SIAM J. SCI. COMPUT.
Vol. 37, No. 1, pp. A156–A180

A PARALLEL TWO-SCALE METHOD FOR EIKONAL EQUATIONS*

ADAM CHACON† AND ALEXANDER VLADIMIRSKY†

Abstract. Numerous applications of Eikonal equations prompted the development of many efficient numerical algorithms. The Heap-Cell Method (HCM) is a recent serial two-scale technique that has been shown to have advantages over other serial state-of-the-art solvers for a wide range of problems [A. Chacon and A. Vladimirsky, *SIAM J. Sci. Comput.*, 34 (2012), pp. A547–A578]. This paper presents a parallelization of HCM for a shared memory architecture. The numerical experiments in R^3 show that the parallel HCM exhibits good algorithmic behavior and scales well, resulting in a very fast and practical solver.

Key words. Eikonal equation, parallel algorithms, domain decomposition, causal numerical methods, optimal control

AMS subject classifications. 49L20, 49L25, 65N06, 65N22, 35F30, 65Y05, 68W10

DOI. 10.1137/12088197X

1. Introduction.

The Eikonal equation is a nonlinear first-order static PDE used in a range of applications, including robotic navigation, wavefront propagation, seismic imaging, optimal control, and shape-from-shading calculations. The computational efficiency on a fixed grid is an important practical consideration in many of these applications. Several competing "fast" serial algorithms have been introduced in the last two decades to solve the grid-discretized Eikonal equation. The two most widely used are the Fast Marching Method (FMM) and the Fast Sweeping Method (FSM). The Heap-Cell Method (HCM), introduced in the authors' previous work [8], is a two-scale technique based on combining the ideas of FMM and FSM. The current paper focuses on the parallelization of HCM for a shared memory architecture. We will start by briefly describing the relevant discretization of the Eikonal PDE (section 1.1) and the prior algorithms for solving it (sections 2 and 3). HCM is reviewed in section 4, and the new parallel HCM (pHCM) is described in detail in section 5. The numerical experiments in section 6 demonstrate that pHCM is efficient and achieves good parallel scalability for a wide range of grid resolutions and domain decomposition parameters. Additional experimental results are included in an extended version of this paper [10]. We conclude in section 7 by discussing the limitations of our approach and the directions of future work.

1.1. Eikonal PDE and its upwind discretization.

An important subclass of Hamilton–Jacobi equations is formed by static Eikonal PDEs:

$$\begin{aligned} |\nabla u(\boldsymbol{x})|F(\boldsymbol{x}) &= 1 \text{ on } \Omega \subset R^d, \\ u(\boldsymbol{x}) &= q(\boldsymbol{x}) \text{ on } \partial\Omega. \end{aligned} \tag{1}$$

Classical (smooth) solutions of equation (1) generally do not exist, and weak solutions are not unique [2]. However, existence and uniqueness can be shown for the *viscosity*

*Submitted to the journal's Methods and Algorithms for Scientific Computing section June 21, 2012; accepted for publication (in revised form) September 23, 2014; published electronically January 20, 2015.

http://www.siam.org/journals/sisc/37-1/88197.html

†Center for Applied Mathematics and Department of Mathematics, Cornell University, Ithaca, NY 14853 (adamdante@gmail.com, vlad@math.cornell.edu). The authors' research was supported in part by the National Science Foundation grant DMS-1016150. The first author's research was also supported in part by an Alfred P. Sloan Foundation Graduate Fellowship.

solution [12]. Moreover, the viscosity solution has an important natural interpretation as the value function of an isotropic time-optimal control problem: F can be viewed as a speed of motion, q as an exit-time penalty on the boundary, and $u(\boldsymbol{x})$ as the minimum time-to-exit $\overline{\Omega}$ starting from $\boldsymbol{x} \in \Omega$. The gradient lines of u can be interpreted both as the characteristics of the Eikonal PDE and as the optimal trajectories for the corresponding optimal control problem.

In this paper we will also consider slightly more general problems, where exiting is allowed only through a closed nonempty "exit set" $Q \subset \partial\Omega$, with a prohibitively large exit-time penalty (e.g., $q = +\infty$) on $\partial\Omega \backslash Q$. This corresponds to a time-optimal control problem "state-constrained" to motion inside $\overline{\Omega}\backslash Q$, with u interpreted as a *constrained viscosity solution* on $\overline{\Omega}$. The boundary conditions on Q are satisfied as usual (with $u = q$), while $\partial\Omega\backslash Q$ is treated as a noninflow boundary, where the boundary conditions are "satisfied in a viscosity sense"; see [2].

Several discretizations have been developed for equation (1), but here we focus on simple first-order upwind finite differences (similar to those presented in [28]) on a three-dimensional (3D) uniform Cartesian grid with stepsize h. A typical gridpoint in R^3 will be denoted as $\boldsymbol{x}_{ijk} = (x_i, y_j, z_k) = (ih, jh, kh)$, where $0 \le i, j, k \le n$ with $nh = 1$ and $M = (n+1)^3$ is the total number of gridpoints in $\overline{\Omega} = [0,1]^3$. We will use U_{ijk} as a numerical approximation of $u(x_i, y_j, z_k)$, with U reserved to denote the entire grid solution vector. For simplicity of exposition, we will assume that the exit set Q is well approximated on this grid, and that all gridpoint values outside this computational cube are equal to $+\infty$. Using the above notation, the upwind discretization can be written as

$$\left(\max\left(D^{-x}_{ijk}U,\, -D^{+x}_{ijk}U,\, 0\right)\right)^2 + \left(\max\left(D^{-y}_{ijk}U,\, -D^{+y}_{ijk}U,\, 0\right)\right)^2 + \left(\max\left(D^{-z}_{ijk}U,\, -D^{+z}_{ijk}U,\, 0\right)\right)^2 = \frac{1}{F^2_{ijk}}, \tag{2}$$

$$\text{where} \qquad u_x(x_i, y_j, z_k) \approx D^{\pm x}_{ijk}U = \frac{U_{i\pm1,j,k} - U_{i,j,k}}{\pm h}, \qquad \text{etc.}$$

Equation (2) must hold at each gridpoint $(x_i, y_j, z_k) \in \overline{\Omega}\backslash Q$, yielding a system of coupled, nonlinear equations. Since the set $Q \subset \partial\Omega$ is usually lower-dimensional, the total number of these equations is still $O(M)$. If the values of U at the neighboring gridpoints were available, equation (2) could be solved directly for U_{ijk}. Since those neighboring U values are not a priori known, the resulting system can be solved iteratively (e.g., using Gauss–Seidel iterations), with V_{ijk} denoting temporary values (or "temporary labels"). When these temporary gridpoint values stop changing, the iterative process terminates and $V \equiv U$.

However, the upwind nature of the discretization guarantees that not all neighboring values are relevant; i.e., only those neighboring values *smaller* than U_{ijk} are actually needed to compute it from equation (2). This is also a reflection of the Eikonal equation's *causality property*, which is often exploited in the construction of fast algorithms.

2. Prior serial methods. The literature on serial methods for the Eikonal is vast; see [8] for a recent review. Here we describe only those methods directly relevant to HCM and its parallelization. For simplicity we describe these methods for Eikonal equations on Cartesian grids, but we note that some of them have been developed in much greater generality (see [5, 21, 24] and [23, 32, 33], for example).

The FSM [34, 37] solves system (2) by Gauss–Seidel iterations with an alternating ordering of the gridpoints at each iteration. These orderings, or "sweep directions," are given by the standard loop orderings for Cartesian grids. For example, in two dimensions these are

$$\begin{array}{llll} i=0,\dots,n, & i=n,\dots,0, & i=0,\dots,n, & i=n,\dots,0, \\ j=0,\dots,n, & j=0,\dots,n, & j=n,\dots,0, & j=n,\dots,0. \end{array}$$

There are 2^d loop orderings in d dimensions. The efficiency of FSM is due to the fact that each characteristic of the solution can be divided into a finite number of contiguous portions where the characteristic directions in each portion are within only one quadrant. Every 2^d sweeps all gridpoints along at least one of these portions will obtain their final values U. The number of sweeps to convergence is related to ρ, the maximum number of times a characteristic changes direction from quadrant to quadrant. Also, this number of sweeps is typically largely independent of the grid size as $h \to 0$ [37], resulting in $O(\rho M)$ algorithmic complexity. Unfortunately, even for fixed functions F and q, this ρ is a priori unknown; moreover, ρ is dependent on the grid orientation.

An additional speedup technique, the Locking Sweeping Method (LSM) [1], uses boolean "active flags" at each gridpoint to determine whether or not it will be updated in the next sweep. (A value at a given gridpoint definitely will not change if none of its neighboring values have changed in the previous sweep.) Initially only the gridpoints adjacent to the exit set are marked as "active." This technique reduces the total number of gridpoint computations per sweep but does not reduce the number of sweeps to convergence.

The FMM [30, 31] is a noniterative method that uses the Eikonal equation's causality property to dynamically determine an order in which to process the gridpoints. A set L of "considered" gridpoints is maintained. At each step of the algorithm, the considered gridpoint with the smallest temporary value is permanently "accepted," and its not-yet-accepted neighbors are updated. The set L at each step can be regarded as an approximation to the current level set of u. When L is structured as a min-heap, updating it incurs an $O(\log(m))$ cost, where $m = |L|$. The performance of FMM thus depends on the $(d-1)$-dimensional volume of the level sets of u; the upper bound for complexity is $O(M \log M)$. While the performance of FMM is more robust with respect to changes in the speed function F, domain geometry, and grid orientation, this method is not much faster for simpler problems, e.g., when the characteristics are straight lines (the regime where FSM is most efficient).

The HCM is a serial two-scale method that was introduced to combine the strengths of FMM and FSM on different scales. On the coarse scale, a Fast Marching-like method is used to order the subdomains, and on the fine scale, sweeping (specifically LSM) is used on each subdomain separately; see section 4 for a thorough description. The informal motivation for this is that sufficiently zooming in on a portion of the domain reveals that characteristics are approximately straight lines on that length scale, so sweeping restricted to that portion will converge quickly. Even though the original purpose of the domain decomposition in HCM was to exploit the structure of the PDE serially, in this paper we show that the parallelization of the HCM is a natural byproduct and proves to be a very effective strategy; see section 6.

