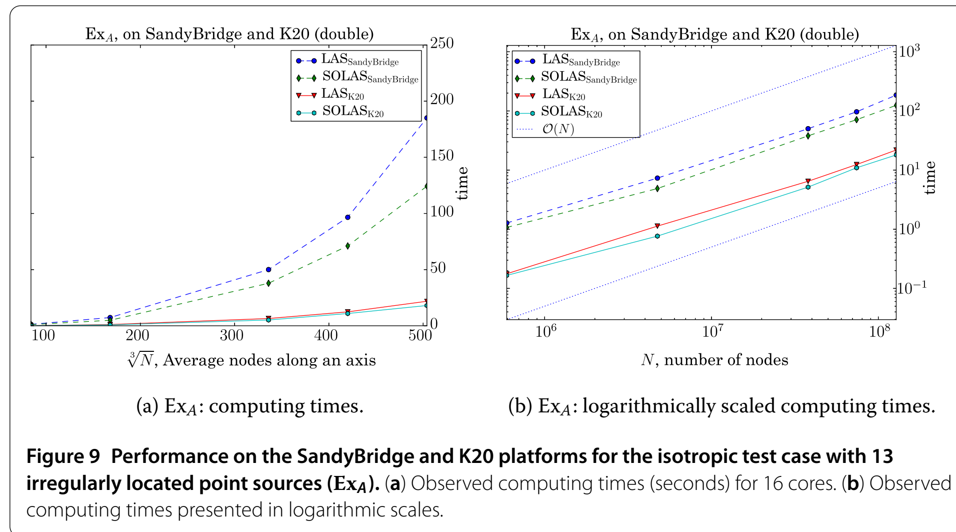
3.1.1 Example with analytic solution

The first example uses the anisotropic formulation (3), with constant velocity parameters $F = 1.4$, and $\psi\mathbf{a} = (0.9, -0.75, -0.07)$. As boundary condition we used 13 point sources spread out irregularly in the rectangular domain, defined by $(0,0,0) \leq \mathbf{x} \leq (10,13,9)$. The ratio between the maximum and minimum velocities, referred to as the anisotropy coefficient, is 11.2 for this problem. A visualization of the anisotropic solution is given in Figure 8.

As shown for the Ex_A case in Figure 9(a), the GPU implementations always outperform its CPU counterparts. Notice that SOLAS performs slightly better than LAS for large grids across both platforms. The difference between these two solvers is more prominent on the CPU than in the GPU-based computations.

The efficiency[a] for eight and 16 cores is 83-88% for SOLAS, whereas it drops down to 65-75% for LAS. Many subdomains are active simultaneously in Ex_A, so the additional ordering of SOLAS is worthwhile. These efficiency ratios are good, taken into account the sequential construction of schedules and that threads are synchronised between the ComputeSchedule() and SyncFromSchedule() procedures. Going from CPU to GPU architecture, we observe 6.9 and 8.5 times faster computations for the Ex_A case when employing the SOLAS and LAS solver on the largest grids, respectively. We also run the same problem with an isotropic setting, $\psi\mathbf{a} = 0$. For the isotropic case, LAS is instead slightly faster than SOLAS on large grids. Compared to the isotropic counterpart, the anisotropic computing time increases with a factor of 7-12 on the CPU and 5-8 on the GPU. The anisotropic stencil is significantly more complicated than the isotropic, and the solution dependencies in the anisotropic case increase the total number of times that subdomains are activated.

In Figure 9(b), we have repeated the computing times from Figure 9(a), using logarithmic scales on both axes. The thin dotted lines illustrate linear relations to N. That is, the figure indicates that all the tested methods have a computational cost that scales with the



(a) Ex$_A$: computing times.

(b) Ex$_A$: logarithmically scaled computing times.

Figure 9 Performance on the SandyBridge and K20 platforms for the isotropic test case with 13 irregularly located point sources (Ex$_A$). (**a**) Observed computing times (seconds) for 16 cores. (**b**) Observed computing times presented in logarithmic scales.

grid size close to $O(N)$, or even *superlinearly*, that is, the computing time seem to increasing less than linearly with growing problem size. Similar behaviour is observed for all our numerical investigations. There are several factors that can help explain this observation. Small data sets cause more computing units to be idle, thereby reducing the observed computational speed. When the data set grows in size, the level of parallelism increases and a better load balance between the processing units is achieved. The time needed to transfer data to and from the GPU is included in the time measurements. The overhead associated with (synchronous) data transfers between the CPU and GPU decreases as the data set gets larger. The cost of data transfer is therefore relatively larger for smaller data sets. Although we have not formally analysed the computational cost of the new algorithms, they will likely have linear complexity due to being semi-ordered and of sweeping type. The superlinear behaviour is thus caused by other artefacts as explained above.

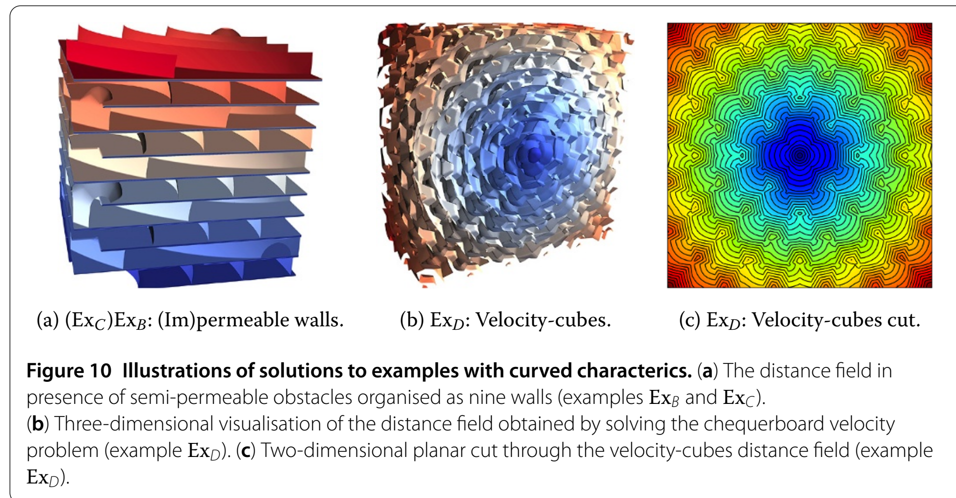
Accuracy and convergence This example has analytical solutions [30], making it possible to study how the numerical solution converges towards the true solution. Let h_k be the edge length of the kth grid in a sequence of increasingly finer grids used for numerical solution of the problems. Denoting the corresponding error $e^{(k)}$, the estimated rate of convergence is

$$p = \frac{\log(\|e^{(k)}\|_{L_2}/\|e^{(k+1)}\|_{L_2})}{\log\left(h_k/h_{k+1}\right)}. \tag{13}$$

For the grid sizes in this study, our convergence estimates are 0.70, 0.74, 0.76 and 0.77, for both the anisotropic and isotopic results. Convergence estimates for single and double precision solutions are for all practical purposes identical. To reach first order convergence, the amount of nodes initially assigned analytic values should be kept constant [16]. The estimated convergence rates are therefore satisfactory.

3.1.2 *Examples with curved characteristics*

Iterative algorithms are sensitive to curved characteristics [32]. It is therefore common to investigate how algorithms perform for problems with obstacles that force the characteristics to turn sharply [18, 29, 38].



