EIKONAL EQUATIONS: NEW TWO-SCALE ALGORITHMS AND ERROR ANALYSIS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Adam Chacon

January 2014

EIKONAL EQUATIONS: NEW TWO-SCALE ALGORITHMS AND ERROR ANALYSIS

Adam Chacon, Ph.D.

Cornell University 2014

Hamilton-Jacobi equations arise in a number of seemingly disparate applications, from front propagation to photolithography to robotic navigation. Eikonal equations fall into an important subset representing isotropic optimal control and often are used as a first benchmark for numerical methods. Many of the interesting geometrical properties of Eikonal and related equations are exploited in two families of popular algorithms: the single-pass Fast Marching Methods and the iterative Fast Sweeping Methods. We start by developing a class of two-scale hybrid algorithms that combine the ideas of these prior methods on different scales. These hybrid methods are shown to have a clear advantage compared to other serial algorithms, but more importantly, one of them ("HCM") is very suitable for parallelization on a shared memory architecture. Our extensive numerical experiments benchmark this parallel HCM against current serial methods and another parallel state-of-the-art solver for the same computer architecture. We demonstrate the robustness of the parallel HCM on a wide range of problems, its good scaling in the number of processors, and its efficiency in solving a problem from exploratory geophysics. In the last part, we focus on estimating the error committed by fast approximate methods that introduce boundary data pollution. Examples include domain restriction methods for recovering only a single optimal path between a source/target pair and a domain decomposition method that creates subdomains whose boundaries are approximately characteristic. In simple cases we use a novel technique to estimate the sensitivity of a gridpoint to other gridpoints in its computational domain of dependence and use this to bound the error.

BIOGRAPHICAL SKETCH

Adam Chacon began attending the University of Texas at Austin in 2003 and graduated in 2006 with a Bachelor of Science in Applied Mathematics. In 2007 he enrolled in the Center for Applied Mathematics at Cornell University as a PhD student under the guidance of Dr. Alexander Vladimirsky. In 2010 he received a Master of Science degree in Applied Mathematics. His dissertation is on efficient numerical methods for a class of partial differential equations and preliminary error bounds for particular approximate numerical methods.

To my mom, dad, and brother.

ACKNOWLEDGEMENTS

Firstly and most importantly, I would like to thank Alex Vladimirsky for his patient training. Together we spent hundreds of hours at the whiteboard in his office discussing interesting problems, which gradually evolved into a productive collaboration. It was especially through his inspiration that I explored new realms of mathematics and its applications.

I am grateful to David Bindel for guidance with parallel computing and for often being the first source through which I discovered new aspects of scientific computing.

I give my thanks to Zach Clawson for a valuable and enjoyable collaboration that allowed us to freely present ideas to each other.

I thank John Guckenheimer for his guidance with numerical analysis, for helping me to see my research in the bigger picture, and for very useful discussions regarding my future plans.

I thank Charlie Van Loan for always being approachable and for our brainstorming that evolved into an important development in the later part of my thesis.

I thank the Cornell Diversity Programs in Engineering for going beyond their administrative duties on multiple occasions to support my endeavors.

I would like to acknowledge Andrew Dolgert (formerly) from Cornell's Center for Advanced Computing for parallel computing and account support. I would also like to acknowledge XSEDE for an allocation on the Stampede supercomputer at the Texas Advanced Computing Center along with the TACC support team.

Finally, I deeply thank Lina, Tipaluck, Sahoko, and my family for much-needed nonacademic support throughout the writing of this dissertation.

This research was supported by the Alfred P. Sloan Foundation Minority PhD Fellowship, the National Science Foundation Grant DMS-1016150, various teaching assistantships with Cornell's Department of Mathematics, and Cornell's Center for Applied Mathematics.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Static Hamilton-Jacobi PDEs arise in a surprisingly wide range of applications: robotic path planning, optimal control, front propagation, shape-from-shading computations, seismic imaging; see [56] and references therein for a detailed description. As a result, efficient numerical methods for Eikonal PDEs are of interest to many practitioners and numerical analysts. In this chapter we briefly introduce the ideas behind three new hybrid methods intended to blend the best properties of the most popular current approaches (Fast Marching and Fast Sweeping).

These methods are built to solve the nonlinear boundary value problem[1]

$$\begin{aligned} |\nabla u(\boldsymbol{x})| F(\boldsymbol{x}) &= 1, \text{ on } \Omega \subset R^2; \\ u(\boldsymbol{x}) &= q(\boldsymbol{x}), \text{ on } \partial\Omega. \end{aligned} \tag{1.1}$$

A discretized version of equation (1.1) is posed at every gridpoint, using upwind divided differences to approximate the partial derivatives of u. The exact form of this discretization is introduced in section 1.2; here we simply note that these discretized equations form a system of M coupled nonlinear equations (where M is the number of gridpoints) and that the key challenge addressed by many "fast" methods is the need to solve this system efficiently. Of course, an iterative approach is certainly possible, but its most straightforward and naive implementation typically leads to $O(M^2)$ algorithmic complexity for Eikonal PDE (and potentially much worse for its anisotropic generalizations). This is in contrast to the "fast" methods, whose worst-case computational complexity is $O(M)$ or $O(M \log M)$.

Interestingly, most fast Eikonal-solvers currently in use are directly related to

[1]For simplicity, we will restrict our exposition to first-order accurate discretizations of these problems on Cartesian grids in R^2.

the fast algorithms developed much earlier to find shortest paths in directed graphs with nonnegative edge-lengths; see, e.g., [1], [9, 10]. Two such algorithmic families are particularly prominent: *label-setting methods*, which have the optimal worst-case asymptotic computational complexity, and *label-correcting methods*, whose worst-case asymptotic complexity is not as good, but the practical performance is at times even better than that of label-setting. We provide a basic overview of both families in section 1.1. The prior fast Eikonal-solvers based on label-setting and label-correcting are reviewed in sections 1.5 and 1.4-1.6 respectively.

The most popular methods from these two categories, Fast Marching and Fast Sweeping, have been shown to be efficient on a wide range of Eikonal equations. However, each of these methods has its own preferred class of problems on which it significantly outperforms the other. Despite experimental comparisons already conducted in [37] and [34], the exact delineation of a preferred problem-set for each method is still a matter of debate. The Fast Sweeping Method (FSM), reviewed in section 1.4, is usually more efficient on problems with constant characteristic directions. But for general functions $F(\boldsymbol{x})$, its computational cost is impacted by the frequency of directional changes of characteristic curves. The Fast Marching Method (FMM), reviewed in section 1.5, is generally more efficient on domains with complicated geometry and on problems with characteristic directions frequently changing. Its causal algorithmic structure results in a provably converged solution on explicitly determined parts of the computational domain even before the method terminates – a very useful feature in many applications. Moreover, its efficiency is much more "robust"; i.e., its computational cost is much less affected by any changes in functions F and q or the grid orientation. But as a result, FMM is also not much faster in simple cases, such as when most characteristics are straight lines – a scenario where FSM is very efficient.

The fundamental idea underlying our hybrid two-scale methods is to take advantage of the best features of both marching and sweeping. Suppose the domain is split into a moderate number of cells such that F is almost-constant on each of them. (Such cell splitting is possible for any piecewise-smooth F.) First, a version of Fast Marching is used on a coarse grid, with each gridpoint representing a cell of the fine grid. Then, once the ordering of coarse gridpoints is established, Fast Sweeping is applied on the fine grid inside individual cells in the same order. This is the basis of our Fast Marching-Sweeping Method (FMSM) described in section 2.1. 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 ought to converge quickly.

Unfortunately, the coarse grid ordering captures the information flow through the fine grid cells only approximately: a coarse gridpoint $\boldsymbol{y}_i$ might be "accepted" by Fast Marching before another coarse gridpoint $\boldsymbol{y}_j$, even if on the fine grid the characteristics cross both from cell i to cell j and from cell j to cell i. The "one-pass" nature of Fast Marching prevents FMSM from acting on such interdependencies between different cells even if they are revealed during the application of Fast Sweeping to these cells. To remedy this, we introduce the Heap-Cell Method (HCM) described in section 2.2.3. The idea is to allow multiple passes through the cells, which are sorted by the representative "cell-values" and updated as a result of cell-level fast sweeping. We also describe its heuristic version, the Fast Heap-Cell Method (FHCM), where the number of cell-level sweeps is determined based on the cell-boundary data.

Similarly to Fast Marching and Fast Sweeping, our HCM provably converges to the exact solution of the discretized equations on the fine scale. In contrast, the even faster FHCM and FMSM usually introduce additional errors. But based on our extensive numerical experiments (sections 2.3 and 3.4.8), these additional errors are small compared to the errors already present due to discretization. The key advantage of all three new methods is their computational efficiency – with properly chosen cell sizes, the hybrid methods significantly outperform both Fast Sweeping and Fast Marching on examples difficult for those methods, while matching their performance on the examples which are the easiest for each of them.

As the scope of applications for Eikonal equations broadens, the more recent literature has focused on developing parallel methods. But Fast Marching has proven difficult to parallelize directly, and a parallel scalable sweeping method for shared-memory computer architectures was only recently discovered [26]. The centerpiece of chapter 3 is the parallel Heap-Cell Method (pHCM)– a scalable version of an already-efficient serial algorithm. Even though the original purpose of the domain decomposition in HCM was to exploit the structure of the PDE serially, we show that parallelization is a natural byproduct; see section 3.4 for numerical results.

In Chapter 4 we are concerned with the problem of recovering the solution at a single point S in the domain rather than in all of Ω. A suite of fast methods [20] addresses this by using heuristic domain restriction techniques similar to those used in the A* algorithms for shortest paths on graphs. Since we again discretize (1.1) using upwind finite differences, the use of domain restriction techniques effectively produces a new, smaller system of coupled equations. Due to the dependency structure among the gridpoints, it turns out that these techniques cause an error at S in addition to the discretization error already present. The goal of the analysis is to estimate this error by using a type of *backward error analysis.* Currently the only analytical results available are for linear advection equations and elementary Eikonal problems where the characteristics are straight and parallel, but we are optimistic that this analysis will generalize sufficiently and even be applicable to the discretizations of other PDE.

Chapters 1, 2, and 3 are based largely on two papers [16, 18] coauthored with

Alex Vladimirsky. Chapter 4 highlights my contribution to work (still-in-progress) together with Zach Clawson and Alex Vladimirsky; it contains excerpts from [20].

In Chapter 5 we discuss the current limitations and future directions of the hybrid methods. We also outline several directions of future work and make sober speculations about the scope of the error analysis.

1.1 Fast algorithms for paths on graphs

Before immediately discussing the Eikonal PDE and current algorithms, we provide a brief review of common fast methods for the classical shortest/cheapest path problems on graphs. Our exposition follows [9] and [10], but with modifications needed to emphasize the parallels with the numerical methods in section 1.2 and Chapter 2.

Consider a directed graph with nodes $X = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_M\}$. Let $N(\boldsymbol{x}_i)$ be the set of nodes to which $\boldsymbol{x_i}$ is connected. We will assume that $\kappa \ll M$ is an upper bound on outdegrees; i.e., $|N(\boldsymbol{x}_i)| \leq \kappa$. We also suppose that all arc-costs $C_{ij} = C(\boldsymbol{x}_i, \boldsymbol{x}_j)$ are positive and use $C_{ij} = +\infty$ whenever $\boldsymbol{x}_j \notin N(\boldsymbol{x}_i)$. Every path terminates upon reaching the specified exit set $Q \subset X$, with an additional exit-cost $q_i = q(\boldsymbol{x}_i)$ for each $\boldsymbol{x}_i \in Q$. Given any starting node $\boldsymbol{x}_i \in X$, the goal is to find the cheapest path to the exit starting from $\boldsymbol{x}_i$. The *value function* $U_i = U(\boldsymbol{x}_i)$ is defined to be the optimal path-cost (minimized over all paths starting from $\boldsymbol{x}_i$). If there exists no path from $\boldsymbol{x}_i$ to Q, then $U_i = +\infty$, but for simplicity we will assume henceforth that U_i is finite $\forall\ i$. The optimality principle states that the "tail" of every optimal path is also optimal; hence,

$$\begin{aligned} U_i &= \min_{\boldsymbol{x}_j \in N(\boldsymbol{x}_i)} \{C_{ij} + U_j\}, \qquad \text{for } \forall \boldsymbol{x}_i \in X \backslash Q; \\ U_i &= q_i, \qquad \text{for } \forall \boldsymbol{x}_i \in Q. \end{aligned} \tag{1.2}$$

This is a coupled system of M nonlinear equations, but it possesses a nice "causal" property: if $\boldsymbol{x}_j \in N(\boldsymbol{x}_i)$ is the minimizer, then $U_i > U_j$.