It is well known that many of the methods for Eikonal equations are directly related to prior algorithms for finding shortest paths on graphs.[1] Here we simply

[1]Such graph algorithms are usually called *label-setting* and *label-correcting*. To reflect this, we are using the terms gridpoint "value" and "label" interchangeably.

acknowledge this connection, but it is explored in detail in [8]. In this framework, FMM and the prior noniterative method in [35] are analogous to Dijkstra's method [14]. The Small Labels First/Large Labels Last (SLF-LLL) [3] is another fast method originally designed for graph problems but also extended to Eikonal PDEs [27]; this iterative algorithm was designed to mimic the acceptance order of nodes in Dijkstra's method while avoiding the costs associated with min-heap data structures. Even though the worst-case complexity of SLF-LLL is not as good as that of Dijkstra's, in practice its performance can be better on many types of graphs.

3. Prior parallel methods. Several interesting approaches have been used to design parallel methods for Eikonal and related PDEs. Although a careful performance/scalability comparison of *all* such methods would clearly be valuable for practitioners, it remains beyond the scope of the current paper. Here we give a brief overview of prior approaches primarily to put pHCM in context. In section 6 we also use one of them as a benchmark for comparison with our own approach.

Two different parallelizations of FSM were introduced in [38]. The first parallelization performs a domain decomposition and uses separate processors to run the serial FSM on each subdomain. Subdomains are preassigned to processors, and communication takes place along the shared boundaries. The second approach presented in [38] does not use domain decomposition and performs all 2^d sweeps simultaneously on separate copies of the domain; these copies are then synchronized after each iteration by assigning the minimum value for each gridpoint.

The method of [13] is a more recent parallel sweeping technique (which we call the Detrixhe Fast Sweeping Method or DFSM) that utilizes the fact that, for the upwind scheme in three dimensions (equation (2)), gridpoints along certain planar slices through the computational domain do not directly depend on each other. The planes are given by

$$\alpha_i i + \alpha_j j + \alpha_k k = C$$

for $\alpha_i, \alpha_j, \alpha_k \in \{-1, 1\}$ and $C \in \mathbb{Z}$. The choice of α's determines one of the 2^3 sweeping directions; once the α's are fixed, the sweeping is performed by incrementing C (which corresponds to translating the plane in the sweep direction). This is a *Cuthill–Mckee* [29] ordering of the gridpoints. Inside any such plane the gridpoint updates are "embarrassingly parallel," but the resulting method is *synchronous* since a barrier is required after processing each plane. Unlike the methods in [38], this algorithm requires exactly the same number of sweeps as the serial FSM and also exhibits much better scalability. This appears to be the current state-of-the-art in parallel sweeping methods for a shared-memory architecture; thus, we have chosen to benchmark our results against it in section 6. We note that a similar parallelization approach can also be used with the regular (lexicographic) gridpoint ordering but with an appropriately extended stencil/discretization. This idea was previously used in [36] for distance computations on parametric surfaces, and more recently in [16] to parallelize the sweeping for more general (anisotropic) problems.

As for marching approaches, the canonical FMM is inherently serial (as is Dijkstra's method) and relies on a causal ordering of computations. Several parallelizations of FMM have been developed employing fixed (problem-independent) domain decompositions and running the serial FMM locally by each processor on a preassigned subdomain(s) (e.g., [18] and [6]). In the absence of a strictly causal relationship between subdomains, this inevitably leads to erroneous gridpoint values, which can be later fixed by rerunning the FMM whenever the boundary data changes. A very

recent massively parallel implementation for distributed memory architecture in [15] uses coarse grid computations to find a good subdomain preassignment, attempting to exploit nonstrict causality to improve the efficiency; the approach is then reused recursively to create a multilevel framework.

The main difficulty with making the most effective use of a domain decomposition for the Eikonal equation is that the direction of information flow at subdomain boundaries is not known a priori. If the domain is decomposed so that there is exactly one subdomain per processor, the loads may not be balanced. Additionally, a problem shared by all algorithms using a fixed domain decomposition is the existence of mutually dependent subdomains with a high degree of dependency; see Figure 1. Nevertheless, domain decomposition is often preferred as a parallelization approach to improve the cache locality and to avoid the use of fine-grain mutual exclusion.

A recent approach aims to minimize the interdomain communications by creating problem-dependent causal domain decompositions. The "Patchy FMM" developed in [7] for feedback control systems uses coarse grid computations to build (almost) causal subdomains, which are then processed independently. The disadvantages of this approach include complicated subdomain geometries, additional errors along subdomain boundaries, and frequent load balancing issues (since the causal subdomains are often very different in size).

In principle, it is also possible to parallelize some prior Eikonal solvers (e.g., the Dial-like algorithm [35] and the Group Marching Method [22]) without resorting to domain decompositions. But we are not aware of any existing parallel implementations, and the scalability is likely to be very limited due to the focus on gridpoint-level parallel computations. For shortest path problems on graphs, examples of asynchronously parallelizable algorithms include the threshold method and the SLF-LLL method [4]. The idea in parallelizing the latter is to let each processor run a serial SLF-LLL method on its own local queue, but with a heuristic used to determine which queue is to receive each graph-node tagged for updating. A mutex is used for every node to prevent multiple processors from attempting to modify it simultaneously. This parallel design inspired our own (cell-level) approach in the pHCM.

Several parallel algorithms were also developed for other computer architectures. One method proposed in [36], intended for SIMD and GPU architectures, computes shortest geodesic paths on parametric surfaces. In this Parallel Marching Method (PMM) the subdomains are processed serially with a dynamic ranking procedure similar to that of FMM. Each time a subdomain is processed, the values of all gridpoints within it are updated using parallel "raster scans," which are similar to the parallel sweeps in [38] and [13].

Another method intended for massively parallel (SIMD and GPU) architectures is the Fast Iterative Method (FIM) developed in [20]. In FIM, an unsorted list L of active gridpoints is maintained, and at each iteration all gridpoints on L are updated in parallel using Jacobi updates. A variant, the Block FIM, maintains blocks of gridpoints on L, and all blocks on L are updated in parallel. New blocks are added based on whether any of their gridpoints received updates. Blocks are used to take advantage of the SIMD parallelism.

4. Heap-cell method. To simplify the exposition, we give the following description of HCM in two dimensions. The translation to higher dimensions is straightforward. We first introduce some relevant notation:

- $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M\}$, the grid (same as the grid used in FMM or FSM). This single-subscript notation is meant to emphasize a gridpoint ordering, rather than the

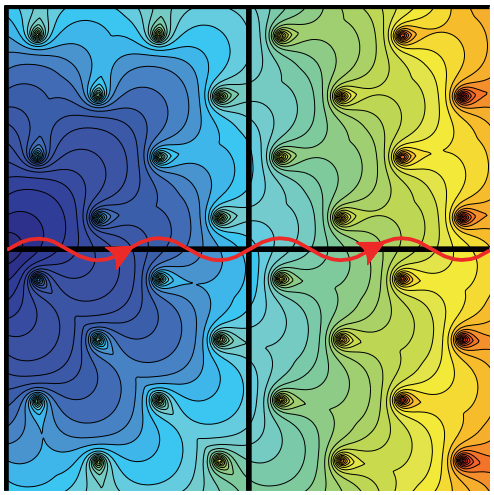

FIG. 1. *Level sets for an Eikonal problem in two dimensions with cell boundaries and a characteristic curve. Since the characteristic repeatedly crosses the subdomain boundary, any method that solves this problem using the given domain decomposition will require a large number of iterations.*

geometric position indicated by the subscripts in formula (2). The corresponding gridpoint values are denoted as $V_i = V(\boldsymbol{x}_i)$.

- $Q' = X \cap Q$, the set of "exit gridpoints," whose values are prescribed.
- $Z = \{c_1, \ldots, c_J\}$, the set of cells (or "nonoverlapping box-shaped subdomains").
- $Q^c = \{c \in Z \mid c \cap Q' \neq \emptyset\}$.
- $N(\boldsymbol{x}_j)$, the grid neighbors of $\boldsymbol{x}_j$, i.e., the gridpoints that exist to the north, south, east, and west of $\boldsymbol{x}_j$.
- $N^c(c_i)$, the set of neighboring cells of c_i, i.e., the cells that exist to the north, south, east, and west of c_i.
- $N(c_i)$, the grid neighbors of c_i, i.e.,

$$N(c_i) = \left\{ \boldsymbol{x}_j \in X \mid \boldsymbol{x}_j \notin c_i \text{ and } N(\boldsymbol{x}_j) \bigcap c_i \neq \emptyset \right\}.$$

- V^c, the cell label.
- h_x^c and h_y^c, the two cell dimensions (assume $h_x^c = h_y^c = h^c$).
- r, the number of gridpoints per cell side.

To ensure that each gridpoint belongs to one and only one cell, the cell boundaries are not aligned with gridlines, and $\Omega^c = \bigcup_{j=1,\ldots,J} \overline{c_j}$ must be a superset of $\overline{\Omega}$; see Figure 2.

The original HCM, presented in [8], is a serial two-scale method. When the two-dimensional analogue of system (2) is solved on a cell c (using any method), if the values of $N(c)$ are already correct, then all $\boldsymbol{x}_i \in c$ will receive their final values U_i. Each cell is therefore dependent on a subset of $N^c(c)$, and the hyperbolic nature of the problem suggests that there is a *preferred order* of processing the cells. The motivation for employing a serial domain decomposition using sweeping on the grid is that, if the cell sizes are small enough, the characteristics within each cell will be approximately straight lines, and sweeping will converge in very few iterations.

