Journal of Computational Physics 322 (2016) 199–223

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

Hybrid massively parallel fast sweeping method for static Hamilton–Jacobi equations

Miles Detrixhe [a,b,*], Frédéric Gibou [a,b,c,d]

[a] Department of Mechanical Engineering, United States
[b] University of California Santa Barbara, Santa Barbara, CA, 93106, United States
[c] Department of Computer Science, United States
[d] Department of Mathematics, United States

ARTICLE INFO

Article history:
Received 8 December 2015
Received in revised form 14 June 2016
Accepted 15 June 2016
Available online 30 June 2016

Keywords:
Parallel computing
Hamilton–Jacobi
Fast sweeping
Eikonal equation
Optimal control
Dynamic games

ABSTRACT

The fast sweeping method is a popular algorithm for solving a variety of static Hamilton–Jacobi equations. Fast sweeping algorithms for parallel computing have been developed, but are severely limited. In this work, we present a multilevel, hybrid parallel algorithm that combines the desirable traits of two distinct parallel methods. The fine and coarse grained components of the algorithm take advantage of heterogeneous computer architecture common in high performance computing facilities. We present the algorithm and demonstrate its effectiveness on a set of example problems including optimal control, dynamic games, and seismic wave propagation. We give results for convergence, parallel scaling, and show state-of-the-art speedup values for the fast sweeping method.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Static Hamilton–Jacobi (HJ) equations are directly applicable in a wide range of applications including optimal control [4,32], seismic imaging [28,31], and computer vision [29,22]. They are also used in the reinitialization procedure of the level-set method [25].

The first fast algorithms for solving static HJ equations were the method of Tsitsiklis [34], the fast marching method (FMM) [30], and fast sweeping method (FSM) [37] for the Eikonal equation. The marching methods are single-pass, Dijkstra-like [14] algorithms that aim to update each grid point in its order of dependence on the boundary data. The fast sweeping method, however, is an iterative method that builds upon the earlier idea of alternating Gauss–Seidel iterations for linear PDEs [23] or HJ equations [5]. Quickly, researchers adapted the two methods into a family of Ordered Upwind Methods [32,16,11,1] and sweeping methods [19,20,33,10,20,27,15], respectively, to solve more general HJ problems. These more powerful techniques often came with added computational cost; either from an increase in iterations required or from a more expensive local discretization. With serial methods, these problems became intractable on relatively fine grids. As parallel computing technology became standard, parallel marching [6,18] and sweeping [38,13,12] methods were created. Recently, the Heap Cell Method (HCM) [8] was developed to combine the advantages of both the FSM and FMM into a single algorithm. A shared memory parallel technique for the HCM [9] soon followed.

* Corresponding author.
E-mail addresses: mdetrixhe@engineering.ucsb.edu (M. Detrixhe), fgibou@engineering.ucsb.edu (F. Gibou).

http://dx.doi.org/10.1016/j.jcp.2016.06.023
0021-9991/© 2016 Elsevier Inc. All rights reserved.

Each of these fast algorithms relies on updating grid points in a specific order in order to propagate the solution from a boundary to the whole domain along the characteristic curves. Parallel algorithms either decompose a domain into disjoint subdomains or aim to update groups of nodes that are independent of (or weakly dependent on) each other. Both of these strategies present challenges and limitations to the efficacy of an algorithm. Domain decomposition leads to an increased number of iterations required for convergence. The strategy of updating clusters of independent nodes is not easily implemented on a distributed memory system and is, therefore, limited by shared memory computer architecture. Existing parallel methods have been shown to be effective on certain problems and have moderate published speedup factors ($\lesssim 30$ for CPUs and $\lesssim 120$ for GPUs).

Here we present the hybrid massively parallel fast sweeping method (HMP-FSM) that utilizes a coarse grained distributed technique to partition the domain among available compute nodes and a fine grained shared memory method within each subdomain. By combining the two methods, we can take advantage of the benefits of the efficient shared memory parallelization while lessening the efficiency penalty of distributed methods. We demonstrate that the hybrid method is more efficient than either constituent method on identical hardware. We detail the algorithm and present speedup results for several example problems. We implement the method on large state-of-the-art supercomputers and present scaling results giving speedup factors orders of magnitude larger than previously published results. We show that our hybrid method is algorithmically superior to domain decomposition alone and more efficiently uses the available hardware.

The structure of this article is as follows. In section 2 we discuss static Hamilton–Jacobi equations and the three specific equations we consider. In section 3 we discuss the two individual parallel methods and the hybrid algorithm, which is a combination of the two. In section 4 we detail the example problems used for all numerical experiments and validations in this work. In section 5 we demonstrate convergence of the method to the analytic solutions of some example problems. In section 6 we present the scaling and efficiency results of our method and compare it to existing methods. Then, the method is used to study two specific example applications in section 7. Finally, we make concluding remarks in section 8.

2. Static Hamilton–Jacobi equations

In [26], Osher provided the link between time dependent and static HJ equations. Consider a time dependent Hamilton–Jacobi equation

$$\phi_t + H(\nabla\phi, \mathbf{x}, t) = 0,$$

with $H > 0$. The zero level set of ϕ at time t is precisely the set of points for which $u(\mathbf{x}) = t$, where $u : \mathbb{R}^n \to \mathbb{R}$ is the solution of the static HJ equation:

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

This general PDE has numerous forms, but for the purposes of this article, we have chosen to consider three specific HJ equations to best demonstrate our method. They each present different mathematical and computational challenges. The fast sweeping method for each of these problems has been developed previously. In each case, we cite the relevant work and include the numerical discretization in Appendix A. Our aim is to choose a set of problems that show the strengths and limitations of our method as well the breadth of problem types to which it can be applied. The first is the solution of the Eikonal equation, which has numerous applications including determining the signed distance function for level set reinitialization or to compute the first arrival time of seismic waves. The second example equation is the Hamilton–Jacobi–Bellman equation for optimal control of a dynamical system. It gives the optimal value function, and indirectly, the optimal control for reaching a target state. The third example equation is the Hamilton–Jacobi–Isaacs equation, which gives the value function and the optimal controls for two adversarial agents engaged in a dynamic game.

2.1. Eikonal equation

The Eikonal equation is an important static Hamilton–Jacobi equation and has applications in optimal control [22,21], computer vision [7], seismology [35,36,31,24], and level-set methods [30]. The FSM and FMM were designed to solve the Eikonal equation. It is a first order, nonlinear, hyperbolic PDE:

$$\begin{aligned} |\nabla u(\mathbf{x})| &= f(\mathbf{x}) \quad \text{for } \mathbf{x} \in \Omega \subset \mathbb{R}^n, \\ u(\mathbf{x}) &= g(\mathbf{x}) \quad \text{for } \mathbf{x} \in \Gamma \subset \Omega, \end{aligned} \tag{2}$$

where Ω is the computational domain, $f(\mathbf{x})$ is a given function, and $g(\mathbf{x})$ is the boundary value prescribed on the boundary Γ. In the case where $g(\mathbf{x})$ is zero, the Eikonal equation describes the physical problem of a moving wavefront where the solution u is the arrival time of a wave that advances monotonically from Γ at a speed equal to $\frac{1}{f(\mathbf{x})}$. The Godunov discretization of equation (2) is given in equation (A.1).

2.2. Hamilton–Jacobi–Bellman equation

Consider a control system governed by the state equation:

$$\begin{aligned} \mathbf{y}'(t) &= \mathbf{f}(\mathbf{y}(t), \mathbf{a}(t)), \quad t > 0, \\ \mathbf{y}(0) &= \mathbf{x}, \end{aligned} \tag{3}$$

where the control $\mathbf{a}$ is any measurable function of t with values in the control space $\mathcal{A}$ and $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$. Assume that the dynamics $\mathbf{f}$ are such that for any choice of the control $\mathbf{a}$ and initial condition $\mathbf{x}$ there exists a unique solution to the state equation (3). This control system is subject to a running cost given by a function $l(\mathbf{x})$. The goal is to control this system to a target set Γ where there is a terminal cost g. It can be shown [4] that the minimum total cost to reach the target is the viscosity solution of the Hamilton–Jacobi–Bellman (HJB) equation:

$$\begin{aligned} H(\mathbf{x}, \nabla u) = \min_{\mathbf{a} \in A} \{\nabla u(\mathbf{x}) \cdot \mathbf{f}(\mathbf{x}, \mathbf{a})\} &= l(\mathbf{x}), && \mathbf{x} \in \Omega \\ u(\mathbf{x}) &= g(\mathbf{x}), && \mathbf{x} \in \Gamma \subset \Omega. \end{aligned} \tag{4}$$

The value function or "cost-to-go" function is u and the optimal control for any state is the value of $\mathbf{a}$ that minimizes the Hamiltonian. For a complete derivation of the HJB equation and a discussion on viscosity solutions, see [4]. The discretization and update formula for the HJB equation is given in A.2.

2.3. Hamilton–Jacobi–Isaacs equation

The Hamilton–Jacobi–Isaacs (HJI) equation is very similar to the HJB, but it concerns dynamic games rather than strictly optimal control. In this case, the dynamical system is

$$\begin{aligned} \mathbf{y}'(t) &= \mathbf{f}(\mathbf{y}(t), \mathbf{a}(t), \mathbf{b}(t)), \quad t > 0, \\ \mathbf{y}(0) &= \mathbf{x}. \end{aligned} \tag{5}$$

The first player can control the system with the control variable $\mathbf{a}$ and the second player with $\mathbf{b}$ in the control spaces $\mathcal{A}$ and $\mathcal{B}$, respectively. Similar to the HJB, there is a target set with a terminal cost, which when reached, ends the game. The goal of the first player is to maximize the total cost and the goal of the second player is to minimize it. The Hamilton–Jacobi–Isaacs equation (6) gives the upper bound on the total cost to reach the target set. The lower bound can be obtained by switching the $\max_{a \in A}$ and the $\min_{b \in B}$. For many problems, the upper bound and the lower bound are equivalent and then the solution gives the exact cost-to-go. For a thorough discussion and derivation, see [4].

$$\begin{aligned} \max_{\mathbf{a} \in A} \min_{\mathbf{b} \in B} \{(\nabla u(\mathbf{x})) \cdot \mathbf{f}(\mathbf{x}, \mathbf{a}, \mathbf{b})\} &= l(\mathbf{x}) \ , \ \mathbf{x} \in \Omega \subset \mathbb{R}^d, \\ u(\mathbf{x}) &= g(\mathbf{x}) \ , \ \mathbf{x} \in \Gamma \subset \Omega. \end{aligned} \tag{6}$$

As with the HJB equation, u gives the value function, $\mathbf{f}$ is the dynamics, l is the running cost, and g is the terminal cost. The discretization and update formula for the HJI equation is given in A.2.

3. Parallel models

In this section we present the hybrid massively parallel fast sweeping method (HMP-FSM). The method uses a distributed memory method on a coarse scale and a shared memory method at the fine scale. We discuss the advantages and disadvantages of the coarse and fine grained methods, while highlighting the benefits of the hybrid method. With arguments based on Amdahl's Law [2] and the geometry of the characteristic curves of the problems considered, we will provide estimates and bounds on parallel efficiency scaling.

A uniform Cartesian grid with N_i evenly spaced grid points in the ith coordinate direction is used for all numerical examples herein. All domain decomposition is done in a uniform manner, i.e., the computational domain is split into an equal number of partitions in each coordinate direction. We denote the total number of partitions by P, the total number of processors by p, and the number of compute cores per partition by c. For each of the three algorithms, the total number of processors is given by $p = P \times c$. For the distributed memory method, there is only one processor per partition, so $c = 1$ is an algorithmic limitation, and the only free parameter is P. For the shared memory method, there is no partitioning so necessarily $P = 1$ and the free parameter is c. The HMP-FSM combines the other two methods and, therefore, we can choose p and c to be any value within the limits of the hardware.

3.1. Distributed memory model

A domain decomposition method for parallelizing the fast sweeping method was first introduced in [38]. The algorithm is to decompose the domain into P uniform subdomains and assign each one to a separate processor with its own memory.