In principle, this system could be solved by “value iterations”; this approach is unnecessarily expensive (and is usually reserved for harder *stochastic* shortest path problems), but we describe it here for methodological reasons, to emphasize the parallels with “fast” iterative numerical methods for Eikonal PDEs. An operator T is defined on $\boldsymbol{R}^M$ component-wise by applying the right hand side of equation (1.2). Clearly, $U = \begin{bmatrix} U_1 \\ \vdots \\ U_M \end{bmatrix}$ is a fixed point of T and one can, in principle, recover U by value iterations:

$$W^{k+1} := T\,W^k \qquad \text{starting from any initial guess } W^0 \in \boldsymbol{R}^M. \tag{1.3}$$

Due to the causality of system (1.2), value iterations will converge to U regardless of W^0 after at most M iterations, resulting in $O(M^2)$ computational cost. This is easy to show by induction; e.g., after one iteration at least one of the neighboring nodes of Q will receive its final value. A Gauss-Seidel relaxation of this iterative process is a simple practical modification, where the entries of W^{k+1} are computed sequentially and the new values are used as soon as they become available: $W_i^{k+1} = T_i(W_1^{k+1}, \ldots, W_{i-1}^{k+1}, W_i^k, \ldots, W_M^k)$. The number of iterations required to converge will now heavily depend on the ordering of the nodes (though M is still the upper bound). We note that, again due to causality of (1.2), if the ordering is such that $U_i > U_j \implies i > j$, then only one full iteration will be required (i.e., $W^1 = U$ regardless of W^0). Of course, U is not known in advance and thus such a causal ordering is usually not available a priori (except in acyclic graphs). If several different node orderings are somehow known to capture likely dependency chains among the nodes, then a reasonable approach would be to perform Gauss-Seidel iterations alternating through that list of preferred orderings – this might potentially result in a substantial reduction in the number of needed iterations. In section 1.4 we explain how such preferred orderings arise from the geometric structure of PDE discretizations, but no such information is typically available in problems on graphs. As a result, instead of alternating through a list of predetermined orderings, efficient methods on graphs are based on finding advantageous orderings of nodes *dynamically.* This is the basis for *label-correcting* and *label-setting* methods.

A generic label-correcting method is summarized below in algorithm 1. It is easy

Algorithm 1 Generic Label-Correcting pseudocode.

```
1: Initialization:
2: for each node $\boldsymbol{x}_i$ do
3:     if $\boldsymbol{x}_i \in Q$ then
4:         $V_i \leftarrow q_i$
5:     else
6:         if $N(\boldsymbol{x}_i) \bigcap Q \neq \emptyset$ then
7:             $V_i \leftarrow \min_{\boldsymbol{x}_j \in N(\boldsymbol{x}_i) \bigcap Q} \{C_{ij} + q_j\}$
8:             add $\boldsymbol{x}_i$ to the list $L$
9:         else
10:            $V_i \leftarrow \infty$
11:        end if
12:    end if
13: end for
14:
15: Main Loop:
16: while $L$ is nonempty do
17:     Remove a node $\boldsymbol{x}_j$ from the list $L$
18:     for each $\boldsymbol{x}_i \notin Q$ such that $\boldsymbol{x}_j \in N(\boldsymbol{x}_i)$ and $V_j < V_i$ do
19:         $\widetilde{V} \leftarrow C_{ij} + V_j$
20:         if $\widetilde{V} < V_i$ then
21:             $V_i \leftarrow \widetilde{V}$
22:             if $\boldsymbol{x}_i \notin L$ then
23:                 add $\boldsymbol{x}_i$ to the list $L$
24:             end if
25:         end if
26:     end for
27: end while
```

to prove that this algorithm always terminates and that upon its termination $V = U$;

e.g., see [9]. Many different label-correcting methods are obtained by using different choices on how to add the nodes to the list L and which node to remove (in the first line inside the while loop). If L is implemented as a queue, the node is typically removed from the top of L. Always adding the nodes at the bottom of L yields the *Bellman-Ford method* [6]. (This results in a first-in/first-out policy for processing the queue.) Always adding nodes at the top of L produces the *depth-first-search* method, with the intention of minimizing the memory footprint of L. Adding nodes at the top if they have already been in L before, while adding the "first-timers" at the bottom yields *D'Esopo-Pape method* [49]. Another interesting version is the so called *small-labels-first* (SLF) method [8], where the node is added at the top only if its value is smaller than that of the current top node and at the bottom otherwise. Another variation is *large-labels-last* (LLL) method [11], where the top node is removed only if its value is smaller than the current average of the queue; otherwise it's simply moved to the bottom of the queue instead. Yet another popular approach is called the *thresholding method*, where L is split into two queues, nodes are removed from the first of them only and added to the first or the second queue depending on whether the labels are smaller than some (dynamically changing) threshold value [32]. We emphasize that the convergence is similarly obtained for all of these methods and their worst-case asymptotic complexity is $O(M^2)$, but their comparative efficiency for specific problems can be dramatically different.

Label-setting algorithms can be viewed as a subclass of the above with an additional property: nodes removed from L never need to be re-added later. Dijkstra's classical method [27] is the most popular in this category and is based on removing the node with the smallest label of those currently in L at each iteration and marking it as ACCEPTED. The fact that this results in no re-entries into the list is yet another consequence of the causality, and the inductive proof is simple; e.g., see [9]. The need to find the smallest label entails additional computational costs. A common implementation of L using heap data structures will result in $O(M \log M)$ overall asymptotic complexity of the method on sparsely connected graphs (i.e., provided $\kappa \ll M$). Another version, due to Dial [25], implements L as a list of "buckets", so that all nodes in the current smallest bucket can be safely removed simultaneously, resulting in the overall asymptotic complexity of $O(M)$. The width of each bucket is usually set to be $\delta = \min_{i,j} C_{ij}$ to ensure that the nodes in the same bucket could not influence or update each other even if they were removed sequentially.

We note that several label-correcting methods were designed to mimic the "no-re-entry" property of label-setting, but without using expensive data structures. (E.g., compare SLF/LLL to Dijkstra's and thresholding to Dial's.) Despite the lower asymptotic complexity of label-setting methods, label-correcting algorithms can be more efficient on many problems. Which types of graphs favor which of these algorithms remains largely a matter of debate. We refer readers to [9, 10] and references therein for additional details and asynchronous (parallelizable) versions of label-correcting algorithms.

When interested in the solution at only a single node S in the domain, one must wonder if there is any way to reduce the problem size. One can always simply halt Dijkstra's algorithm once S becomes ACCEPTED. Or alternatively, if an overestimate $\mathcal{O}$ of the value function $U(S)$ is available, there is a quick way to determine whether a node $\boldsymbol{x}_i$ is relevant to the computation of $U(S)$: if $U_i > \mathcal{O}$, then surely $\boldsymbol{x}_i$ does not lie on the optimal path (by the causality property of (1.2)).

The goal of the A* methods on graphs is to use such under/overestimates to accelerate the computation by considering only a small neighborhood of nodes on the optimal path. Here we do not discuss these algorithms in detail; we simply note that the translation of A* to an Eikonal solver is still a current area of research, and that domain restriction methods for Eikonal equations generally cause an error at $U(S)$

(see section 1.3). A study of the error and computational savings of different A* techniques can be found in [20].

1.2 Eikonal PDE, upwind discretization & prior fast methods

Static Hamilton-Jacobi equations frequently arise in exit-time optimal control problems. The Eikonal PDE (1.1) describes an important subset: isotropic time-optimal control problems. The goal is to drive a system starting from a point $\boldsymbol{x} \in \Omega$ to exit the domain as quickly as possible. In this setting, $F : \Omega \to \boldsymbol{R}_+$ is the local speed of motion, and $q : \partial\Omega \to \boldsymbol{R}$ is the exit-time penalty charged at the boundary. We note that more general control problems (with an exit-set $Q \subset \partial\Omega$ and trajectories constrained to remain inside Ω until reaching Q) can be treated similarly by setting $q = +\infty$ on $\partial\Omega\backslash Q$.

The *value function* $u(\boldsymbol{x})$ is defined to be the minimum time-to-exit starting from $\boldsymbol{x}$, and a formal argument shows that u should satisfy the equation (1.1). Moreover, characteristics of this PDE, coinciding with the gradient lines of u, provide the optimal trajectories for moving through the domain. Unfortunately, Equation (1.1) usually does not have a classical (smooth) solution on the entire domain, while weak solutions are not unique. Additional test conditions are used to select among them the unique *viscosity solution*, which coincides with the value function of the original control problem [22, 21]. A detailed treatment of general optimal control problems in the framework of viscosity solutions can be found in [4].

Many discretization approaches for the Eikonal equation have been extensively studied, including first-order and higher-order Eulerian discretizations on grids and meshes in $\boldsymbol{R}^n$ and on manifolds [53, 55, 42, 58], semi-Lagrangian discretizations

[30, 33], and the related approximations with controlled Markov chains [43, 13]. For the purposes of this thesis, we will focus on the simplest first-order upwind discretization on a uniform Cartesian grid X (with gridsize h) on $\overline{\Omega} \subset \boldsymbol{R}^2$. To simplify the description of algorithms, we will further assume that both $\partial\Omega$ and Q are naturally discretized on the grid X. Our exposition here closely follows [57, 56].

We will also consider slightly more general problems, where exiting is only allowed 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 non-inflow boundary, where the boundary conditions are "satisfied in a viscosity sense"; see [4].

To introduce the notation, we will refer to gridpoints $\boldsymbol{x}_{ij} = (x_i, y_j)$, value function approximations $U_{ij} = U(\boldsymbol{x}_{ij}) \approx u(\boldsymbol{x}_{ij})$, and the speed $F_{ij} = F(\boldsymbol{x}_{ij})$. A popular first-order accurate discretization of (1.1) is obtained by using upwind finite-differences to approximate partial derivatives:

$$\left(\max\left(D_{ij}^{-x}U,\, -D_{ij}^{+x}U,\, 0\right)\right)^2 \;+\; \left(\max\left(D_{ij}^{-y}U,\, -D_{ij}^{+y}U,\, 0\right)\right)^2 \;=\; \frac{1}{F_{ij}^2}, \tag{1.4}$$

$$\text{where} \qquad u_x(x_i, y_j) \approx D_{ij}^{\pm x}U = \frac{U_{i\pm1,j} - U_{i,j}}{\pm h}; \qquad u_y(x_i, y_j) \approx D_{ij}^{\pm y}U = \frac{U_{i,j\pm1} - U_{i,j}}{\pm h}.$$

If the values at four surrounding gridpoints are known, this equation can be solved to recover U_{ij}. This is best accomplished by computing updates from individual quadrants as follows. Focusing on a single node $\boldsymbol{x}_{ij}$, we will simplify the notation by using $U = U_{ij}$, $F = F_{ij}$, and $\{U_E, U_N, U_W, U_S\}$ for the values at its four neighbor gridpoints.

First, suppose that $\max\left(D_{ij}^{-x}U,\, -D_{ij}^{+x}U,\, 0\right) = 0$ and $\max\left(D_{ij}^{-y}U,\, -D_{ij}^{+y}U,\, 0\right) =$

$-D_{ij}^{+y}U$. This implies that $U \geq U_N$ and the resulting equation yields

$$U = h/F + U_N. \tag{1.5}$$

To compute "the update from the first quadrant", we now suppose that $\max\left(D_{ij}^{-x}U,\, -D_{ij}^{+x}U,\, 0\right) = -D_{ij}^{+x}U$ and $\max\left(D_{ij}^{-y}U,\, -D_{ij}^{+y}U,\, 0\right) = -D_{ij}^{+y}U$. This implies that $U \geq U_N$ and $U \geq U_E$. The resulting quadratic equation is

$$\left(\frac{U - U_E}{h}\right)^2 + \left(\frac{U - U_N}{h}\right)^2 = \frac{1}{F^2}. \tag{1.6}$$

We define "the update from the first quadrant" U^{NE} to be the root of the above quadratic satisfying $U \geq \max(U_N, U_E)$. If no such root is available, we use the smallest of the "one-sided" updates, similar to the previous case; i.e., $U^{NE} = h/F + \min(U_N, U_E)$. If we similarly define the updates from the remaining three quadrants, it is easy to show that $U = \min(U^{NE}, U^{NW}, U^{SW}, U^{SE})$ satisfies the original equation (1.4).

