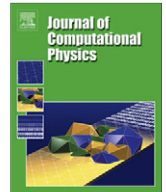
Journal of Computational Physics 237 (2013) 46–55

Contents lists available at SciVerse ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp



A parallel fast sweeping method for the Eikonal equation

Miles Detrixhe [a], Frédéric Gibou [a,b], Chohong Min [c,*]

[a] Mechanical Engineering Department, University of California, Santa Barbara, CA 93106, United States
[b] Computer Science Department, University of California, Santa Barbara, CA 93106, United States
[c] Mathematics Department, Ewha Womans University, Seoul 120-750, Republic of Korea

ARTICLE INFO

Article history:
Received 14 December 2011
Received in revised form 28 November 2012
Accepted 29 November 2012
Available online 19 December 2012

Keywords:
Eikonal equation
Cuthill–McKee ordering
Parallel implementation
Fast sweeping method

ABSTRACT

We present an algorithm for solving in parallel the Eikonal equation. The efficiency of our approach is rooted in the ordering and distribution of the grid points on the available processors; we utilize a Cuthill–McKee ordering. The advantages of our approach is that (1) the efficiency does not plateau for a large number of threads; we compare our approach to the current state-of-the-art parallel implementation of Zhao (2007) [14] and (2) the total number of iterations needed for convergence is the same as that of a sequential implementation, i.e. our parallel implementation does not increase the complexity of the underlying sequential algorithm. Numerical examples are used to illustrate the efficiency of our approach.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

The Eikonal equation is fundamental in many applications including optimal control, computer vision [2,9], image processing [7,11], path planning [1,6], and interface tracking [8]. In n spatial dimensions, this equation reads:

$$\begin{aligned} |\nabla u(\mathbf{x})| &= f(\mathbf{x}) \quad \text{for } \mathbf{x} \in \Omega \subset \mathbb{R}^n, \\ u(\mathbf{x}) &= g(\mathbf{x}) \quad \text{for } \mathbf{x} \in \Gamma \subset \Omega, \end{aligned} \tag{1}$$

where u is the unknown, f a given "inverse velocity field" and g the value of u at an irregular interface Γ. Within the level-set technology, the Eikonal equation is used to "reinitialize" the level-set function, i.e. transform an arbitrary Lipschitz continuous function into a signed distance function. This intermediate step must be repeated several times in the course of a typical free boundary problem solution so efficient algorithms are important. Efficient solutions are also needed in the case of the aforementioned applications so that effective control strategies can be designed. There exist several fast algorithms to solve the Eikonal equation sequentially but parallel implementations are less common. Given the ever increasing need of efficiency and, in some cases, the limitation of sequential computer resources, it is desirable to develop an efficient parallel method for the Eikonal equation. This paper introduces a simple, yet efficient approach.

The fast marching method (FMM) [12] is a popular technique that updates nodes on a front in the order of propagating solutions. The method requires a sorting algorithm that is $O(\log N)$, where N is the number of grid points, giving the FMM an algorithmic complexity of $O(N \log N)$. The fast iterative method (FIM) [4] solves the Eikonal equation by iterating over a list of nodes, adding nodes as the list propagates and subtracting nodes as their solution's value converges. FIM has complexity $O(N)$ and can be implemented in parallel [5], but may not have optimal parallel efficiency on all problems (e.g. when the active list is small during much of the computation). The fast sweeping method (FSM) uses a nonlinear upwind difference

* Corresponding author. Tel.: +82 2 3277 2292.
E-mail address: chohong@ewha.ac.kr (C. Min).

0021-9991/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jcp.2012.11.042

scheme and alternating sweeping orderings of Gauss–Seidel iterations over the entire domain until convergence [13]. This process is $O(N)$ and the current state-of-the-art parallel implementation has been given in Zhao [14]. In this paper, we present a novel parallel fast sweeping method for solving Eq. (1) that is simple to implement and achieves speedup near $O(N/p)$ in the ideal case, where p is the number of threads.

In Section 2 we recall the serial FSM algorithm and the parallel implementation of Zhao [14] before we describe our parallel implementation. We then present numerical results and compare them to the work of Zhao [14] in Section 3. We draw our conclusions in Section 4.

2. Parallel fast sweeping algorithm

2.1. Fast sweeping method

In our exposition of the fast sweeping algorithm and its parallelization, we will restrict the problem to $\mathbb{R}^2$, but explain the extension to $\mathbb{R}^3$ when it is not straightforward. Consider a computational domain discretized into a grid with I and J nodes in the x- and y-directions, respectively. Let Γ be a one-dimensional interface describing the initial location from which the solution propagates. The FSM uses a Godunov upwind differencing scheme [9] on the interior nodes:

$$\left[\left(u_{i,j}^{new}-u_{xmin}\right)^{+}\right]^{2}+\left[\left(u_{i,j}^{new}-u_{ymin}\right)^{+}\right]^{2}=f_{i,j}^{2}h^{2},\quad \text{for } i=2,\ldots,I-1,\quad \text{and } j=2,\ldots,J-1, \tag{2}$$

where h is the grid spacing and the following notations are used:

$$u_{xmin}=\min(u_{i-1,j},u_{i+1,j}),\quad \text{and}\quad (x)^{+}=\begin{cases}x, & x>0,\\ 0, & x\leqslant 0.\end{cases}$$