Algorithm 1 Domain decomposition parallel fast sweeping method.

```
initialize_Parallel_Environment(p)          ▷ Spawn p parallel processes.
pid = get_Processor_ID()
get_Domain_Partition(pid)                   ▷ Uniform partition (Fig. 1)
initialize_Problem()                        ▷ Initialize U, RHS in each partition according to [37]
while convergence_Check()==false do         ▷ Iterate until convergence
    for sweep_number=1:2^d do               ▷ Number of sweeping directions is 2^d
        for visit_All_Points_In_Order(sweep_number) do   ▷ FSM alternating ordering
            update_Point()                  ▷ Solve the appropriate update function
        end for
    end for
    wait_For_All_Processors()               ▷ Synchronize. E.g. MPI_Barrier.
    send_Updated_Shared_Points_To_Neighbor_Partitions()   ▷ Send updated values
    update_Ghost_Points()                   ▷ Update ghost points as in [38]
end while
```

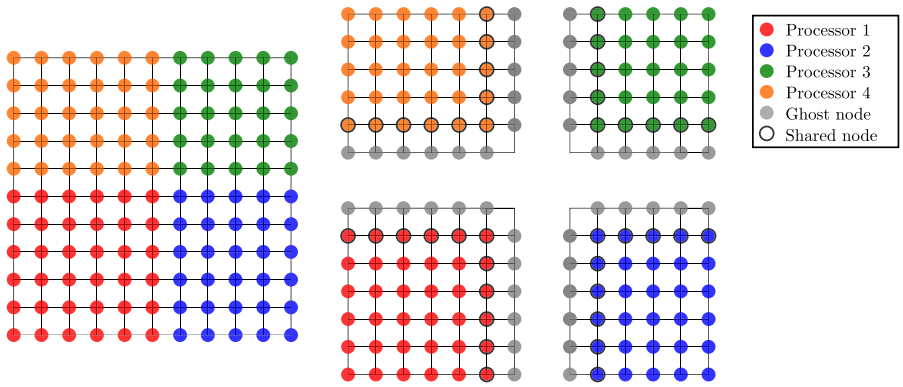

Fig. 1. A uniform 2D grid with $N_x = 11$, $N_y = 11$ partitioned among four processors. Ghost points for each partition are shown in gray and shared points have a black outline.

The standard FSM is applied within each domain, and information along the boundaries is shared between partitions after each iteration. Because all numerical Hamiltonians discussed here are monotone, shared points are updated by choosing the minimum value across both processors. We choose this domain decomposition parallel fast sweeping method (DDP-FSM) as the coarse grained parallel component of the hybrid method. Algorithm 1 gives a pseudocode listing of the DDP-FSM. Fig. 1 shows a 2D domain partitioned among four processors. Each processor is assigned a color and the grid points that are updated from neighboring partitions (ghost points) are shown in gray. Grid points for which information must be sent to neighbors (shared points) are outlined in black.

This algorithm is attractive because it is simple to implement and is compatible with modern supercomputing clusters and the Message Passing Interface (MPI). The major drawback is that the total amount of work increases with the number of partitions. In the sequential FSM, information can propagate along characteristics across the entire domain in a single iteration. In the DDP-FSM, however, the domain is decomposed into smaller subdomains, reducing the distance over which information can propagate in a single iteration. In fact, for the case of the Eikonal equation with constant right-hand-side and uniform partitions, it can be shown that the number of iterations scales as $\sqrt[d]{P}$ where P is the number of partitions and d is the dimensionality. This leads to a parallel efficiency that scales as $P^{-1/d}$. As the number of uniform partitions increases, the parallel efficiency decreases. We investigate this further in section 5. With nonuniform partitioning, it may be possible to choose a partitioning scheme based on the specific problem that mitigates this effect, however, nonuniform partitioning is beyond the scope of this article. Therefore, in order to maximize parallel efficiency with a uniform domain decomposition method, the number of partitions must be minimized.

3.2. Shared memory model

In a previous work [13], we developed a shared memory parallel fast sweeping method for the Eikonal equation. Here, we use the same technique to parallelize the solution of more general HJ equations. We denote the method the hyperplane stepping parallel fast sweeping method (HSP-FSM). A brief description of the method is given below; for a full description, see [13]. A pseudocode listing of the shared memory method is given in Algorithm 2.

We would like to point out that the fine-grained portion of the HMP-FSM was chosen to efficiently use a single compute node on a modern supercomputer (specifically Stampede at the Texas Advanced Computing Center). In section 6.1, we show that the HSP-FSM from [13] is very efficient on all static HJ examples considered and nearly ideally efficient on the HJB and HJI examples. We have chosen to use this method throughout the paper. However, any parallel method that solves the

Algorithm 2 Hyperplane stepping parallel fast sweeping method.

```
initialize_Problem()                                   ▷ Initialize U, RHS in each partition according to [37]
compute_Mapping()                                      ▷ Index points by level.
while convergence_Check()==false do                    ▷ Iterate until convergence
  for sweep_number=1:2^d do                            ▷ Number of sweeping directions is 2^d
    for level=1:max_level do
      parallel for point=1:number_Of_Points(level) do  ▷ Update single level in parallel
        update_Point()                                 ▷ Solve the appropriate update function
      end parallel for
    end for
  end for
end while
```

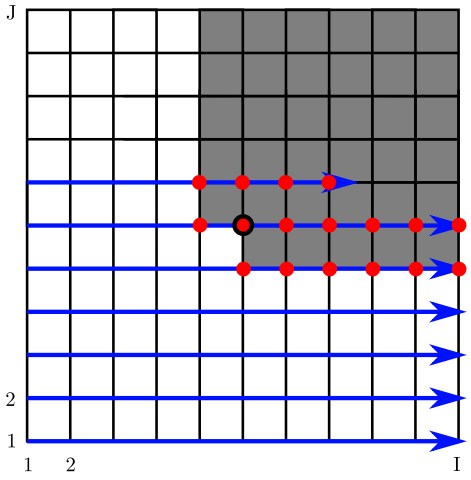

Fig. 2. A depiction of the serial FSM for example 1 on a coarse grid during the first sweep. Data is propagated from the origin (Γ) up and to the right. Blue lines indicate sweeping direction. Red dots indicate a correctly updated grid point. The solution will be propagated into the entire shaded region after the first sweep. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

static HJ equation efficiently within a single partition could be used to the same effect. For instance, the parallel two-scale method [9] or the fast iterative method [18] or the method in section 2.1 of [38] could be substituted for the HSP-FSM.

The standard FSM [37] updates each point in a specific order in a sequence of alternating sweeps. Each sweep propagates the solution along a distinct family of characteristics. The hyperplane stepping parallel fast sweeping method identifies groups of points that are independent of each other and depend only on points that have already been updated on the current sweep and updates them in parallel. On a uniform grid, these points belong to a line, a plane, or a hyperplane in 2D, 3D, and higher dimension, respectively. Fig. 2 illustrates the first sweep of the standard FSM on a 2D problem. The first sweep updates points beginning in the lower left corner and sweeps to the right and up. For the prototype problem, the solution will be propagated from the boundary data (at the center) to the $\nearrow$ corner. The HSP-FSM updates all points within a *level* simultaneously. For the first sweep, a level is defined as the set of all points for which $i + j + \ldots = C$ where C is an integer constant. Fig. 3 illustrates the parallel sweeping ordering for a 2D problem. The inset shows that for a particular point, all of its neighbors are in a separate level and all neighboring points earlier in the sequential ordering belong to the previous level. Therefore all points within a level are independent and can be computed in parallel. The HSP-FSM computes the *exact* same solution as the serial FSM in the *exact* same number of sweeps.

The strength of the HSP-FSM is that it does exactly the same amount of work as the serial method and produces exactly the same solution. It can, theoretically, achieve ideal parallel speedup. A shortcoming of this method is that it requires frequent spawning and syncing of parallel threads and there is no obvious way to decompose the domain among processors. For this reason, it is an effective approach on shared memory architectures, but not on distributed memory machines.

As the problem size increases, this method's theoretical efficiency approaches unity. Assuming a discretization with an equal number of grid points in each dimension, the total computational work is proportional to the number of grid points: $O(N_x^d)$ in d dimensions. The span (or T_∞) is the longest sequential path, which is equal to the total number of levels: $O(N_x)$. Potential parallelism is a measure of the amount of parallel work per sequential step. It is given by the ratio of work to span and is

$$pp = \frac{O(N_x^d)}{O(N_x)} = O(N_x)^{d-1}.$$

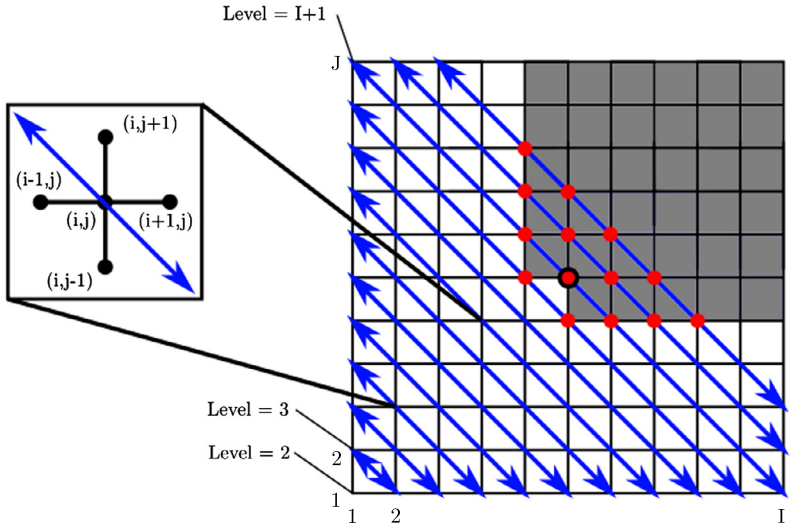

Fig. 3. An illustration of the HSP-FSM for example 1 on a coarse grid during the first sweep. Blue lines indicate a *level* is updated in parallel. The inset shows that points within a level are independent. The solution is propagated to exactly the same grid points as the serial method after the first sweep. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Algorithm 3 Hybrid massively parallel fast sweeping method.

```
P = p/c                                                  ▷ Compute number of partitions based on architecture.
initialize_Parallel_Environment(P)                       ▷ Spawn P parallel processes.
pid = get_Processor_ID()
get_Domain_Partition(pid)                                ▷ Uniform partition (Fig. 1)
initialize_Problem()                                     ▷ Initialize U, RHS in each partition according to [37]
compute_Mapping()                                        ▷ Index nodes by level.
while convergence_Check()==false do                      ▷ Iterate until convergence
  for sweep_number=1:2^d do                              ▷ Number of sweeping directions is 2^d
    for level=1:max_level do
      parallel for node=1:number_Of_Nodes(level) do      ▷ Divide nodes among c threads.
        update_Node()                                    ▷ Solve the appropriate update function
      end parallel for
    end for
  end for
  wait_For_All_Processes()                               ▷ Synchronize. E.g. MPI_Barrier.
  send_Updated_Shared_Nodes_To_Neighbor_Partitions()     ▷ Send updated values
  update_Ghost_Nodes()                                   ▷ Update ghost nodes as in [38]
end while
```

Not surprisingly the parallelism increases with N_x and increases exponentially with d. This analysis indicates that as the problem size increases, the amount of parallelism increases quickly, especially for problems with high dimensionality. Our numerical experiments confirm that this method is increasingly effective on higher dimensional problems.

3.3. Hybrid parallel model

The DDP-FSM and HSP-FSM each have their own strengths and shortcomings. Namely, the distributed memory method can be implemented on large computing clusters and be applied to memory-intense problems, but its efficiency decreases with the number of partitions. The shared memory method can achieve ideal efficiency, but the number of threads, and therefore the maximum speedup, is limited by current shared memory architectures.

Here, we present the hybrid massively parallel fast sweeping method (HMP-FSM) that combines the strengths of the two methods and lessens their shortcomings. A common layout for modern supercomputers is to have a large distributed array of compute nodes, each with their own memory shared among a small number of processors and/or a coprocessor. The HMP-FSM is specifically designed to excel with this architecture.

The algorithm is to decompose the domain and assign each partition to a cluster of processors with shared memory. Within each partition, the HSP-FSM is applied with the available processors. The total number of partitions is given by $P = p/c$. We call this a coarse-grained domain decomposition with fine-grained parallelism within each partition. The coarse-grained method (DDP-FSM) allows the algorithm to be implemented on large numbers of processors, and the fine-grained method (HSP-FSM) reduces the total number of partitions, increasing parallel efficiency. The complete pseudocode listing is given in Algorithm 3.

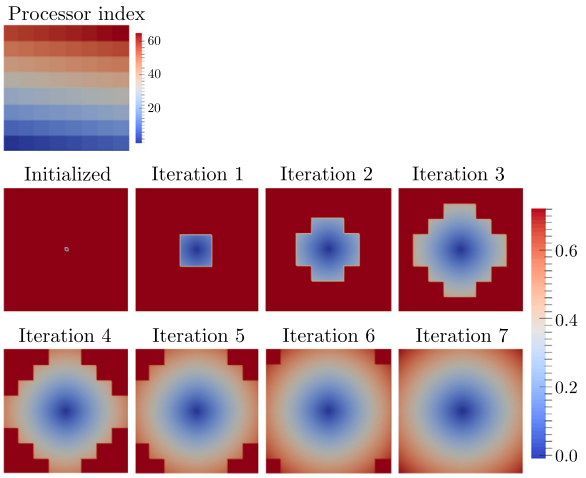

Fig. 4. Iterative convergence of the solution to example 1 with the DDP-FSM on 64 processors ($p = 64$, $P = 64$, $c = 1$).

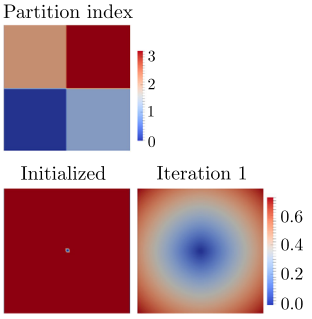

Fig. 5. Iterative convergence of the solution to example 1 with the HMP-FSM on 64 processors ($p = 64$, $P = 4$, $c = 16$).