It is also easy to verify that this discretization is

- *consistent*, i.e., suppose both sides of (1.4) are multiplied by h^2; if the true solution $u(\boldsymbol{x})$ is smooth, it satisfies the resulting discretized equation up to $O(h^2)$;
- *monotone*, i.e., U is a non-decreasing function of each of its neighboring values;
- *causal*, i.e., U depends only on the neighboring values smaller than itself [55, 56].

The consistency and monotonicity can be used to prove the convergence to the viscosity solution $u(\boldsymbol{x})$; see [5].

However, since (1.4) has to hold at every gridpoint $\boldsymbol{x}_{ij} \in X \backslash Q$, this discretization results in a system of M coupled nonlinear equations, where M is the number of gridpoints in the interior of Ω. In principle, this system can be solved iteratively (similarly to the value iterations process described in (1.3)) with or without Gauss-Seidel relaxation, but a naive implementation of this iterative algorithm would be unnecessarily expensive, since it does not take advantage of the causal properties of the discretization. Several competing approaches for solving the discretized system efficiently are reviewed in the following subsections.

1.3 The dependency structure of the gridpoints

Suppose all gridpoints in X are ordered. We will slightly abuse the notation by using a single subscript (e.g., $\boldsymbol{x}_i$) to indicate the particular gridpoint's place in that ordering. The double subscript notation (e.g., $\boldsymbol{x}_{ij}$) will still be reserved to indicate the physical location of a gridpoint in the two-dimensional grid.

Consider discretization (1.4) and suppose that the solution U has been computed everywhere. Each $\boldsymbol{x}_i$ depended on one or two of its immediate neighboring gridpoints, determined by which quadrant was used for a two-sided update (similar to (1.6)), and if a one-sided update was used (similar to (1.5)). This allows us to define a *dependency digraph* G on the vertices $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M$ with a link from $\boldsymbol{x}_i$ to $\boldsymbol{x}_j$ indicating that U_j was needed to compute U_i. The causality of the discretization (1.4) guarantees that G is always acyclic. We also refer to $G(\boldsymbol{x}_i)$ as the subgraph of G containing only the computational domain of dependence of the gridpoint $\boldsymbol{x}_i$.

As for the domain restriction techniques, it may at first seem unintuitive that there is any resulting error at all; after all, the A* methods on graphs recover the value function exactly. Furthermore, for the physical PDE the domain of dependence of a point x is exactly the characteristic connecting it to the boundary, so a domain restriction away from the optimal path would not cause any error. But given the discretized system (1.4), the notion of the dependency graph explains why a domain restriction would cause error: the *computational* domain of dependence can be quite large, even including gridpoints quite far away from the optimal path.

As an example, consider a problem where $F \equiv 1$ and Q consists of only the single point in the lower left corner of the domain. In this case all characteristics passing through interior gridpoints point towards the southwest, and correspondingly all the updates of these gridpoints are two-sided SW updates. See Figure 1.1.

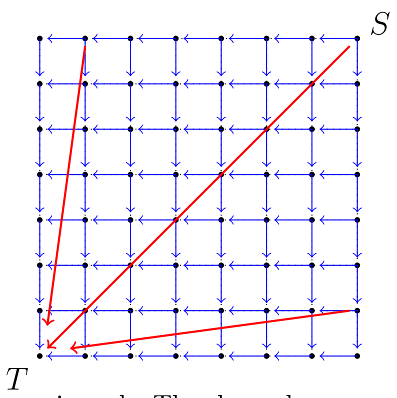


Figure 1.1: Optimal path drawn in red. The dependency graph of S is the entire grid in this example, even under refinement.

It is easy to visualize the dependency structure among the gridpoints for this problem: S depends on its south and west neighbors, which in turn depend on their south and west neighbors, etc. If we solve this problem at S by a domain restriction that cuts out *any* gridpoint in the domain, it is clear that the resulting local error ought to propagate towards S. In Chapter 4 we analyze the effect of this kind of error for the much simpler linear advection equations with constant coefficients, which corresponds to Eikonal problems where the characteristics are straight and parallel.

1.4 Fast Sweeping Methods

Suppose we were to order the gridpoints in such a way that $i > j \implies$ there is no path in G from $\boldsymbol{x}_j$ to $\boldsymbol{x}_i$. Then a single Gauss-Seidel iteration would correctly solve the full system in $O(M)$ operations. However, we see that unless U was already computed, the dependency digraph G will not be generally known in advance. Thus, basing a gridpoint ordering on it is not a practical option. Instead, one can alternate through a list of several "likely" orderings while performing Gauss-Seidel iterations. A geometric interpretation of the optimal control problem provides a natural list of likely orderings: if all characteristics point from SW to NE, then ordering the gridpoints bottom-to-top and left-to-right within each row will ensure the convergence in a single iteration (a "SW sweep").

The "Fast Sweeping Methods" [62, 70] perform Gauss-Seidel iterations on the system (1.4) in alternating directions (sweeps). Let m be the number of gridpoints in the x-direction and n be the number in the y-direction, and $\boldsymbol{x}_{ij}$ will denote a gridpoint in a uniform Cartesian grid on $\Omega \subset R^2$. There are four alternating sweeping directions: from SW, from SE, from NE, and from NW. For the above described southwest sweep, the gridpoints $\boldsymbol{x}_{ij}$ will be processed in the following order: `i=1:1:m, j=1:1:n` (MATLAB index notation). All four orderings are similarly defined in algorithm 2.

Algorithm 2 Sweeping Order Selection pseudocode.

```
1: $sweepDirection \leftarrow sweepNumber \texttt{ mod } 4$
2: if $sweepDirection == 0$ then
3:     $iOrder \leftarrow (1:1:m)$
4:     $jOrder \leftarrow (1:1:n)$
5: else if $sweepDirection == 1$ then
6:     $iOrder \leftarrow (1:1:m)$
7:     $jOrder \leftarrow (n:-1:1)$
8: else if $sweepDirection == 2$ then
9:     $iOrder \leftarrow (m:-1:1)$
10:    $jOrder \leftarrow (n:-1:1)$
11: else
12:    $iOrder \leftarrow (m:-1:1)$
13:    $jOrder \leftarrow (1:1:n)$
14: end if
```

The alternating sweeps are then repeated until convergence. The resulting algorithm is summarized in 3.

Algorithm 3 Fast Sweeping Method pseudocode.

```
Initialization:
for each gridpoint $\boldsymbol{x}_{ij} \in X$ do
    if $\boldsymbol{x}_{ij} \in Q$ then
        $V_{ij} \leftarrow q(\boldsymbol{x}_{ij})$.
    else
        $V_{ij} \leftarrow \infty$.
    end if
end for

Main Loop:
sweepNumber ← 0
repeat
    changed ← FALSE
    Determine iOrder and jOrder based on sweepNumber
    for $i = iOrder$ do
        for $j = jOrder$ do
            if $\boldsymbol{x}_{ij} \notin Q$ then
                Compute a temporary value $\widetilde{V}_{ij}$ using upwinding discretization
                (1.4).
                if $\widetilde{V}_{ij} < V_{ij}$ then
                    $V_{ij} \leftarrow \widetilde{V}_{ij}$
                    changed ← TRUE
                end if
            end if
        end for
    end for
    sweepNumber ← sweepNumber + 1
until changed == FALSE
```

The idea that alternating the order of Gauss-Seidel sweeps might speed up the convergence is a centerpiece of many fast algorithms. For Euclidean distance computations it was first used by Danielsson in [24]. In the context of general HJB PDEs it was introduced by Boue and Dupuis in [13] for a numerical approximation based on controlled Markov chains. More recently, a number of papers by Cheng, Kao, Osher, Qian, Tsai, and Zhao introduced related Fast Sweeping Methods to speed up the iterative solving of finite-difference discretizations [62, 70, 39]. The key challenge for these methods is to find a provable and explicit upper bound on the number of iterations. As of right now, such a bound is only available for boundary value problems in which characteristics are straight lines. Experimental evidence suggests that these methods can be also very efficient for other problems where the characteristics are "largely" straight. The number of necessary iterations is independent of M and equal to the number of times the characteristics "switch directions" (i.e., change from one directional quadrant to another) inside Ω. However, since the quadrants are defined relative to the grid orientation, the number of iterations will generally be grid-dependent.

We note that the sweeping approach can be in principle useful for a very wide class of problems. For example, the method introduced in [39] is applicable to problems with nonconvex Hamiltonians corresponding to differential games; however, the amount of required artificial viscosity is strongly problem-dependent and the choice of consistently discretized boundary conditions can be complicated. Sweeping algorithms for discontinuous Galerkin finite element discretizations of the Eikonal PDE can be found in [44, 69].

The Fast Sweeping Method performs particularly well on problems where the speed function F is constant, since in this case the characteristics of the Eikonal PDE will be straight lines regardless of the boundary conditions. (E.g., if $q \equiv 0$, then the quickest path is a straight line to the nearest boundary point.)

It might seem that the recomputation of V_{ij} from (1.4) will generally require solving 4 quadratic equations to compare the updates from all 4 quadrants. However, the monotonicity property noted above guarantees that only one quadrant needs to be considered. E.g., if $U_S < U_N$ then $U^{SE} \leq U^{NE}$ and the latter is irrelevant even if we are currently sweeping from NE. Thus, the relevant quadrant can be always found by using $\min(U_S, U_N)$ and $\min(U_E, U_W)$. We note that this shortcut is not directly applicable to discretizations on unstructured meshes nor for more general PDEs. Interestingly, Alton and Mitchell showed that the same shortcut can also be used with Cartesian grid discretizations of Hamilton-Jacobi PDEs with grid-aligned anisotropy [2].

One of the problems in this basic version of the Fast Sweeping Method is the fact that the CPU time might be wasted to recompute V_{ij} even if none of $\boldsymbol{x}_{ij}$'s neighbors have changed since the last sweep. To address this, one natural modification is to introduce "active flags" [2] for individual gridpoints and to update the currently *active* gridpoints only [3]. Briefly, all gridpoints but those immediately adjacent to Q start out as *inactive*. When an *active* gridpoint $\boldsymbol{x}_{ij}$ is processed during a sweep, if U_{ij} changes, then all of its larger neighbors are marked *active*. The gridpoint $\boldsymbol{x}_{ij}$ is then itself marked *inactive* regardless of whether updating U_{ij} resulted in activating a neighbor. We note that a similar mechanism was previously introduced by Falcone in the context of semi-Lagrangian discretizations [30].

This modification does not change the asymptotic complexity of the method nor the total number of sweeps needed for convergence. Nevertheless, the extra time and memory required to maintain and update the active flags are typically worthwhile since their use allows to decrease the amount of CPU-time wasted on parts of the domain, where the iterative process already produced the correct numerical solutions. In Chapters 2 and 3 we will refer to this modified version as Locking Sweeping Method (LSM) to distinguish it from the standard implementation of the FSM.

1.5 Label-setting methods for the Eikonal

Causality is the basis of Dijkstra-like methods for the Eikonal PDE. Just as in shortest path problems on graphs, Dijkstra-like methods dynamically decouple system (1.4)

[2]We avoid the use of the word "locks" (the original terminology of [3]) to avoid confusion with the mutex locks used later in this document in the context of parallel programming.

in such a way that if $U_i < U_j$, then $\boldsymbol{x}_i$ is processed before $\boldsymbol{x}_j$.