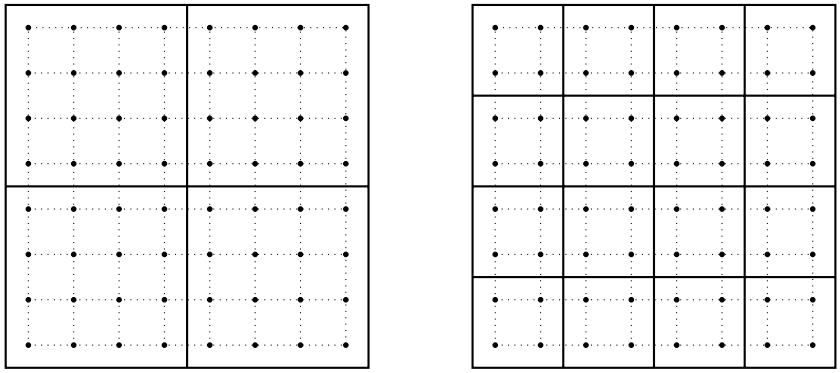

FIG. 2. *Two examples with different domain decompositions. Both* (a) *and* (b) *are based on the same grid (dotted), with* $M = 8^2$ *and* $h = 1/7$. *Figure* (a) *uses the cell size* $h^c = 4/7$, *the total number of cells* $J = 2^2$, *and* $r = 4$ *gridpoints per cell side. Figure* (b) *uses* $h^c = 2/7$, $J = 4^2$, *and* $r = 2$.

The HCM maintains a list L of cells-to-be-processed, initially populated with Q^c. The entire grid is initialized only once, in the same way as it is for LSM.[2] At each iteration of the main algorithm, a cell c is chosen from L and equation (2) is solved by LSM on $X \cap c$ with the boundary conditions specified by the current values on $N(c_i)$. The order of processing of the cells is determined dynamically based on heuristically assigned and updated cell values. The name "Heap-Cell" comes from organizing L as a min-heap data structure. HCM is designed to mimic FMM on the cell level (though previously processed cells may re-enter L; see Algorithm 1 for the pseudocode). Since in typical cell decompositions $J \ll M$, the cost of maintaining the heap L is small compared to the cost of grid computations. The experimental evidence in [8] shows that HCM is very efficient for a wide range of M and J values.

Algorithm 1. Heap-Cell Method main loop.

```
Initialize cell values and grid values
Add all c ∈ Q^c cells to L
while L nonempty do
    Remove the cell c with the smallest cell value from L
    V^c(c) ← +∞
    Perform modified LSM on c until convergence and populate
    the list DN of currently downwind neighboring cells        //see Algorithm 2

    for each neighbor c_k ∈ DN do
        Update V^c(c_k), the cell value of c_k
        Add c_k onto L if not already there
        Update the preferred sweeping directions of c_k
    end for
end while
```

We say that a cell B is *currently downwind* from a cell A if **(1)** A was the last processed cell, **(2)** there exist neighboring border gridpoints $\boldsymbol{x}_i \in A$ and $\boldsymbol{x}_j \in B$ such

[2]That is, all $\boldsymbol{x}_i \notin Q'$ have $V_i = \infty$; the active flags of gridpoints in $\{\boldsymbol{x} \in N(\boldsymbol{x}_i) | \boldsymbol{x}_i \in Q', \boldsymbol{x} \notin Q'\}$ are set to "active"; and the active flags of all other gridpoints are set to "inactive."

that the value of V_i has changed the last time A was processed, and **(3)** $V_i < V_j$. See Figure 3. We note that, since this relationship is based on the temporary labels V, it is entirely possible that the same A might be also *downwind* from B at a different stage of the algorithm.

Unfortunately, a good dependency ordering of cells may not exist even if we could base it on permanent gridpoint labels U or even on the continuous viscosity solution $u(\boldsymbol{x})$. We will say that B *depends* on A if there exists some optimal trajectory crossing the cell boundary from B to A on its way to Q. This allows us to construct a dependency graph on the set of cells. We will say that a cell decomposition is *strictly causal* if this dependency graph is acyclic. A strictly causal decomposition ensures that there exists an ordering of cells such that each of them needs to be processed only once.

Figure 1 shows that, for many generic problems and large h^c, neighboring cells A and B are likely to be interdependent, resulting in multiple alternating reprocessings of A and B. As h^c decreases, the decomposition becomes *weakly causal*—most cell boundaries become either purely inflow or purely outflow. Additionally, if the ordering is such that most dependents are processed after the cells on which they depend, the average number of times each cell is processed becomes close to one. As confirmed by the numerical evidence in [8], weakly causal domain decompositions are very useful in decreasing the computational costs of serial numerical methods.

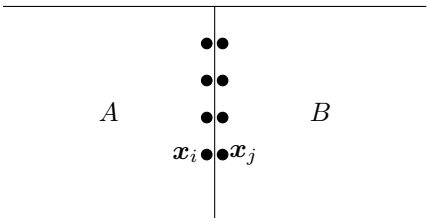

FIG. 3. *Suppose that as a result of processing the cell A an eastern border value V_i becomes updated. If $V_i < V_j$ and $\boldsymbol{x}_j \notin Q'$, then cell B is* downwind *of cell A. In this case cell B will be added onto L unless already there, its value will be updated, and its preferred sweeping directions will be updated.*

Processing cells by using Fast Sweeping Methods. Sweeping using LSM [1] is performed on the cell c by using the neighboring grid values as boundary data. Precisely, the domain for processing c is $\tilde{c} = c \cup N(c)$, with the boundary conditions defined as $\tilde{q}(\boldsymbol{x}_i) = q(\boldsymbol{x}_i)$ on $c \cap Q'$ and $\tilde{q}(\boldsymbol{x}_i) = V_i$ on $N(c)$. The sweeping processes gridpoints one at a time, with the gridpoint update procedure detailed in Algorithm 2.

As in the usual LSM, we loop through different sweeping directions, using a new one in each iteration. However, by the time a cell B needs to be processed, the boundary information from its previously processed neighboring cells can be used to determine the preferred directions to start sweeping, with the likely effect of reducing the total number of sweeps needed to converge in B. This is accomplished by having each cell maintain a list of boolean *preferred-sweep-direction* flags, and by LSM beginning sweeping only from the directions marked `TRUE`. If the convergence is not achieved after performing sweeps in these preferred directions, we revert back to a standard loop (i.e., in two dimensions the default standard loop would be SW, SE, NE, NW). After a cell is processed, all sweep-direction flags are set to `FALSE`. A sweep-direction flag of a cell B is updated to `TRUE` only at the time a neighboring cell

Algorithm 2. Modified LSM update at a gridpoint $\boldsymbol{x}_i$.

```
if x_i is inactive then
    Do nothing
else
    Set x_i inactive
    Compute a possible new value Ṽ for x_i by solving equation (2)
    if Ṽ < V(x_i) then
        V(x_i) ← Ṽ
        for each x_j ∈ N(x_i)\Q' do
            if V(x_j) > V(x_i) then
                Set x_j active
                if x_j is in a different cell from x_i then
                    Tag that cell as part of the list DN of currently downwind
                    cells
                end if
            end if
        end for
    end if
end if
```

A tags B as *downwind*. The directions that are updated depend on the location of A relative to B. For example, if B is downwind from A as in Figure 3, then both A-relevant sweep-direction flags in B (i.e., both NW and SW) will be set to `TRUE`.

In principle, the actual values of the border gridpoints could also be used to further restrict the list of preferred sweep directions (with the goal of avoiding unnecessary sweeping). The *Fast Heap-Cell Method* introduced in [8] uses one such acceleration technique by checking the "monotonicity" of boundary data. Since this technique is more costly in R^3, we are not using it in the current implementation of HCM.

Assigning cell values. Cell values are computed heuristically and intended to capture the direction of information flow. If a cell B depends on a cell A, then ideally $V^c(A) < V^c(B)$ should hold to ensure that A is processed earlier. We emphasize that the choice of a particular cell value heuristic *does not* affect the final output of the HCM (see [8] for a proof of convergence), but may affect the method's overall efficiency. An ideal heuristic would reflect the inherent causal structure. For example, if the cell decomposition is strictly causal, using a good cell value heuristic would result in exactly J heap removals. For weakly causal cell decompositions (attained for all problems once h^c becomes sufficiently small), a good cell value heuristic ensures that the average number of heap removals per cell becomes closer to 1; see [8] and sections 6.1 and 6.2 of the current paper for experimental evidence.

In this paper, our treatment of the cell value differs from that in [8] in two ways: **(1)** whenever a cell B is removed from L, we reset $V^c(B)$ to $+\infty$, and **(2)** we assign $V^c(B)$ as the smallest of the newly updated gridpoint values in $N(B)$; see formula (3). The logic is that cells should be ranked by the currently most upwind inflow. We reset $V^c(B)$ so that if B is to be processed again, the later time-of-processing will be determined only by new inflow information. This heuristic appears to be very efficient for a variety of examples and easily generalizes to higher dimensions. Most importantly, it seems to be effective at handling discontinuities in the speed function that do not align with the cell boundaries, which was a weakness of the cell value

in [8]. Our current cell value update formula is

$$\widetilde{V}^c(B) \leftarrow \min_{j \in A_{new}} V(\boldsymbol{x}_j), \qquad V^c(B) \leftarrow \min(V^c(B), \widetilde{V}^c(B)), \tag{3}$$

where A_{new} is the set of newly updated "inflow for B" gridpoints of A along the relevant cell border; i.e., $A_{new} = \{\boldsymbol{x}_i \in N(B) \cap A \mid \text{recently updated } U_i < U_j \text{ for some } \boldsymbol{x}_j \in B \cap N(\boldsymbol{x}_i)\}$. An efficient implementation of this heuristic relies on updating the current minimum border value of B at line 12 of Algorithm 2. The equivalent of formula (3) was also previously used to determine the processing order of large "charts" in [36].