Consider the two-dimensional Eikonal equation (2) implemented on four compute nodes, each with 16 shared memory cores. This example problem is fully specified in section 4.1. The goal is to parallelize the problem with all 64 processors. The HSP-FSM alone is incompatible with this architecture. The four compute nodes do not have a single shared memory and an attempt to implement this method with MPI would be dominated by communication costs. At best, the method can be implemented on a single node and be limited to a speedup of $S = p = 16$. Figs. 4 and 5 illustrate the iterative convergence of the simple example problem parallelized with the DDP-FSM (left) and the HMP-FSM (right) on 64 processors. In both cases, the solution is initialized at a small number of nodes near the origin and then the sweeping method is iterated until convergence. A single iteration includes sweeps in each direction within each partition. I.e. one iteration in 2D includes all four sweeping directions. Both methods use all 64 processors, but the hybrid method uses a factor of 16 fewer partitions ($c = 16$). For this particular problem, the hybrid method converges in a single iteration, while the standard domain decomposition method requires 7 iterations for convergence. The reason for the extra iterations is that information is shared along partition boundaries only between each iteration. The sweeping method can only propagate information within a partition during a single iteration. If both algorithms were implemented with negligible parallel overhead costs, the hybrid method would give a factor of 7 larger parallel speedup with identical hardware.

We note that [38] provides two separate methods and a hybrid distributed/shared method with $c = 4$ for a 2D example. Fig. 4 does not fairly exemplify the methods of [38], but rather compares the hybrid method to a pure domain decomposition method.

In that particular case, the hybrid algorithm gives a factor of 7 improvement, but in general we can estimate the algorithm's benefit. For a cluster of p processors arranged into nodes, each with c processors with shared memory, the hybrid method would decompose the domain into $P = p/c$ total partitions. In section 3.2, we established that the HSP-FSM can theoretically achieve ideal speedup, while the DDP-FSM cannot. For the Eikonal equation with constant right-hand-side, the characteristics are straight lines. Since information must propagate along characteristics, then the number of iterations must scale with the maximum number of partition boundaries crossed by a characteristic curve. This results in a number of iterations that scale as $\sqrt[d]{p}$. From this, we can compute the speedup of the domain decomposition method to be $S_p = p^{1-1/d}$ where p is the total number of processors. For the hybrid method, the number of partitions is not equal to the number of processors and the scaling law becomes $S_p = c^{1/d} p^{1-1/d}$. Consequently, for the same number of processors, the hybrid method has a speedup larger than that of the pure domain decomposition method by a constant factor equal to $c^{1/d}$.

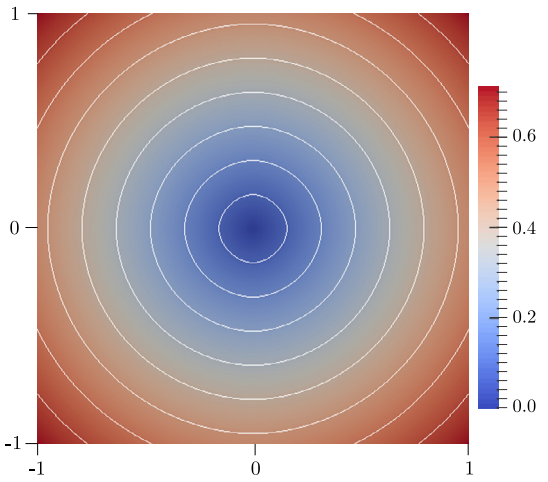

Fig. 6. Solution to example 1: the distance from the origin in two dimensions. The solution is indicated by color, while the isocontours of the distance are shown in white. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

The HMP-FSM is general to *any* fast sweeping application or discretization on a uniform grid, with the exception of discretizations that utilize diagonal neighbors.

4. Example problems

In this section we detail the specific problems used for numerical examples. Each problem poses its own unique numerical and mathematical challenges. The set of problems below was selected to show the versatility and effectiveness of our hybrid parallel fast sweeping method. For each example, we briefly discuss the problem and its physical representation if it has one. Then we provide a list of the parameters and functions in order to fully specify the mathematical problem.

4.1. Distance from a point in $\mathbb{R}^n$

The solution of the Eikonal equation (2) can be formulated as the arrival time of a wave propagating monotonically with speed equal to $\frac{1}{f(\mathbf{x})}$ from a boundary Γ. When the wave speed is equal to unity, then the arrival time and the distance are equal. The first examples we consider have constant right-hand-side equal to 1 and give the distance from a single point at the origin. We use these examples as a prototype problem for convergence tests, benchmarking, and illustrative purposes.

The solution to examples 1, 2, 3 are all the Euclidean distance from the origin in 2, 3, and 4 dimensions, respectively. The solution of example 1 is shown in Fig. 6. The solution is indicated by color and several evenly spaced contours are shown in white.

Example 1.

Equation:	Eikonal (2)
Dimension:	$n = 2$
Boundary:	$\Gamma = \{\mathbf{0}\}$
Boundary data:	$g(\mathbf{x}) = 0$
RHS:	$f(\mathbf{x}) = 1$
Domain:	$\Omega = [-1, 1] \times [-1, 1]$

Example 2.

Equation:	Eikonal (2)
Dimension:	$n = 3$
Boundary:	$\Gamma = \{\mathbf{0}\}$
Boundary data:	$g(\mathbf{x}) = 0$
RHS:	$f(\mathbf{x}) = 1$
Domain:	$\Omega = [-1, 1] \times [-1, 1] \times [-1, 1]$

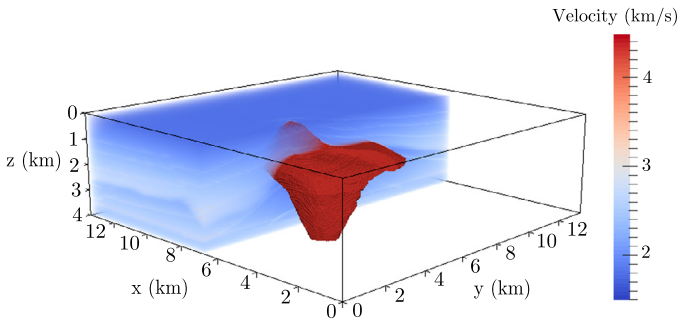

Fig. 7. A visualization of the SEG/EAGE salt dome model [3]. Wave velocity is indicated by color. The high velocity salt formation is in the center and a volume render overlay is shown in the $x > 6750\,\text{m}$ half-space. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.).

Example 3.

Equation:	Eikonal (2)
Dimension:	$n = 4$
Boundary:	$\Gamma = \{\mathbf{0}\}$
Boundary data:	$g(\mathbf{x}) = 0$
RHS:	$f(\mathbf{x}) = 1$
Domain:	$\Omega = [-1, 1] \times [-1, 1] \times [-1, 1] \times [-1, 1]$

4.2. First arrival time of seismic waves

Since the solution to the Eikonal equation is the arrival time of a wave propagating monotonically, it can be applied to the problem of elastic waves traveling through a medium. Here, we formulate the solution as the first arrival time of seismic waves propagating through a model of a section of Earth's crust. In particular, we are using the SEG/EAGE salt dome model [3]. The model gives the speed of propagation of elastic waves through the medium. This model has very high resolution data and large variation in wave speed. We chose this model because simulations at this level of resolution are very computationally expensive and require large computing resources.

The model is based on a geologic formation called a "salt dome". Fig. 7 depicts a volume render illustrating the velocity throughout the 3D domain. The high-velocity salt formation is shown in the center surrounded by relatively low-velocity rock. The model also features several thin horizontal bands of high velocity material. These require a high resolution grid to resolve. The original data are on a $13.5\,\text{km} \times 13.5\,\text{km} \times 4.0\,\text{km}$ grid with 20 m resolution in all directions. In our example problem, we simulate a wave emanating from the surface at $(9720\,\text{m}, 6860\,\text{m}, 0\,\text{m})$ and propagating through the domain. When simulating on grids other than the original, we use velocity values linearly interpolated from the original data. The discretization is given in A.1.

Example 4.

Equation:	Eikonal (2)
Dimension:	$n = 3$
Boundary:	$\Gamma = (9720\,\text{m}, 6860\,\text{m}, 0\,\text{m})$
Boundary data:	$g(\mathbf{x}) = 0$
RHS:	SEG/EAGE model from [3]
Domain:	$\Omega = [0, 13.5\,\text{km}] \times [0, 13.5\,\text{km}] \times [0, 4.0\,\text{km}]$

4.3. Hamilton–Jacobi–Bellman example

Example 5 is a Hamilton–Jacobi–Bellman equation which was also used as an example in [20]. This particular example has oscillatory characteristics and, therefore, requires more iterations for convergence than a problem with simpler characteristic curves. In [37], it was shown for that the number of iterations for the fast sweeping method to converge depends on the number of "turns" or direction changes of the characteristic curves. Since this example is oscillatory, it has many turns and requires many iterations to converge. The oscillatory speed function is given by:

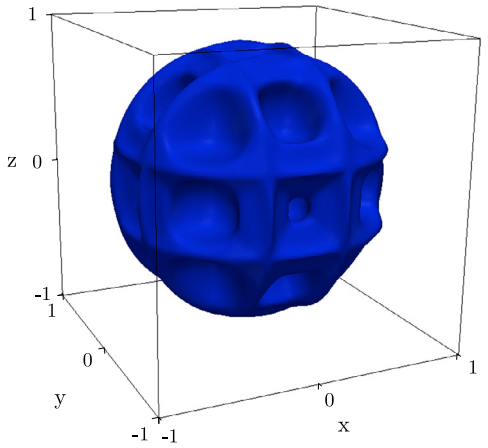

Fig. 8. Isocontour of the solution of example 5 at $u = 1.8$.

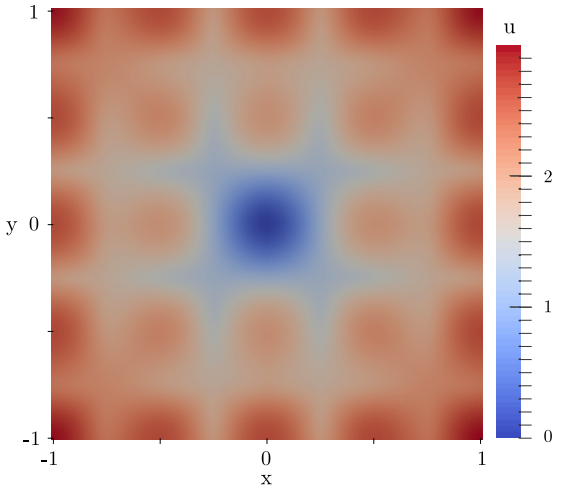

Fig. 9. Cross section of the numerical solution of example 5 at $z = 0$. Value of u is indicated by color. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

$$\mathbf{f}(\mathbf{x}, \mathbf{a}) = \mathbf{a}\left(1 + (\nabla h(\mathbf{x}) \cdot \mathbf{a})^2\right)^{-1/2}, \tag{7}$$

where

$$h(x, y, z) = \cos(2\pi x)\cos(2\pi y)\cos(2\pi z),$$

and

$$\mathcal{A} = \{\mathbf{v} \in \mathbb{R}^3 \mid \|\mathbf{v}\| \leq 1\}.$$

The domain is $[-1, 1]^3$ and the boundary value is equal to zero and imposed at the origin. A contour of the solution is shown in Fig. 8, and a cross section of the solution at $z = 0$ is shown in Fig. 9. The discrete form of equation (4) and the update formula is given in A.2.

Example 5.

Equation:	Hamilton–Jacobi–Bellman (4)
Dimension:	$n = 3$
Boundary:	$\Gamma = \mathbf{0}$
Boundary data:	$g(\mathbf{x}) = 0$
Speed function:	equation (7)

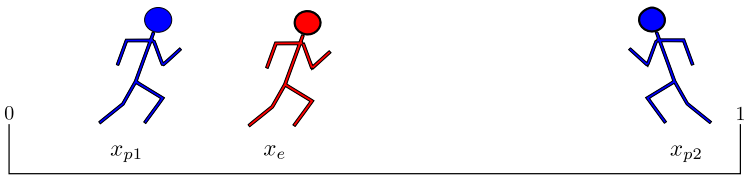

Fig. 10. Schematic of the tag-chase game along a line with one evader and two pursuers.

Cost function:	$l(\mathbf{x}) = 1$
Domain:	$\Omega = [-1, 1] \times [-1, 1] \times [-1, 1] \times [-1, 1]$

4.4. Tag-chase game on a line

The Hamilton–Jacobi–Isaacs equation gives the value function of a two-agent dynamic game. The first game we consider is the game of tag-chase for players restricted to a line segment. This example has two pursuers and one evader. The goal of the evader is to avoid capture, or being "tagged", for as long as possible. The goal of the pursuers is to tag the evader in minimum time. Fig. 10 illustrates the game.