(a) (Ex_C)Ex_B: (Im)permeable walls. (b) Ex_D: Velocity-cubes. (c) Ex_D: Velocity-cubes cut.

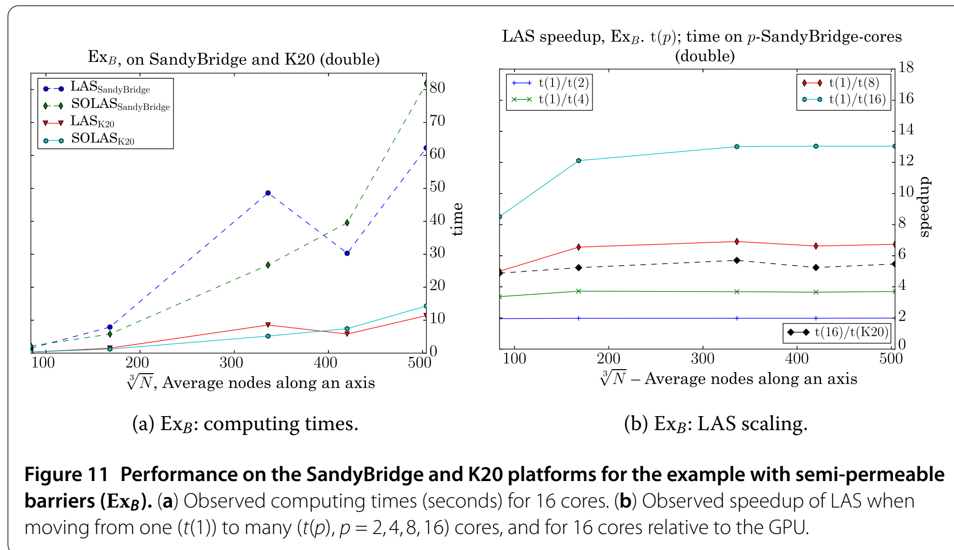
Figure 10 Illustrations of solutions to examples with curved characteristics. (**a**) The distance field in presence of semi-permeable obstacles organised as nine walls (examples Ex_B and Ex_C). (**b**) Three-dimensional visualisation of the distance field obtained by solving the chequerboard velocity problem (example Ex_D). (**c**) Two-dimensional planar cut through the velocity-cubes distance field (example Ex_D).

In the test cases Ex_B and Ex_C, we consider a cubic volume divided by nine walls with small openings in either the upper left or lower right corners. The walls are one node layer thick, and are either semi-permeable ($F = 0.03dz$ for Ex_B) or impermeable ($F = 0$ for Ex_C). Outside these walls, the isotropic velocity is set to $F = 1$. The fastest path from the starting point in a corner of the domain to the opposite corner is by zigzagging through the small openings in the walls. A sample solution is visualised in Figure 10(a) [38, 52].

In the final example, Ex_D, the computational domain is divided into a set of equally sized cubes. The velocity is constant within each cube, having the value of either 1 or 2. For a cube with velocity value 1, all adjacent cubes sharing a side with it will have a velocity equal to 2, and vice versa. We have placed in total 11^3 such velocity cubes within a computational domain measuring 10 length units along each axis. The boundary condition is one point source at the center of the domain. Figure 10(b) shows a volumetric view of the computed solution, while Figure 10(c) shows a planar cut of the solution through the center of the domain [34, 40].

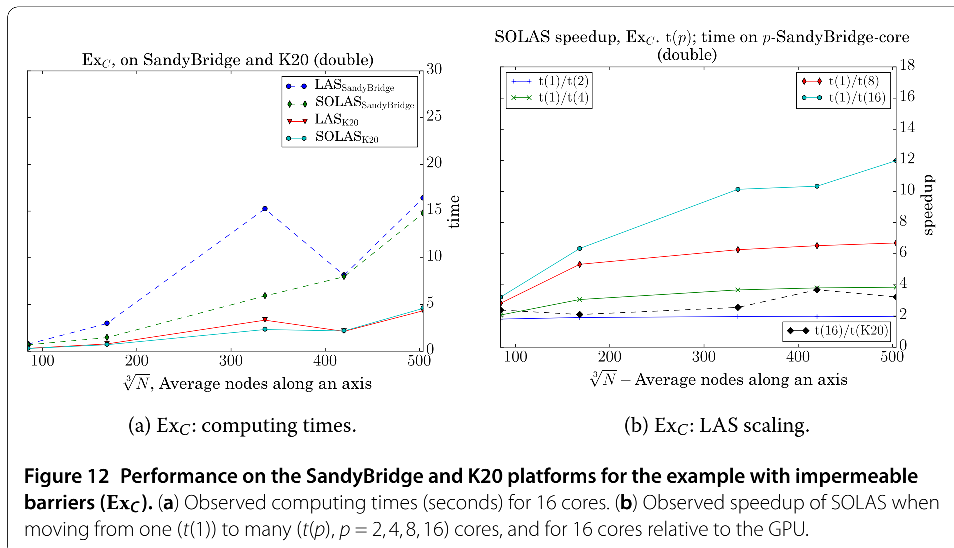
Performance for the obstacle problems The examples Ex_B and Ex_C, having barriers embedded in the computational domains, are challenging since the front is forced to travel along irregularly shaped paths. Figures 11 and 12 summarise the computing times and the speedup factors for these two problems.

The barriers are only one node thick for all grids, and the fraction of nodes that are obstacles reduces as the grid gets finer. The front is small since it only propagates between two layers of walls. For larger grids, more subdomains fit between two barriers, and the parallel processing possibilities increases as the front activates more subdomains. Still, many schedules are built in the computation of these examples. The changing dynamics of the problems causes the compute times to change rather peculiarly with grid sizes, especially LAS since the extra ordering in SOLAS reduce the effects. At a large enough grid size, the LAS solvers changes to such an extent that the solve time decreases for a larger grid size. The irregular behaviour of LAS is present for both the CPU and the GPU implementations. Although the variations for each platform is at the same level when seen relative to the platform's computational power, this issue is most pressing in the CPU environment due to generally longer computing times.



(a) Ex_B: computing times. (b) Ex_B: LAS scaling.

Figure 11 Performance on the SandyBridge and K20 platforms for the example with semi-permeable barriers (Ex_B). (**a**) Observed computing times (seconds) for 16 cores. (**b**) Observed speedup of LAS when moving from one ($t(1)$) to many ($t(p)$, $p = 2, 4, 8, 16$) cores, and for 16 cores relative to the GPU.