Finally, we use a natural initialization of cell values before the main loop of the algorithm:

$$V^c(c) \leftarrow \begin{cases} \min\{V(\boldsymbol{x}_j) \mid \boldsymbol{x}_j \in c \cap Q'\} & \text{if } c \in Q^c, \\ +\infty & \text{otherwise.} \end{cases}$$

5. Parallelization. There are several different approaches one can take to parallelize HCM. It is possible, for instance, to parallelize the sweeping scheme within an individual cell. Our choice for pHCM was to have multiple subdomains processed simultaneously. Each processor p essentially performs the serial HCM on its own local cell-heap L_p, but with one important difference: when a cell c is tagged for reprocessing, we attempt to add it to the heap L_j with the lowest current number of cells. Except for some modifications explicitly described below, most of the subroutines of the serial HCM can be directly reused in pHCM as well. In Algorithm 3, all data is shared unless stated otherwise.

The described algorithm gives rise to occasional (benign) data race conditions. But before explaining why they have no impact on correctness/convergence, we highlight several main design decisions:

- To ensure efficiency/scalability, there is no synchronization mechanism at the gridpoint level.
- Unlike many other parallel Eikonal solvers, pHCM is *asynchronous*; i.e., no barriers are used to ensure that cells are processed in some specific order.
- There are two separate "individual cell operations" that must be serialized: **(1)** the movement of a cell onto/off of/within a heap, and **(2)** the update of gridpoint values within that cell. However, both of these can be safely performed simultaneously. Thus, each cell maintains both a "compute" lock and a "position" lock to allow for the overlapping of these operations.
- Adding a cell onto the heap with fewest elements ensures good load balancing. But if that heap is currently locked, waiting for the lock to be released might have the opposite effect on the method's performance. Since we can assign the cell to another heap without drastically altering the balance, we attempt to obtain the lock using the `omp_test_lock` subroutine, and move on to the next heap if that attempt was unsuccessful; see Algorithm 5. Profiling shows that this approach always results in better performance than using the `omp_set_lock`.
- The *activeCellCount* is decremented on line 29 of Algorithm 3 (rather than around line 10) to prevent other threads from quitting prematurely.
- The cell update (lines 15–17 of Algorithm 3) is exactly the same sweeping procedure as in HCM. Just as in HCM, any other method that solves system

Algorithm 3. Parallel Heap-Cell Method pseudocode.

```
Cell Initialization: same as in HCM (divide cells Q^c evenly among all heaps L_p)
Fine Grid Initialization: same as in HCM
P ← number of threads
activeCellCount ← |Q^c|
PARALLEL SECTION
while activeCellCount > 0 do
    while L_p is nonempty do
        Lock heap L_p
            Position-lock cell c at the top of L_p
                Remove c from L_p
                V^c(c) ← +∞
            Position-unlock c
        Unlock L_p
        Compute-Lock c
            Perform modified LSM on c and populate the (local) list DN
            of currently downwind neighboring cells        //see Algorithm 2
            Set all preferred sweeping directions of c to FALSE
        Compute-Unlock c
        for each c_k ∈ DN do
            Compute a possible new (local) cell value Ṽ for c_k
            if Ṽ < V(c_k) then
                Set Cell Value (c_k, Ṽ)        //see Algorithm 4
            end if
            if c_k is not on a heap then
                Add Cell (c_k)        //see Algorithm 5
            end if
            Update sweeping directions of c_k based on location of c
        end for
        activeCellCount --        (atomic)
    end while
end while
```

(2) within a cell c may be substituted in place of LSM. However, if the grid value updates inside c also involve updating any grid-level data in $N^c(c)$, the potential race conditions must be handled carefully. Below we explain how this issue is handled in LSM for the active flag updates across cell boundaries.

5.1. Efficiency and data race conditions. There is always a delicate tradeoff between performance-boosting heuristics in the serial realm and the synchronization penalty they would incur in the parallel implementation. The serial HCM has several features (the use of LSM within cells, the use of preferred sweeping directions, the accuracy of cell values at predicting information flow) that could cause contention when parallelized. In this section we describe how we chose to handle those features in designing pHCM. Since there is no synchronization at the gridpoint level, we have actually allowed several data races to be present in the algorithm. We first check the convergence of the algorithm in the presence of these data races.

For all of the following arguments we assume a *sequentially consistent* memory model, meaning that the instructions in Algorithm 3 are executed in the order in

Algorithm 4. Set cell value $(c_k, \widetilde{V})$.

```
success ← FALSE
while success == FALSE do
    if c_k is not on a heap then
        Position-lock c_k
        if c_k is still not on a heap then
            V(c_k) ← min(Ṽ, V(c_k))
            success ← TRUE
        end if
        Position-unlock c_k
    else
        j ← index of the heap of c_k
        Lock L_j
        Position-lock c_k
        if c_k is still on L_j then
            V(c_k) ← min(Ṽ, V(c_k))
            Heap-sort L_j
            success ← TRUE
        end if
        Position-unlock c_k
        Unlock L_j
    end if
end while
```

Algorithm 5. Add cell (c_k).

```
j ← index of heap with fewest elements (no locking; counts may be outdated
    during search);
testCount ← 0
while Lock L_{(j+testCount)%P} can not be immediately obtained do
    testCount++
end while

Position-Lock c_k
if c_k is still not on a heap then
    Add c_k onto L_{(j+testCount)%P}
    activeCellCount ++          (atomic)
end if
Position-Unlock c_k
Unlock L_{(j+testCount)%P}
```

which they appear. On modern platforms it is possible that compilers or hardware will reorder the program's instructions. While these optimizations are innocuous in serial codes, in a multithreaded environment this can lead to unexpected results.[3]

First consider a more basic version of pHCM that uses FSM within cells instead of LSM. There is still a possibility of data races along the boundary of each cell: updating

[3]Indeed, in our implementation it was actually necessary to explicitly prevent such reordering of certain lines of code (using the OpenMP "flush" pragma).

a border gridpoint by equation (2) requires reading information in a neighboring cell. However, it is easy to see that the monotonicity of gridpoint value updates makes such data races harmless. Suppose two cells A and B are being simultaneously swept by processors p_A and p_B, respectively (see Figure 3). Suppose also that B undergoes its final sweep. First, the most obvious outcome is that

a. p_A updates $\boldsymbol{x}_i$ (and writes V_i);
b. p_A checks V_j and finds $V_i < V_j$, $\Rightarrow$ tags B to be added onto a heap.

So, B will have a chance to use the new boundary information V_i the next time it is processed. Now, suppose neighbors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are updated simultaneously (i.e., Algorithm 2 is executed in parallel at $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ by the different processors). Suppose also that the final sweep in A leaves $V_i < V_j$. Then either

a. p_A writes $V(\boldsymbol{x}_i)$;
b. p_B writes $V(\boldsymbol{x}_j)$;
c. p_A checks V_j and finds $V_i < V_j$, $\Rightarrow$ tags B to be added onto a heap;
d. p_B checks V_i and finds $V_i < V_j$, $\Rightarrow$ does nothing

or

a. p_B writes $V(\boldsymbol{x}_j)$;
b. p_B checks V_i and finds $V_j < V_i$, $\Rightarrow$ tags A to be added onto a heap;
c. p_A writes $V(\boldsymbol{x}_i)$;
d. p_A checks V_j and finds $V_i < V_j$, $\Rightarrow$ tags B to be added onto a heap.

In the latter case the cell A is unnecessarily added onto a heap, but this redundancy does not impact the convergence. Therefore, a cell with new inflow boundary information is always guaranteed to be reprocessed at some later point.

But our reliance on the Locking Sweeping technique introduces an additional issue: it is also necessary to ensure that all relevant boundary gridpoints in that yet-to-be-reprocessed cell will be marked as "active"—since otherwise the first cell-sweep will not touch them. Recall that p_A will only set the gridpoint values within A, but because of LSM, it might also change the active flags of gridpoints in $N(A) \cap B$. What if $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are updated simultaneously, p_A makes $\boldsymbol{x}_j$ active, but p_B immediately resets it as inactive and V_j is never recomputed based on the new value of V_i? The order of operations in Algorithm 2 makes this scenario impossible, since setting a gridpoint inactive is immediately *followed* by the recomputation of that gridpoint's value.

Finally, there is one additional design choice we have made that causes a race condition at the cell level when setting the cell's preferred-sweep-direction flags. After processing a cell A, we typically need to update the preferred sweeping directions of its neighboring cells. If one of these neighboring cells B is simultaneously processed using LSM, the preferred directions data might be overwritten. We could avoid this scenario by obtaining B's computation lock before updating its preferred directions. Our implementation does not use this idea because the preferred directions only reduce the number of sweeps without affecting the convergence, and because the additional contention would dominate the savings for most M/J ratios. Since all other access to cell-level data is lock protected, pHCM converges.

6. Numerical experiments. In this section we present and compare the performance of FMM, FSM, LSM, HCM, DFSM (a parallel sweeping method), and pHCM on three qualitatively different examples. Our primary goal is to test the "strong scalability" of pHCM with various cell decompositions. Sections 6.1 and 6.2 provide a more detailed performance analysis of the serial and parallel methods, respectively.

Our source code and scripts for all methods and examples in this paper are publicly available from http://www.math.cornell.edu/∼vlad/papers/pHCM/.

Benchmark problems. We consider three Eikonal examples with an exit set $\{(0.5, 0.5, 0.5)\}$ on a unit cube domain $\overline{\Omega} = [0,1] \times [0,1] \times [0,1]$. In all three cases, the boundary conditions are $q = 0$ in the center and $q = +\infty$ on the boundary of the cube. Since the center of the computational domain is not a gridpoint (i.e., M is even), we have initialized U on the set Q of the eight gridpoints closest to the center. Since J values are also even, the set Q^c contains eight cells in all of the examples.

The speed functions are as follows:

1. $F \equiv 1$.
2. $F(x, y, z) = 1 + .5 \sin(20\pi x) \sin(20\pi y) \sin(20\pi z)$.
3. $F(x, y, z) = 1 + .99 \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$.

These examples are "representative" in the sense that their respective viscosity solutions are qualitatively very different. In example 1, all characteristics are straight lines. In example 2, the characteristics are highly oscillatory and might weave through cell boundaries many times. The third example has more moderate behavior, with curved characteristics that do not oscillate rapidly.