The dynamics in this game are simple. Each player can move in either direction up to their maximum speed instantaneously. The state is specified by the locations of each player $\mathbf{x} = (x_e, x_{p1}, x_{p2})$. Formally, the dynamics are given by equation (8)

$$\mathbf{f}(\mathbf{x}, a, \mathbf{b}) = \begin{pmatrix} v_e a \\ v_p b_1 \\ v_p b_2 \end{pmatrix}, \tag{8}$$

where the max pursuer velocity is $v_p = 2$, the max evader velocity is $v_e = 1$, the control spaces are $\mathcal{A} = \{v \in \mathbb{R}^1 \mid |v| \leq 1\}$ and $\mathcal{B} = \{\mathbf{v} \in \mathbb{R}^2 \mid \|v\|_\infty \leq 1\}$, and $\mathbf{b} = (b_1, b_2)$. In order to enforce the constraints that the agents are confined to a line, the control spaces are modified at the boundaries. Specifically, $a > 0$ when $x_e = 0$, $a < 0$ when $x_e = 1$, $b_1 > 0$ when $x_{p1} = 0$, $b_1 < 0$ when $x_{p1} = 1$, $b_2 > 0$ when $x_{p2} = 0$, and $b_2 < 0$ when $x_{p2} = 1$. In this problem, the target set is given by $\Gamma = \{\mathbf{x} \mid x_{p1} = x_e \vee x_{p2} = x_e\}$. The discretization of equation (6) and the update formula is given in A.3.

Example 6.

Equation:	Hamilton–Jacobi–Isaacs (6)
Dimension:	$n = 3$
Boundary:	$\Gamma = \{\mathbf{x} \mid x_{p1} = x_e \ \vee \ x_{p2} = x_e\}$
Boundary data:	$g(\mathbf{x}) = 0$
Cost function:	$l(\mathbf{x}) = 1$
Dynamics:	equation (8)
Domain:	$\Omega = [0, 1] \times [0, 1] \times [0, 1]$

4.5. Tag-chase game on a plane

This example is another dynamic game governed by the HJI equation. In this example there is one pursuer and one evader constrained to a square. The four dimensional state is specified by the (x, y) location of each player: $\mathbf{x} = (x_p, y_p, x_e, y_e)$. A schematic of the game is given in Fig. 11.

The dynamics are that each player can move in any direction at any speed up to its maximum:

$$\mathbf{f}(\mathbf{x}, \mathbf{a}, \mathbf{b}) = \begin{pmatrix} v_p s_p \cos\theta_p \\ v_p s_p \sin\theta_p \\ v_e s_e \cos\theta_e \\ v_e s_e \sin\theta_e \end{pmatrix}, \tag{9}$$

where the control for each player is a speed and direction: $\mathbf{a} = (s_e, \theta_e)$, $\mathbf{b} = (s_p, \theta_p)$ and both are constrained to $\mathcal{A} = \mathcal{B} = [0, 1] \times [0, 2\pi)$. The dynamics are modified at the boundaries such that each player must stay within the box. E.g. when the evader is at the $x = 2$ wall, its direction of motion is confined to the left half plane: $\theta_e \in [\pi/2, 3\pi/2]$. The maximum velocity of the pursuer is given by $v_p = 2$ and the maximum evader velocity is $v_e = 1$. The target set is all points where capture has occurred: $\Gamma = \{\mathbf{x} \mid x_p = x_e \ \wedge \ y_p = y_e\}$.

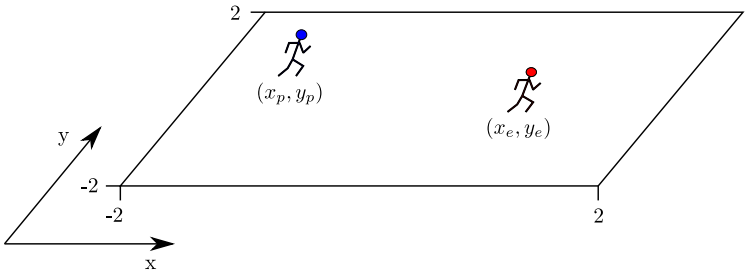

Fig. 11. Schematic of the tag-chase game on a plane. The red player is the evader and the blue player is the pursuer. The state is specified by the vector $\mathbf{x} = (x_p, y_p, x_e, y_e)$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Example 7.

Equation:	Hamilton–Jacobi–Isaacs (6)
Dimension:	$n = 4$
Boundary:	$\Gamma = \{\mathbf{x} \mid x_p = x_e \ \wedge \ y_p = y_e\}$
Boundary data:	$g(\mathbf{x}) = 0$
Cost function:	$l(\mathbf{x}) = 1$
Dynamics:	equation (9)
Domain:	$\Omega = [-2, 2] \times [-2, 2] \times [-2, 2] \times [-2, 2]$

The final example is the same tag-chase game, except that there is a physical barrier inside the domain that neither player can penetrate. The barrier is defined as the set of points in Λ. We can enforce this constraint by slightly modifying the dynamics to reflect that a player is *frozen* inside the barrier. It will always be advantageous for a player to avoid being frozen, so this method is equivalent to choosing the dynamics such that they never point into the barrier. The modified dynamics are given by

$$\mathbf{f}(\mathbf{x}, \mathbf{a}, \mathbf{b}) = \begin{pmatrix} \hat{v_p} s_p \cos\theta_p \\ \hat{v_p} s_p \sin\theta_p \\ \hat{v_e} s_e \cos\theta_e \\ \hat{v_e} s_e \sin\theta_e \end{pmatrix}, \tag{10}$$

where

$$\hat{v_p} = \begin{cases} 0 & \text{if } (x_p, y_p) \in \Lambda \\ v_p & \text{otherwise} \end{cases}, \tag{11}$$

and

$$\hat{v_e} = \begin{cases} 0 & \text{if } (x_e, y_e) \in \Lambda \\ v_e & \text{otherwise} \end{cases}. \tag{12}$$

The barrier Λ is defined as the set of points inside two concentric semi-rings of width $w = .25$ and radii $r_1 = 1$ and $r_2 = .5$. The outer ring exists between the angles $\pi/6$ and $5\pi/6$. The inner ring exists between the angles of 0 and $5\pi/12$ and between $7\pi/12$ and π. Angles are measured counterclockwise from the positive x-axis. Fig. 12 depicts a schematic of the domain including the barrier Λ. The discretization used for both examples 7 and 8 is given in A.3.

Example 8.

Equation:	Hamilton–Jacobi–Isaacs (6)
Dimension:	$n = 4$
Boundary:	$\Gamma = \{\mathbf{x} \mid x_p = x_e \ \wedge \ y_p = y_e\}$
Boundary data:	$g(\mathbf{x}) = 0$
Dynamics:	equation (10)
Domain:	$\Omega = [-2, 2] \times [-2, 2] \times [-2, 2] \times [-2, 2]$

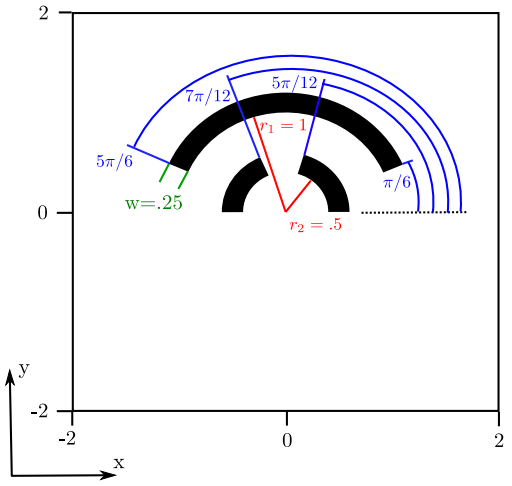

Fig. 12. Schematic for barrier in example 8. Both agents in the dynamic game are constrained to always be outside the black region.

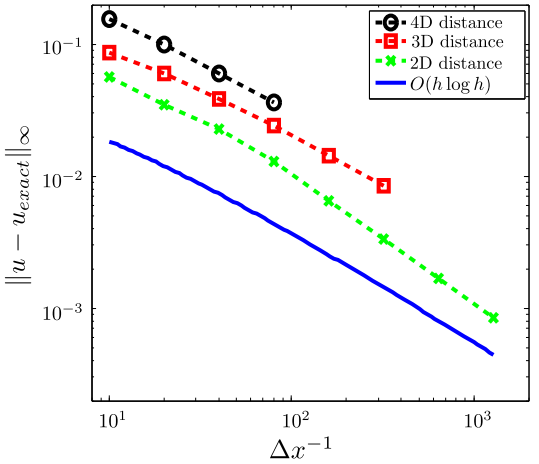

Fig. 13. Convergence of the maximum norm of error for examples 1, 2, and 3. Numerical solutions computed with the serial FSM. Also shown is a curve representing an $O(|h \log h|)$ convergence rate.

5. Validation

In this section we provide convergence results showing that the FSM converges to the exact solution for the static HJ equations for which we have an analytic solution. We then compare the solutions from the HMP-FSM to conclude that it converges to the *exact* numerical solution as the FSM.

We have an analytic solution to the simple Eikonal examples 1, 2 and 3. The exact solution is simply the Euclidean distance from the origin:

$$u(\mathbf{x}) = \|\mathbf{x}\|_2. \tag{13}$$

We computed the numerical solution for examples 1, 2, and 3 using the serial FSM. In Fig. 13, we plot the maximum norm of error as the grid is refined. In each case, the method converges to the analytic solution and appears to be doing so at a rate of $O(|h \log h|)$, which was proven to be the convergence rate in [37].

Example 6 also has an analytic solution. The problem is a dynamic game describing three "players" constrained to a line (see section 4.4). By considering each of the possible configurations of the order of the three players along the line segment, we can determine the capture time for all cases. Briefly, if $x_e < x_{p1} < x_{p2}$ then it is optimal for each player to move in the negative x direction at maximum speed until capture or stopping at the wall. If $x_{p1} < x_e < x_{p2}$, then pursuer 1 moves in the positive x direction, pursuer 2 moves in the negative x direction (both towards the evader), and the evader moves away from the nearest pursuer all at maximum speed. The other cases are determined by symmetry: let the line segment be measured in the opposite direction and then let the pursuers be interchanged. In this manner, we can enumerate the individual cases and compute the time to capture for each initial condition. Fig. 14 shows that the numerical solution converges to the exact solution as the grid is refined.

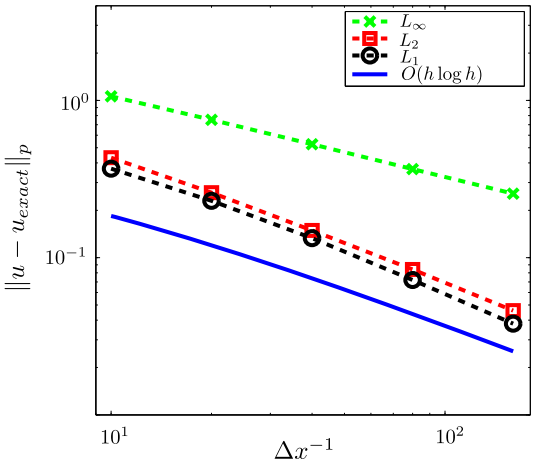

Fig. 14. Error convergence for example 6. Solution computed with the serial FSM.

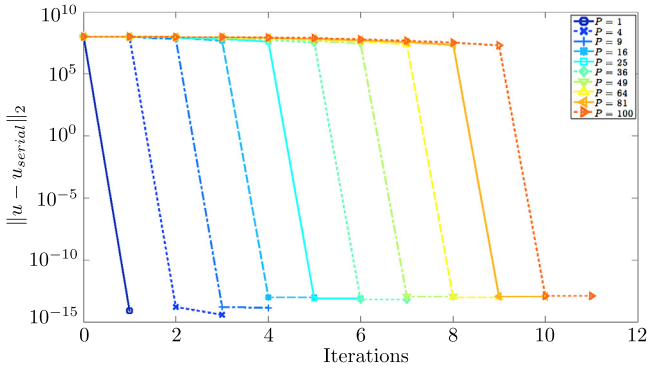

Fig. 15. As the number of partitions is increased, the number of iterations for the DDP-FSM to converge to the numerical solution increases. The method converges to the same numerical solution (to machine precision) regardless of the number of partitions.

It was proved in [13] that the HSP-FSM converges to the exact same numerical solution as the serial method in the exact same number of iterations. Therefore, for the hybrid method, the total iterations to convergence is independent of c. The ghost layer in the domain decomposition method guarantees that it also converges to the same numerical solution. The DDP-FSM will, however, require additional iterations to converge in some cases. We experiment on example 1. We recorded the maximum error between the converged numerical solution and the solution from the DDP-FSM after each iteration. Fig. 15 shows that the DDP-FSM converges to the same numerical solution as the serial method, but requires an increasing number of iterations to do so as the number of partitions is increased. In each case, the two solutions are identical within approximately machine epsilon after a finite number of iterations.