The problem is initialized by assigning exact (or interpolated) values at the grid points nearest the interface Γ [3], and large positive values at all other grid nodes. The algorithm proceeds by sweeping through all the nodes and assigning a new value to u by solving Eq. (2) (see [13] for the explicit solution's formula) and updating the solution as $u_{i,j}=\min(u_{i,j},u_{i,j}^{new})$. Fig. 1 illustrates the fact that sweeping ordering directly leads to the propagation of the boundary data (on Γ) in the direction of the associated characteristics. Alternating sweeping ordering is used to ensure that the information is being propagated along all four (or eight in three dimensions) classes of characteristics. Zhao showed that this method converges in 2^n sweeps in $\mathbb{R}^n$ [13] and that convergence is independent of grid size. The natural choice for sweep directions is:

$$i=1:I,\ j=1:J,\quad \text{then}\quad i=I:1,\ j=1:J,\quad \text{then}\quad i=I:1,\ j=J:1,\quad \text{then}\quad i=1:I,\ j=J:1. \tag{3}$$

In the case where obstacles are present, such as in the case of Fig. 2, the process of four sweeps must be iterated until convergence. The algorithm is described in Algorithm 1 (left).

2.2. The parallel implementation of [14]

In [14], Zhao presented a parallel implementation of the FSM method described in Section 2.1: The strategy is to sweep through all four orderings simultaneously, assigning each ordering to a separate thread (see Fig. 5). This generates a separate temporary solution $u^{i,new},\ i=1,2,3,4$ for each thread. After each sweeping is finished, the data is synchronized according to

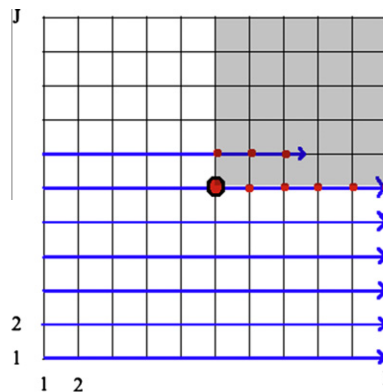


Fig. 1. Hypothetical distance function calculation from a source point (center). This figure is a snapshot of the calculation midway through sweeping ordering 1. Nodes with red dots indicate an updated solution. This illustrates that each ordering corresponds to a class of characteristic curve. It can be seen that ordering scheme 1 propagates information along the ↗ or first quadrant characteristics into the entire shaded (upper right) region.

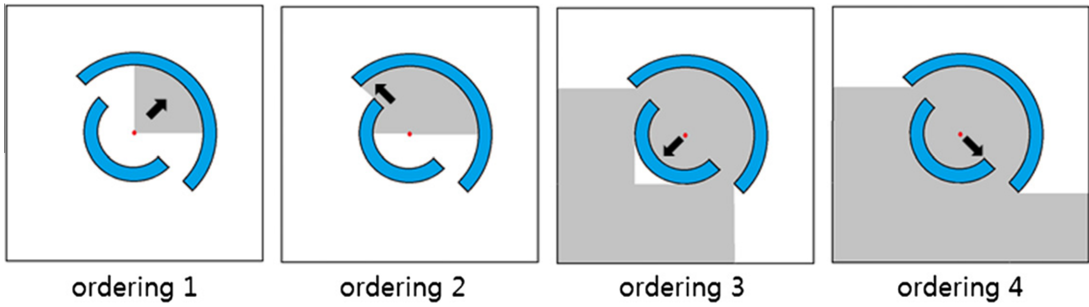


Fig. 2. Illustration of how data propagates from a source point (center, red) through the sequence of sweeps given by (3) in the case of the serial FSM. The blue arcs represent obstacles. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

$u_{i,j}^{new} = \min(u_{i,j}^{1,new}, u_{i,j}^{2,new}, u_{i,j}^{3,new}, u_{i,j}^{4,new})$ for all i,j. Algorithm 1 (center) gives a pseudocode. This method can generate up to four threads in two spatial dimensions and eight threads in three spatial dimensions. In order to utilize an arbitrary number of threads in parallel, Zhao [14] uses a domain decomposition strategy where the domain is split into rectangular subdomains so that computations in each subdomain can be carried out simultaneously. In the next section we present a highly parallel FSM method that does not require domain decomposition. Of course, we could in addition use a domain decomposition approach as in [14]. However, domain decomposition may in some cases limit the distance data can propagate in a single iteration and may lead to poor performance as the number of processors becomes large (i.e. the decomposed domain size becomes small). This notion motivates our choice of a benchmark method for comparison with the approach of Zhao [14] without domain decomposition.

2.3. Present method

Our parallel fast sweeping method follows the same procedure as that presented in Section 2.1 except for one significant change: the present work still uses four (resp. eight) sweeping orderings in two spatial dimensions (resp. three spatial dimensions) to represent all classes of characteristics, but chooses the direction of sweepings in a manner which allows for sets of nodes to be updated simultaneously. In turn, this leads to performances that do not plateau as the number of threads increases. An illustration is given in Fig. 5.

Specifically, we use a Cuthill–McKee type ordering [10], an example of which is depicted in Fig. 3. Here we define the level of a node (i,j) as `level=` $i+j$. In $\mathbb{R}^n$, this ordering allows for a $(2n+1)$-point stencil to be updated independently of any other points within a single level. Fig. 3 illustrates the ordering in two spatial dimensions and the inset depicts that the discretization is independent of any other nodes within a level (along a solid blue line in Fig. 3). This property allows for the update of Eq. (2) to be executed in parallel for all the nodes in one level with no threat of a data race or corrupted data.