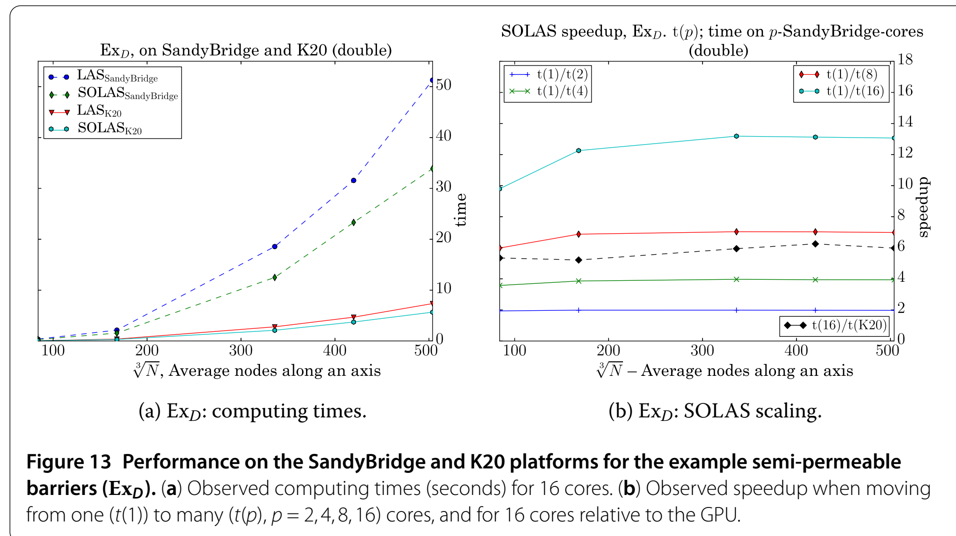


(a) Ex_C: computing times. (b) Ex_C: LAS scaling.

Figure 12 Performance on the SandyBridge and K20 platforms for the example with impermeable barriers (Ex_C). (**a**) Observed computing times (seconds) for 16 cores. (**b**) Observed speedup of SOLAS when moving from one ($t(1)$) to many ($t(p)$, $p = 2, 4, 8, 16$) cores, and for 16 cores relative to the GPU.

For Ex_B, where the barriers are semi-permeable, LAS turns out to be the most efficient solver for large N, regardless of platform. For the impermeable barriers in Ex_C, SOLAS would be the preferred method, although the behaviour comes very close to that of LAS when the grid gets large. For the 504^3 grid, the GPU versions of LAS and SOLAS compute the solution in 4.5 seconds, while the CPU implementations require about 15 seconds. It is noted that the semi-permeable case requires about two to three times as long computing time as the impermeable problem, regardless of solvers and platforms.

Figure 11(b) shows that LAS applied to the Ex_B problem gives a very good speedup, even on eight and 16 cores for which the efficiency is 82% and 84%, respectively. Moving on to the GPU, the LAS method runs 5.5 times faster than the 16 CPU cores. The speedup of SOLAS on eight cores is the same as for LAS, but there is a drop to 77% efficiency when going up to 16 cores. Similar to the observation we made for Ex_A, this drop of speedup does not propagate over to the GPU, where SOLAS performs at same level as LAS. As shown in Figure 12(b), the computing times obtained for applying LAS and SOLAS to



(a) Ex$_D$: computing times.

(b) Ex$_D$: SOLAS scaling.

Figure 13 Performance on the SandyBridge and K20 platforms for the example semi-permeable barriers (Ex$_D$). (**a**) Observed computing times (seconds) for 16 cores. (**b**) Observed speedup when moving from one ($t(1)$) to many ($t(p)$, $p = 2, 4, 8, 16$) cores, and for 16 cores relative to the GPU.

Ex$_C$ scale with the number of cores quit similar to the factors observed for Ex$_B$. However, the performance gain of moving from 16 cores to the GPU is less for the impermeable problem in Ex$_C$, where the speedup factor is in the range 3.2-3.8.

Performance for the velocity-cubes problem For the velocity-cubes problem Ex$_D$, Figure 13 illustrates smooth behaviour of all algorithms with respect to the computing time as a function of the grid size. As for the two examples with embedded barriers, SOLAS proves to be an efficient solver. It performs consistently better than LAS, in particular on the CPU platform. On the GPU, the solution can be computed for the largest grid in 5.7 seconds, compared to the 33.9 seconds required by the 16 CPU cores. Both LAS and SOLAS demonstrate excellent speedup on the CPU, delivering 80%-88% of the theoretical potential for eight and 16 cores. For both methods, the GPU calculations runs 6-7 times faster than the 16 cores.

Overall performance for the synthetic examples Combining the observations from the synthetic test cases discussed above, we conclude that the proposed methods are well suited for efficient computations on multicore CPUs and GPUs. Even for complicated problems these algorithms offer computational efficiency that can support interactive software applications, also when handling finely spaced grids. This behaviour is due to the methods being designed to efficiently exploiting the causality of a propagating front.

For anisotropic problems, the GPU implementations of LAS and SOLAS spend on the average around 96% of the time on computing, 3-4% on synchronisation, and less than 1% on list building. For isotropic problems, the computations take around 90% of the total time on the GPUs. Computations dominate even more on CPUs where 98-99% of the total time is spent on subdomain computations and the remaining time is used for synchronisation. List creation is then less than 0.1% of the total time, and is therefore negligible. As to be expected, the synchronisation of values is slightly more costly for SOLAS than for the LAS algorithm.

Regardless of the test cases, the GPU implementations offer consistently faster computations than the CPU implementation running on up to 16 cores. The speedup factors when moving from 16 cores to the GPU ranges from 3 to 8 with an average of 6. For single

precision computations, this speedup factor varies from 6 to 16 with an average of 10. LAS is preferred for the isotropic or mildly anisotropic problems, whereas SOLAS is the best choice for problems with complicated velocity profiles.

3.2 Single versus double precision arithmetics

The difference in peak performance between single and double precision of the SandyBridge CPU assumes a perfectly vectorised code. Unfortunately, the large amount of logical branching prohibits a vectorisation of the computationally dominating update step. By hardware design, GPUs compute substantially faster in single precision than in double precision. In particular, Nvidia's Kepler architecture, such as our K20 platform detailed in Table 2, is able to perform three times as many operations per second in single precision than in double precision. In addition to the raw processing capabilities, values represented in single precision need only half the memory space that values stored in double precision need. Therefore, one can fit larger data sets into the limited memory of a GPU if single precision is used. Data-intensive applications will thereby get an additional improvement of performance due to better utilisation of fast memory and less shuffling of data between the GPU and the host computer. Across the range of synthetic problems investigated in Section 3.1, we have observed an average speedup of 2.1 when comparing single and double precision computations on the K20 platform. In comparison, the corresponding speedup of 1.2 for the 16-cores SandyBridge CPU is rather modest. On both platforms, these speedup factors are independent of the solvers. This observation agrees with the fact that the total time spent by the algorithms is heavily dominated by arithmetic operations, rather than memory transfers and synchronisation.

Given the advantage of single precision arithmetics, it is relevant to investigate whether computations in single precision can provide a satisfactory level of numerical accuracy when solving problems of the form (3). Empirically, we have observed that the single precision solution, T_f, is practically identical to the double precision solution, T_d. For the largest grid, $N = 504^3$, we have observed that $\|T_d - T_f\|_{L_2} = O(10^p)$, where $p = -7$ for Ex_A, $p = -6$ for Ex_A and Ex_D, and $p = -4$ for the complicated cases Ex_B and Ex_C. By the triangle inequality, we have

$$\begin{aligned} \|T - T_f\|_{L_2} &= \|T - T_f + T_d - T_d\|_{L_2} && (14) \\ &\le (1+\varepsilon)\|T - T_d\|_{L_2}, && (15) \end{aligned}$$

where $\varepsilon = \|T_d - T_f\|_{L_2} / \|T - T_d\|_{L_2}$. That is, if $\varepsilon < 1$, the single precision error will be of the same order as for the computations in double precision. We have estimated ε empirically for Ex_A, which have analytic solutions. Depending on the problem and the grid size, this entity stays in the range from 10^{-5} to 10^{-3} and is thus significantly smaller than one. Moreover, we have computed the ratio $\beta = \|T_d - T_f\|_{L_2} / \|T_d\|_{L_2}$, which is obtainable also for problems that can not be solved analytically. We have then observed that β stays between 10^{-5} and 10^{-8} for Ex_A-Ex_D across all grid sizes, and that β gets smaller as N increases, thus indicating convergence.