For the three simple Eikonal examples in section 4.1, we can predict the number of iterations the method will require. In [37], the author proves that, for the distance problem, the FSM will converge to the correct solution after one iteration. For our simple test problems, the algorithm is initialized at the origin, and the data propagates outward to the rest of the domain. In the first iteration, only the partitions that contain the initial boundary data will be updated. In the second iteration, the data propagates into the next 'layer' of partitions. Fig. 4 illustrates this for the 2D problem on 16 processors. Recall that with uniform partitions, it will take $O(\sqrt{p})$ iterations for the information to propagate through all the partitions. In a d dimensions, this becomes $O(\sqrt[d]{p})$.

To confirm this, we carried out a numerical experiment on the distance problem in 2D, 3D, and 4D. We tested the number of iterations to convergence on a range of total partitions. Fig. 16 shows the results. The markers are the actual iterations from the experiment and the dashed lines represent a least square power fit. The equations of the fitted curves are indicated on the plot. Based on our analysis, we would expect the iterations to scale with $p^{1/2}$, $p^{1/3}$, $p^{1/4}$ in 2D, 3D, and 4D, respectively. The results very closely agree with our analysis. A power fit of the experimental results gives exponents of 0.50, 0.34, and 0.28 in 2D, 3D, and 4D, respectively.

The numerical experiments in this section demonstrate that the FSM converges to the solution of static Hamilton–Jacobi PDEs when using the Godunov discretization schemes outlined in Appendix A. We have also demonstrated that the HMP-FSM converges to the same numerical solution as the original serial method, but may require more iterations. Using the

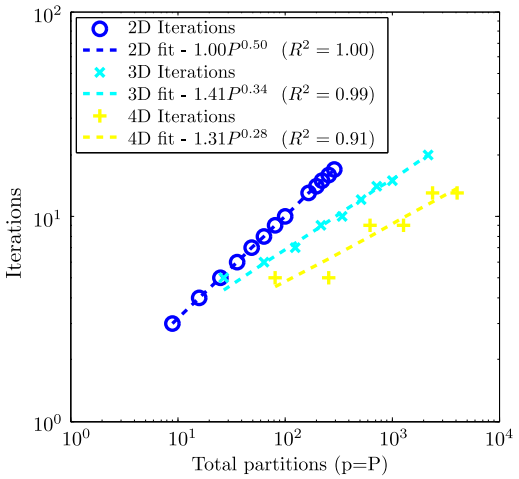

Fig. 16. The iterations to convergence depend predictably on the number of uniform partitions. Tests were conducted on examples 1, 2 and 3.

model problem, we provided an estimate for the relationship between the number of iterations and the number of domain partitions. Numerical experiments confirm the validity of that analysis.

6. Scaling and efficiency

The hybrid FSM is powerful because it can scale on large numbers of parallel processors better than the standard domain decomposition algorithm. Even in cases where the parallel efficiency is sub-ideal, it is useful because it can solve memory bottlenecks while utilizing hardware more efficiently than existing methods. The shared memory parallel fast sweeping method was invented for the Eikonal equation [13]. On a typical compute node with 16 cores, the method can solve the Eikonal equation on a grid large enough to saturate the available memory before it becomes time prohibitive. Many applications require fast computation of a solution to the Eikonal equation on domains that require more memory than is available on a single compute node. In this case, a domain decomposition strategy is needed. The hybrid parallel fast sweeping method provides a means to doing this while limiting the negative effects from standard domain partitioning techniques.

The Hamilton–Jacobi–Bellman and Hamilton–Jacobi–Isaacs equations require computing an expensive numerical minimization during the update step. On a single compute node, these problems have a prohibitive time cost before the memory is saturated. The shared memory parallel FSM is well suited for these problems because it introduces no additional work and allows for a parallel shared memory machine to quickly compute the solution. The total amount of speedup, however, is limited with the shared memory model alone. The hybrid method can be applied to eliminate the limit and compute solutions to problems that were previously intractable.

In this section, we first provide scaling results for each component of the hybrid method separately, and then the full hybrid method. Analyzing the scaling results for the coarse and fine parallel components individually allows us to better understand the hybrid method, its shortcomings, and its advantages. We implemented the algorithm in C++; the shared memory component is parallelized with OpenMP and the distributed memory component is MPI. We used the Intel C++ Compiler 13.2. All tests were done on the Stampede supercomputing cluster at the Texas Advanced Computing Center (TACC). A single compute node of stampede is configured with two Xeon E5-2680 processors with 32 GB of host memory.

6.1. Fine grained parallelism

The hybrid method uses the HSP-FSM to parallelize on the scale of a single partition. In this section we analyze the scaling of just the shared memory component for our example problems.

We solve example 2 for the distance from point in a 3D domain. This problem has an extremely low computational cost to update each node (equation (A.1)). Due to the low computational cost per node, parallel overhead and communication costs are significant in all of our tests. Fig. 17 shows the parallel speedup and efficiency on up to 16 threads on a single Stampede compute node for a range of problem sizes. The results show that parallel efficiency is poor for small problems where parallel overhead is dominant. As the problem size increases, the computation expense begins to outweigh the communication overhead.

For the 160^3 problem size, the efficiency is very high on up to 8 threads ($> 88\%$). The large drop in efficiency at 16 threads is due to the specific computer architecture. Compute nodes of Stampede consist of two Xeon E5 processors with 8 cores each. The cost to communicate between the 8 cores of a single processor is much lower than to communicate between the two processors. These results indicate that the fine scale parallelism for the Eikonal equation is most efficient on 8 cores and when the problem size per partition is $\geq 160^3$.

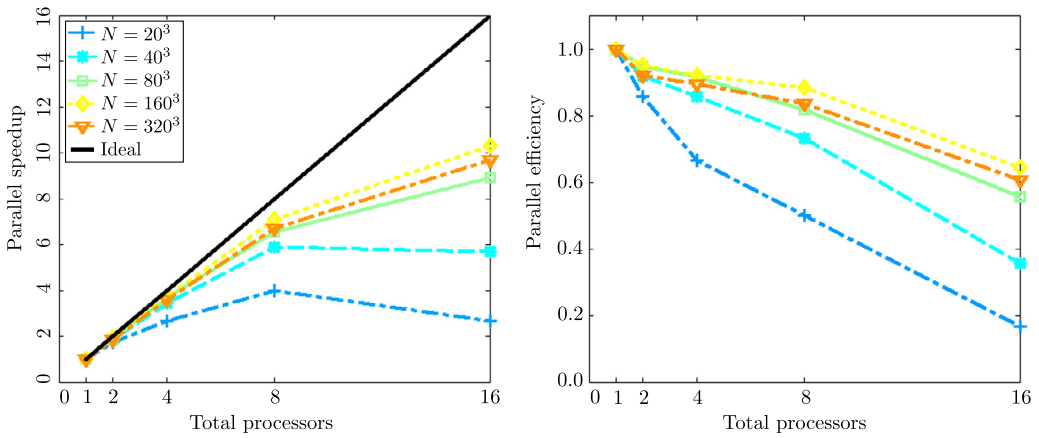

Fig. 17. Shared memory method scaling for example 2. Parallel speedup (left) and efficiency (right) for a range of problem sizes on a single compute node.

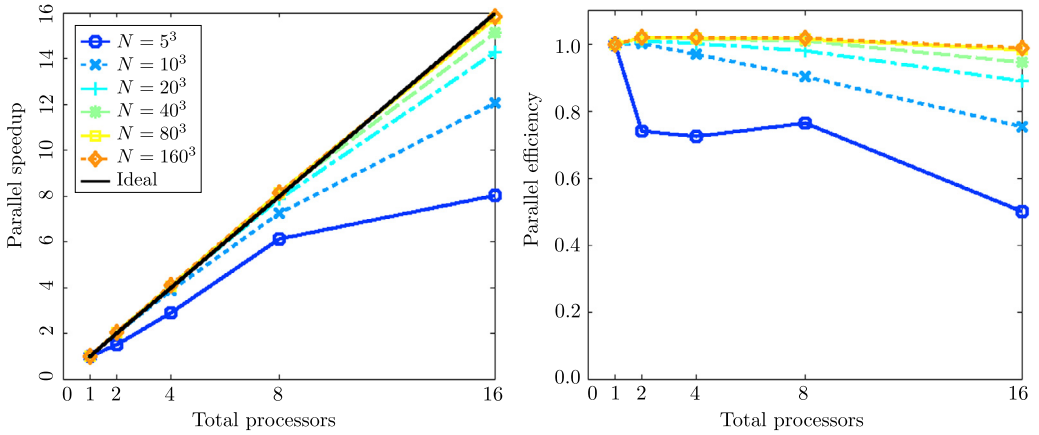

Fig. 18. Shared memory method scaling for example 5. Parallel speedup (left) and efficiency (right) for a range of problem sizes on a single compute node.

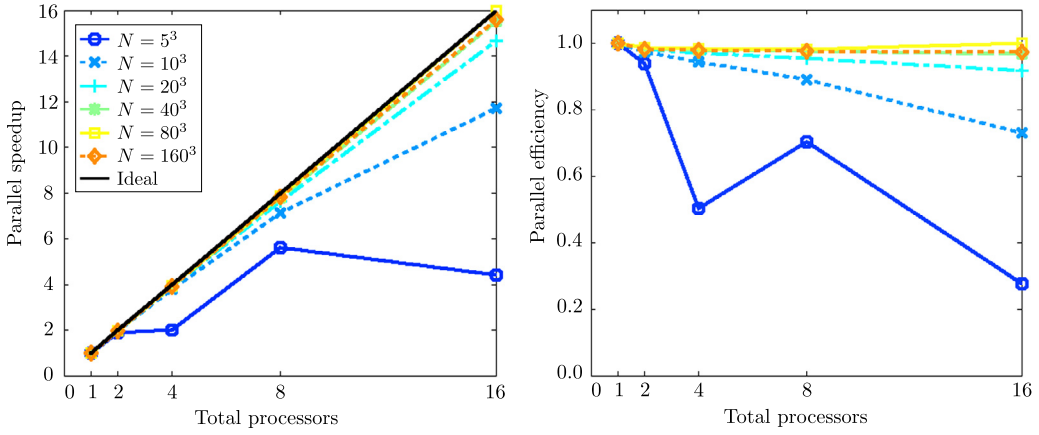

Fig. 19. Shared memory method scaling for example 6. Parallel speedup (left) and efficiency (right) for a range of problem sizes on a single compute node.

The 3D HJB and HJI examples (5) and (6) are much more computationally intensive. We apply the shared memory method to those examples and show the results in Figs. 18 and 19. In both cases, efficiency quickly approaches unity as the problem size is increased. For problems $\gtrsim 40^3$, the efficiency of the fine scale parallelism is near ideal. Also, since computational expense is much higher, the communication cost between processors is not significant even when utilizing all 16 threads on a compute node. These results indicate that the method will be near ideally efficient when each partition is larger than 40^3.

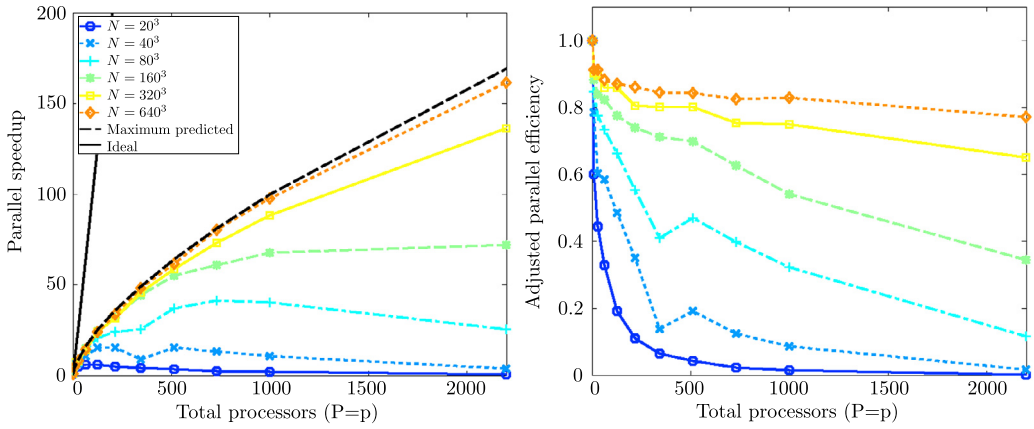

Fig. 20. Distributed memory method scaling for example 2. Parallel speedup (left) and adjusted efficiency (right) for a range of problem sizes on a large distributed computing cluster.

Our results show that for the Eikonal examples, the fine scale parallel method loses efficiency drastically when using more than 8 cores and parallel overhead is significant for even the large problems. For the HJB and HJI examples, parallel overhead is negligible for problems larger than 40^3 and is efficient on up to 16 cores.

6.2. Coarse parallelism

In this section, we present results for the pure distributed memory algorithm. The drawback to this method is that by partitioning the domain into subdomains, one reduces the distance over which information can propagate during a single sweep. The most significant factor in determining the efficiency of this method is the additional iterations it requires to reach convergence.

In section 5 we determined analytically and numerically that the number of iterations to convergence scales as $O(\sqrt[d]{P})$. With knowledge of how the number of iterations scale with the number of partitions, we can estimate the parallel speedup and efficiency. Parallel speedup is given by $S_p = \dfrac{T_1}{T_p}$. The execution time T_p will scale linearly with the number of iterations and, ideally, inversely with the number of processors, so we expect the parallel speedup to scale as $p^{1-1/d}$ for a problem in d dimensions.