In $\mathbb{R}^n$, a pseudocode for our parallel implementation is given in Algorithm 1 (right), where `update`$(u_{i,j})$ solves Eq. (2) at the grid node (i,j). First, the problem is initialized exactly as in the original FSM; note that this step is embarrassingly

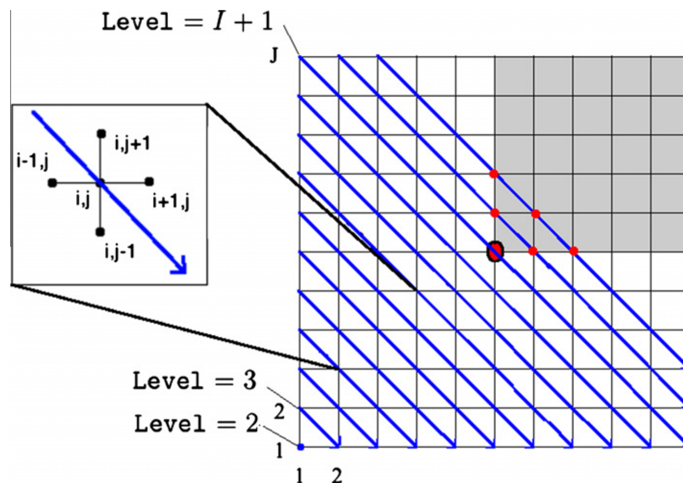


Fig. 3. Example of Cuthill–McKee sweeping ordering for a 2D problem (computation of the distance from a center source point). The inset illustrates that nodes along a level are uncoupled from all other nodes along that level. The data from the source point will propagate into the entire shaded region.

parallel. To iterate through the different sweeping directions, we first rotate the axes to correspond to the correct ordering, then sweep though the domain in parallel with the Cuthill–McKee ordering. The process is aborted when it has converged to within a desired tolerance. The notation `parallel for`, is borrowed from OpenMP and simply divides all the iterations of the loop among the available threads. For values of `level`, $\gg p$ (where p is the number of threads) the division of work among processors is very near equal and speedup is expected to be linear. When `level`, $< p$, the work cannot be shared by all processors and speedup suffers. For large problem, however, these cases become rare and we expect speedup to be near linear, giving the method a complexity $O(N/p)$, with N the total number of grid points.

Algorithm 1. FSM algorithms considered in this work

Serial FSM	Parallel implementation of [14]	Present work
initialize (u)	initialize (u)	initialize (u)
for (iter = 1:max_iter) do	for (iter = 1:max_iter) do	for (iter = 1:max_iter) do
for ordering=1 : 2^2	**parallel for** ordering=1 : 2^2	for ordering=1 : 2^2
rotate_axes (ordering)	$u^{\text{ordering}} = u$	rotate_axes (ordering)
for i = 1:I do	rotate_axes (ordering)	for level = 2:I + J do
for j = 1:J do	for i = 1:I do	I_1=max (1,level-J)
update (u_{ij})	for j = 1:J do	I_2=min (I, level-1)
	update $\left(u_{ij}^{\text{ordering}}\right)$	**parallel for** i= $I_1 : I_2$ do
	for i = 1:I do	j = level-i
	for j = 1:J do	update (u_{ij})
	$u_{ij} = \min\left(u_{ij}^1, u_{ij}^2, u_{ij}^3, u_{ij}^4\right)$	

2.4. Load balancing

We use the following strategy to assign groups of nodes to different threads in a balanced fashion in three spatial dimensions[1]: Referring to Fig. 4(a), let's assume that we want to distribute the load on 3 threads. Along the constant i-direction, the number of nodes involved are 2 nodes for $i = 1$ plus 2 nodes for $i = 2$ plus 3 nodes for $i = 3$ plus 2 nodes for $i = 4$ plus 1 node for $i = 5$. Those are represented in Fig. 4(b) (Step 1) by the black symbols. The second step seeks to iteratively split the nodes in groups of comparable sizes using a bisection approach. In the case of Fig. 4(b), the first iteration (Step 2a) groups the nodes for $i = 1$ and $i = 2$ together for a total of 4 nodes on one hand (red symbols), and the nodes for $i = 3,\ i = 4$ and $i = 5$ together for a total of 6 nodes on the other hand (blue symbols). Note that grouping the nodes for $i = 3$ with the first group would give a grouping of 7 nodes on one hand and a grouping of 3 nodes on the other hand, hence a bias towards one group. The process repeats iteratively, subdividing the first group into two subgroups of 2 nodes each; and subdividing the second group into a subgroup of 3 nodes and another of 2 + 1 nodes, as illustrated in Fig. 4(b) (Step 2b). This iteration process ends when the number of subgroup is greater or equal to the number of threads.

In the case where the number of subgroup is greater than the number of threads, one needs to collapse some of the subgroups together in such a way as to make the number of subgroups equal to the number of threads. We first locate the smallest subgroup, e.g. in the case of Fig. 4(b) (Step 3), we choose between one of the subgroups with 2 nodes by looking at which subgroup has two neighboring subgroups. In general, if more than one exists, we simply choose one randomly (i.e the first one encountered). Then we collapse with the smallest subgroup chosen, its neighboring subgroup with the less number of nodes (Step 4). In this fashion the number of nodes assigned to each thread is 4, 3 and 3, which gives a balanced load. We note that other approaches could be used and refer the interested reader to [10] and the references therein.