Experimental setup and implementation details. All experiments were performed on the Texas Advanced Computing Center's "Stampede" computer, using a single Dell PowerEdge R820 node with four E5-4650 8-core 2.7 GHz processors and 1TB of DDR3 memory. We implemented all methods in C++ and compiled with the `-O2` level of optimization using the Intel Composer XE compiler v13.0. All solutions were computed and stored using *double* precision. The speed $F(x, y, z)$ was computed by a separate function call as needed, instead of precomputing and storing it for every gridpoint. HCM and pHCM use Locking Sweeping, which is experimentally always much faster than regular Fast Sweeping. In all iterative methods, the sweeps were continued as long as some gridpoints received updated values. In benchmarking all parallel methods, we have used one thread per core, up to a total of 32 cores. In addition, for some r values, the performance of pHCM may be significantly influenced by both system-level background processes and variations in the effective speed of the cores. To fully reflect this, each pHCM test was performed 30 times, and we report both the median values and the max/min "error bars."

We compare our method's performance/scaling to a parallelization of the sweeping methods. Our implementation largely follows the method described in [13], but with two exceptions:

- Detrixhe, Gibou, and Min have not tested a "locking sweeping" version of their method; our implementation of DLSM is based on a straightforward substitution of LSM updates for FSM updates.
- Our implementations of DFSM and DLSM use the default OpenMP static loop scheduling ("omp for") to divide the work among threads instead of the manual load balancing procedure described in [13].

We have also conducted many additional numerical tests including

- benchmarking with *single* precision data/computations;
- benchmarking with *"early termination criterion"* for sweeping methods;
- benchmarking on a different shared memory architecture;
- benchmarking with a different cell value heuristic;
- additional test problems (with piecewise constant F), including 3D versions of the "checkerboard" examples from [8, 9, 26].

The results (included in the expanded version of this paper [10]) are sufficiently similar qualitatively, and we omit them here for the sake of brevity.

Layout of experimental results. The HCM tests were run using $J = M/2^3$, $M/4^3$, $M/8^3$, $M/16^3$, and $M/32^3$, so there are $r = 2, 4, 8, 16, 32$ gridpoints per cell side. "HCMr" and "pHCMr" in the figure legends mean HCM and pHCM with $J = M/r^3$. (This notation emphasizes the amount of work per cell, but it is different from the format previously used in [8, 9], where the table headings directly stated J rather than r.) On each test problem the performance of pHCM depends on three problem parameters: M, r, and P, the number of processors. The performance/scaling plots for pHCM2 are omitted to improve the readability of all figures.

Figures 4, 5, 6, and 7 are organized so that columns present different examples and rows give different comparison metrics.[4] Figure 4 compares the performance of serial methods by plotting the ratio of FMM CPU time to other methods' times for $M = 128^3, 192^3, 256^3$, and 320^3. Since we are interested in strong scalability, we test pHCMr with a fixed problem size while varying P. In Figure 5, ***M** is frozen at* $\mathbf{320^3}$. The first row reports the speedup factors of the parallel methods over the serial methods; these are (HCMr time/pHCMr time), (FSM time/DFSM time), and (LSM time/DLSM time). The second row of Figure 5 provides the performance comparison of all serial and parallel methods. The growth of parallel overhead and the change in total work (as functions of P) are presented for each pHCMr in Figure 6. Plots similar to Figure 5 but computed for $M = 128^3$ are presented in subsection 6.2.

Main observations.

1. LSM significantly outperforms FMM on example 1 (Figure 4(a)), and its advantage grows with M. FMM greatly outperforms LSM on example 2 (Figure 4(b)) for all values of M. Their performance is more comparable on the third example (Figure 4(c)).
2. The performance ranking among serial HCMr methods is problem-dependent (Figures 4(a)–4(c)).
3. Figures 5(d)–5(f) demonstrate that pHCM has a large advantage over all serial methods for most r and P combinations. On these three examples with $M = 320^3$, the median performance for pHCM8 on 32 threads was between 34 and 84 times faster than for FMM, between 7.7 and 166 times faster than for LSM, and between 18.4 and 436 times faster than for FSM.
4. Generally, the pHCM speedup over HCM is greater when there is more work per cell. We see in Figures 5(a)–5(c) that the experiments with higher gridpoints-per-cell number r exhibit better parallelization, and the speedup of pHCM4 is always the worst.
5. In Figure 5 the position of each curve relative to its error bar reveals the most likely outcome. For example, the pHCM4 scaling plummets in the worst cases and plateaus in the best cases. At 32 threads, since the median is near the bottom of the error bar in all examples, the good cases are relatively rare.
6. As shown in Figure 5, for most r values pHCM scales much better than DFSM/DLSM. Since DFSM is a synchronous parallel algorithm, it comes as no surprise that using the LSM does not boost performance significantly—LSM only reduces the amount of work performed by a subset of the threads. Better scaling in DLSM would likely be achieved if it were possible to apply

[4] Table versions of the same benchmarking results are also included in the supplementary materials.

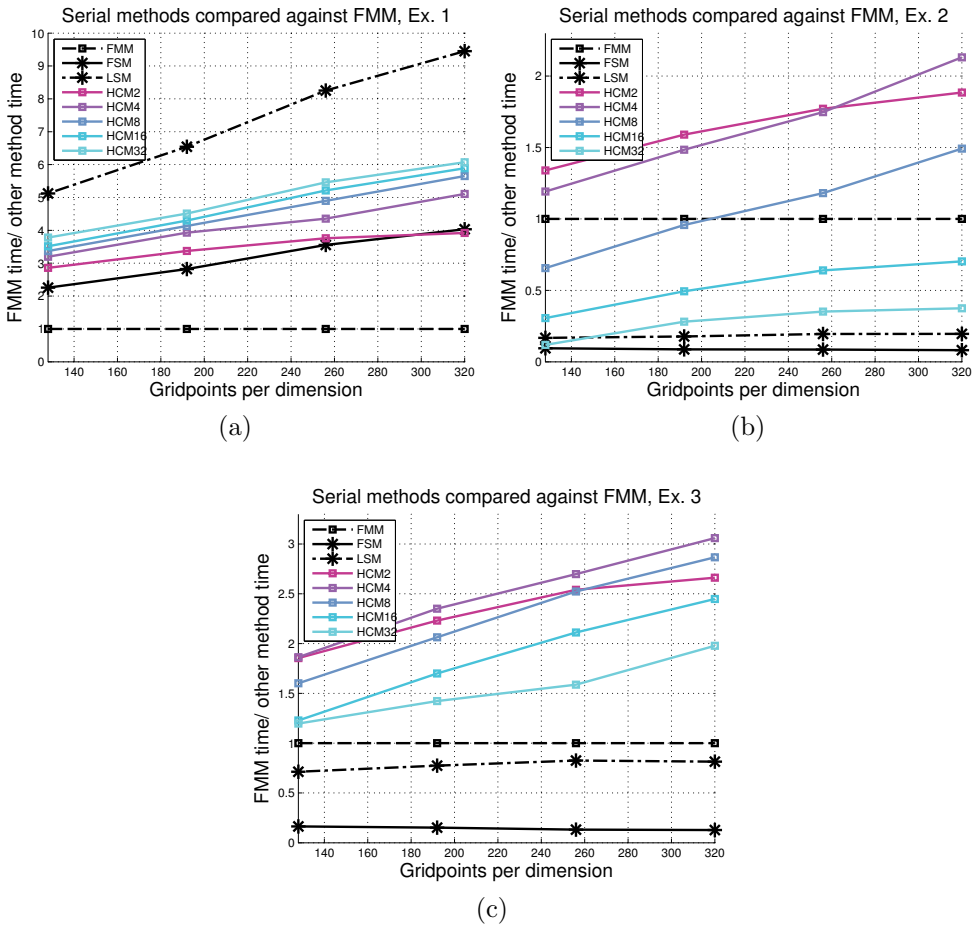

FIG. 4. *Performance of the serial methods for different M.* (a) $F \equiv 1$; (b) $F = 1 + .5\sin(20\pi x)\sin(20\pi y)\sin(20\pi z)$; *and* (c) $F = 1 + .99\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$. *The data is given as a ratio of FMM's CPU time to the times of all other methods.*

a special load balancing procedure based on the set of currently "active" gridpoints.

6.1. Further comments on performance of serial methods.

1. *Tradeoffs between FMM and LSM.* It is well known that Marching and Sweeping methods are each advantageous on their own subsets of Eikonal problems. The exact delineation remains a matter of debate. The reader can find careful comparative studies in [17, 19, 9]. In each example (Figures 4(a)–4(c)) we observe that, as M increases, the ratio of FMM time to LSM time increases due to the greater cost of each heap-sort operation. However, FMM's performance is much more robust to the qualitative differences in the solution; FMM's raw times for $M = 320^3$ ranged between 32s (Ex. 1) and 51s (Ex. 2), while the LSM times were between 3s (Ex. 1) and 363s (Ex. 2). FMM is also usually much more efficient on problems with complicated domain geometry (e.g., on domains containing multiple impenetrable obstacles).
2. *Grid memory layout and caching issues.* Large grids, particularly common in higher-dimensional problems, present an additional challenge for all (serial and parallel) methods implemented on a shared memory architecture.