We ran numerical experiments, computing the solution of the Eikonal equation with the domain decomposition method. Fig. 20 (left) shows the parallel speedup over a range of processors. For comparison, we plot ideal speedup ($S_p = p$) and predicted speedup ($S_p = p^{1-1/d}$). When the size of the problem becomes large, the speedup very closely agrees with the predicted value.

Parallel efficiency is often used as a measure of how efficiently compute resources are being used. Parallel efficiency on p processors is given by $E_p = \dfrac{T_1}{pT_p}$. A parallel program with ideal speedup will have parallel efficiency equal to unity. For the domain decomposition FSM, the algorithm introduces more work (iterations) as the number of processors increases, so even if there were no parallel overhead, the efficiency would still be sub-ideal. To determine how well we are utilizing the parallel resources, we use a new measure that is normalized by the total number of iterations. The *adjusted parallel efficiency* is given by:

$$\hat{E}_p = \frac{E_p I_1}{I_p} = \frac{T_1 I_1}{p I_p T_p}, \tag{14}$$

where I_p is the number of iterations on p processors.

This new measure allows us to distinguish between cases where efficiency loss is due to the algorithm or the parallel overhead associated with the implementation. If the adjusted parallel efficiency is close to 1, then the parallel overhead is small and the implementation is efficiently using the processors. Adjusted efficiency results for example 2 are shown in Fig. 20 (right). The results show that for large problems, our implementation is effectively using the available processors. Even on 2197 processors, the adjusted efficiency is $\approx 80\%$. The large discrepancy between ideal speedup and actual speedup is due to the additional iterations, which are inherent in the domain decomposition method.

We conducted the same experiment in 4D (example 3) on up to 4096 processors. Fig. 21 (left) shows that the parallel speedup is clearly less than ideal, but as the problem size increases, it approaches the predicted speedup. In Fig. 21 (right), we show the adjusted efficiency which demonstrates that even on 4096 processors, this approach is quite efficient ($\hat{E}_{4096} \approx 73\%$ for the 80^4 grid).

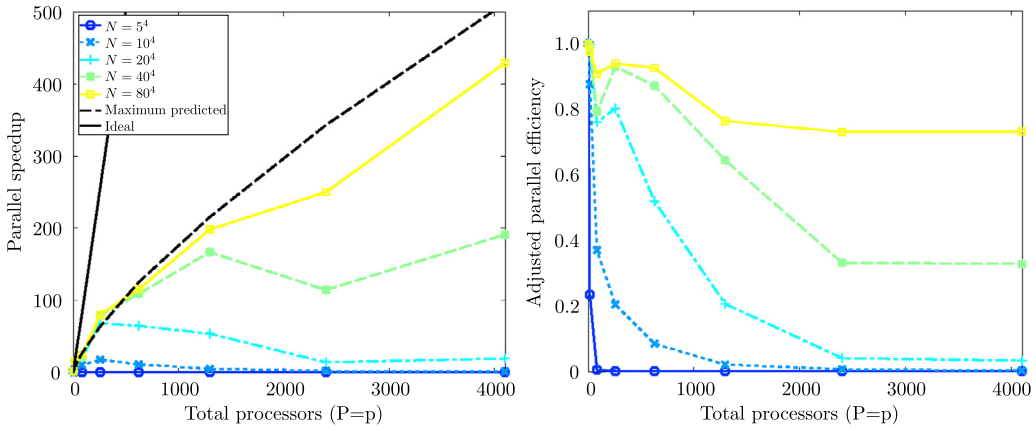

Fig. 21. Distributed memory method scaling for example 3. Parallel speedup (left) and adjusted efficiency (right) for a range of problem sizes on a large distributed computing cluster.

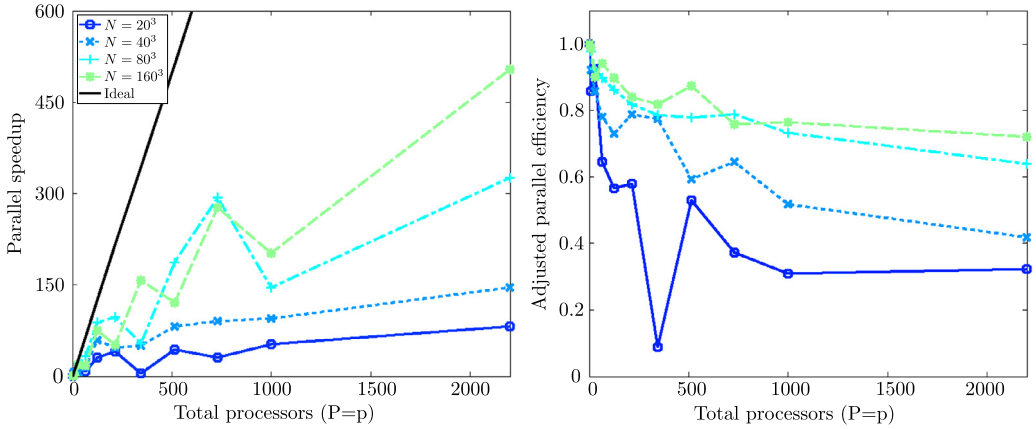

Fig. 22. Distributed memory method scaling for example 5. Parallel speedup (left) and adjusted efficiency (right) for a range of problem sizes on a large distributed computing cluster.

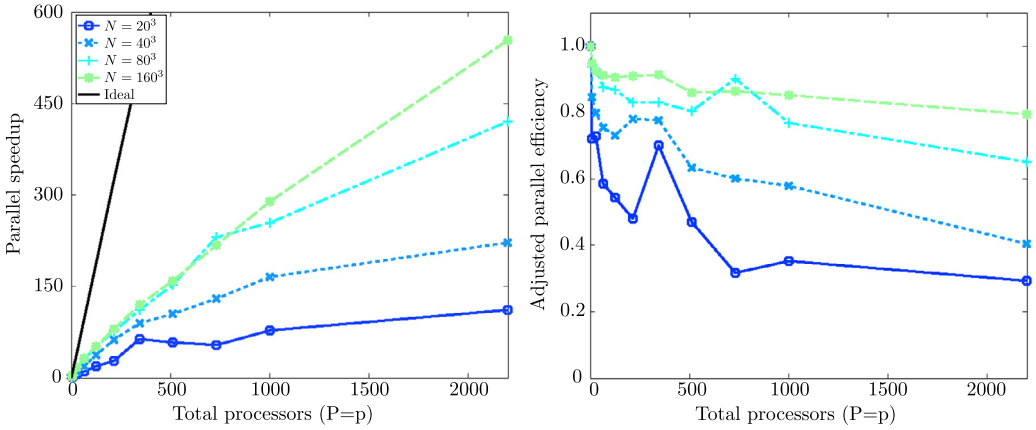

Fig. 23. Distributed memory method scaling for example 6. Parallel speedup (left) and adjusted efficiency (right) for a range of problem sizes on a large distributed computing cluster.

We did the same experiment for the 3D HJI and HJB example problems. Figs. 22, 23 show the scaling and adjusted efficiency for examples 5 and 6, respectively. In both cases, the adjusted efficiency is very high for moderate to large size problems.

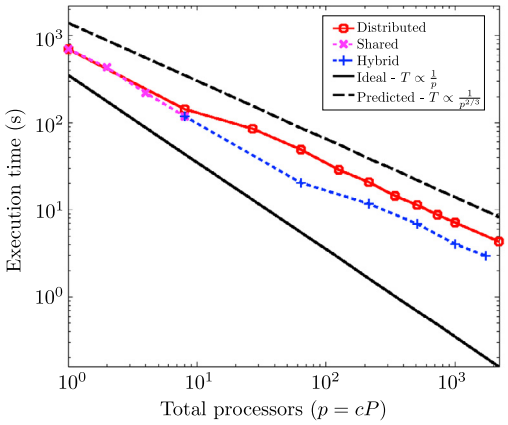

Fig. 24. Strong scaling for the 3D Eikonal equation example (example 2) on a grid with $N = 640^3$.

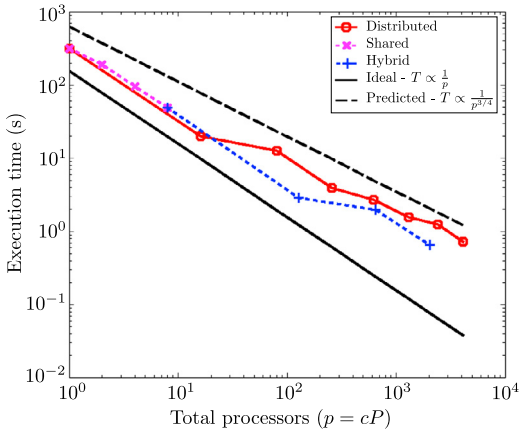

Fig. 25. Strong scaling for the 4D Eikonal equation example (example 3) with $N = 80^4$.

6.3. Hybrid method

In previous sections, we have motivated the hybrid method and discussed the drawbacks of both the shared memory parallel fast sweeping method and the domain decomposition fast sweeping method. We also tested the scaling of both methods, which are the fine and coarse grained parallel components respectively. In this section we present the scaling and efficiency results of the hybrid method.

We solved the 3D distance problem (example 2) on a grid $N = 640^3$ with the hybrid method on up to approximately 2000 processors. We can choose the number of shared cores per partition up to the limitations of the hardware. For the compute nodes of the Stampede supercomputer, 16 is the maximum shared memory cores per compute node. In most cases, it is best to choose $c = 16$ because that produces the largest constant factor of improvement from our analysis. However, for the Eikonal equation, our results from section 6.1 showed that the most efficient method was the hybrid method with $c = 8$. This is because the computation work per grid point is very small and the shared interconnect within the compute node is inefficient for this particular case. Fig. 17 shows the steep drop off in efficiency for 16 shared processors. Fig. 24 shows the total execution time for the three methods for a range of processors. This one plot illustrates all the drawbacks of the existing methods and the benefits of the hybrid method. The shared memory method scales well, but is strictly limited to small numbers of processors. The domain decomposition FSM can be extended to large numbers of processors, but scales at a less than ideal rate. The hybrid method achieves an approximately constant factor of improvement over the domain decomposition FSM. Later in this section we will quantify this constant factor for all examples. For reference, all speedup plots will include lines indicating ideal scaling ($T \propto 1/p$) and predicted scaling ($T \propto 1/p^{1-1/d}$).

The results for the 4D problem are similar. Fig. 25 shows the scaling results for example 3 with $N = 80^4$. The execution time scales as expected, but the hybrid method gives an approximately constant factor of improvement.

The HJI and HJB examples have much more computation per node, and showed better fine grained scaling results. We choose $c = 16$ and expect the hybrid method to have an even more significant improvement over the domain decomposition FSM. Figs. 26 and 27 show the strong scaling results on the 3D HJB and HJI problems (examples 5 and 6, respectively) computed on grids with $N = 160^3$. In both cases, the hybrid method gives a significant improvement. The non-smooth

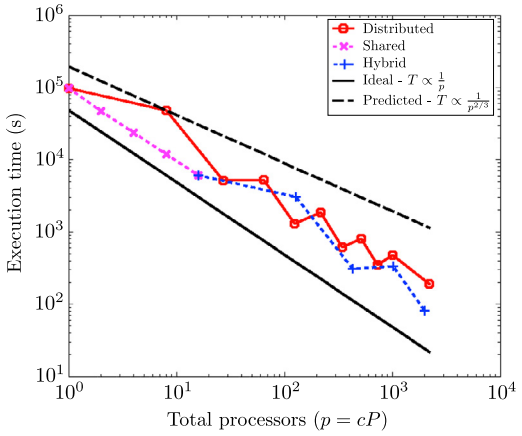

Fig. 26. Strong scaling results for the 3D HJB example problem (example 5) on a grid with $N = 160^3$.

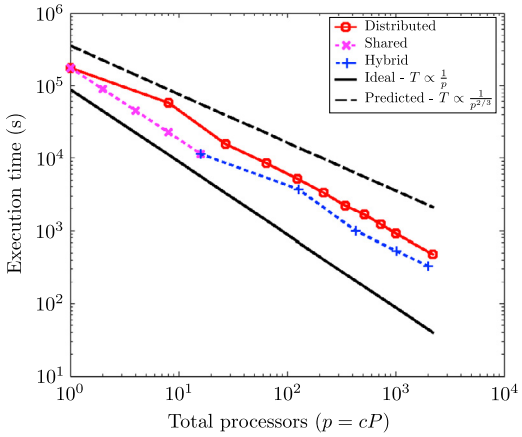

Fig. 27. Strong scaling results for the 3D HJI example problem (example 6) on a grid with $N = 160^3$.

scaling rate on the 3D HJB problem is due to the nature of the problem. This particular problem has oscillatory characteristic curves. In [37], the author showed that the number of iterations for the FSM to converge depends on the number of "turns" of the characteristic curves of the Eikonal equation. It can also be shown for general HJ equations that the number of iterations of the domain decomposition method depends on the number of processor boundary crossings. When a partition boundary happens to cross a characteristic curve multiple times, the number of iterations sharply increases. When the characteristic goes through the center of most partitions, then fewer iterations are required. The scaling curves in Fig. 26 are erratic because for one decomposition, the characteristic curves' "turns" may be aligned with a partition boundary, while for another decomposition, the turns are not.