Based on these empirical studies, we conclude that single precision computations provide high enough accuracy when using the algorithms proposed in this paper with first order stencils to simulate front propagation based on (3). More subdomains are reactivated for computation when using double precision arithmetics than in the case of single

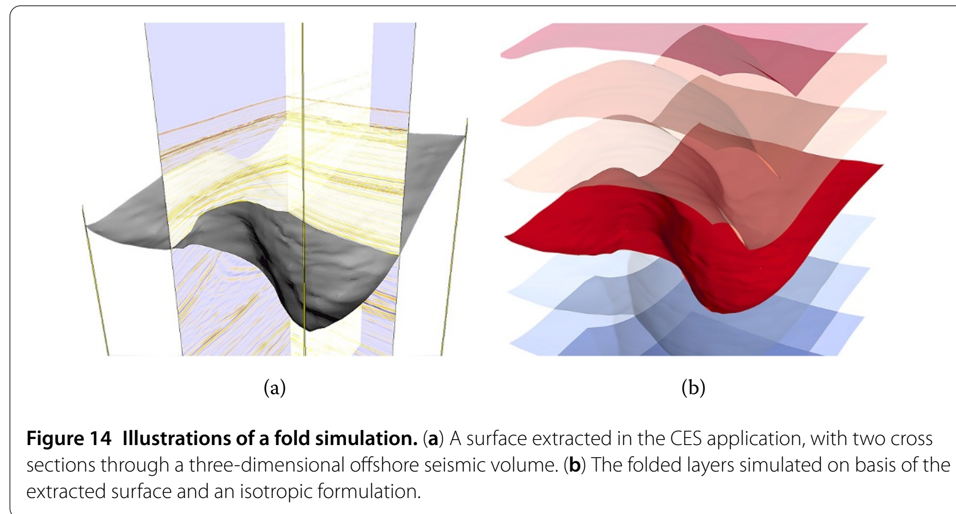


Figure 14 Illustrations of a fold simulation. (**a**) A surface extracted in the CES application, with two cross sections through a three-dimensional offshore seismic volume. (**b**) The folded layers simulated on basis of the extracted surface and an isotropic formulation.

precision. A subdomain is only reactivated when it receives a new value. Since the smallest noticeable change in single precision is effectively 10^{-7}, the subdomains with smaller changes will not be reactivated in single precision, but only in double precision computations. The increase of reactivations contributes to the increase of time needed by the solvers in double precision compared to single precision. Note that a subdomain could be reactivated only if the new value t_{new} differs from the old value t_{old} more than a chosen threshold-limit in order to limit the amount of reactivations. In other words, to reduce the number of subdomain computations, a subdomain can be activated only if $t_{\text{old}} - t_{\text{new}} > c$ for some empirically chosen value c. In all our numerical experiments we have used $c = 0$.

3.3 Fold simulation

Using the CES software, a user can easily segment structured surfaces from seismic data sets by autotracking. Figure 14(a) shows an autotracked surface in the CES software, extracted from a three-dimensional seismic volume as indicated by the transparent seismic cross sections. The physical domain measures 6700, 4975, and 11055 meter along the x, y and z axes, respectively. Values in the nodes immediately above and below the extracted horizon were computed in CES for grids with 256^3 and 512^3 nodes, respectively. These values were then used as boundary conditions in our numerical experiments. Parallel folded layers were simulated by solving the isotropic eikonal equation for the different grids. An example of the computed solution is visualised in Figure 14(b).

We have compared our new algorithms with the CES solver. The fold simulation solver in CES is based on a tracking algorithm, and is therefore sequential. When measuring the performance of the CES solver, we used a 4 core, 3.5 GHz Intel Ivy Bridge Core i7-3770K CPU with 8MB L3 cache. This is the fastest platform available that can run the CES software. Applying the new algorithms on K20 and SandyBridge, we have measured the computing times for both single and double precision arithmetics. Because of the sizing of subdomains, the grids used by the new algorithms are slightly larger than the grids used by the CES solver. The new grids consist of 266^3 and 518^3 nodes, respectively. These grids represent from 343,000 to more than 138 million unknowns. Table 3 shows the computing time for double and single precision arithmetics, together with the speedup achieved by replacing the Ivy-based CES solver. The speedup factors are prefixed with the symbol '$\times$.'

Table 3 Comparison of computing times (in seconds) for the CES solver on Ivy and LAS and SOLAS solver on SandyBridge and K20, including the speedup factors ('×') relative to CES.

Solver	Device	Double precision				Single precision			
		266^3		518^3		266^3		518^3	
CES	Ivy	27.68	×1.0	354.9	×1.0	24.91	×1.0	319.42	×1.0
LAS	K20	0.87	×32.0	6.2	×57.3	0.44	×57.2	2.99	×107.0
SOLAS	K20	0.84	×33.1	6.0	×59.6	0.43	×58.6	2.98	×107.2
LAS	SandyBridge	5.22	×5.3	36.0	×9.9	4.31	×5.8	28.46	×11.2
SOLAS	SandyBridge	4.66	×5.9	32.8	×10.8	3.81	×6.5	26.67	×12.0

The CES time for single precision computations has been empirically estimated as 90% of the CES time for double precision.

Reviewing the results, we observe that the difference between solution times for LAS and SOLAS are negligible when run on the same platform and with the same arithmetic resolution. We also observe that the speedup is higher for the GPU platforms than for the CPU platforms. As noted in Section 3.2, there is no difference in solution quality between single and double precision computations. However, the computing times are considerably lower in single precision computations, especially on the GPU. The computing time for the GPU is roughly halved when moving from double to single precision computations. For the largest grid in single precision, the K20 implementation of SOLAS is more than 100 times faster than the CES solver on Ivy.[b] We do not expect the speedup to be as impressive in all fold simulations. However, the user experience is significantly improved when fold simulations are performed with the new algorithms. Even on the largest grids, the computational time is a few seconds instead of minutes when LAS or SOLAS uses K20 instead of the current solution method. Based on this and several other numerical experiments with seismic field data, the proposed algorithms are now being implemented in the industrial code.

The task of simulating a folded volume is rather different from the synthetic examples discussed initially. For fold simulations, the initial geological surface defining the boundary condition will contain a significant number of subdomains. Because of this, the list used for subdomain scheduling will be quite long already from the start. When this list is long, it is easier to make full use of the available computing resources, thus resulting in a substantial effect of parallelisation. For simpler problems, where the boundary condition is represented by one or a few point sources, only a few subdomains will be scheduled as active. Thus, some of the available computing resources will be idling until later stages of the solution process when the lists grow longer. Since GPUs have an intrinsic capability of processing a large number of subdomains simultaneously, the simulation of geological folds based on LAS or SOLAS is particularly suitable for GPUs.