The first such method was introduced by Tsitsiklis for isotropic control problems using first-order semi-Lagrangian discretizations on uniform Cartesian grids [63, 64]. The Fast Marching Method was introduced by Sethian [55] using first-order upwind-finite differences in the context of isotropic front propagation. A detailed discussion of similarities and differences of these approaches can be found in [60]. Sethian and collaborators have later extended the Fast Marching approach to higher-order discretizations on grids and meshes [57], more general anisotropic Hamilton-Jacobi-Bellman PDEs [59, 60], and quasi-variational inequalities [61]. Similar methods were also introduced for semi-Lagrangian discretizations [23]. The Fast Marching Method for the Eulerian discretization (1.4) is summarized below in Algorithm 4.

As explained in section 1.1, the label-setting Dijkstra's method can be considered as a special case of the generic label-correcting algorithm, provided the current smallest node in L is always selected for removal. Of course, in this case it is more efficient to implement L as a binary heap rather than a queue. The same is also true for the Fast Marching Method, and a detailed description of an efficient implementation of the heap data structure can be found in [56]. The re-sorting of *Considered* nodes upon each update involves up to $O(\log M)$ operations, resulting in the overall computational complexity of $O(M \log M)$.

Unfortunately, the discretization (1.4) is only causal in the sense that there exists no $\delta > 0$ such that $U^{NE} > \delta + \max(U^N, U^E)$ whenever $U^{NE} > \max(U^N, U^E)$. Thus, no safe "bucket width" can be defined and Dial-like methods are not applicable to the resulting discretized system. In [64] a Dial-like method is introduced for a similar discretization but using an 8-neighbor stencil. More recently, another Dial-related method for the Eikonal PDE on a uniform grid was introduced in [41]. A more general formula for the safe bucket-width to be used in Dial-like methods on unstructured

Algorithm 4 Fast Marching Method pseudocode.

```
1: Initialization:
2: for each gridpoint $\boldsymbol{x}_{ij} \in X$ do
3:     if $\boldsymbol{x}_{ij} \in Q$ then
4:         Label $\boldsymbol{x}_{ij}$ as Accepted and set $V_{ij} = q(\boldsymbol{x}_{ij})$.
5:     else
6:         Label $\boldsymbol{x}_{ij}$ as $Far$ and set $V_{ij} = \infty$.
7:     end if
8: end for
9: for each $Far$ neighbor $\boldsymbol{x}_{ij}$ of each Accepted node do
10:     Label $\boldsymbol{x}_{ij}$ as $Considered$ and put $\boldsymbol{x}_{ij}$ onto the Considered List $L$.
11:     Compute a temporary value $\widetilde{V}_{ij}$ using the upwinding discretization.
12:     if $\widetilde{V}_{ij} < V_{ij}$ then
13:         $V_{ij} \leftarrow \widetilde{V}_{ij}$
14:     end if
15: end for
16: End Initialization
17:
18: while $L$ is nonempty do
19:     Remove the point $\bar{x}$ with the smallest value from $L$.
20:     for $\boldsymbol{x}_{ij} \in N(\bar{x})$ do
21:         Compute a temporary value $\widetilde{V}_{ij}$ using the upwinding discretization.
22:         if $\widetilde{V}_{ij} < V_{ij}$ then
23:             $V_{ij} \leftarrow \widetilde{V}_{ij}$
24:         end if
25:         if $\boldsymbol{x}_{ij}$ is $Far$ then
26:             Label $\boldsymbol{x}_{ij}$ as $Considered$ and add it to $L$.
27:         end if
28:     end for
29: end while
```

acute meshes was derived in [65]. Despite their better computational complexity, Dial-like methods often perform slower than Dijkstra-like methods, at least on single processor architectures.

Finally, we note another convenient feature of label-setting methods: if the execution of the algorithm is stopped early (before the list L becomes empty), all gridpoints previously removed from L will already have provably correct values. This property (unfortunately not shared by the methods in sections 1.4-1.6) is very useful in a number of applications: e.g., when computing a quickest path from a single source to a single target or in problems of image segmentation [56].

1.6 Other fast methods for Eikonal equations

Ideas behind many label-correcting algorithms on graphs have also been applied to discretizations of Eikonal PDEs. Here we aim to briefly highlight some of these connections.

Perhaps the first label-correcting methods developed for the Eikonal PDE were introduced by Polymenakos, Bertsekas, and Tsitsiklis based on the logic of the discrete SLF/LLL algorithms [51]. On the other hand, Bellman-Ford is probably the simplest label-correcting approach and it has been recently re-invented by several numerical analysts working with Eikonal and more general Hamilton-Jacobi-Bellman PDEs [12], [3], including implementations for massively parallel computer architectures [38]. In [3] another "2-queues method" is also introduced, essentially mimicking the logic of thresholding label-correcting algorithms on graphs. While such algorithms clearly have promise and some numerical comparisons of them with sweeping and marching techniques are already presented in the above references, more careful analysis and testing is required to determine the types of examples on which they are the most efficient. We emphasize, however, that even though many of these prior methods are less well-known, in practice they are sometimes (not uncommonly) faster than either FSM or FMM.

All of the above methods produce the exact same numerical solutions as FMM and FSM. In contrast, two of the three new methods introduced in Chapter 2 aim to gain efficiency even if it results in small additional errors. We know of only one prior numerical method for Eikonal PDEs with a similar trade-off: in [68] a Dial-like method is used with buckets of unjustified width δ for a discretization that is not δ-causal. This introduces additional errors (analyzed in [52]), but decreases the method's running time. However, the fundamental idea behind our new two-scale methods is quite different, since we aim to exploit the geometric structure of the speed function.

There are lately also many efficient parallel algorithms for solving (1.4) and related discretizations. We defer our review of these until section 3.1 in order to put the parallel Heap-Cell Method into context.

CHAPTER 2

NEW HYBRID TWO-SCALE METHODS

We present three new hybrid methods based on splitting the domain into a collection of non-overlapping rectangular "cells" and running the Fast Sweeping Method on individual cells sequentially. The motivation for this decomposition is to break the problem into sub-problems, with F nearly constant inside each cell. If the characteristics rarely change their quadrant-directions within a single cell, then a small number of sweeps should be sufficient on that cell; see Figure 2.1. But to compute the value function correctly within each cell, the correct boundary conditions (coming from the adjacent cells) should be already available. In other words, we need to establish a causality-respecting order for processing the cells. The Fast Marching Sweeping Method (FMSM) uses the cell-ordering found by running the Fast Marching Method on a coarser grid, while the Heap-Cell Methods (HCM and FHCM) determine the cell-ordering dynamically, based on updates along the cell-boundaries.

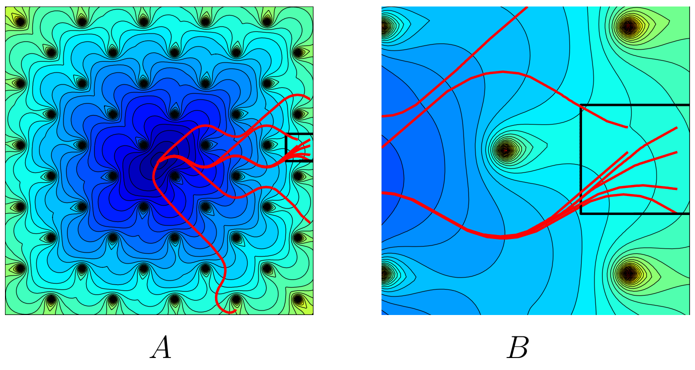


Figure 2.1: Left: level sets of a particular value function with select characteristics drawn in red (A). Right: a zoom on a cell. Globally the Eikonal problem in A is very difficult for sweeping methods, whereas a sweeping method restricted to a tiny subdomain (B) ought to converge relatively quickly.

We first introduce some relevant notation:

- $X = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_M\}$, 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 ge-

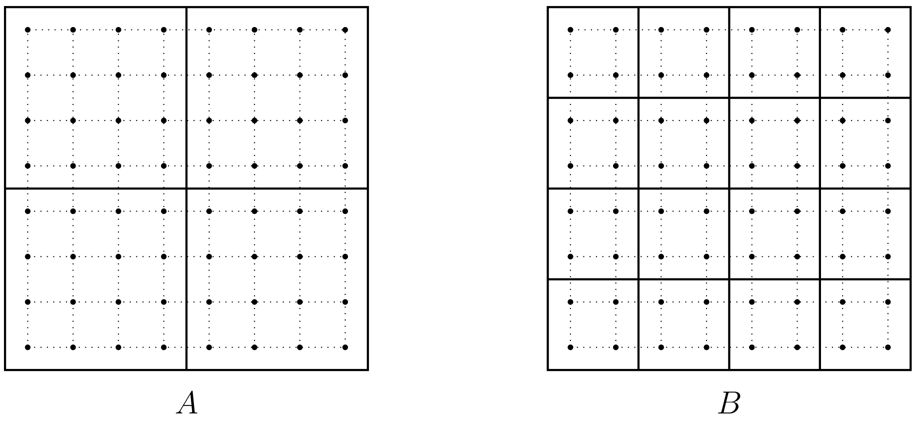


Figure 2.2: Two examples with different domain decompositions. Both A and B are based on the same grid (dotted), with $M = 8^2$ and $h = 1/7$. Figure A uses the cell size $h^c = 4/7$, the total number of cells $J = 2^2$, and $r = 4$ gridpoints per cell-side. Figure B uses $h^c = 2/7$, $J = 4^2$, and $r = 2$.

ometric position indicated by the subscripts in formula (1.4). 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.
- $Z = \{c_1, ..., c_J\}$, the set of cells (or "non-overlapping 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., $N(c_i) = \{\boldsymbol{x}_j \in X \mid \boldsymbol{x}_j \notin c_i \text{ and } N(\boldsymbol{x}_j) \bigcap c_i \neq \emptyset\}$.
- V^c, the cell label.
- h^c_x and h^c_y, the two cell dimensions (assume $h^c_x = h^c_y = h^c$).
- 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 $\Omega^c = \bigcup\limits_{j=1,...,J} \overline{c_j}$ must be a superset of $\overline{\Omega}$; see Figure 2.2.

2.1 Fast Marching-Sweeping Method (FMSM)

This algorithm uses a coarse grid and a fine grid. Each "coarse gridpoint" is taken to be the center of a cell of "fine gridpoints". Fast Marching is used on the coarse grid, and the acceptance-order of coarse gridpoints is recorded. A sweeping method is then used on the corresponding cells in the same order. An additional speed-up is obtained by running only a fixed number of sweeps on each cell based on the upwind directions determined on the coarse grid. Before providing the details of our implementation, we introduce additional notation relevant to FMSM:

- $X^c = \{\boldsymbol{x}_1^c, ..., \boldsymbol{x}_J^c\}$, the coarse grid, corresponding to the centers of cells.
- U^c, the solution of the discretized equations on the coarse grid.
- V^c, the temporary label of the coarse gridpoints.
- $\pi : \{1, ..., J\} \to \{1, ..., J\}$, a permutation on the coarse gridpoint indices.

In this section only, the value at a cell $V^c(c)$ is determined by the label of the coarse gridpoint.We assume for simplicity that the exit set is representable on the coarse gridpoints lying in exit set cells Q^c; in section 2.8, where this is not the case, we use interpolation to assign the initial values U_i^c. We still reserve the notation U, h, etc. for the fine grid. Since Fast Marching is used on the coarse grid only, the heap L will contain coarse gridpoints only.

Remark 1. The "Modified Fast Sweeping" procedure applied to individual cells in Algorithm 5 follows the same idea as the FSM described in section 1.4. For all the cells containing parts of Q (i.e., the ones whose centers are Accepted *in the initialization* of the FMM on the coarse grid) we use the FSM without any changes. For all the remaining cells, our implementation has 3 important distinctions from Algorithm 3:

1. No initialization of the fine gridpoints within $\tilde{c}$ is needed since the entire fine grid is initialized in advance.