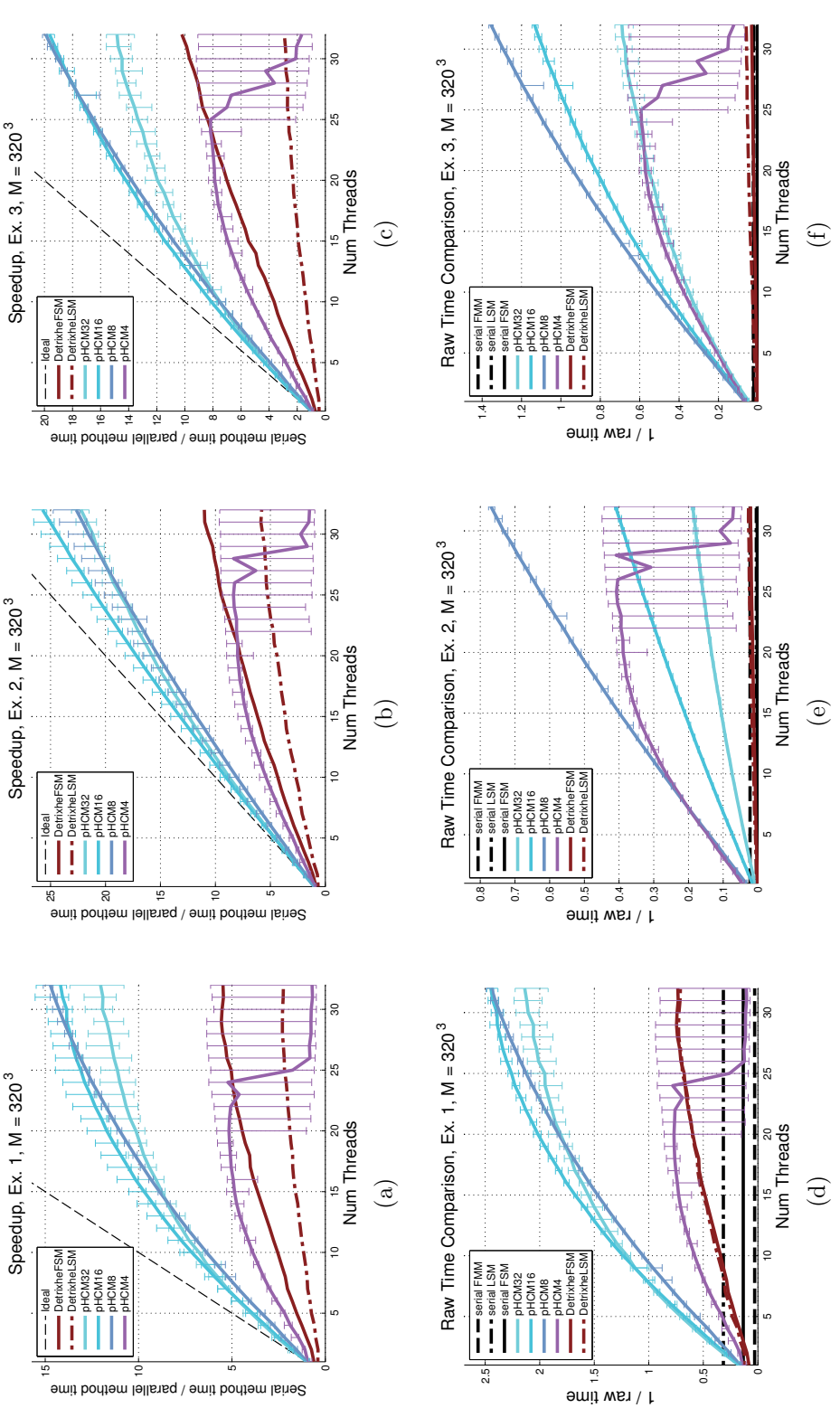

FIG. 5. *Scaling and performance for pHCM at* $M = 320^3$. *The first column has* $F \equiv 1$, *the second has* $F = 1 + .5\sin(20\pi x)\sin(20\pi y)\sin(20\pi z)$, *and the third has* $F = 1 + .99\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$.

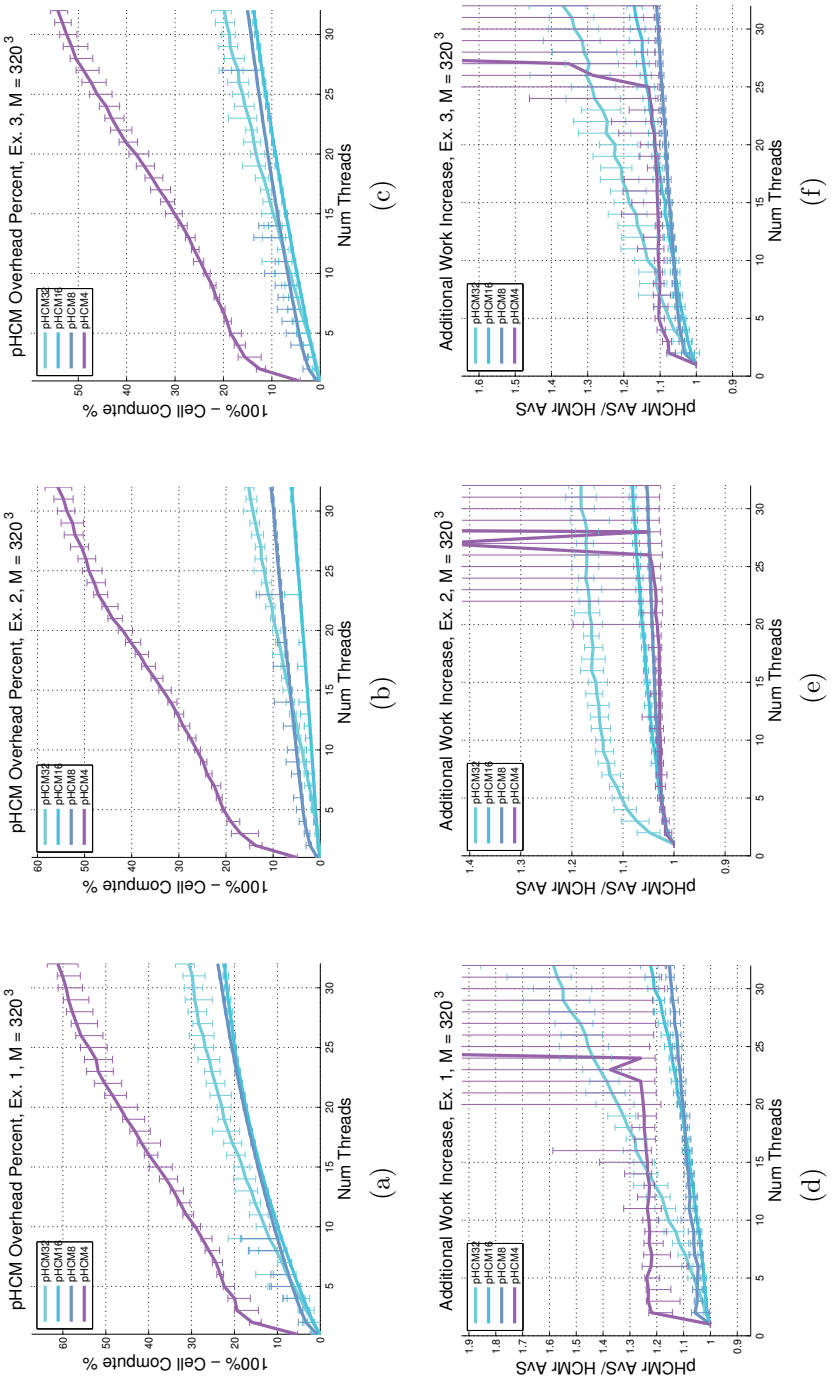

FIG. 6. *Overhead percentages and additional work in pHCMr for different P for the three examples, with $M = 320^3$. In* (a), (b), *and* (c) *the value at Num Threads* $= 1$ *of each curve approximately gives the part of the overhead accounted for by heap maintenance alone; the* parallel *overhead would be given approximately by subtracting it from each curve.*

Solving equation (2) requires accessing the U values for all gridpoints neighboring $\boldsymbol{x}_{ijk}$, but the geometric neighbors can be far apart in memory when the higher-dimensional grid is stored lexicographically. This results in frequent cache-swapping, ultimately impacting the computational cost. More detailed profiling (not included here) confirms the resulting slow-down in all serial methods, including LSM. In other applications space-filling curves have been successfully used to alleviate this problem (e.g., [25]), but we are not aware of any successful use in Eikonal solvers. We believe that allocating the fine grid separately per cell would be advantageous for a robust extension of HCM/pHCM to higher dimensions. However, our current implementation of heap-cell methods does not take advantage of this idea.

3. *FMM scaling in M.* Since the length of the heap increases with M, the number of flops per heap operation increases, too. In addition, FMM is affected by additional caching issues: the time per heap-related memory access increases, since the parent/child relationships of heap entries do not translate to memory proximity of the corresponding gridpoints. Profiling shows that the cache miss rate increases noticeably with M.
4. *HCM scaling in M.* For most cell decompositions, when $J \ll M$, the heap maintenance is negligible. As J becomes large (e.g., for $r = 2$), HCMr is affected by the same issues described for FMM above.
5. *Optimal J in HCM.* As cell sizes decrease, the causality among cells becomes stronger (see the end of section 4), and our cell value heuristic does a better job of capturing the dependency structure; the average number of times each cell is processed tends to 1. Additionally, the characteristics within each cell become approximately straight lines, so the per-cell LSM converges quickly. On the other hand, if J is large enough, the overhead due to heap maintenance becomes significant; this is quantified in Tables 1–3 ("Heap maintenance %" means the percentage of execution time spent outside of sweeping cells). Turning to individual examples, we note the following:
   (a) Ex. 1: HCM with larger cell sizes performs better. See Figure 4(a) and Table 1. This is due to a very special property of $F \equiv 1$: since there is exactly one heap removal per cell regardless of J, the maintenance of the heap is the dominant factor affecting the performance. Correspondingly, LSM performs the best. (LSM is equivalent to HCM using only one cell.)

TABLE 1
Performance analysis of HCM on Ex. 1, $M = 320^3$.

	HCM32	HCM16	HCM8	HCM4	HCM2
Avg. sweeps per cell	4.84	4.92	4.96	4.98	4.12
Heap maintenance %	1.09	1.12	1.66	5.88	33.9

   (b) Ex. 2: Due to the oscillatory nature of characteristics, HCM performs better with *smaller* cell sizes. The ranking among HCMr methods is more or less the reverse of that for Ex. 1, and the sweeping methods are the slowest. See Figure 4(b) and Table 2.
   (c) Ex. 3: Figure 4(c) and Table 3 show that the performance among the HCMr methods is qualitatively different from that of the previous examples. A weakly causal ordering already exists here for moderately sized cells.

TABLE 2
Performance analysis of HCM on Ex. 2, $M = 320^3$.