4 Conclusions

Two new parallel solvers are presented, together with several numerical experiments conducted on both multicore CPUs and GPUs. All solvers use a domain decomposition approach, where each subdomain is surrounded by a layer of ghost nodes. A subdomain needs to be processed only if its local boundary condition has changed since the previous computation. Due to this observation, a computed subdomain is locked for further computations until its boundary condition changes, and the subdomain is reactivated.

The presented algorithms differ mainly in the way subdomains are scheduled for computations. LAS places all active subdomains in a list, and updates the subdomains in this

list in parallel. The GPU implementation of LAS builds the lists on the CPU, but performs all computations on the GPU. SOLAS is very similar to LAS, but creates the lists slightly differently to enforce a more causal ordering of the subdomain computations.

From empirical studies, the sequential solution times seem to scale linearly with the total number of nodes. The parallel solvers appears to scale somewhat superlinearly, accredited to the increase in parallel processing efficiency obtained for larger grids. Some algorithmic details are crucial for achieving good performance. For instance, if a schedule is only used once by LAS and SOLAS, many more and longer lists are built. For anisotropic cases, repeated activations between neighbouring subdomains are more common than for isotropic cases. It is therefore important not to enforce a too strict ordering of subdomains. For some examples the LAS ordering is not very different from the SOLAS ordering, and the extra overhead of the SOLAS ordering is then not worthwhile.

We have investigated the potential for accelerating an industrial software for simulation of geological folding by introducing new algorithms for numerical solution of static Hamilton-Jacobi equations. The computing times are reduced from several minutes to seconds, enabling the software to be used interactively even for large three-dimensional data sets.

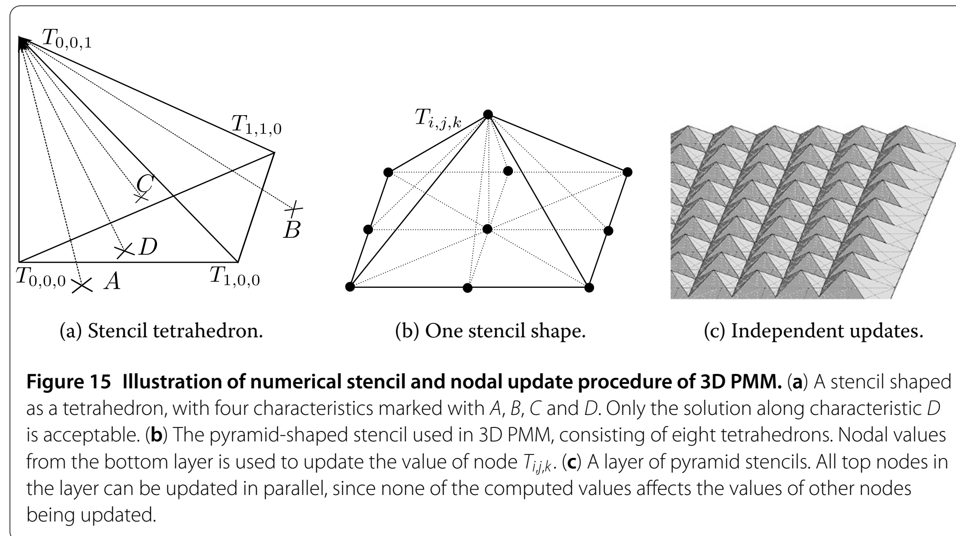
Possible extensions of the presented work

In our implementation, the computing and synchronisation procedures are entirely completed before the list building starts. Future implementations can overlap the building of new schedules with the processing of previous schedules. That is, one processor can build the new list while other processors reuse the old list. This will result in reduced overhead of list building and processor synchronisation. On the GPU, the copying of data to and from the device can be done asynchronously with the computations, and the utilisation of the CPU will increase.

The ghost nodes assure that subdomains can be updated simultaneously by different processors. No data is communicated between subdomains during the update procedure. Therefore, a parallel implementation iterates through the grid in the same way as a sequential implementation. The computed solutions are independent of how many processors are used, and the computed solution is always of good quality.

Ghost nodes carry the drawback of needing a synchronisation procedure. Before a synchronisation of nodal values, all computations must be finalised and the processing threads must be synchronised. The pHCM method avoids synchronisations of threads by allowing memory to be shared between different threads [34], resulting in so called *false sharing*. Because of the stable upwind stencils, the pHCM method converge to the correct solution. However, the algorithmic behaviour changes noticeably depending on by which processor, and in which order, memory is accessed. Similarly to pHCM and FIM, the ghost nodes can be avoided in LAS and SOLAS. Some minor changes of the reactivation system is then needed, and the synchronization can be entirely avoided. Values are instead 'synchronised' between adjacent subdomains directly when computed. We expect the performance when running on a few threads to improve significantly, but the number of iterations will change with the number of threads.

Other methods than 3D PMM should be investigated for the nodal updates within each subdomain. It is difficult to foresee which type of methods would be better suited. Iterative methods such as SOFI, FIM and Locked Sweeping [40] might perform well since the



(a) Stencil tetrahedron. (b) One stencil shape. (c) Independent updates.

Figure 15 Illustration of numerical stencil and nodal update procedure of 3D PMM. (**a**) A stencil shaped as a tetrahedron, with four characteristics marked with *A*, *B*, *C* and *D*. Only the solution along characteristic *D* is acceptable. (**b**) The pyramid-shaped stencil used in 3D PMM, consisting of eight tetrahedrons. Nodal values from the bottom layer is used to update the value of node $T_{i,j,k}$. (**c**) A layer of pyramid stencils. All top nodes in the layer can be updated in parallel, since none of the computed values affects the values of other nodes being updated.

subdomains are rather small. Second order stencils can be implemented with almost no algorithmic alterations, thereby increasing accuracy further.

Appendix 1: Efficient formulation of the eikonal 'pyramid' stencil

The most common way to propagate a front numerically is through use of semi-Lagrangian [53] or conditional upwind [16, 30] discretisations. The semi-Lagrangian approach searches through all possible characteristics for the one yielding the minimal distance. The conditional upwind stencils first solve for a solution, and thereafter accepts the new estimate only if the associated characteristic originates from within the spatial grid element, defined as the convex hull of the nodes supporting the stencil. Both approaches are identical on eikonal formulations but may differ in anisotropic cases [30]. If the characteristic curve originate from outside of the element, the new estimate is not based on upwind values, and should not be accepted. Due to the isotropic nature of the eikonal problem one can often formulate upwind conditions for the acceptance of a new estimate [16]. These conditions can significantly reduce the amount of computations needed to update a nodal value.

In this appendix we present an efficient stencil for the eikonal equation, derived as a conditional upwind stencil. The stencil element is shaped as a tetrahedon defined by the corner nodes $\mathcal{X}_{i,j,k}$ with values $T_{i,j,k} = T(\mathcal{X}_{i,j,k})$, as visualised in Figure 15(a). The value $T_{0,0,1}$ for the top node is to be updated using the values $T_{0,0,0}$, $T_{1,0,0}$ and $T_{1,1,0}$. Four characteristic curves (dashed arrows) are shown with their entrance points on the *bottom* surface defined by nodes $\mathcal{X}_{i,j,0}$. Only the solution associated with the characteristic curve D would be acceptable since the other curves originate from outside the stencil element. When updating a node in 3D PMM, the tetrahedron shape is used eight times in a configuration that resembles a *pyramid*, shown in Figure 15(b).