Algorithm 5 Fast Marching-Sweeping Method pseudocode.

```
1:  Part I:
2:  Run FMM on $X^c$ (see algorithm 4).
3:  Build the ordering $\pi$ to reflect the Acceptance-order on $X^c$.
4:
5:  Part II:
6:  Fine grid initialization:
7:  for each gridpoint $\boldsymbol{x}_i \in X$ do
8:      if $\boldsymbol{x}_i \in Q'$ then
9:          $V_i \leftarrow q_i$;
10:     else
11:         $V_i \leftarrow \infty$;
12:     end if
13: end for
14:
15: for $j = \pi(1) : \pi(J)$ do
16:     Define the fine-grid domain $\tilde{c} = c_j \bigcup N(c_j)$.
17:     Define the boundary condition as
18:             $\tilde{q}(\boldsymbol{x}_i) = q(\boldsymbol{x}_i)$ on $c_j \bigcap Q'$ and
19:             $\tilde{q}(\boldsymbol{x}_i) = V_i$ on $N(c_j)$.
20:     Perform Modified Fast Sweeping (see Remark 1) on $\tilde{c}$ using boundary conditions $\tilde{q}$.
21: end for
22:
```

2. Instead of looping through different sweeps until convergence, we use at most four sweeps and only in the directions found to be "upwind" on the coarse grid. As illustrated by Figure 2.3, the cells in $N^c(c_i)$ whose centers were accepted prior to $\boldsymbol{x}_i^c$ determine the sweep directions to be used on c_i.

3. When computing V_i during the sweeping, we do not employ the usual sweeping procedure described in section 1.4 to find the relevant quadrant. Instead, we use "sweep-directional updates"; e.g., if the current sweeping direction is from the NE, we always use the update based on the northern and eastern neighboring fine gridpoints. The advantage is that we will have processed both of them within the same sweep.

Before discussing the computational cost and accuracy consequences of these im-

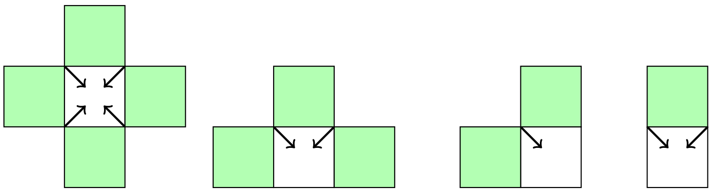
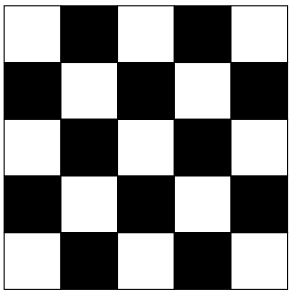


Figure 2.3: Sweeping directions on c_i chosen based on the neighboring cells accepted earlier than c_i (shown in green). Note that 2 sweeping directions are conservatively used in the case of a single accepted neighbor.

plementation choices, we illustrate the algorithm on a specific example: a "checkerboard" speed function with $F = 1$ in the white checkers and $F = 2$ in black checkers, with the exit set being a single point in the center of the domain see Figure 2.4). This example was considered in detail in [48]. The numerical results and the performance of our new methods on the related test problems are described in detail in section 2.5. As explained in Remark 1.2, we do not sweep until convergence on each cell; e.g., the sweeps for the cell # 1 in Figure 2.4 will be from northwest and southwest, while the cell #14 will be swept from northeast only.



21	18	12	19	24
15	7	3	6	16
10	2	0	1	11
14	8	4	5	13
23	17	9	20	22

Figure 2.4: Left: checkerboard speed function (with $K = 5$) with a source point in the slow checker in the center. Right: The order of cell-acceptance in Part I of FMSM, assuming $h^c = 1/K$

The resulting algorithm clearly introduces additional numerical errors – in all but the simplest examples, the FMSM's output is not the exact solution of the discretized system (1.4) on X. We identify three sources of additional errors: the fact that the coarse grid computation does not capture all cell interdependencies, and the two cell-sweeping modifications described in Remark 1. Of these, the first one is by far the most important. Focusing on the fine grid, we will say that the cell c_i *depends on* $c_j \in N^c(c_i)$ if there exists a gridpoint $\boldsymbol{x}_k \in c_i$ such that U_k directly depends on

U_l for some gridpoint $\boldsymbol{x}_l \in c_j$. In the limit, as $h \to 0$, this means that c_i depends on c_j if there is a characteristic going from c_j into c_i (i.e., at least a part of c_i's boundary shared with c_j is *inflow*). For a specific speed function F and a fixed cell-decomposition Z, a causal ordering of the cells need not exist at all. As shown in Figure 2.5, two cells may easily depend on each other. This situation arises even for problems where F is constant on each cell; see Figure 2.11. Moreover, if the cell refinement is performed uniformly, such non-causal interdependencies will be present even as the cell size $h^c \to 0$. This means that every algorithm processing each cell only once (or even a fixed number of times) will unavoidably introduce additional errors at least for some speed functions F.

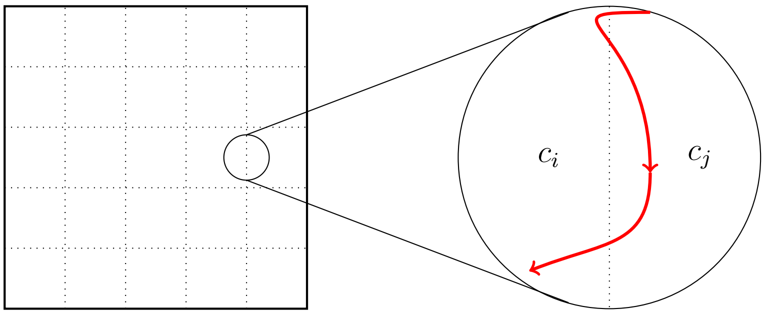


Figure 2.5: Two mutually dependent cells.

One possible way around this problem is to use the characteristic's vacillations between c_i to c_j to determine the total number of times that these cells should be alternately processed with FSM. This idea is the basis for heap-cell methods described in the next section. However, for FMSM we simply treat these "approximate cell-causality" errors as a price to pay for the higher computational efficiency. Our numerical experiments with FMSM showed that, as $h^c \to 0$, the effects due to the approximate cell-causality dominate the errors stemming from using a finite (coarse-grid determined) number of sweeps. I.e., when the cells are sufficiently small, running FSM to convergence does not decrease the additional errors significantly, but does noticeably increase the computational cost. The computational savings due to our use of "sweep-directional updates" are more modest (we simply avoid the necessity to examine/compare all neighbors of the updated node), but the numerical evidence indicates that it introduces only small additional errors and usually only near the shock lines, where ∇u is undefined. Since characteristics do not emanate from shocks, the accuracy price of this modification is even more limited if the errors are measured in L_1 norm. In section 2.3 we show that on most of X the cumulative additional errors in FMSM are typically much smaller than the discretization errors, provided h^c is sufficiently small.

The monotonicity property of the discretization ensures that the computed solution V will always satisfy $V_i \geq U_i$. The numerical evidence suggests that V becomes closer to U as h^c decreases, though this process is not always monotone.

The computational cost of Part I is relatively small as long as $J \ll M$. However, if h and M are held constant while h^c decreases, this results in $J \to M$, and the total computational cost of FMSM eventually increases. As of right now, we do not have any method for predicting the optimal h^c for each specific example. Such a criterion would be obviously useful for realistic applications of our hybrid methods, and we hope to address it in the future.

2.2 Label-correcting methods on cells

The methods presented in this section also rely on the cell-decomposition $Z = \{c_1, \dots, c_J\}$, but do not use any coarse-level grid. In what follows, we will define "cell values" to represent coarse-level information about cell dependencies. Unlike in finite volume literature, here a "cell value" is not necessarily synonymous with the average of a function over a cell.

2.2.1 A generic cell-level convergent method

To highlight the fundamental idea, we start with a simple "generic" version of a label-correcting method on cells. We maintain a list of cells to be updated, starting with the cells in Q^c. While the list is non-empty, we choose a cell to remove from it, "process" that cell (by any convergent Eikonal-solver), and use the new grid values near the cell boundary to determine which neighboring cells should be added to the list. The criterion for adding cells to the list is illustrated in Figure 2.6. All other implementation details are summarized in Algorithm 6.

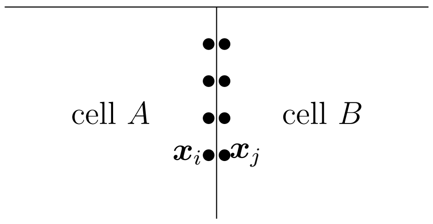


Figure 2.6: Suppose that, as a result of processing the cell A an eastern border value V_i becomes updated. If $V_i < V_j$ and $\boldsymbol{x}_j \notin Q$, the cell B will be added to L unless already there.

It is easy to prove by induction that this method terminates in a finite number of steps; in Theorem 2 we show that upon its termination $V = U$ on the entire grid X, regardless of the specific Eikonal-solver employed to process individual cells (e.g., FMM, FSM, LSM or any other method producing the exact solution to (1.4) will do). We emphasize that the fact of convergence also does not depend on the specific selection criteria for the next cell to be removed from L. However, even for a fixed cell-decomposition Z, the above choices will significantly influence the total number of list removals and the overall computational cost of the algorithm. One simple strategy is to implement L as a queue, adding cells at the bottom and always removing from the top, thus mirroring the logic of Bellman-Ford algorithm. In practice, we found the version described in the next subsection to be more efficient.

Theorem 2. *The generic cell-based label-correcting method converges to the exact*

Algorithm 6 Generic Label-Correcting on Cells pseudocode.

```
Cell Initialization:
for each cell $c_k$ do
    if $c_k \cap Q \neq \emptyset$ then
        add $c_k$ to the list $L$
    end if
end for

Fine Grid Initialization:
for each gridpoint $\boldsymbol{x}_i$ do
    if $\boldsymbol{x}_i \in Q$ then
        $V_i \leftarrow q(\boldsymbol{x}_i)$
    else
        $V_i \leftarrow \infty$
    end if
end for

Main Loop:
while $L$ is nonempty do
    Remove a cell $c$ from the list $L$.
    Define a domain $\tilde{c} = c \cup N(c)$.
    Define the boundary condition as
            $\tilde{q}(\boldsymbol{x}_i) = q(\boldsymbol{x}_i)$ on $c \cap Q$ and
            $\tilde{q}(\boldsymbol{x}_i) = V_i$ on $N(c)$.
    Process $c$ by solving the Eikonal on $\tilde{c}$ using boundary conditions $\tilde{q}$.
    for each cell $c_k \in N^c(c) \backslash L$ do
        if  $\exists\, \boldsymbol{x}_i \in (c \cap N(c_k))$ AND $\boldsymbol{x}_j \in (c_k \cap N(\boldsymbol{x}_i) \backslash Q)$ such that
                ( $V_i$ has changed OR ($\boldsymbol{x}_i \in Q$ AND $c$ is removed from $L$ for the first time) )
                AND ($V_i < V_j$) then
            Add $c_k$ to the list $L$.
        end if
    end for
end while
```

solution of (1.4).

Proof. First we describe notation and recall from section 1.3 the dependency digraph G.

- We say $\boldsymbol{x}_j$ *depends on* $\boldsymbol{x}_i$ if U_i is used to compute U_j (see discussion of formulas (1.5) and (1.6)).

- $\Gamma_{\boldsymbol{x}} = \{\text{nodes in } G \text{ on which } \boldsymbol{x} \text{ depends directly}\}$. For each node $\boldsymbol{x}$, the set $\Gamma_{\boldsymbol{x}}$ will have 0, 1, or 2 elements. If $\boldsymbol{x} \in Q$, then $\Gamma_{\boldsymbol{x}}$ is empty. If a one-sided update was used to compute $U(\boldsymbol{x})$ (see formula (1.5)), then there is only one element in $\Gamma_{\boldsymbol{x}}$.

- $G_{\boldsymbol{x}}$ denotes the subgraph of G that is reachable from the node $\boldsymbol{x}$.

- We define the cell transition distance $d(\boldsymbol{x}) = max_{\boldsymbol{x}_i \in \Gamma_{\boldsymbol{x}}} \{d(\boldsymbol{x}_i) + \text{cell\_dist}(\boldsymbol{x}, \boldsymbol{x}_i)\}$, where $\text{cell\_dist}(\boldsymbol{x}, \boldsymbol{x}_i) = 0$ if both $\boldsymbol{x}$ and $\boldsymbol{x}_i$ are in the same cell and 1 otherwise. Note that in general $d(\boldsymbol{x}) < M$, but in practice $\max d(\boldsymbol{x})$ is typically much smaller. In the continuous limit $d(\boldsymbol{x})$ is related to the number of times a characteristic that reaches $\boldsymbol{x}$ crosses cell boundaries.