2.5. Pros and cons

An advantage of the parallel implementation of [14] is its simplicity. In the case where the number of dedicated threads is 4 (in 2D) or 8 (in 3D), straightforward modifications of the serial implementation will result in a speedup of the algorithm. Our method is less straightforward but offers significant advantages: (1) The updates are the same as those in the serial case, i.e. they use the newly computed values of previous updates. In turn the number of iterations needed in our parallel implementation is equal to that of the serial algorithm. This is in contrast with the update mechanism of [14], which translates into more iterations needed for the algorithm to converge.[2] This point is illustrated in Fig. 5. (2) The memory use of the present work is the same as for the serial implementation, whereas the implementation of [14] requires 4 (in 2D) and 8 (in 3D) times the

[1] The two dimensional case is straightforward.
[2] We note that this was first pointed out in [14].

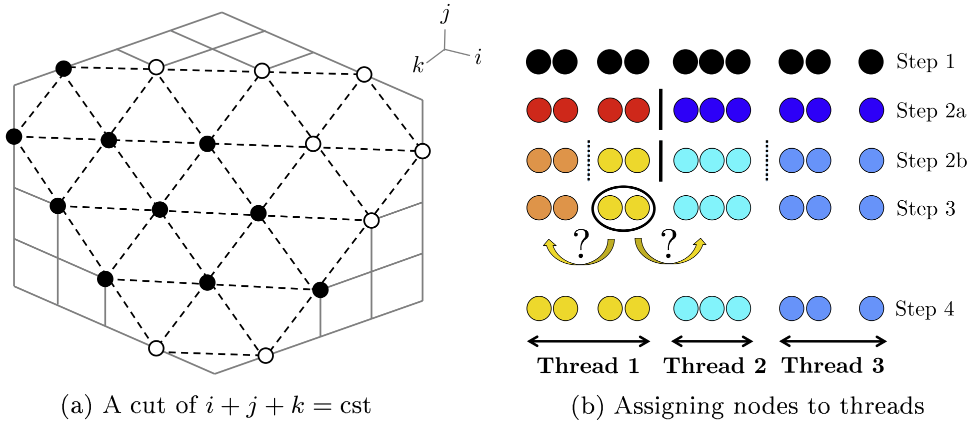


Fig. 4. Left: Cuthill–McKee ordering in three spatial dimensions. The solid symbols represent the grid points in Ω^- at which the solution is to be computed and the open symbols represent region of obstacles. Right: schematic of the strategy to balance the load on 3 threads.

resource of memory. (3) This overhead in memory also translates in CPU overhead. Our computational experiments seem to indicate that the parallel overhead in [14] adds computational time comparable to that induced by the creation and synchronization of threads in our algorithm. (4) More importantly, any number of threads will be effective in our case and will not suffer from a plateau effect in performance. These points are illustrated in the next section.

3. Numerical experiments

In this section we demonstrate the effectiveness of the method and compare it to the benchmark parallel method of [14]. All tests were run on DL580 nodes with 4 Intel X7550 eight core processors each.

3.1. Example problems

3.1.1. Example 1

Eq. (1) is solved in two dimensions with variable and discontinuous propagation speed. The problem is specified by:

$$f(\mathbf{x}) = \begin{cases} \sqrt{13x^2+13y^2+24xy}, & \mathbf{x}\in\{\mathbf{x}|x_1>0, x_2>0\}, \\ 2\sqrt{x^2+y^2}, & \text{otherwise}, \end{cases} \quad \mathbf{x}\in\Omega=[-1,1]^2,$$
$$g(\mathbf{x}) = 0, \quad \mathbf{x}\in\Gamma=\mathbf{0},$$

3.1.2. Example 2

We consider the computation the distance function from a center source point in a three dimensional cubic domain with four concentric spherical obstacles. The mathematical description is given by:

$$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{x}\in A, \\ \infty, & \text{otherwise}, \end{cases} \quad \mathbf{x}\in\Omega=[-1,1]^3,$$

$$g(\mathbf{x}) = 0, \quad \mathbf{x}\in\Gamma=\mathbf{0},$$

where $A = A_1\cup A_2\cup A_3\cup A_4$ with

$$A_1 = \{\mathbf{x}|(.3<|\mathbf{x}|<.3+\texttt{width},)\}\setminus((x_1^2+x_2^2<.1)\cap x_3<0),$$