In line with general consensus, the velocity F is assumed to be constant within the element. When updating $T_{0,0,1}$, as in Figure 15(a), only nodes with strictly smaller T values than $T_{0,0,1}$ are used. A larger value implies that the front moves away from $\mathcal{X}_{0,0,1}$, and towards the position of the larger value. Initially we assume that no new value is found, that

is

$$t_{\text{new}} \leftarrow T_{0,0,1}. \tag{16}$$

From the characteristic equations (6) we know that the characteristic coincides with the gradient $\mathbf{n}$, and since $\mathbf{n} = \nabla T/\|\nabla T\| = F\nabla T$, we get the following estimates

$$\mathbf{n}_x \leftarrow F\frac{T_{1,0,0} - T_{0,0,0}}{dx}, \qquad \mathbf{n}_y \leftarrow F\frac{T_{1,1,0} - T_{1,0,0}}{dy}. \tag{17}$$

These estimates and the assumption of constant velocity correspond to modelling an arriving front as planar [29]. If $\mathbf{n}_y > 0$, the ray reaching $\mathcal{X}_{0,0,1}$ cuts the bottom surface outside of the tetrahedron, in areas with negative i offset (ray A). The shortest distance from within the tetrahedron then originates from the side with $i = 0$, that is, the new estimate is given from a two-dimensional diagonal stencil solution $\mathtt{s}_{2D}(\mathcal{X}_{0,0,0}, \mathcal{X}_{1,0,0})$ [16]. A similar observation holds for $\mathbf{n}_x$; in order for the characteristic to cut below the line connecting $\mathcal{X}_{0,0,0}$ and $\mathcal{X}_{1,1,0}$, the condition $\mathbf{n}_x\, dy < \mathbf{n}_y\, dx$ must hold. Otherwise (ray C), the minimal new estimate is $\mathtt{s}_{2D}(\mathcal{X}_{0,0,0}, \mathcal{X}_{1,1,0})$. These conditions results in the following upwind conditions

$$\mathtt{if}\ (\mathbf{n}_y > 0)\big\{t_{\text{new}} \leftarrow \min\big(t_{\text{new}}, \mathtt{s}_{2D}(\mathcal{X}_{0,0,0}, \mathcal{X}_{1,0,0})\big)\big\}, \tag{18}$$

$$\mathtt{else\ if}\ (\mathbf{n}_x\, dy > \mathbf{n}_y\, dx)\big\{t_{\text{new}} \leftarrow \min\big(t_{\text{new}}, \mathtt{s}_{2D}(\mathcal{X}_{0,0,0}, \mathcal{X}_{1,1,0})\big)\big\}. \tag{19}$$

Next, $\mathbf{n}_z$ is estimated using the unit argument of the normal, $\mathbf{n}_z^2 = 1 - \mathbf{n}_x^2 - \mathbf{n}_y^2$. To make sure that the ray originates from a location with $i \leq 1$ (to remove ray B), the following upwind condition is needed

$$\mathtt{else\ if}\ \big(\mathbf{n}_z^2\, dx^2 < \mathbf{n}_x^2\, dz^2\big)\big\{t_{\text{new}} \leftarrow \min\big(t_{\text{new}}, \mathtt{s}_{2D}(\mathcal{X}_{1,0,0}, \mathcal{X}_{1,1,0})\big)\big\}. \tag{20}$$

This condition also assures that $\mathbf{n}_z^2 > 0$. If all upwind conditions are true, the characteristic curve that cuts node $\mathcal{X}_{0,0,1}$ enters the tetrahedron through its base (ray D), and the new arrival time estimate is

$$\mathtt{else}\ \left\{t_{\text{new}} \leftarrow T_{0,0,0} + \mathbf{n}_z\frac{dz}{F}\right\}. \tag{21}$$

Details for the anisotropic stencil

The anisotropic stencil for equation (3) uses a direct conditional upwind discretisation in our implementation. Note that nodes upwind of the new solution point can be used in the anisotropic stencil, as long as the point where the characteristic curve enters the stencil lies upwind of the updated node [30].

Appendix 2: Subdomains

Subdomains are surrounded by a layer of virtual nodes referred to as *ghost nodes*. These ghost nodes are copies of nodes in adjacent subdomains. Since each ghost node exists at multiple locations, their values must be kept equal at all locations. We refer to the procedure of comparing such nodal values as synchronisation. Figure 16 is a 2D illustration

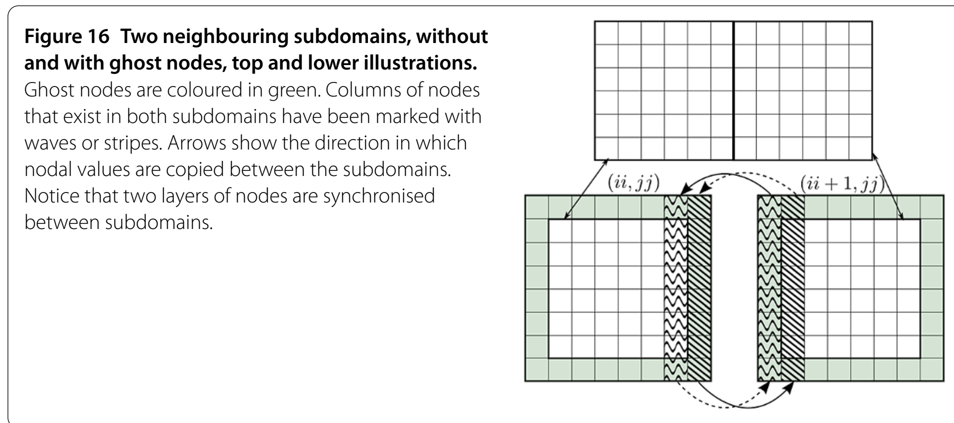


Figure 16 Two neighbouring subdomains, without and with ghost nodes, top and lower illustrations. Ghost nodes are coloured in green. Columns of nodes that exist in both subdomains have been marked with waves or stripes. Arrows show the direction in which nodal values are copied between the subdomains. Notice that two layers of nodes are synchronised between subdomains.

of subdomains in their logical grid (top) and with the ghost nodes marked by light green (bottom). Columns of nodes that exists in both subdomains have been marked with waves and stripes in the lower figure.

Nodes belonging to a subdomain, including the ghost nodes, will during computations only be accessed, and written to, by the thread(s) updating that subdomain. Two subdomains can be computed simultaneously without any risk of memory interference since all data is private to the updating thread(s). Ghost nodes have been used and discussed in relation to front propagation problems in other works [23, 24]. Our use is different, and details regarding the synchronisation of ghost nodes are given in Section B.1. The computation of a subdomain using 3D PMM is discussed in Section B.2.