- $D_s = \{\boldsymbol{x} \in G \mid d(\boldsymbol{x}) = s\}$. See Figure 2.7 for an illustration of $G_{\boldsymbol{x}}$ split into $D_0, D_1, \ldots, D_{d(\boldsymbol{x})}$.

- $\widetilde{D}_s = \{\boldsymbol{x}_j \in D_s \mid \exists \boldsymbol{x}_i \in D_{s-1}$ such that $\boldsymbol{x}_j$ depends on $\boldsymbol{x}_i\}$, i.e., the set of gridpoints in D_s that depend on a gridpoint in a neighboring cell. Note that $\widetilde{D}_0 = \emptyset$.

- $\widehat{D}_s = \{\boldsymbol{x}_i \in D_s \mid \exists \boldsymbol{x}_j \in D_{s+1}$ such that $\boldsymbol{x}_j$ depends on $\boldsymbol{x}_i\}$, i.e., the set of gridpoints in D_s that influence a gridpoint in a neighboring cell.

- $\star$ denotes any method that exactly solves the Eikonal on $\tilde{c}$ (see line 20 of algorithm 6).

Recall that by the monotonicity property of the discretization (1.4), the temporary labels V_j will always be greater than or equal to U_j throughout algorithm 6. Moreover, once V_j becomes equal to U_j, this temporary label will not change in any subsequent applications of $\star$ to the cell c containing $\boldsymbol{x}_j$. The goal is to show that $V_j = U_j$ for all $\boldsymbol{x}_j \in X$ upon the termination of Algorithm 6.

To prove convergence we will use induction on s. First, consider $s = 0$ and note

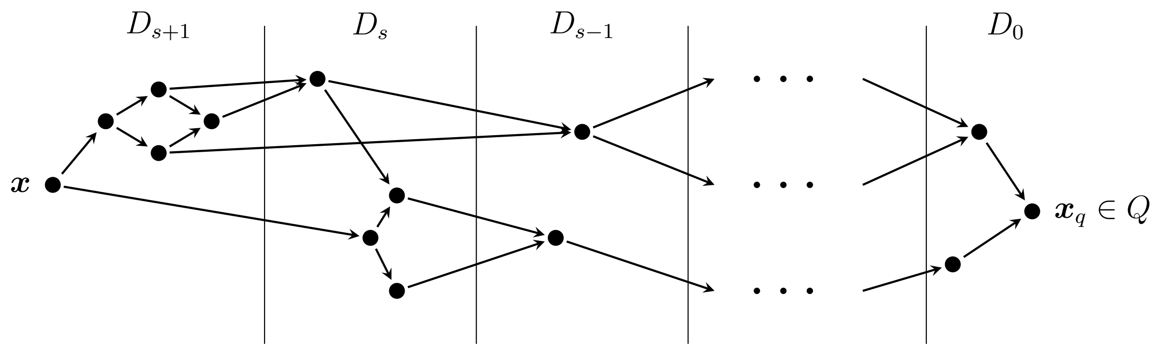


Figure 2.7: A schematic view of dependency digraph $G_{\boldsymbol{x}}$.

that every cell c containing some part of D_0 is put in L at the time of the cell initialization step of the algorithm. When c is removed from L and $\star$ is applied to it, every $\boldsymbol{x} \in D_0 \cap c$ will obtain its final value $V(\boldsymbol{x}) = U(\boldsymbol{x})$ because $G_{\boldsymbol{x}}$ contains no gridpoints in other cells by the definition of D_0.

Now suppose all $\boldsymbol{x} \in D_k$ already have $V(\boldsymbol{x}) = U(\boldsymbol{x})$ for all $k \leq s$. We claim that:

1) If a cell c contains any $\boldsymbol{x} \in D_{s+1}$ such that $V(\boldsymbol{x}) > U(\boldsymbol{x})$, then this cell is guaranteed to be in L at the point in the algorithm when the last $\boldsymbol{x}_i \in D_s \cap N(c)$ receives its final update.

2) The next time $\star$ is applied to c, $V(\boldsymbol{x})$ will become equal to $U(\boldsymbol{x})$ for all $\boldsymbol{x} \in D_{s+1} \cap c$.

To prove 1), suppose $D_{s+1} \cap c \neq \emptyset$ and note that there exist $\boldsymbol{x}_j \in \widetilde{D}_{s+1} \cap c$ and $\boldsymbol{x}_i \in \Gamma_{\boldsymbol{x}_j}$ with $\boldsymbol{x}_i \in \widehat{D_s} \cap \hat{c}$ for some neighboring cell $\hat{c}$. Indeed, if each gridpoint $\boldsymbol{x} \in D_{s+1} \cap c$ were to depend only on those in D_{s+1} (gridpoints within the same cell) and/or those in D_k for $k < s$, this would contradict $\boldsymbol{x} \in D_{s+1}$ (it is not possible for $\Gamma_{\boldsymbol{x}} \subset \cup_{k<s} D_k$; see Figure 2.7). At the time the *last such* $\boldsymbol{x}_i$ receives its final update, we will have $V_j \geq U_j > U_i = V_i$ since $\boldsymbol{x}_i \in \Gamma_{\boldsymbol{x}_j}$. Thus, c is added to L (if not already there) as a result of the add criterion in Algorithm 6.

To prove 2), we simply note that all nodes in $(G_{\boldsymbol{x}} \backslash c) \subset (\bigcup_{k=0}^{s} D_k)$ will already have correct values at this point.

$\square$

Remark 3. We note that the same ideas are certainly applicable to finding shortest paths on graphs. Algorithm 1 can be similarly modified using a collection of non-overlapping subgraphs instead of cells, but so far we were unable to find any description of this approach in the literature.

2.2.2 Heap-Cell Method (HCM)

HCM is a particular label-correcting method on cells that aims to decouple the cells through the use of cell values. Unlike FMSM, the dependency among the cells is discovered dynamically; like FMSM and unlike the generic label-correcting method on cells, HCM is designed to mimic FMM on the cell level; like the generic method and unlike FMSM the previously processed cells may re-enter L.

As described above in the proof of Theorem 2, when system (1.4) 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 list L of cells-to-be-processed is again initially populated with Q^c. The entire grid is initialized only once, in the same way as it is for LSM[1]. At each iteration of the main algorithm, a cell c is chosen from L and equation (1.4) 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

[1]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"; the active flags of all other gridpoints are set to "inactive".

a min-heap data structure. Again, 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 2.3 and 3.4 shows that HCM is very efficient for a wide range of M and J values.

Algorithm 7 Heap-Cell Method main loop.

```
 1: Initialize cell-values and grid-values
 2: Add all $c \in Q^c$ cells to $L$
 3: while $L$ nonempty do
 4:     Remove the cell $c$ with the smallest cell value from $L$
 5:     $V^c(c) \leftarrow +\infty$
 6:     Perform modified LSM on $c$ until convergence and populate
        the list $DN$ of currently downwind neighboring cells        //see Algorithm 8
 7:     for each neighbor $c_k \in DN$ do
 8:         Update $V^c(c_k)$, the cell value of $c_k$
 9:         Add $c_k$ onto $L$ if not already there
10:         Update the preferred sweeping directions of $c_k$
11:     end for
12: end while
```

We say that a cell B is *currently downwind* from a cell A, if **(1)** A was the last processed cell and **(2)** there exist neighboring border gridpoints $\boldsymbol{x}_i \in A$ and $\boldsymbol{x}_j \in B$ such that the value of V_i has changed the last time A was processed and **(3)** $V_i < V_j$. 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.

As noted earlier, 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 3.1 shows that, for many generic problems and large h^c, neighboring cells A and B are likely to be interdependent, resulting in multiple alternating re-processings 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 they depend on, the average number of times each cell is processed becomes close to one. As confirmed by the numerical evidence in section 2.3, weakly causal domain decompositions are very useful in decreasing the computational costs of serial numerical methods.

Processing cells by using Fast Sweeping Methods: Sweeping using LSM [3] 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 8.

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 2D 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 A tags B as

Algorithm 8 Modified LSM update at a gridpoint $\boldsymbol{x}_i$.

```
1:  if $\boldsymbol{x}_i$ is inactive then
2:      Do nothing
3:  else
4:      Set $\boldsymbol{x}_i$ inactive
5:      Compute a possible new value $\widetilde{V}$ for $\boldsymbol{x}_i$ by solving equation (1.4)
6:      if $\widetilde{V} < V(\boldsymbol{x}_i)$ then
7:          $V(\boldsymbol{x}_i) \leftarrow \widetilde{V}$
8:          for each $\boldsymbol{x}_j \in N(\boldsymbol{x}_i) \backslash Q'$ do
9:              if $V(\boldsymbol{x}_j) > V(\boldsymbol{x}_i)$ then
10:                 Set $\boldsymbol{x}_j$ active
11:                 if $\boldsymbol{x}_j$ is in a different cell from $\boldsymbol{x}_i$ then
12:                     Tag that cell as part of the list $DN$ of currently downwind
                        cells
13:                 end if
14:             end if
15:         end for
16:     end if
17: end if
```

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 2.6, then both A-relevant sweep-direction flags in B (i.e., both NW and SW) will be set to `TRUE`.

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 [16] 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 section 2.3 and sections 3.4.1, 3.4.2 of the next chapter for experimental evidence.

In FMSM of section 2.1, the cell values were defined by running FMM on the coarse grid. That approach is not very suitable here, since each cell c_k might enter the list more than once and it is important to re-evaluate V_k^c each time this happens. Instead, we define and update V_k^c using the boundary values in the adjacent cells, and line 8 of Algorithm 7 is executed as follows:

Let $\boldsymbol{b}_k$ be a unit vector pointing from the center of c in the direction of c_k's center and suppose that $\boldsymbol{x}_i$ has the largest current value among the gridpoints inside c but adjacent to c_k; i.e., $\boldsymbol{x}_i = \operatorname*{argmax}_{\boldsymbol{x}_j \in (c \cap N(c_k))} V_j$. Define $\boldsymbol{y}_i = \boldsymbol{x}_i + \frac{h+h^c}{2}\boldsymbol{b}_k$. Then

$$\begin{aligned} \widetilde{V}_k^c &\leftarrow V_i + \frac{(h+h^c)/2}{F(\boldsymbol{y}_i)}; \\ V_k^c &\leftarrow \min\left(V_k^c, \widetilde{V}_k^c\right). \end{aligned} \tag{2.1}$$

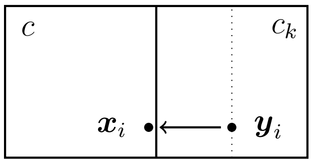


Figure 2.8: An illustration corresponding to Equation (2.2) (the estimate for a cell value) with $\boldsymbol{b}_k = (1, 0)$.

The concept of a cell value is useful even if L is implemented as a queue and the cells are always removed from the top. Indeed, V_k^c can still be used to decide whether c_k should be added at the top or at the bottom of L. This is the SLF/LLL strategy previously used to solve the Eikonal PDE on the grid-level (i.e., without any cells) by Polymenakos, Bertsekas, and Tsitsiklis [51]. We have also implemented this strategy and found it to be fairly good, but on average less efficient than the HCM described above. (The performance comparison is omitted to save space.) The intuitive reason is that the SLF/LLL is based on mimicking the logic of Dijkstra's method, but without the expensive heap-sort data structures. However, when $J \ll M$, the cost of maintaining the heap is much smaller than the cost of occasionally removing/processing less influential cells from L.

The performance and accuracy data in section 2.3 shows that, for sufficiently small h and h^c, HCM often outperforms both FMM and FSM on a variety of examples, including those with piecewise continuous speed function F. This is largely due to the fact that the average number of times a cell enters the heap tends to 1 as $h^c \to 0$. Further extensions, including an improved cell value heuristic, are introduced in Chapter 3; see section 3.4.

2.2.3 Fast Heap-Cell Method (FHCM)

Here we develop an accelerated version of HCM by using the following modifications:

1. Each newly removed cell is processed using at most four iterations – i.e., it is only swept once in each of the preferred directions instead of continuing to iterate until convergence.
2. Directional flags in all cells containing parts of Q are initialized to `TRUE`.
3. To further speed up the process, we use a "Monotonicity Check" on cell-boundary data to further restrict the preferred sweeping directions. For concreteness, assume that A and B are related as in Figure 2.6. If the grid values in $N(B) \cap A$ are monotone non-decreasing from north to south, we set B's NW preferred direction flag to `TRUE`; if those grid values are monotone non-increasing we flag SW; otherwise we flag both NW and SW. In contrast, both HCM and FMSM always use two sweeps in this situation; see Figure 2.3. We note that the set $c \cap N(B)$ already had to be examined to compute an update to $V^c(B)$ and the above Monotonicity Check can be performed simultaneously.