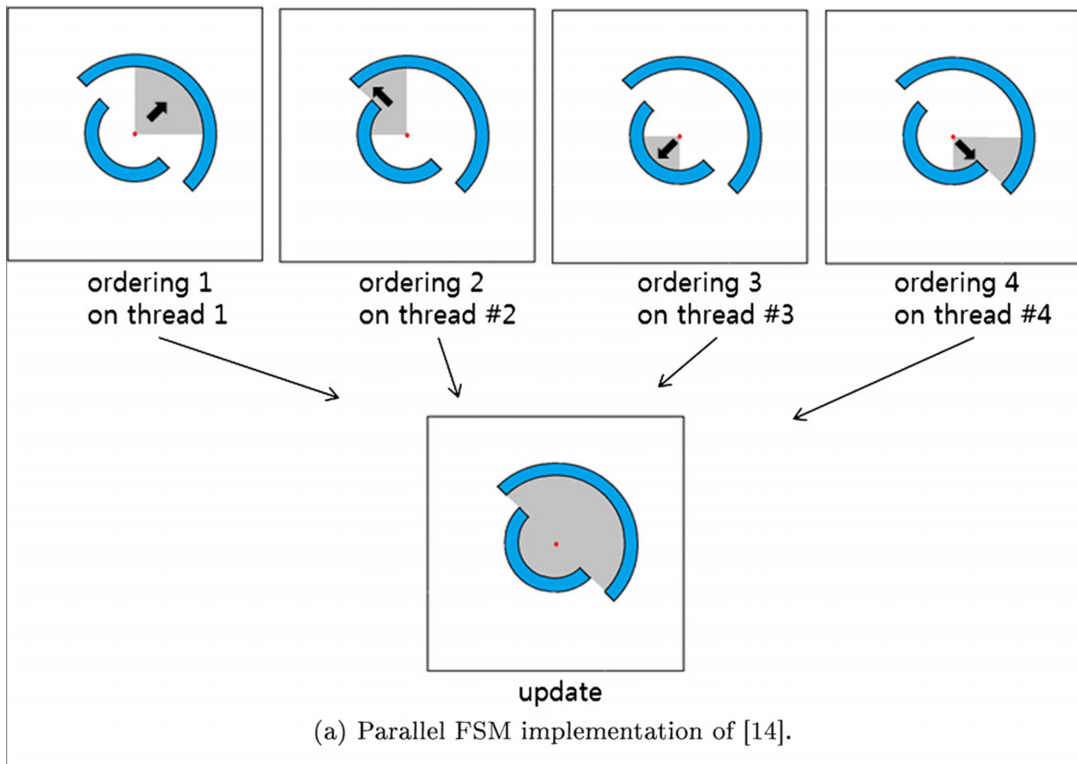
$$A_2 = \{\mathbf{x}|(.5<|\mathbf{x}|<.5+\texttt{width},)\}\setminus((x_1^2+x_2^2<.1)\cap x_3<0),$$

$$A_3 = \{\mathbf{x}|(.7<|\mathbf{x}|<.7+\texttt{width},)\}\setminus((x_1^2+x_2^2<.1)\cap x_3<0),$$

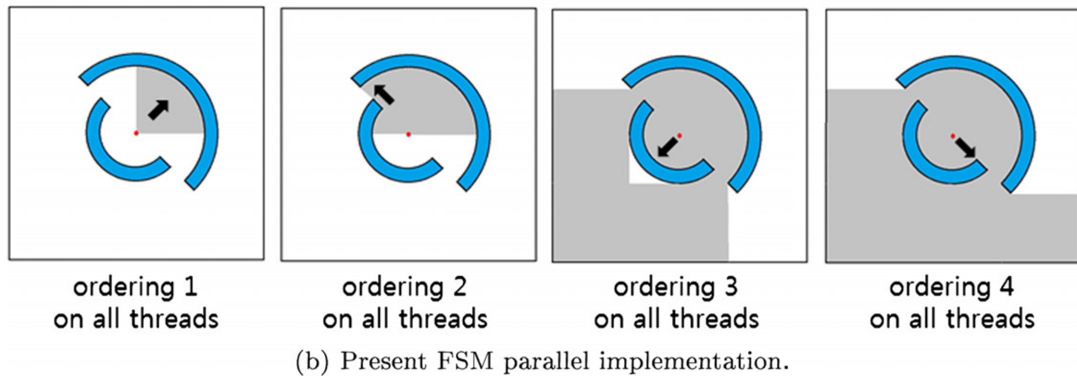
$$A_4 = \{\mathbf{x}|(.9<|\mathbf{x}|<.9+\texttt{width},)\}\setminus((x_1^2+x_2^2<.1)\cap x_3<0)$$

and `width` = 1/12.

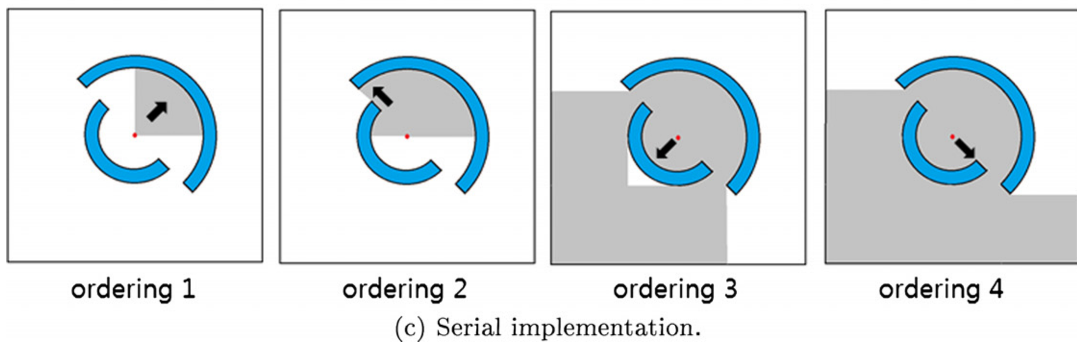
Note that we have used the \ operator to indicate a set theoretic difference.



(a) Parallel FSM implementation of [14].



(b) Present FSM parallel implementation.



(c) Serial implementation.