The scaling results from the 4D HJI example (example 7) computed on a grid with $N = 40^4$ are shown in Fig. 28. Again, both the DDP-FSM and HMP-FSM scale at the predicted rate, but the HMP-FSM is faster by an approximately constant factor.

Our scaling results demonstrate that our hybrid parallel method is faster and more efficient than existing parallel methods when implemented on the exact same hardware. This hybrid method is designed to take advantage of computer architecture that is very common to modern supercomputers. Standard domain decomposition does not utilize the compute resources ideally. Each of the strong scaling graphs shows that the hybrid method is faster than the domain decomposition method by an approximate constant factor. In order to quantify that improvement, we fit a curve to the data and measure the improvement with the fitted curve. The scaling data behave like a power curve $T = ap^b$ where T is execution time, p is processors, and a, b are free parameters to fit. If $f_1(p)$ is the curve fitted to the domain decomposition data and $f_2(p)$ is the curve fitted to the hybrid method, then the *algorithmic speedup* is given by $S_{alg} = \frac{f_1}{f_2}$. Fig. 29 illustrates the fitting and speedup measurement procedure on two example problems.

The domain decomposition method and the hybrid method were tested on a range of processors from $p = 1$ to $p = 4096$. The algorithmic speedup is approximately constant over the range of p, but varies slightly. Table 1 gives the algorithmic speedup for several example problems at an intermediate number of processors: $p = 100$. In all cases, the hybrid method gives a significant speedup over the domain decomposition method.

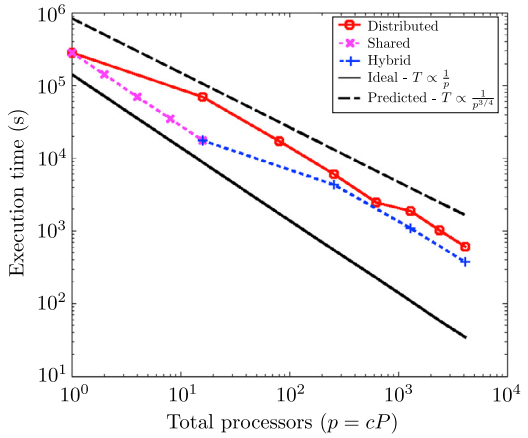

Fig. 28. Strong scaling results for the 4D HJI example problem (example 7) on a grid with $N = 40^4$.

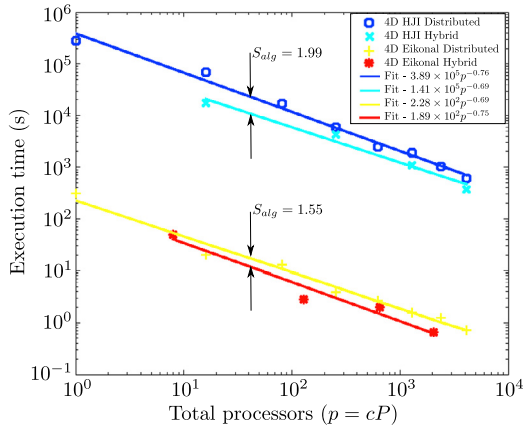

Fig. 29. Computing the algorithmic speedup for two example problems at $p = 100$.

Table 1
Algorithmic speedup for several example problems.

Example	Grid size	p	c	Predicted $S_{alg} = \sqrt[d]{c}$	Measured S_{alg}
2	640^3	100	8	2.00	1.72
3	80^4	100	8	1.68	1.55
5	160^3	100	16	2.52	1.49
6	160^3	100	16	2.52	1.79
7	40^4	100	16	2.00	1.99

7. Example applications

In the previous section, we demonstrated that the hybrid parallel FSM is more efficient than existing methods and can be applied to very large problems. Here, we choose two example applications that require a highly parallel method and present the results.

7.1. First arrival time of seismic waves

The first example application we consider is computing the first arrival times of seismic waves in a large domain with a complex wave velocity profile (example 4). This example presents a computational challenge because the domain and associated data are huge. The velocity model data itself is approximately 1 GB. If we want to simulate waves at a resolution equal to or higher than this, memory quickly becomes a limiting factor. The hybrid parallel FSM is uniquely equipped to solve this problem while limiting the negative effects of domain decomposition.

Fig. 30 shows three snapshots of a seismic wave traversing the SEG/EAGE salt dome model. Figs. 31 and 32 show a cross section of the velocity profile with evenly spaced contours of the wave arrival time. This example problem has been studied previously; our Figs. 31 and 32 are analogous to Figs. 6 and 8 from [31]. Notice that the wavefront quickly expands

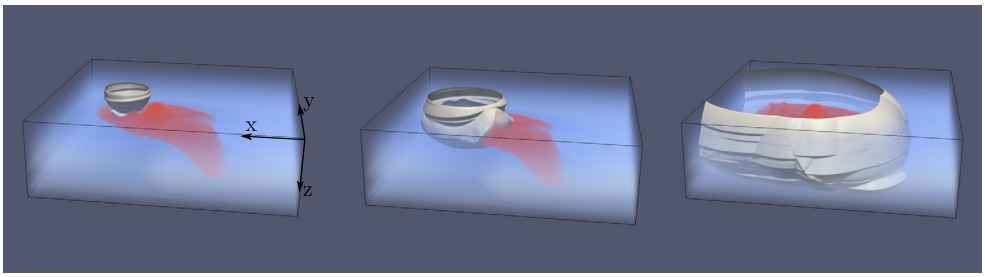

Fig. 30. Visualization of the seismic wavefront traversing the SEG/EAGE salt dome model [3] (example 4). The white surface is the wavefront pictured at $t = 0.8$ s (left), $t = 1.3$ s (center), and $t = 2.6$ s (right). Wave velocity is indicated by color of the semi-transparent volume render. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

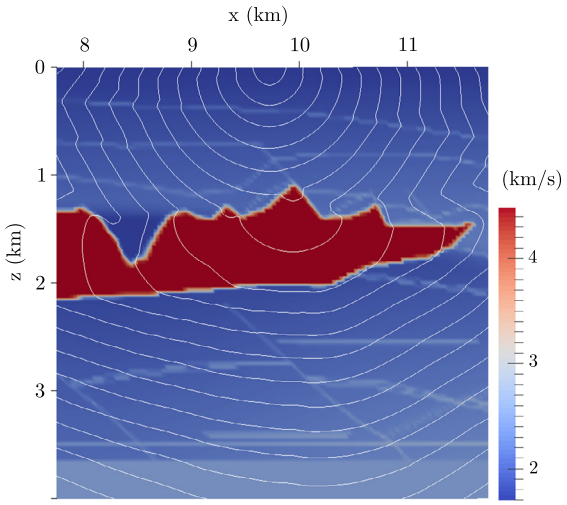

Fig. 31. Cross section of the solution to example 4 at $y = 6.86$ km. Color indicates wave velocity and the while lines are evenly spaced arrival time contours. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

into the high velocity region within the salt dome. This computation also resolves the wave behavior in the relatively high-velocity horizontal striations. All of these visualizations show results from a computation on a grid with $\Delta x = \Delta y = \Delta z = 10$ m. We also computed the solution on a grid with 5 m resolution and approximately 20 billion total grid points to demonstrate the viability of the method on extremely large grids. The memory requirements were approximately 1TB and the computation took 90 seconds on 4096 processors on the Stampede supercomputer. The extremely fine resolution allows for the simulation to resolve very fine physical features.

7.2. Dynamic games

The second example application we consider is computing the cost-to-go function, and indirectly the optimal control for each agent, in a dynamic game. In section 5, we confirmed that the HJI discretization is convergent for the 3D tag chase game, example 6. Here we compute results on the 4D problem. Examples 7 and 8 are the tag-chase game with one pursuer and one evader constrained to a 2D plane. These problems are four dimensional and require a very expensive minimization operation in the update function (equation (A.3)) and become compute bound very quickly. To compute the solution on a grid with 40 nodes in each dimension takes on the order of several days for the serial method. The hybrid parallel FSM is well suited to solve this problem in a reasonable amount of time and make tractable the solution of larger problems.

During the sweeping procedure, the arguments **a** and **b** of the minimization can be recorded at each grid node, giving the behavior of the evader and pursuer, respectively, at each state **x**. Then we can compute trajectories of a single game by choosing an initial condition and evolving the system according to the dynamics (equation (9)) until the target state (capture) is reached. Once the value function and control have been found in the entire state space, we can use the results to compute all trajectories beginning in our domain.

We computed the solution to examples 7 and 8. The two problems are identical except for the introduction of a barrier for example 8. The computation was done on a grid 40^4 and took ≈ 6 minutes on 4096 processors. Fig. 33 shows the outcome of the game from four initial conditions (a), (b), (c), and (d), respectively: $\mathbf{x} = (-1.04, -0.47, 1.79, -1.94)$, $\mathbf{x} = (1.62, 0.35, -0.26, -0.96)$, $\mathbf{x} = (-1.24, -0.50, 1.19, 1.33)$, and $\mathbf{x} = (1.54, 1.68, -1.55, -0.36)$. Fig. 34 shows the equivalent results from the game with the barrier. Notice that the trajectories (a) and (b) do not change with the addition of the

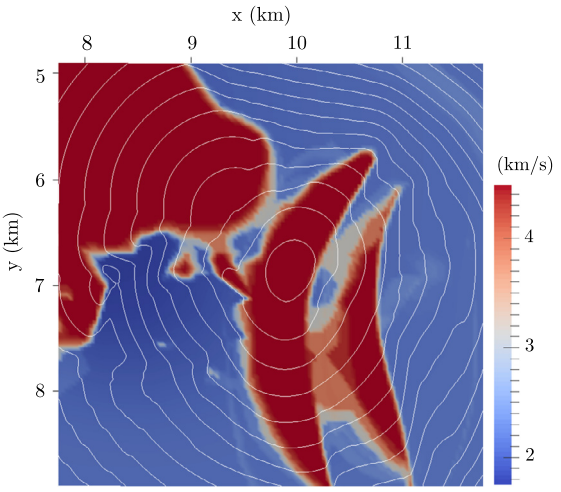

Fig. 32. Cross section of the solution to example 4 at $z = 1.38$ km. Color indicates wave velocity and the while lines are evenly spaced arrival time contours. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

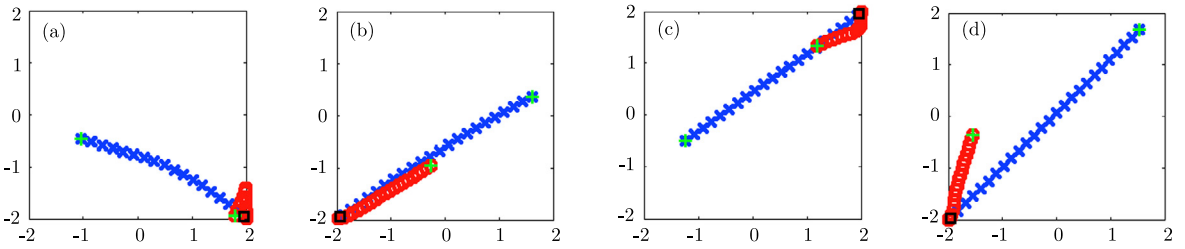

Fig. 33. Trajectories for four initial states of the dynamic game example 7. A blue x indicates the location of the pursuer at an instant of time and a red circle indicates the evader. Each agent's initial location is at the green cross. The game ends at capture, which is indicated by the black square. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

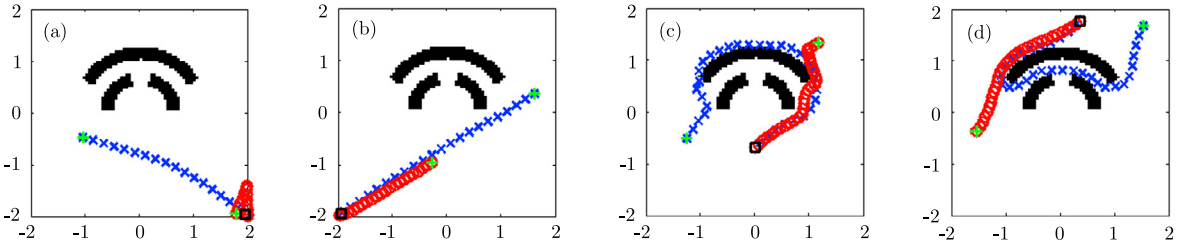

Fig. 34. Trajectories for four initial states of the dynamic game example 8. A blue x indicates the location of the pursuer at an instant of time and a red circle indicates the evader. Each agent's initial location is at the green cross. The game ends at capture, which is indicated by the black square. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

barrier. For these initial conditions, the barrier has no impact on the outcome. For the other cases, however, both agents' behavior is drastically altered by the introduction of the barrier.