B.1 Synchronisation of nodal values

Consider the two-dimensional case when comparing values from nodes in subdomain (ii, jj) to values in subdomain $(ii+1, jj)$, as shown with the two lower arrows in Figure 16. When a computed nodal value, t_s in (ii, jj) is compared to its copy t_t in $(ii+1, jj)$, the copy is only changed if its value is larger than the computed nodes, that is $t_t = \min(t_t, t_c)$. If at least one node of the inner part of $(ii+1, jj)$ (lower solid arrow in Figure 16) receives a new value, the subdomain is activated by releasing the compute lock, $\texttt{CL}(ii+1, jj) \leftarrow \texttt{Open}$. A subdomain that receives new values only on its out-most boundary (dashed arrow) does not need to be activated.

In SOLAS we need the minimum nodal value that activates a subdomain. The minimum activation value is computed efficiently in our GPU implementation using parallel reduction. Note that if a tetrahedron (diagonal) stencil is used, the corners will be shared and updated by eight (four) blocks in three (two) spatial dimensions. However, by synchronising values in one direction at a time, the corner nodes receive correct solution values.

B.2 Computation of a subdomain

In all our numerical experiments, 3D PMM has been used for the computation of new nodal values in the subdomains. Here we give a brief description of the method [30, 54].

A specific *pyramid* shape of the stencil is used, as shown in Figure 15(b), where the value $T_{i,j,k}$ for the top node of the pyramid is updated. 3D PMM computes *layers* of nodes, as illustrated in Figure 15(c). Nodes in a layer have one index (i, j or k) in common. All nodes in one layer can be updated entirely in parallel since there are no dependencies between the computed nodes, since the value of a top node does not depend on other top

Algorithm 17 *i*SubSweep(sd,dir)

comment: Subsweep in a subdomain using in 3D PMM.
Stencil($B_{\mathcal{X}}$) returns an estimate using values on nodes in the set $B_{\mathcal{X}}$.

```
T ← T values in subdomain sd
if dir is 1
  then order ← 0 to b_x
if dir is −1
  then order ← b_x + 1 downto 1
for i ← order
  do { for each j ← 0 to b_y + 1 and k ← 0 to b_z + 1, in parallel
       do { B_X ← { All X_{i,j±a,k±b}, a ∈ {0,1}, b ∈ {0,1} within sd }
            t_new ← Stencil(B_X)
            if t_new < T_{i+dir,j,k}
              then T_{i+dir,j,k} = t_new
```

Algorithm 18 ComputeSubdomain(sd)

comment: Computation of nodal values in subdomain sd with 3D PMM.

```
T ← T values in subdomain sd
if CL(sd) is Open)
  then { CL(sd) ← Locked
         for dir ← 1 and −1
           do { iSubSweep(sd, dir)
                jSubSweep(sd, dir)
                kSubSweep(sd, dir)
```

nodes, see Figure 15(c). After processing a layer, the computed values are used as bottom-pyramid values to compute the next layer of nodes. An iteration through the domain for an increasing or decreasing common index is referred to as a *subsweep*. A set of all six subsweeps, increasing and decreasing i, j, k indices, are referred to as a *sweep*.

Conceptually, the ghost nodes in a subdomain define the boundary condition of the subdomain. In our implementation we compute new values also for the ghost nodes. When the ghost nodes are updated, no values from other subdomains are used in the computations. Therefore, 'half'-pyramids are used at the sides, and 'quarter'-pyramids in the corners. Pseudocode for a subsweep in the i direction is given in Algorithm 17, where the layers defined by $i = 0$ and $i = b_x + 1$ contains only ghost nodes.

A subdomain that is activated receives (at least) one surface with two layers of nodes in the synchronisation. Assuming that these 'thick' boundary values are correct, the characteristic curves are extrapolated to the rest of the subdomain in the sweeps as new solution values are computed. These 'thick' boundary conditions explain why the solution converges fast. On a local scale, the characteristic curves are straight lines, or only slightly curved, and the solution within a subdomain will often converge in only one sweep. The final solution remains the same in all our examples whether one, two, or more sweeps are

used when computing subdomains. Since the total computational time is minimal when only one sweep is used, we have used only one sweep in the numerical experiments.

After processing a subdomain, the compute lock is closed. As explained above, it can be opened again during a synchronisation. Algorithm 18 presents pseudocode for the computation of new nodal values in subdomain `sd`, using 3D PMM. This algorithm, named ComputeSubdomain(), is called from the procedure in Algorithm 3.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
The main idea of this paper was proposed by TG, based on earlier research by ØH. TG and MS contributed with the implementation and numerical experiments. TG and AM performed the analysis, and prepared the manuscript initially. All authors contributed to the manuscript, and have read and approved the final manuscript.

Acknowledgements
The presented work was funded by Statoil ASA through the Akademia program, and the Research Council of Norway under grant 202101/I40. The work has been conducted at Kalkulo AS, a subsidiary of Simula Research Laboratory.

Endnotes

[a] Here, the term 'efficiency' refers to the ratio between observed speedup and the theoretically possible speedup. For instance, the LAS speedup of 6.0 for $\mathbf{Ex}_A$ running eight cores gives an efficiency of $6.0/8 \approx 75\%$.

[b] The CES software is only available on the Ivy platform running in double precision. Experiments indicate that single precision computations on Ivy runs at about 90% of the time used in double precision. Therefore, the CES times for single precision experiments are constructed by multiplying the observations for double precision with a factor 0.9.

Received: 24 February 2014 Accepted: 30 June 2014 Published: 24 Jul 2014

References

1. Sethian JA: *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. 2nd edition. Cambridge: Cambridge University Press; 1999.
2. Rawlinson N, Hauser J, Sambridge M: **Seismic ray tracing and wavefront tracking in laterally heterogeneous media**. *Adv. Geophys.* 2009, **49**:203-267.
3. Gillberg T: **A semi-ordered fast iterative method (SOFI) for monotone front propagation in simulations of geological folding**. In *MODSIM2011, 19th International Congress on Modelling and Simulation*; 2011:641-647.
4. Gillberg T: **Fast and accurate front propagation for simulation of geological folds**. *PhD thesis*. University of Oslo; 2013.
5. Natvig JR, Lie K-A: **Fast computation of multiphase flow in porous media by implicit discontinuous Galerkin schemes with optimal ordering of elements**. *J. Comput. Phys.* 2008, **227**(24):10108-10124.
6. Kornhauser ET: **Ray theory for moving fluids**. *J. Acoust. Soc. Am.* 1953, **25**(5):945-949.
7. Podvin P, Lecomte I: **Finite difference computation of traveltimes in very contrasted velocity models: a massively parallel approach and its associated tools**. *Geophys. J. Int.* 1991, **105**:271-284.
8. Li S, Mueller K, Jackowski M, Dione D, Staib L: **Physical-space refraction-corrected transmission ultrasound computed tomography made computationally practical**. In *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2008*. Berlin: Springer; 2008:280-288. [*Lecture Notes in Computer Science*, vol. 5242.]
9. Wallman M, Smith NP, Rodriguez B: **A comparative study of graph-based, eikonal, and monodomain simulations for the estimation of cardiac activation times**. *IEEE Trans. Biomed. Eng.* 2012, **59**(6):1739-1748.
10. Jeong W-K, Fletcher PT, Tao R, Whitaker R: **Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware Hamilton-Jacobi solver**. *IEEE Trans. Vis. Comput. Graph.* 2007, **13**(6):1480-1487.
11. Rawlinson N, Sambridge M: **Multiple reflection and transmission phases in complex layered media using a multistage fast marching method**. *Geophysics* 2004, **69**(5):1338-1350.
12. Petersen SA, Hjelle Ø, Hustoft S, Haubiers M: **Process based data-restoration and model-reconstruction workflow for seismic interpretation and model building**. In *EAGE 74th Conference & Exhibition, Extended Abstracts*; 2012.
13. Petersen SA, Hjelle Ø: **Earth recursion, an important component in shared earth model builders**. In *EAGE 70th Conference & Exhibition, Extended Abstracts*; 2008.
14. Cerveny V: *Seismic Ray Theory*. Cambridge: Cambridge University Press; 2001.
15. Hjelle Ø, Petersen SA: **A Hamilton-Jacobi framework for modeling folds in structural geology**. *Math. Geosci.* 2011, **43**(7):741-761.
16. Gillberg T, Hjelle Ø, Bruaset AM: **Accuracy and efficiency of stencils for the eikonal equation in earth modelling**. *Comput. Geosci.* 2012, **16**(4):933-952.
17. Chopp D: **Recent advances in the level set method**. In *Handbook of Biomedical Image Analysis*. Edited by Micheli-Tzanakou E, Suri JS, Wilson DL, Laxminarayan S. New York: Springer; 2005:201-256.
18. Tsitsiklis JN: **Efficient algorithms for globally optimal trajectories**. *IEEE Trans. Autom. Control* 1995, **40**(9):1528-1538.
19. Cacace S, Cristiani E, Falcone M: **Requiem for local single-pass methods solving stationary Hamilton-Jacobi equations**? 2013 [arXiv:1301.6775]
20. Sethian JA, Vladimirsky A: **Ordered upwind methods for static Hamilton-Jacobi equation: theory and algorithms**. *SIAM J. Numer. Anal.* 2003, **41**(1):325-363.