Fig. 5. Schematics of how data propagates from a source point (center, red) using our parallel implementation and that of [14]. The blue arcs represent obstacles. At the end of one iteration (the four sweeps), our parallel implementation has updated the solution of (1) in a larger region than does the implementation of [14]. This also exemplifies that the implementation of Zhao [14] will in some cases require a little more iterations to converge than the serial version. In contrast, our parallel implementation gives after one iteration the same result as that of the serial implementation, while utilizing all available threads. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

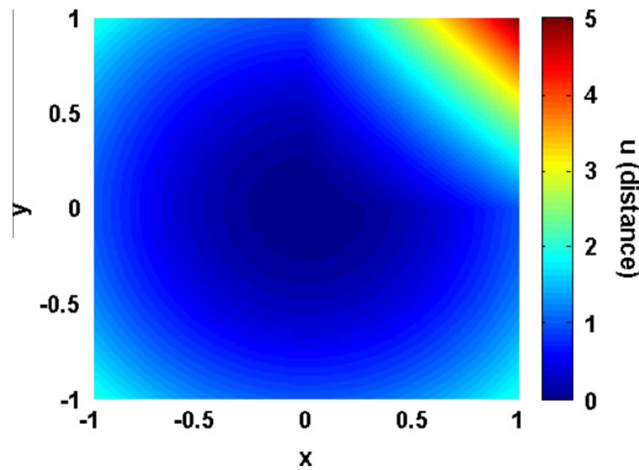


Fig. 6. Solution to Example 1.

3.1.3. Example 3

We consider the computation of the distance to 40 randomly generated points in a cubic domain and is specified by:

$$f(\mathbf{x}) = 1, \quad \mathbf{x} \in \Omega = [-1,1]^3,$$
$$g(\mathbf{x}) = 0, \quad \mathbf{x} \in \Gamma = \{\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \ldots, \tilde{\mathbf{x}}_{40}\}, \text{ where the } \tilde{\mathbf{x}}_i\text{'s are randomly generated points in } \Omega.$$

3.2. Results

We chose Example 1 to illustrate the solution of Eq. (1) with variable $f(x)$ and to show the convergence rate to the true solution. The computed solution is given in Fig. 6. The analytical solution of 1 is given by Eq. (4). In Fig. 7 we show the convergence of the parallel FSM to the true solution. As expected, the algorithm exhibits first order accuracy.

$$u(\mathbf{x}) = \begin{cases} x^2 + y^2 + 3xy, & \mathbf{x} \in \{\mathbf{x} | x_1 > 0, x_2 > 0\} \\ x^2 + y^2, & \text{otherwise}. \end{cases} \tag{4}$$

Examples 2 and 3 were solved in serial and in parallel on up to 32 threads (their computed solutions are given in Figs. 8 and 9 respectively). For the purposes of these tests, parallel speedup is defined as the wall clock time[3] for the serial FSM method divided by the wall clock time for each parallel method on p processors (T_1/T_p). We denote by N the total number of grid points.

The results of a convergence analysis of the sequential FSM, as well as the present work and the benchmark parallel implementations of [14] are shown in Table 1. This table gives the number of Gauss–Seidel iterations to converge to an approximate solution. In the table, we define one iteration as all the sweep orderings, i.e. one iteration performs $8 \times N$ operations in three spatial dimensions. Each method is iterated until it converges to an approximate solution within the same precision. As discussed in Section 2.5, the present work converges in as many iterations as the serial FSM, while the benchmark parallel method of [14] requires more iterations.

In three dimensional problems, the implementation of [14] (without domain decomposition) has $2^3 = 8$ operations that can be performed simultaneously. Fig. 10 depicts the resulting plateaus in parallel speedup at two, four, and eight threads. For the specific cases of $p = 2, 4$ and 8, our implementation performs equal to or better than the benchmark [14], and outperforms it for all other choices of p.

Fig. 11 illustrates that as N become large, the parallel speedup for the present work trends closer to the linear limit. Note that for two and three dimensional problems, the solution of Eq. (2) requires very little computation, therefore, the problem size becomes prohibitive for the machine we are using before parallel overhead is negligible. For that reason, linear speedup is not explicitly shown experimentally.

3.3. A remark on parallel architectures

The present method is not specific to a type of parallel architecture, however, we implemented it with OpenMP on shared memory. This choice was motivated by the nature of the test problems. For distance functions, $f = 1$ in Eq. (1). This leads to a very simple computation at each node. We note that this type of problem may be well suited for GPU architectures and plan to investigate that in the future. The overhead associated with moving data in a message passing environment is much higher than in a shared memory environment. For problems in which more computation is required (e.g. f is not prescribed and

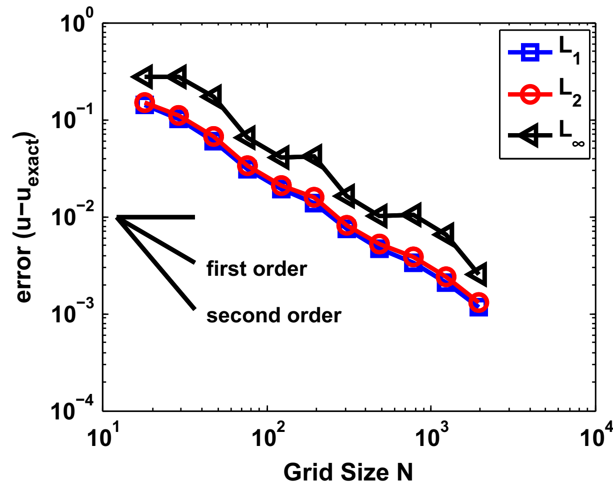


Fig. 7. Error analysis for Example 1.