	HCM32	HCM16	HCM8	HCM4	HCM2
Avg. sweeps per cell	223	100	31.1	12.9	6.97
Heap maintenance %	0.076	0.214	0.954	4.95	30.6

TABLE 3
Performance analysis of HCM on Ex. 3, $M = 320^3$.

	HCM32	HCM16	HCM8	HCM4	HCM2
Avg. sweeps per cell	29.3	14.6	9.37	7.14	5.02
Heap maintenance %	0.292	0.424	0.914	4.55	28.5

The sweeping methods can be accelerated by stopping at the iteration where the maximum change over gridpoint values is less than or equal to a certain threshold $\kappa \geq 0$. If $\kappa > 0$, the method will terminate "early," and the output will be different from the true solution of the discretized system (2). Ideally, κ should be chosen based on the L_∞-norm discretization error, but since the latter is a priori unknown, a common practical approach is to use a small heuristically selected constant (e.g., [37]). We note that, for a fixed $\kappa > 0$, the number of needed iterations can be quite different for different h, and there is currently no proof that the early-terminated numerical values are within κ from the correct solution. See also the discussion and experimental results in [9]. The results reported here were obtained with $\kappa = 0$. Testing with $\kappa = 10^{-8}$ reduces the number of sweeps, but the scaling behavior remains largely the same; see [10].

6.2. Detailed performance analysis of parallel methods. Two key factors that affect the speedup of parallel methods are the amount of parallel overhead (contention, interthread communication, etc.) and the change in the amount of work performed from serial to parallel. In this section we focus on both the overhead analysis and the algorithmic differences between pHCM and HCM. The overhead is the sum of the parallel overhead and the "base" heap maintenance. The latter is given above in Tables 1–3.

We define the following terms used in Figure 6:

- AvS $= \Sigma_{p=0}^{P-1}$ (total number of sweeps performed by processor p)$/J$.
- Cell Compute % = percent of total time spent on sweeping cells alone.
- Overhead % = 100% − Cell Compute %, i.e., percent of total time spent beyond sweeping cells.

1. *Effects of P on overhead.* As P increases, contention and network communication increase. If more threads are used for a given cell discretization, it is more likely for a processor $\hat{p}$ to wait to obtain a lock (e.g., as in line 8 of Algorithm 3).
2. *Effects of J on overhead.* The overhead percentage can be large if either **(1)** J is large, so processors spend more time doing heap sorts and contending with each other to obtain locks to shared data structures, or **(2)** J is small and P is large, so there is not enough total work to be divided among the processors. In this case a processor may spend a significant amount of time outside the main loop just waiting for work.

3. *Effect of a strong causal structure.* The order of processing the cells is different for pHCM and HCM. On Ex. 1 (Figure 4(a)) there is a strict causal relationship among cells, resulting in exactly one heap removal per cell in HCM. For pHCM the AvS is larger since cells are not generally processed in their strict causal order. In fact, on *any* problem for which HCM has exactly one heap removal per cell, pHCM will almost surely see an increase in the total number of heap removals. A cell is added to some processor's heap when one of its neighbors updates a gridpoint on the inflow boundary. But with a strictly causal cell decomposition, this may well result in avoidable/premature computations if that cell is actually processed before *all* of its inflow boundary data is finalized. This situation is particularly common when P is large and J is relatively small.
4. *Effects of multiple caches.* Even by comparing only the time spent on cell-level sweeping (and accounting for differences in the total AvS), one sees that the speedup factor is closer to P but not exact. When P is larger, it is more likely that adjacent cells will be processed simultaneously, a situation whereby individual sweeps may become slower than their serial counterparts. Referring back to Figure 3, suppose that in the process of updating a border gridpoint $\boldsymbol{x}_i \in A$ the value of its neighbor $\boldsymbol{x}_j \in B$ is loaded into the cache of the local processor p_A. If $\boldsymbol{x}_j$ changes value as a result of sweeps on cell B, the value stored in p_A will either need to be invalidated or have the new value communicated to it [11]. This operation is orders of magnitude slower than simply updating a cached value without communication.
5. *Robustness of pHCM performance.* There is a possibility of the total amount of work increasing significantly if processor speeds vary. Suppose processor $\hat{p}$ is slow or has become slow and is processing a high-priority cell A. The other fast processors will not be able to do useful work on cells downwind from A. What is more, there is a cascade effect: cells downwind from the downwind neighbors of A will need to be readded, etc. This effect is more commonplace for small cells, as seen in Figures 6(d)–6(f). The nonrobust performance of pHCM4 appears to be due entirely to this effect—the error bars for the work are large, while those for the overhead are small. Not surprisingly, pHCM2 (omitted in this paper) shows even less robustness than the reported pHCMr. For small cells and large P, a synchronous parallel implementation may be a wiser choice.
6. *Coarser grids.* The charts in Figure 7 present the same information as in Figure 5, but for $M = 128^3$. The speedup of the parallel methods here is expectedly worse than for $M = 320^3$. Indeed, for a fixed r and P, a smaller M yields a smaller number of cells J. For larger values of P, smaller J results in both an increased overhead and premature processing of cells; see items 2 and 3 above. A good illustration of this is the pHCM32 curve in Figures 7(a) and 7(c). Since $M = 128^3$ here, the cell decomposition for pHCM32 is only four cells per domain side; the scaling plateaus at a low number of threads.
7. *Parallel sweeping.* As reported in [13], the algorithmic complexity of Detrixhe Sweeping is constant in the number of threads; for DFSM and DLSM, charts like those in Figures 6(d)–6(f) would all show a constant value of 1. Unfortunately, the performance is also affected by the fact that memory access patterns are more complicated for DFSM/DLSM than for FSM/LSM, which may prevent the compiler from taking advantage of data locality. Based on our own OpenMP implementation on a shared memory architecture, the scal-

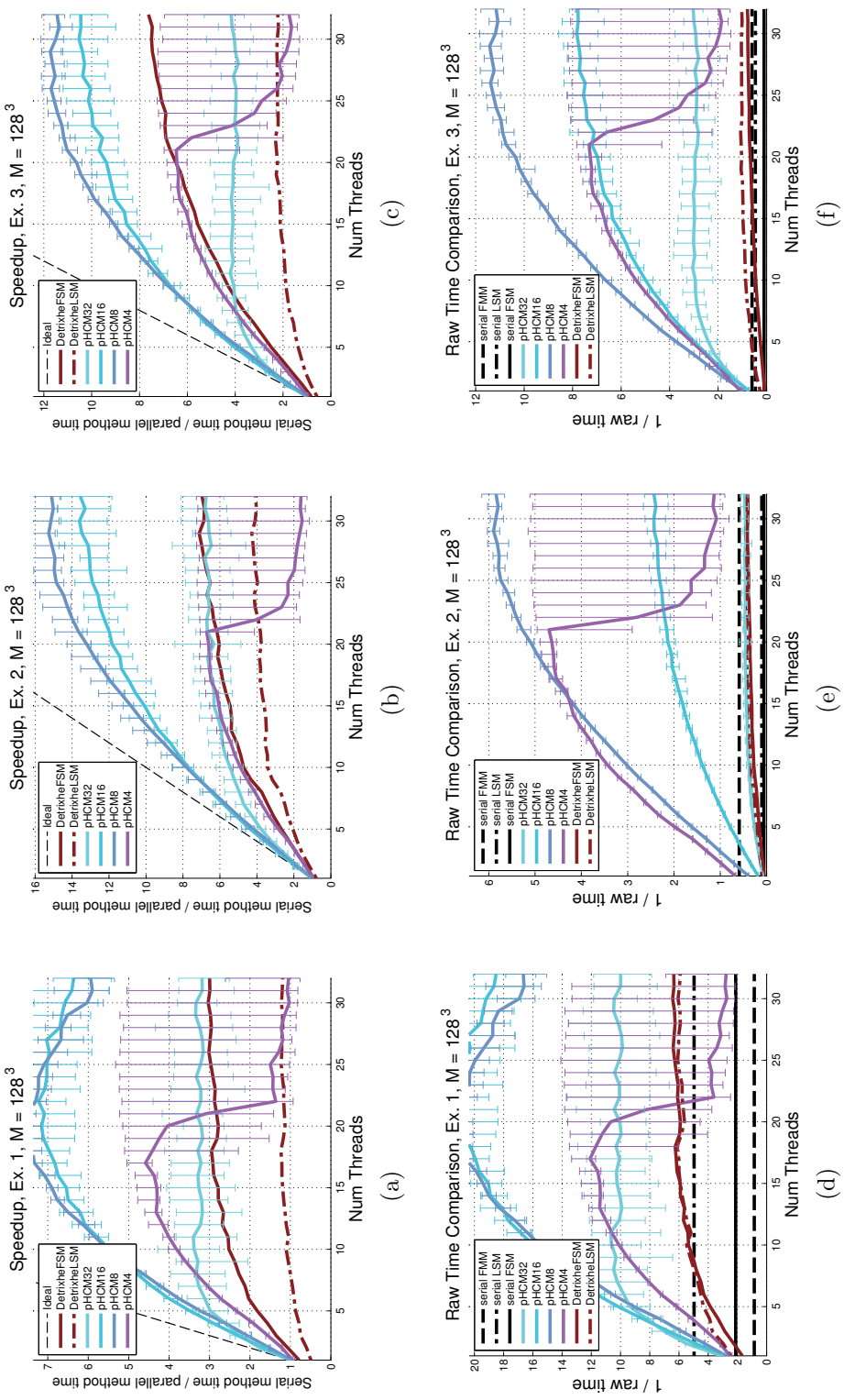

FIG. 7. *Scaling and performance for pHCM at* $M = 128^3$. *The first column has* $F \equiv 1$, *the second has* $F = 1 + .5\sin(20\pi x)\sin(20\pi y)\sin(20\pi z)$, *and the third has* $F = 1 + .99\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$.

ability is also sensitive to hardware properties of the specific platform. In [10] we have also observed somewhat better scaling of DFSM/DLSM on a different hardware system with a smaller ratio of memory bandwidth to CPU speed. We note that the authors of [13] have also implemented their method in lower-level memory languages (MPI, CUDA) to alleviate this sensitivity.