The resulting Fast Heap-Cell Method (FHCM) is significantly faster than HCM, but at the cost of introducing additional errors (see section 2.3).

The Monotonicity Checks result in a considerable increase in performance since, for small enough h^c, most cell boundaries become monotone. However, generalizing this procedure to higher dimensional cells is less straightforward. For this reason we decided against using Monotonicity Checks in our implementation of HCM. FHCM is summarized in Algorithm 9.

Algorithm 9 Fast Heap-Cell Method pseudocode.

```
1: Cell Initialization:
2: if cell $c_k \ni \boldsymbol{x}$ for $\boldsymbol{x} \in Q^f$ then
3:     Add $c_k$ to the list $L$;
4:     Tag all four sweeping directions of $c_k$ as *true*;
5:     Assign a cell value $V_k^{cell} := 0$;
6: else
7:     Assign a cell value $V_k^{cell} := \infty$;
8: end if
9: Fine Grid Initialization:
10: if $\boldsymbol{x}_i^f \in Q^f$ then
11:     $V_i^f := q_i^f$;
12: else
13:     $V_i^f := \infty$;
14: end if
15:
16: while $L$ is nonempty do
17:     Remove cell at the top of $L$;
18:     Perform Non-Directional Fast Sweeping within the cell according to its directions marked *true*, then set all directions to *false* and:
19:     for Each cell border N,S,E,W do
20:         if the value of a gridpoint $\boldsymbol{x}_i^f$ along a border changes and $V_i^f < V_j^f$ for $\boldsymbol{x}_j^f$ a neighboring gridpoint across the border then
21:             Add the cell $c_k$ containing $x_j^f$ onto $L$ if not already there.
22:             Update the planned sweeping directions for $c_k$ based on the location of the cell containing $\boldsymbol{x}_i^f$ (more about this later).
23:         end if
24:         Compute a value $v$ for the neighbor cell $c_k$ (more about this later)
25:         if $v < V_k^{cell}$ then
26:             $(V_k^{cell}) \leftarrow v$
27:         end if
28:     end for
29: end while
```

As an illustration, we consider another 5×5 checkerboard example (this time with a fast checker in the center) and show the contents of the heap in Figure 2.9.

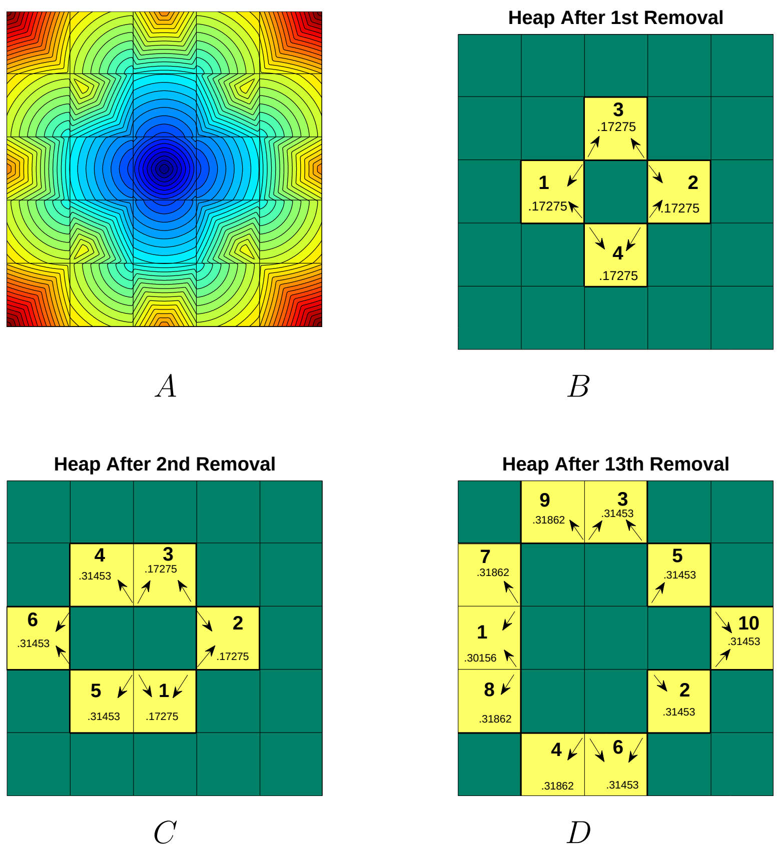


Figure 2.9: FHCM on a 5×5 checkerboard example. The level sets of the solution are shown in subfigure A. The state of the cell-heap, current cell values and tagged preferred sweeping directions are shown after 1, 2, and 13 cell removals in subfigures B, C, and D.

Here we take the cells coinciding with checkers; finer cell-decompositions are numerically tested in section 2.5. The arrows indicate flagged sweeping directions for each cell, and the smaller font is used to show the current cell values. Similarly to Dijkstra's method and FMM, the heap data structure is implemented as an array; the bold numbers represent each cell's index in this array. In the beginning the central cell is the only one in L; once it is removed, it adds to L all four of its neighbors, all of them with the same cell value. Once the first of these (to the west of the center) is removed, it adds three more neighbors[2] (but not the central cell since there are no characteristics passing through the former into the latter). This is similar to the execution path of FMSM, however, with heap-cell methods the cells may generally enter the heap more than once. Thus, additional errors introduced by FHCM are usually smaller than those in FMSM.

Remark 4. To conclude the discussion of our heap-cell methods we briefly describe a recent algorithm with many similar features, but different goals and implementation details. The "Raster scan algorithm on a multi-chart geometry image" was introduced in [67] for geodesic distance computations on parametric surfaces. Such surfaces are frequently represented by an atlas of overlapping charts, where each chart has its own parametric representation and grid resolution (depending on the detail level of the underlying surface). The computational subdomains corresponding to charts are typically large and the "raster scan algorithm" (similar to the traditional FSM with a fixed ordering of sweep directions) is used to parallelize the computations within each chart. The heuristically defined chart values are employed to decide which chart will be raster-scanned next.

In [67] the emphasis is on providing the most efficient implementation of raster scans on each chart for a SIMD/GPU parallel architecture. The use of several large, parametrization/resolution-defined charts typically results in complicated chart interdependencies since most chart boundaries are generally both inflow and outflow. Moreover, if this method is applied to any Eikonal problems beyond the geodesic distance computations, the monotonicity of characteristic directions will generally not hold and a high number of sweeps will be needed on each chart. In contrast, our focus is on reducing the cell interdependencies and on the most efficient cell ordering: when h^c is sufficiently small, most cell boundaries are either completely inflow or

[2]We note that the cell indexed 4 after the first removal is indexed 2 immediately after the second. This is the consequence of the performing *remove_the_smallest* using the *down_heap* procedure in the standard implementation of the heap; see [57].

outflow, providing a causal relationship among the cells. Relatively small cell sizes also ensure that F is approximately constant, the characteristics are approximately straight lines, and only a small number of sweeps is needed on each cell. Furthermore, the cell orderings are also useful to accelerate the convergence within each cell by altering the sweep-ordering based on the location of upwinding cells (as in FMSM and HCM) or fine gridpoint boundary data (as in FHCM). The hybrid methods introduced here show that causality-respecting domain decompositions can accelerate even serial algorithms on single processor machines. Finally, as explained in the next chapter, the operations that are parallelized in PMM are in a sense opposite those that are parallelized in the parallel HCM: in PMM many gridpoints on one chart receive updates simultaneously in a massively parallel way, while in pHCM multiple cells are processed simultaneously.

2.3 Numerical Experiments

All examples were computed on a unit square $[0,1]\times[0,1]$ domain with zero boundary conditions $q=0$ on the exit set Q (defined separately in each case). In each example that follows we have fixed the grid size h, and only the cell size h^c is varied. Since analytic formulas for viscosity solutions are typically unavailable, we have used the Fast Marching Method on a much finer grid (of size $h/4$) to obtain the "ground truth" used to evaluate the errors in all the other methods.

Suppose e_i is the absolute value of the error-due-to-discretization at gridpoint $\boldsymbol{x}_i$ (i.e., the error produced by FSM or FMM when directly executed on the fine grid), and suppose E_i is the absolute value of the error committed by one of the new hybrid methods at the same $\boldsymbol{x}_i$. Define the set $X_+ = \{\boldsymbol{x}_i \in X \mid e_i \neq 0\}$ and let $M_+ = |X_+|$ be the number of elements in it. (We verified that $\boldsymbol{x}_i \notin X_+ \Rightarrow E_i = 0$

in all computational experiments.) To analyze the "additional errors" introduced by FMSM and FHCM, we report

• the *Maximum Error Ratio* defined as $\mathcal{R} = \max_i(E_i/e_i)$, where the maximum is taken over $\boldsymbol{x}_i \in X_+$;

• the *Average Error Ratio* defined as $\rho = \frac{\sum(E_i/e_i)}{M_+}$, where the sum is taken over $\boldsymbol{x}_i \in X_+$;

• the *Ratio of Maximum Errors* defined as $R = \frac{\max_i(E_i)}{\max_i(e_i)}$.

R is relevant since on parts of the domain where e_i's are very small, additional errors might result in large $\mathcal{R}$ even if E_i's are quite small compared to the L_∞ norm of discretization errors. In the ideal scenario, with no additional errors, $\mathcal{R} = \rho = R = 1$.

For the Heap-Cell algorithms we also report

• *AvHR*, the average number of heap removals per cell,

• *AvS*, the average number of sweeps per cell, and

• *Mon %*, the percentage of times that the "cell-boundary monotonicity" check was successful.

Finally, we report the number of sweeps needed in FSM and LSM for each problem.

Performance analysis of competing numerical methods is an obviously delicate undertaking since the implementation details as well as the choice of test problems might affect the outcome. We have made every effort to select representative examples highlighting advantages and disadvantages of all approaches. All tests in this section were performed on an AMD Turion 2GHz dual-core processor with 3GB RAM. Only one core was used to perform all tests. Our C++ implementations were carefully checked for the efficiency of data structures and algorithms, but we did not conduct any additional performance tuning or Assembly-level optimizations. Our code was compiled using the `g++` compiler version 3.4.2 with compiler options `-O0 -finline`. We have also performed all tests with full compiler optimizations (i.e., with `-O3`) and found the results to be qualitatively similar; we opted to report the performance data for the unoptimized version to make the comparison as compiler-independent as possible. For each method, all memory allocations (for grids and heap data structures) were not timed; the reported CPU times include the time needed to initialize the relevant data structures and run the corresponding algorithm. The speed function $F(\boldsymbol{x})$ was computed by a separate function call whenever needed, rather than precomputed and stored for every gridpoint during initialization. All CPU-times are reported in seconds for Fast Marching (FMM), the standard Fast Sweeping (FSM), Locking Sweeping (LSM), and the three new hybrid methods (HCM, FHCM, and FMSM).

2.4 Comb Mazes

The following examples model optimal motion through a maze with slowly permeable barriers. Speed function $F(x, y)$ is defined by a "comb maze": $F = 1$ outside and 0.01 inside the barriers; see Figure 2.10. The exit set consists of the origin: $Q = \{(0, 0)\}$. The computational cost of sweeping methods is roughly proportional to the number of barriers, while FMM is only minimally influenced by this. The same good property is inherited by the hybrid methods introduced in this chapter. The first example with 4 barriers uses barrier walls aligned with cell boundaries and all hybrid methods easily outperform the fastest of the previous methods (LSM); see Table 2.1.