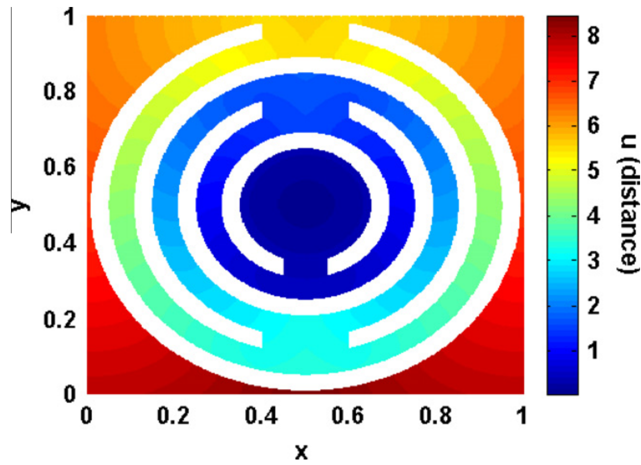


Fig. 8. Cross-section of the solution to Example 2 at $z = .5$. White regions indicate obstacles.

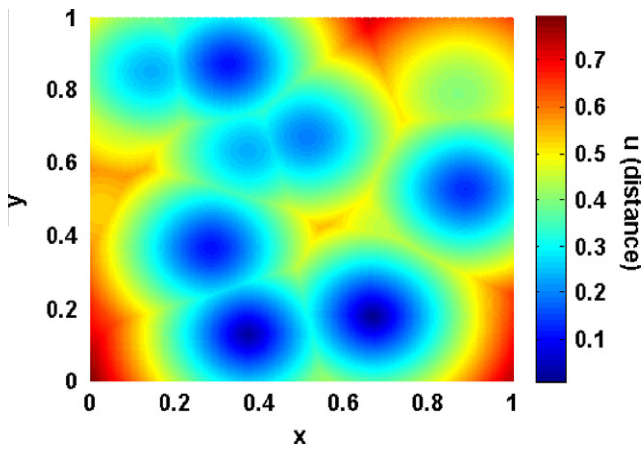
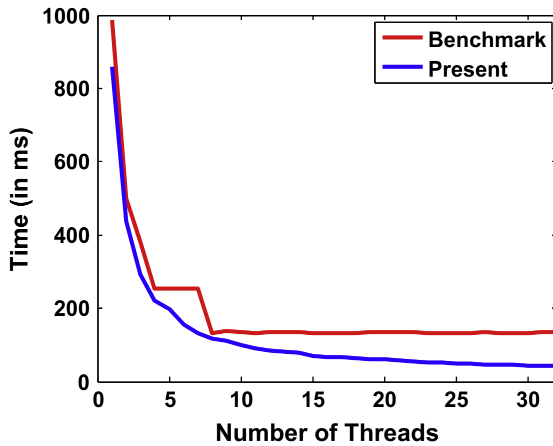


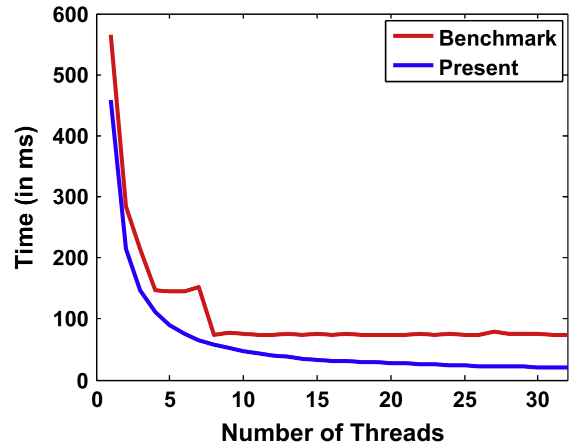
Fig. 9. A cross-section of the solution to Example 3 at $z = .5$.

Table 1
Comparison of the number of iterations necessary to reach convergence for the serial FSM, the parallel implementation of [14] and the present work. The number of iterations of the present work is equal to that of the serial implementation while the parallel implementation of [14] may require more iterations.

	Example 2			Example 3		
	Serial FSM	Present work	Benchmark [14]	Serial FSM	Present work	Benchmark [14]
$N = 40^3$	7	7	12	2	2	4
$N = 80^3$	7	7	12	3	3	5
$N = 160^3$	8	8	12	3	3	5