Choosing the *optimal* cell decomposition for a given problem and grid resolution remains a difficult problem even for the serial HCM. But luckily, as shown in Figure 4 and in [8], a wide range of medium-sized cells exhibits good serial performance *and* parallelizes sufficiently well (Figures 5 and 7). In all cases, the parallelization is better when there is more work per cell (e.g., r is large) and there are enough active cells to keep all processors busy.

7. Conclusions. We introduced a new parallel algorithm for the Eikonal equation based on HCM, a fast two-scale serial solver. The numerical experiments demonstrated that pHCM achieves its best speedup on problems where the amount of work per cell is high; this occurred when cells were sufficiently large or when the sweeping within cells required more than a few iterations. As for performance, the combination of HCM's speed and pHCM's good scalability results in a considerable advantage over some of the best serial methods and the parallelization of FSM/LSM. A comprehensive performance/scaling comparison with other existing parallel Eikonal solvers remains to be performed in the future.

The benchmarking and design of our algorithm was influenced by a particular shared memory architecture; e.g., each thread currently handles the cell-level sweeping serially. An efficient hybrid GPU/multicore implementation could parallelize the individual cell processing on a GPU (e.g., as in [36]), while each CPU core would still maintain its own heap. A possible bottleneck of this approach is the smaller number of GPUs compared to the number of CPU cores in most current systems. Extensions to a distributed memory architecture appear more problematic since communication times would likely dominate the cell processing, at least for the first-order upwind discretization of the Eikonal considered in this paper.

As in HCM, the performance of pHCM for each problem is dependent on a particular cell decomposition. For example, given fixed P and M, what value of J will result in the optimal performance? In this paper we only suggest an answer based on our numerical experiments, but rigorously addressing it will be clearly useful for practitioners. Ideally, we would like to base (possibly adaptive) cell decompositions on a posteriori error estimates. Another interesting direction is the use of noncubic cells to improve the causal properties of decompositions.

The performance analysis in section 6 suggests a number of possible pHCM improvements. A smarter memory allocation strategy can be used to increase the spatial and temporal locality of data (particularly in higher-dimensional problems). Rigorous criteria for early sweeping termination would bring additional performance gains to HCM/pHCM (as well as to FSM/LSM). The methods of [13] can be substituted in place of LSM within cells, especially for problems with large cell sizes. In the longer term, we intend to investigate the applicability of our approach to other PDEs and/or discretizations. Causal problems with a higher amount of work per gridpoint (e.g., discretizations of anisotropic Hamilton–Jacobi) are likely to result in even better pHCM scalability. We expect this to also be the case for extensions of other parallel Eikonal solvers (e.g., DFSM/DLSM).

Finally, we hope that practitioners will find pHCM useful for applications requiring its efficiency.

Acknowledgments. We thank David Bindel for guidance with parallel computing, and Jeffrey Donatelli for useful correspondence regarding FMM and memory access costs. We are also grateful to Miles Detrixhe for his help in implementing parallel Fast Sweeping Methods and analyzing their performance. Finally, we thank the XSEDE for the computing time allocation and the Texas Advanced Computing Center for the use of their "Stampede" supercomputer.

REFERENCES

[1] S. Bak, J. McLaughlin, and D. Renzi, *Some improvements for the fast sweeping method*, SIAM J. Sci. Comput., 32 (2010), pp. 2853–2874.

[2] M. Bardi and I. Capuzzo-Dolcetta, *Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman Equations*, Birkhäuser Boston, Boston, 1997.

[3] D. P. Bertsekas, *A simple and fast label correcting algorithm for shortest paths*, Networks, 23 (1993), pp. 703–709.

[4] D. P. Bertsekas, F. Guerriero, and R. Musmanno, *Parallel asynchronous label-correcting methods for shortest paths*, J. Optim. Theory Appl., 88 (1996), pp. 297–320.

[5] M. Boué and P. Dupuis, *Markov chain approximations for deterministic control problems with affine dynamics and quadratic cost in the control*, SIAM J. Numer. Anal., 36 (1999), pp. 667–695.

[6] M. Breuss, E. Cristiani, P. Gwosdek, and O. Vogel, *An adaptive domain-decomposition technique for parallelization of the fast marching method*, Appl. Math. Comput., 218 (2011), pp. 32–44.

[7] S. Cacace, E. Cristiani, M. Falcone, and A. Picarelli, *A patchy dynamic programming scheme for a class of Hamilton–Jacobi–Bellman equations*, SIAM J. Sci. Comput., 34 (2012), pp. A2625–A2649.

[8] A. Chacon and A. Vladimirsky, *Fast two-scale methods for Eikonal equations*, SIAM J. Sci. Comput., 34 (2012), pp. A547–A578.

[9] A. Chacon and A. Vladimirsky, *Fast Two-Scale Methods for Eikonal Equations*, expanded technical report; available online from http://arxiv.org/pdf/1110.6220 (2011).

[10] A. Chacon and A. Vladimirsky, *A Parallel Heap-Cell Method for Eikonal Equations*, expanded technical report; available online from http://arxiv.org/abs/1306.4743 (2014).

[11] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*, Academic Press, San Diego, CA, 2001.

[12] M. G. Crandall and P.-L. Lions, *Viscosity solutions of Hamilton-Jacobi equations*, Trans. Amer. Math. Soc., 277 (1983), pp. 1–42.

[13] M. Detrixhe, F. Gibou, and C. Min, *A parallel fast sweeping method for the Eikonal equation*, J. Comput. Phys., 237 (2013), pp. 46–55.

[14] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[15] J. Donatelli and J. Sethian, *A Massively Parallel Multilevel Fast Marching Method Framework*, unpublished technical report, 2012.

[16] T. Gillberg, M. Sourouri, and X. Cai, *A new parallel 3D front propagation algorithm for fast simulation of geological folds*, Procedia Comput. Sci., 9 (2012), pp. 947–955.

[17] P. A. Gremaud and C. M. Kuster, *Computational study of fast methods for the eikonal equation*, SIAM J. Sci. Comput., 27 (2006), pp. 1803–1816.

[18] M. Herrmann, *A domain decomposition parallelization of the Fast Marching Method*, in Annual Research Briefs, Center for Turbulence Research, Stanford, CA, 2003, pp. 213–225.

[19] S.-R. Hysing and S. Turek, *The Eikonal equation: Numerical efficiency vs. algorithmic complexity on quadrilateral grids*, in Proceedings of ALGORITMY 2005, Conference on Scientific Computing, Slovak University of Technology, Bratislava, Slovakia, 2005, pp. 22–31.

[20] W.-K. Jeong and R. T. Whitaker, *A fast iterative method for Eikonal equations*, SIAM J. Sci. Comput., 30 (2008), pp. 2512–2534.

[21] C. Y. Kao, S. Osher, and J. Qian, *Lax-Friedrichs sweeping scheme for static Hamilton-Jacobi equations*, J. Comput. Phys., 196 (2004), pp. 367–391.

[22] S. Kim, *An $\mathcal{O}(N)$ level set method for eikonal equations*, SIAM J. Sci. Comput., 22 (2001), pp. 2178–2193.

[23] R. Kimmel and J. A. Sethian, *Computing geodesic paths on manifolds*, Proc. Natl. Acad. Sci. USA, 95 (1998), pp. 8431–8435.

[24] F. Li, C.-W. Shu, Y.-T. Zhang, and H.-K. Zhao, *A second order DGM based fast sweeping method for Eikonal equations*, J. Comput. Phys., 227 (2008), pp. 8191–8208.
[25] J. Mellor-Crummey, D. Whalley, and K. Kennedy, *Convergent difference schemes for nonlinear elliptic and parabolic equations*, Internat. J. Parallel Programming, 29 (2001), pp. 217–247.
[26] A. M. Oberman, R. Takei, and A. Vladimirsky, *Homogenization of metric Hamilton–Jacobi equations*, Multiscale Model. Simul., 8 (2009), pp. 269–295.
[27] L. C. Polymenakos, D. P. Bertsekas, and J. N. Tstsiklis, *Implementation of efficient algorithms for globally optimal trajectories*, IEEE Trans. Automat. Control, 43 (1998), pp. 278–283.
[28] E. Rouy and A. Tourin, *A viscosity solutions approach to shape-from-shading*, SIAM J. Numer. Anal., 29 (1992), pp. 867–884.
[29] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, Philadelphia, 2003.
[30] J. A. Sethian, *A fast marching level set method for monotonically advancing fronts*, Proc. Natl. Acad. Sci. USA, 93 (1996), pp. 1591–1595.
[31] J. A. Sethian, *Fast Marching Methods*, SIAM Rev., 41 (1999), pp. 199–235.
[32] J. A. Sethian and A. Vladimirsky, *Fast methods for the Eikonal and related Hamilton–Jacobi equations on unstructured meshes*, Proc. Natl. Acad. Sci. USA, 97 (2000), pp. 5699–5703.
[33] J. A. Sethian and A. Vladimirsky, *Ordered upwind methods for static Hamilton-Jacobi equations*, Proc. Natl. Acad. Sci. USA, 98 (2001), pp. 11069–11074.
[34] Y.-H. R. Tsai, L.-T. Cheng, S. Osher, and H.-K. Zhao, *Fast sweeping algorithms for a class of Hamilton–Jacobi equations*, SIAM J. Numer. Anal., 41 (2003), pp. 673–694.
[35] J. N. Tsitsiklis, *Efficient algorithms for globally optimal trajectories*, IEEE Trans. Automat. Control, 40 (1995), pp. 1528–1538.
[36] O. Weber, Y. Devir, A. Bronstein, M. Bronstein, and R. Kimmel, *Parallel algorithms for the approximation of distance maps on parametric surfaces*, ACM Trans. Graph., 27 (2008), 104.
[37] H. K. Zhao, *A fast sweeping method for Eikonal equations*, Math. Comp., 74 (2005), pp. 603–627.
[38] H. K. Zhao, *Parallel implementations of the fast sweeping method*, J. Comput. Math., 25 (2007), pp. 421–429.