8. Conclusion

We have presented a hybrid parallel method for static Hamilton–Jacobi equations. By combining two existing parallel approaches, we are able to more efficiently utilize existing computational hardware. For *all* of our example problems, the HMP-FSM outperformed the DDP-FSM on identical computer hardware. This is purely algorithmic speedup, and is generally compatible with any hybrid shared/distributed computer architecture. The algorithmic speedup results presented in this work strongly depend on the hardware on which the method was implemented. More modern hardware could lead to this method becoming more advantageous. For instance, San Diego Supercomputer Center's Comet went online in 2015 and has 24 shared memory cores per compute node ($c \leq 24$). Since we predict the algorithmic speedup to depend on $\sqrt[d]{c}$, we can expect the HMP-FSM to be 15% more effective (in 3D) on Comet than on Stampede. By implementing the HSP-FSM on coprocessors and/or GPGPUs, the benefit could be even greater.

Table 2
Summary of speedup data on largest number of processors.

Example	Grid size	Processors	Cores per partition	Execution time (s)	Speedup	Efficiency
2	640^3	1728	8	2.958	237	0.0958
3	80^4	2048	8	0.658	474.7	0.232
5	160^3	2000	16	82.0	1181	0.590
6	160^3	2000	16	328.5	538.5	0.269

To our knowledge, our results show speedup values at least an order of magnitude larger than any other published work on parallel fast methods for static HJ equations. Table 2 summarizes some of the notable speedup results.

Acknowledgements

This research was supported by ONR N00014-11-1-0027 and NSF DMS 1412695 and by the W.M. Keck Foundation. We would also like to acknowledge support by grant ARO W911NF-16-1-0136 and the NSF under the DMREF program DMR-1534264. The authors acknowledge computational resources from the "Center for Scientific Computing at UCSB" and NSF Grant CNS-0960316.

Appendix A. Discrete update formulae

All of our example problems require a numerical discretization that gives an explicit update formula for a particular node. Here we give the update formula for each of the Eikonal, HJB, and HJI equations. For simplicity, we give the formulae in two dimensions, but the extension to higher dimensions is straightforward. For the derivation of each or further information, see the corresponding reference.

A.1. Discrete Eikonal equation

We use the Godunov upwind discretization of [29,37]. The discrete form of equation (2) in 2D is:

$$\left(\frac{(u_{i,j}-u_{xmin})^+}{\Delta x}\right)^2+\left(\frac{(u_{i,j}-u_{ymin})^+}{\Delta y}\right)^2=f_{i,j}^2, \tag{A.1}$$

where $u_{xmin}=\min(u_{i-1,j},u_{i+1,j})$ and $u_{ymin}=\min(u_{i,j-1},u_{i,j+1})$ and

$$a^+=\begin{cases}a, & a>0\\ -a, & a\le 0\end{cases}.$$

A procedure for solving this quadratic equation for $u_{i,j}$ is given in [37].

A.2. Discrete Hamilton–Jacobi–Bellman equation

We use a Godunov upwind discretization of the Hamilton–Jacobi–Bellman equation. This was first developed in [20], however, we use a slightly different notation that allows for a more general control input α. The discrete form of equation (4) in 2D is:

$$\min_{\alpha\in\mathcal{A}}\left\{(-\mathrm{f}_x)^+\frac{u_{i,j}-u_{i-1,j}}{\Delta x}+(-\mathrm{f}_x)^-\frac{u_{i+1,j}-u_{i,j}}{\Delta x}+(-\mathrm{f}_y)^+\frac{u_{i,j}-u_{i,j-1}}{\Delta y}+(-\mathrm{f}_y)^-\frac{u_{i,j+1}-u_{i,j}}{\Delta y}\right\}=l(\mathbf{x}),$$

which can be solved explicitly for $u_{i,j}$:

$$u_{i,j}=\min_{\alpha\in A}\left\{-\left(\frac{\frac{(\mathrm{f}_x)^-u_{i-1,j}-(\mathrm{f}_x)^+u_{i+1,j}}{\Delta x}+\frac{(\mathrm{f}_y)^-u_{i,j-1}-(\mathrm{f}_y)^+u_{i,j+1}}{\Delta y}-l(\mathbf{x})}{\frac{|\mathrm{f}_x|}{\Delta x}+\frac{|\mathrm{f}_y|}{\Delta y}}\right)\right\}. \tag{A.2}$$

The update equation (A.2) is discontinuous and may contain multiple local minima. A minimization algorithm must be chosen accordingly.

A.3. Discrete Hamilton–Jacobi–Isaacs equation

The Hamilton–Jacobi–Isaacs equation is similar to the HJB except that the goal of one player is to minimize the value function while the goal of the other player is to maximize it. The FSM for the HJI equation has been studied and a semi-Lagrangian discretization has been developed [17,16]. We choose, however, to adapt the Godunov discretization of [20] to the HJI equation. The discrete form of equation (6) reads:

$$\min_{b\in\mathcal{B}}\left\{\max_{a\in\mathcal{A}}\left\{(-\mathrm{f}_x)^+\frac{u_{i,j}-u_{i-1,j}}{\Delta x}+(-\mathrm{f}_x)^-\frac{u_{i+1,j}-u_{i,j}}{\Delta x}+(-\mathrm{f}_y)^+\frac{u_{i,j}-u_{i,j-1}}{\Delta y}+(-\mathrm{f}_y)^-\frac{u_{i,j+1}-u_{i,j}}{\Delta y}\right\}\right\}=l(\mathbf{x}).$$

The corresponding explicit update formula is:

$$u_{i,j}=-\min_{b\in\mathcal{B}}\left\{-\min_{a\in\mathcal{A}}\left\{-\left(\frac{\frac{(\mathrm{f}_x)^-u_{i-1,j}-(\mathrm{f}_x)^+u_{i+1,j}}{\Delta x}+\frac{(\mathrm{f}_y)^-u_{i,j-1}-(\mathrm{f}_y)^+u_{i,j+1}}{\Delta y}-l(\mathbf{x})}{\frac{|\mathrm{f}_x|}{\Delta x}+\frac{|\mathrm{f}_y|}{\Delta y}}\right)\right\}\right\}. \tag{A.3}$$

References

[1] Ken Alton, Ian M. Mitchell, Fast marching methods for stationary Hamilton–Jacobi equations with axis-aligned anisotropy, SIAM J. Numer. Anal. 47 (1) (2008) 363–385.
[2] Gene M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference – AFIPS '67 (Spring), 1967, p. 483.
[3] F. Aminzadeh, N. Burkhard, T. Kunz, L. Nicoletis, F. Rocca, 3-D modeling project: 3rd report, Lead. Edge 14 (February) (1995) 125.
[4] Martino Bardi, Italo Capuzzo-Dolcetta, Optimal Control and Viscosity Solutions of Hamilton–Jacobi–Bellman Equations, Birkhäuser, 2008.
[5] Michelle Boue, Paul Dupuis, Markov chain approximations for deterministic control problems with affine dynamics, SIAM J. Numer. Anal. 36 (3) (1999) 667–695.
[6] Michael Breuß, Emiliano Cristiani, Pascal Gwosdek, Oliver Vogel, An adaptive domain–decomposition technique for parallelization of the fast marching method, Appl. Math. Comput. 218 (1) (September 2011) 32–44.
[7] A. Bruss, The Eikonal equation: some results applicable to computer vision, J. Math. Phys. 23 (5) (1982) 890–896.
[8] Adam Chacon, Alexander Vladimirsky, Fast two-scale methods for Eikonal equations, SIAM J. Sci. Comput. (2012) 1–26.
[9] Adam Chacon, Alexander Vladimirsky, A parallel two-scale method for Eikonal equations, SIAM J. Sci. Comput. 37 (1) (2015) A156–A180.
[10] Weitao Chen, Ching-Shan Chou, Chiu-Yen Kao, Lax–Friedrichs fast sweeping methods for steady state problems for hyperbolic conservation laws, J. Comput. Phys. 234 (February 2013) 452–471.
[11] Emiliano Cristiani, Maurizio Falcone, A fast marching method for pursuit-evasion games, Commun. SIMAI Congr. 1 (2007) 1–6.
[12] Florian Dang, Nahid Emad, Fast iterative method in solving Eikonal equations: a multi-level parallel approach, Proc. Comput. Sci. 29 (2014) 1859–1869.
[13] Miles Detrixhe, Frédéric Gibou, Chohong Min, A parallel fast sweeping method for the Eikonal equation, J. Comput. Phys. 237 (March 2013) 46–55.
[14] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.
[15] Björn Engquist, Brittany D. Froese, Yen-Hsi Richard Tsai, Fast sweeping methods for hyperbolic systems of conservation laws at steady state, J. Comput. Phys. 255 (December 2013) 316–338.
[16] Simone Cacace, Maurizio Falcone, Emiliano Cristiani, A local ordered upwind method for Hamilton–Jacobi and Isaacs equations, World Congress 18 (1) (2011) 6800–6805.
[17] Maurizio Falcone, Emiliano Cristiani, Fully discrete schemes for the value function of pursuit-evasion games with state constraints, in: Odile Pourtallier, Vladimir Gaitsgory, Pierre Bernhard (Eds.), Advances in Dynamic Games and Their Applications, Birkhäuser Boston, Boston, 2009, pp. 177–206.
[18] W.K. Jeong, R.T. Whitaker, A fast iterative method for Eikonal equations, SIAM J. Sci. Comput. 30 (5) (2008) 2512–2534.
[19] C. Kao, Lax–Friedrichs sweeping scheme for static Hamilton–Jacobi equations, J. Comput. Phys. 196 (1) (May 2004) 367–391.
[20] Chiu-Yen Kao, Stanley Osher, Yen-Hsi Tsai, Fast sweeping methods for static Hamilton–Jacobi equations, SIAM J. Numer. Anal. 42 (6) (January 2005) 2612–2632.
[21] Ron Kimmel, James A. Sethian, Computing geodesic paths on manifolds, Proc. Natl. Acad. Sci. USA 95 (15) (July 1998) 8431–8435.
[22] Ron Kimmel, James A. Sethian, Optimal algorithm for shape from shading and path planning, J. Math. Imaging Vis. 14 (2001) 2001.
[23] Pierre Lesaint, Pierre-Arnaud Raviart, On a finite element method for solving the neutron transport equation, in: Carl de Boor (Ed.), Mathematical Aspects of Finite Elements in Partial Differential Equations, Academic Press, New York, 1974, pp. 89–123.
[24] Siwei Li, Alexander Vladimirsky, Sergey Fomel, First-break traveltime tomography with the double-square-root Eikonal equation, Geophysics 78 (6) (2013).
[25] Stanley J. Osher, Ronald Fedkiw, Level Set Methods and Dynamic Implicit Surfaces, Springer-Verlag, New York, NY, 2002.
[26] Stanley J. Osher, A level set formulation for the solution of the Dirichlet problem for Hamilton–Jacobi equations, SIAM J. Math. Anal. 24 (5) (1993) 1145–1152.
[27] Jianliang Qian, Yong-Tao Zhang, Hong-Kai Zhao, A fast sweeping method for static convex Hamilton–Jacobi equations, J. Sci. Comput. 31 (1–2) (March 2007) 237–271.
[28] N. Rawlinson, M. Sambridge, Wave front evolution in strongly heterogeneous layered media using the fast marching method, Geophys. J. Int. 156 (3) (March 2004) 631–647.
[29] E. Rouy, A. Tourin, A viscosity solution approach to shape-from-shading, SIAM J. Numer. Anal. 29 (3) (June 1992) 867–884.
[30] James A. Sethian, A fast marching level set method for monotonically advancing fronts, Proc. Natl. Acad. Sci. 93 (1996) 1591–1595.
[31] James A. Sethian, A. Mihai Popovici, 3-D traveltime computation using the fast marching method, Geophysics 64 (2) (1999) 516–523.
[32] James A. Sethian, Alexander Vladimirsky, Ordered upwind methods for static Hamilton–Jacobi equations: theory and algorithms, SIAM J. Numer. Anal. 41 (1) (January 2003) 325–363.
[33] Yen-Hsi Richard Tsai, Li-Tien Cheng, Stanley Osher, Hong-Kai Zhao, Fast sweeping algorithms for a class of Hamilton–Jacobi equations, SIAM J. Numer. Anal. 41 (2003) 673–694.
[34] John N. Tsitsiklis, Efficient algorithms for globally optimal trajectories, IEEE Trans. Autom. Control 40 (1995) 1528–1538.
[35] John E. Vidale, Finite-difference calculation of travel times, Bull. Seismol. Soc. Am. 78 (6) (1988) 2062–2076.
[36] John E. Vidale, Finite-difference calculation of traveltimes in three dimensions, Geophysics 55 (5) (1990) 521.
[37] Hongkai Zhao, A fast sweeping method for Eikonal equations, Math. Comput. 74 (250) (2004) 603–627.
[38] Hongkai Zhao, Parallel implementations of the fast sweeping method, J. Comput. Math. 25 (2007) 421–429.