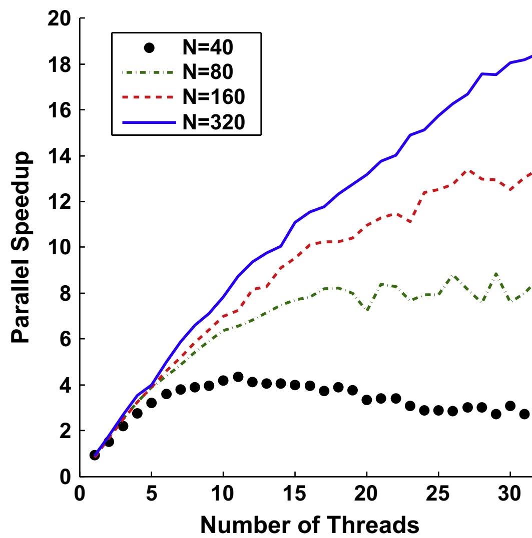


(a) Parallel results for Example 2

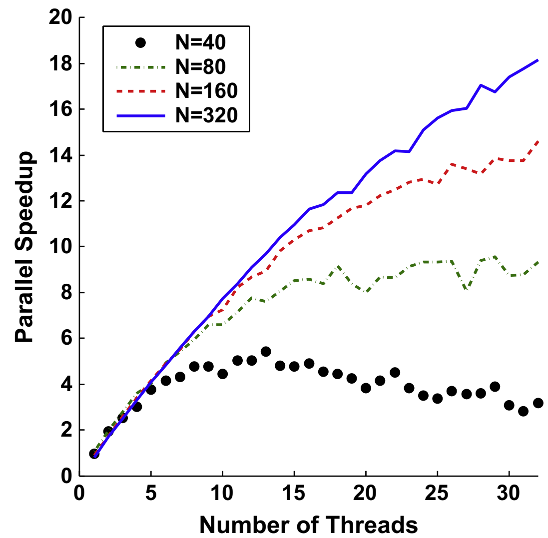


(b) Parallel results for Example 3

Fig. 10. Computational times for the parallel implementation of the present work and the benchmark parallel implementation of [14]. In this test the total number of grid points is $N = 320^3$.



(a) Parallel speedup for Example 1



(b) Parallel speedup for Example 2

Fig. 11. Illustration that, as N becomes large, the parallel speed up of the present work increases. In addition no plateaus are present.

requires a significant amount of computation) one would see parallel efficiency near 1 for relatively low values of N, and it would justify an implementation in a message passing parallel environment.

4. Conclusion

We have introduced a novel parallel method for solving the Eikonal equation and we have compared its performance to the work of [14]. The performance of the method have been shown to scale well with the number of threads. In addition, our implementation requires only as many Gauss–Seidel iterations as the original serial FSM method. Our computational experiments seem to indicate that the parallel overhead is on a par with that of [14]. In the case where the number of dedicated threads divides the number of orderings (4 in 2D and 8 in 3D), both methods have about the same efficiency and the method of Zhao [14] is simpler to implement. However, any number of threads will be effective in our case and will not suffer from a plateau effect in performance. Future work will investigate the generalization of this approach and the study of its efficiency on problems in higher dimensions and on other parallel architectures.

Acknowledgments

The research of M. Detrixhe and F. Gibou was supported in part by ONR N00014-11-1-0027, NSF CHE 1027817, DOE DE-FG02-08ER15991, ICB W911NF-09-D-0001, NSF IGERT Grant DGE-0221715, and by the W.M. Keck Foundation.The work of C. Min was supported by Priority Research Centers Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012-0006691) and by the National Research Foundation of Korea (NRF) Grant funded by the Korea government (MEST) (No. 2012-0003385). We acknowledge support from the Center for Scientific Computing at the CNSI and MRL: an NSF MRSEC (DMR-1121053) and NSF CNS-0960316.

References

[1] K. Alton, I.M. Mitchell, Optimal path planning under different norms in continuous state spaces, in: Proceedings of the 2006 IEEE International Conference on Robotics and Automation, ICRA 2006, May 2006, pp. 866–872.
[2] A. Bruss, The Eikonal equation: some results applicable to computer vision, J. Math. Phys. 23 (5) (1982) 890–896.
[3] D.L. Chopp, Some improvements of the fast marching method, J. Sci. Comput. 23 (1) (2001) 230–244.
[4] Won-Ki Jeong, Ross T. Whitaker, A fast iterative method for Eikonal equations, SIAM J. Sci. Comput. 30 (5) (2008) 2512–2534.
[5] Won ki Jeong, Ross T. Whitaker, A fast Eikonal equation solver for parallel systems, in: SIAM Conference on Computational Science and Engineering, 2007.
[6] Ron Kimmel, James A. Sethian, Optimal algorithm for shape from shading and path planning, J. Math. Imaging Vis. 14 (2001) 2001.
[7] R. Malladi, J.A. Sethian, A unified approach to noise removal, image enhancement, and shape recovery, IEEE Trans. Image Process. 5 (11) (1996) 1554–1568.
[8] Stanley Osher, James A. Sethian, Fronts propagating with curvature dependent speed: algorithms based on Hamilton–Jacobi formulations, J. Comput. Phys. 79 (1) (1988) 12–49.
[9] E. Rouy, A. Tourin, A viscosity solution approach to shape-from-shading, SIAM J. Numer. Anal. 29 (3) (1992) 867–884.
[10] Yousef Saad, Iterative Methods for Sparse Linear Systems, second ed., Society for Industrial and Applied Mathematics, 2003.
[11] J.A. Sethian, Level Set Methods and Fast Marching Methods, Cambridge University Press, 1999.
[12] J.A. Sethian, A fast marching level set method for monotonically advancing fronts, Proc. Natl Acad. Sci. 93 (1996) 1591–1595 (Applied Mathematics).
[13] Hongkai Zhao, A fast sweeping method for Eikonal equations, Math. Comput. 74 (2004) 603–627.
[14] Hongkai Zhao, Parallel implementations of the fast sweeping method, J. Comput. Math. 25 (2007) 421–429.