21. Hjelle Ø, Petersen SA, Bruaset AM: A numerical framework for modeling folds in structural geology. *Math. Geosci.* 2013, **45**(3):255-276.
22. Zhang J, Huang Y, Song L-P, Liu Q-H: Fast and accurate 3-D ray tracing using bilinear traveltime interpolation and the wave front group marching. *Geophys. J. Int.* 2011, **184**(3):1327-1340.
23. Herrmann M: A domain decomposition parallelization of the fast marching method. In *Annual Research Briefs 2003*. Stanford: Center for Turbulence Research, Stanford University; 2003:213-225.
24. Tugurlan MC: Fast marching methods - parallel implementation and analysis. *PhD thesis*. Louisiana State University and Agricultural and Mechanical College; 2008.
25. Zhao H-K: A fast sweeping method for eikonal equations. *Math. Comput.* 2004, **74**(250):603-627.
26. Qian J, Zhang Y-T, Zhao H-K: A fast sweeping method for static convex Hamilton-Jacobi equations. *J. Sci. Comput.* 2007, **31**(1-2):237-271.
27. Zhao H-K: Parallel implementations of the fast sweeping method. *J. Comput. Math.* 2007, **25**(4):421-429.
28. Detrixhe M, Gibou F, Min C: A parallel fast sweeping method for the eikonal equation. *J. Comput. Phys.* 2013, **237**:46-55.
29. Weber O, Devir YS, Bronstein AM, Bronstein MM, Kimmel R: Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Trans. Graph. (TOG)* 2008, **27**(4):104:1-104:16.
30. Gillberg T, Sourouri M, Cai X: A new parallel 3D front propagation algorithm for fast simulation of geological folds. *Proc. Comput. Sci.* 2012, **9**:947-955. Proceedings of the International Conference on Computational Science, ICCS 2012.
31. Gillberg T, Hjelle Ø, Bruaset AM: A parallel 3D front propagation algorithm for simulation of geological folding on GPUs. In *EAGE 74th Conference & Exhibition, Extended Abstracts*; 2012.
32. Gremaud PA, Kuster CM: Computational study of fast methods for the eikonal equation. *SIAM J. Sci. Comput.* 2006, **27**(6):1803-1816.
33. Hysing S-R, Turek S: The eikonal equation: numerical efficiency vs. algorithmic complexity on quadrilateral grids. In *Proceedings of ALGORITMY*; 2005.
34. Chacon A, Vladimirsky A: A parallel heap-cell method for eikonal equations; 2013 [arXiv:1306.4743]
35. Chacon A, Vladimirsky A: Fast two-scale methods for eikonal equations. *SIAM J. Sci. Comput.* 2012, **34**(2):A547-A578.
36. Glover F, Klingman D, Phillips N: A new polynomially bounded shortest path algorithm. *Oper. Res.* 1985, **33**(1):65-73.
37. Dejnozkova E, Dokladal P: A parallel algorithm for solving the eikonal equation. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003*. *Volume* 3; 2003:325-328. IEEE.
38. Jeong W-K, Whitaker RT: A fast iterative method for eikonal equations. *SIAM J. Sci. Comput.* 2008, **30**(5):2512-2534.
39. Fu Z, Jeong W-K, Pan Y, Kirby RM, Whitaker RT: A fast iterative method for solving the eikonal equation on triangulated surfaces. *SIAM J. Sci. Comput.* 2011, **33**(5):2468-2488.
40. Bak S, McLaughlin J, Renzi D: Some improvements for the fast sweeping method. *SIAM J. Sci. Comput.* 2010, **32**(5):2853-2874.
41. Jeong W-K, Whitaker RT: A fast iterative method for a class of Hamilton-Jacobi equations on parallel systems. *Technical report*. University of Utah; 2007.
42. Tanenbaum AS: *2. Modern Operating Systems*. 3rd edition. Upper Saddle River: Prentice Hall; 2007.
43. Kjolstad FB, Snir M: Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ParaPLoP '10. New York: ACM; 2010:4-149.
44. Rivera G, Tseng C-W: Tiling optimizations for 3D scientific computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. Supercomputing '00. Washington: IEEE Computer Society; 2000:1-23.
45. Nvidia: CUDA C programming guide; 2012 [http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html]
46. Michéa D, Komatitsch D: Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophys. J. Int.* 2010, **182**(1):389-402.
47. Nvidia: CUDA C Best Practices Guide; 2012 [http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html]
48. Nvidia: CUDA driver API; 2013 [http://docs.nvidia.com/cuda/cuda-driver-api/index.html]
49. OpenMP.org: OpenMP application program interface; 2013 [http://openmp.org/]
50. Nvidia: What is CUDA; 2013 [https://developer.nvidia.com/what-cuda]
51. Nvidia: CUDA occupancy calculator; 2012.
52. Kadlec B, Dorn G: Leveraging graphics processing units (GPUs) for real-time seismic interpretation. *Lead. Edge* 2010, **29**(1):60-66.
53. Cristiani E, Falcone M: Fast semi-Lagrangian schemes for the eikonal equation and applications. *SIAM J. Numer. Anal.* 2007, **45**(5):1979-2011.
54. Sourouri M: A parallel front propagation method: simulating geological folds on parallel architectures. *Master's thesis*. University of Oslo; 2012.

10.1186/2190-5983-4-10
Cite this article as: Gillberg et al.: Parallel solutions of static Hamilton-Jacobi equations for simulations of geological folds. *Journal of Mathematics in Industry* 2014, 4:10