We note that even the slowest of the HCM trials outperforms FMM, FSM, and LSM on this example. Despite the special alignment of cell boundaries, this example is typical in the following ways:

1. In both Heap-Cell algorithms, as the number of cells increases, the average

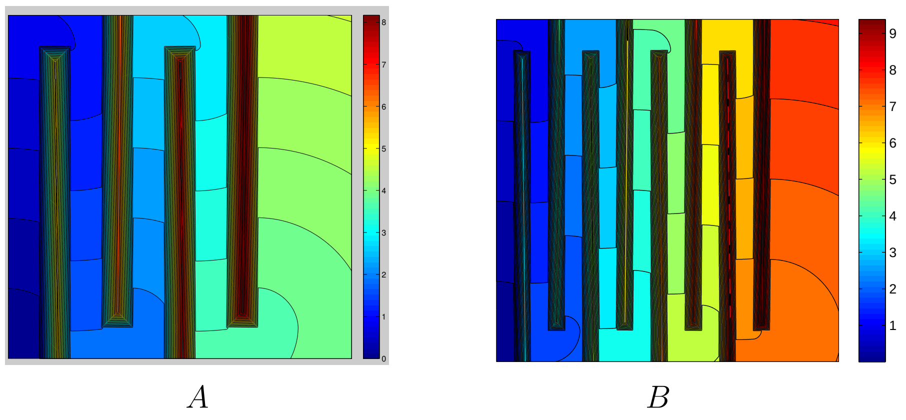


Figure 2.10: Min time to the point $(0, 0)$ on comb maze domains: 4 barriers (A), and 8 barriers (B).

Table 2.1: Performance/convergence results for a 4 wall comb maze example.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	5.9449e-002	1.4210e-002	2.45	6.41	2.05	12

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	1.08				1.151	3.971	
HCM 44 × 44 cells	1.10				1.078	3.724	
HCM 88 × 88 cells	1.08				1.040	3.593	
HCM 176 × 176 cells	1.10				1.020	3.518	
HCM 352 × 352 cells	1.24				1.015	3.496	
HCM 704 × 704 cells	1.63				1.008	3.468	
FHCM 22 × 22 cells	0.79	1.0460	1.0000	1.0000	1.151	1.618	85.5
FHCM 44 × 44 cells	0.74	1.0191	1.0000	1.0000	1.078	1.310	92.6
FHCM 88 × 88 cells	0.74	1.0085	1.0000	1.0000	1.040	1.156	96.2
FHCM 176 × 176 cells	0.78	1.0073	1.0000	1.0000	1.020	1.080	98.4
FHCM 352 × 352 cells	0.95	1.0002	1.0000	1.0000	1.015	1.049	99.3
FHCM 704 × 704 cells	1.41	1.0000	1.0000	1.0000	1.008	1.022	100.0
FMSM 22 × 22 cells	0.58	1.1659	1.0000	1.0000		1.436	
FMSM 44 × 44 cells	0.54	1.0706	1.0000	1.0018		1.218	
FMSM 88 × 88 cells	0.53	1.0821	1.0000	1.0018		1.110	
FMSM 176 × 176 cells	0.57	1.0468	1.0000	1.0008		1.055	
FMSM 352 × 352 cells	0.71	1.0378	1.0000	1.0004		1.028	
FMSM 704 × 704 cells	1.24	1.0064	1.0000	1.0001		1.014	

number of heap removals per cell decreases.

2. In FHCM the average number of sweeps per cell decreases to 1 as h^c decreases.

3. In FHCM the percentage of monotonicity check successes increases as h^c decreases.

4. For timing performance in both HCM and FHCM, the optimal choice of h^c is somewhere in the middle of the tested range.

The reason for #2 is that, as the number of cells J increases, most cells will pass the Monotonicity Check. When the monotonicity percentage is high and each cell has on average 2 "upwinding" neighboring cells, each cell on the heap will have one sweeping direction tagged. This observation combined with #1 explains #2.

Combining #1 and #2 and the fact that the length of the heap also increases with J there is a complexity trade-off that explains #4. As $J \to M$ the complexity of both Heap-Cell algorithms is similar to that of Fast Marching. As $J \to 1$, the complexity of HCM is similar to that of Locking Sweeping.

In the second example we use 8 barriers and the boundaries of the cells are **not aligned** with the discontinuities of the speed function. This example was chosen specifically because it is difficult for our new hybrid methods when using the same cell-decompositions as in the previous example. The performance data is summarized in Table 2.2.

Notice that since the edges of cells do not coincide with the edges of barriers, the performance of the hybrid methods is not as good as in the previous 4-barrier case, where the edges do coincide. In this example the cells that contain a discontinuity of the speed function may not receive an accurate cell value (for either the Heap-Cell algorithms or FMSM) and may often have poor choices of planned sweeping directions (for FHCM & FMSM). For FHCM, since the error is small in most trials, this effect appears to be rectified at the expense of the same cells being added to the heap many times. For FMSM, since each cell is processed only once, large error remains. The non-monotonic behavior of $\mathcal{R}$ in FMSM and FHCM appears to be due to changes in positions of cell centers relative to barrier edges as h^c decreases.

Table 2.2: Performance/convergence results for an 8 wall comb maze example.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	6.5644e-002	1.6865e-002	2.50	11.1	3.20	20

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	2.13				2.795	9.293	
HCM 44 × 44 cells	7.68				8.738	28.046	
HCM 88 × 88 cells	6.68				6.798	22.804	
HCM 176 × 176 cells	5.86				5.655	18.872	
HCM 352 × 352 cells	2.95				2.456	8.314	
HCM 704 × 704 cells	1.74				1.037	3.587	
FHCM 22 × 22 cells	1.75	1.4247	1.0000	1.0000	2.946	4.087	84.7
FHCM 44 × 44 cells	5.86	1.4250	1.0000	1.0000	8.991	10.209	94.0
FHCM 88 × 88 cells	4.54	1.3083	1.0000	1.0000	6.976	7.329	98.1
FHCM 176 × 176 cells	3.96	1.2633	1.0000	1.0000	5.754	5.910	99.1
FHCM 352 × 352 cells	2.13	1.8922	1.0000	1.0000	2.468	2.549	99.1
FHCM 704 × 704 cells	1.48	1.5700	1.0000	1.0000	1.037	1.066	100.0
FMSM 22 × 22 cells	0.68	604.49	6.6555	21.036		1.783	
FMSM 44 × 44 cells	0.59	228.29	3.1529	19.442		1.385	
FMSM 88 × 88 cells	0.56	313.01	2.7666	6.4608		1.195	
FMSM 176 × 176 cells	0.58	381.98	1.7374	5.5944		1.097	
FMSM 352 × 352 cells	0.74	45.397	1.1718	2.0506		1.049	
FMSM 704 × 704 cells	1.26	23.303	1.1738	1.3536		1.024	

In the next chapter we present these comb maze examples again but computed using HCM/FHCM with a different cell value that seems to overcome the limitations of discontinuities in the speed function that are misaligned with the cell boundaries. Briefly, the idea for the new cell value is to rank the cells by whichever has the most upwind inflow, instead of trying to approximate the value of the center of the cell as we do here.

2.5 Checkerboards

We return to the checkerboard example already described in section 2.1. For both the 11×11 and 41×41 checkerboard speed functions the center checker is slow. The speed is 1 in the slow checkers and 2 in the fast checkers. The exit set is the single point $Q = \{(0.5,\, 0.5)\}$.

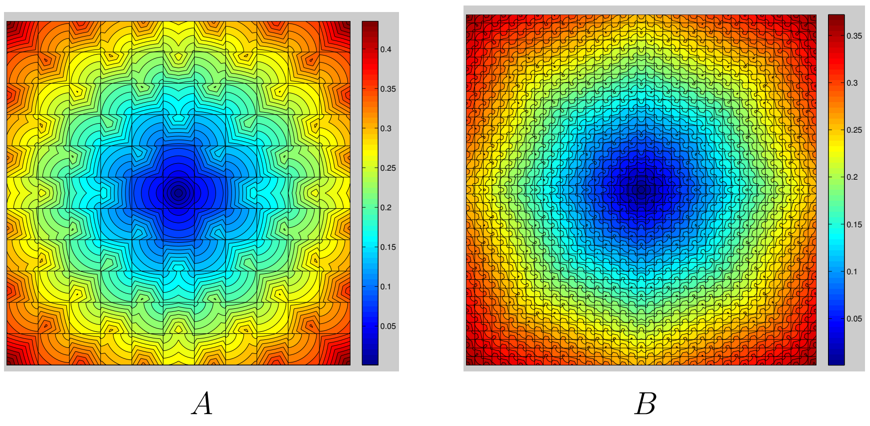


Figure 2.11: Min time to the center on checkerboard domains: 11×11 checkers (A), and 41×41 checkers (B).

Remark 5. Such checkerboard examples arise naturally in the context of front propagation through composite media, consisting of a periodic mix of isotropic constituent materials with different speed function F. The idea of *homogenization* is to derive a homogeneous but anisotropic speed function $\overline{F}(\boldsymbol{n})$, describing the large-scale properties of the composite material. After $\overline{F}(\boldsymbol{n})$ is computed, the boundary value problems can be solved on a coarser grid. An efficient method for this homogenization was introduced in [48], using FMM on the fine scale grid since the characteristics are highly oscillatory and the original implementation of sweeping was inefficient. The same test problems were later attacked in [45] using a version of FSM with gridpoint locking (see Remark **??**). The results in Table 2.4 show that even the Locking-Sweeping Method becomes significantly less efficient than FMM with the increase in the number of checkers.

Table 2.3: Performance/convergence results for 11×11 checkerboard example.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	3.2639e-003	1.7738e-003	3.44	12.3	2.28	16

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	1.84				1.397	5.254	
HCM 44 × 44 cells	1.73				1.209	4.613	
HCM 88 × 88 cells	1.69				1.083	4.117	
HCM 176 × 176 cells	1.72				1.029	3.864	
HCM 352 × 352 cells	1.87				1.009	3.768	
HCM 704 × 704 cells	2.51				1.003	3.746	
FHCM 22 × 22 cells	1.17	1.0122	1.0000	1.0000	1.399	1.779	86.3
FHCM 44 × 44 cells	1.11	1.0208	1.0000	1.0000	1.227	1.535	90.6
FHCM 88 × 88 cells	1.08	1.0111	1.0000	1.0000	1.091	1.247	95.1
FHCM 176 × 176 cells	1.14	1.0050	1.0000	1.0000	1.029	1.103	97.8
FHCM 352 × 352 cells	1.33	1.0006	1.0000	1.0000	1.009	1.043	99.4
FHCM 704 × 704 cells	2.08	1.0000	1.0000	1.0000	1.003	1.020	100.0
FMSM 22 × 22 cells	0.87	40.312	1.5725	13.016		1.269	
FMSM 44 × 44 cells	0.91	18.167	1.0875	7.4581		1.334	
FMSM 88 × 88 cells	0.89	7.6692	1.0113	3.1400		1.222	
FMSM 176 × 176 cells	0.91	5.4947	1.0025	2.4813		1.127	
FMSM 352 × 352 cells	1.07	2.4557	1.0004	1.3888		1.067	
FMSM 704 × 704 cells	1.84	1.5267	1.0000	1.0032		1.035	

Table 2.4: Performance/convergence results for 41×41 checkerboard example.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1312 × 1312	1.2452e-002	6.6827e-003	4.13	58.9	11.7	45

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 41 × 41 cells	4.18				3.261	11.926	
HCM 82 × 82 cells	3.05				1.571	5.939	
HCM 164 × 164 cells	2.84				1.314	4.831	
HCM 328 × 328 cells	2.81				1.080	3.972	
HCM 656 × 656 cells	3.36				1.026	3.768	
FHCM 41 × 41 cells	2.83	1.7506	1.0041	1.7123	3.261	4.600	75.5
FHCM 82 × 82 cells	2.09	1.0299	1.0006	1.0128	1.584	2.147	78.8
FHCM 164 × 164 cells	1.95	1.0103	1.0001	1.0000	1.321	1.670	90.4
FHCM 328 × 328 cells	2.01	1.0173	1.0000	1.0000	1.080	1.236	96.9
FHCM 656 × 656 cells	2.79	1.0075	1.0000	1.0000	1.026	1.106	100.0
FMSM 41 × 41 cells	1.46	12.398	3.4110	3.3991		1.164	
FMSM 82 × 82 cells	1.54	10.551	1.0975	1.7662		1.211	
FMSM 164 × 164 cells	1.70	4.7036	1.0142	1.7123		1.281	
FMSM 328 × 328 cells	1.88	2.0192	1.0020	1.7123		1.242	
FMSM 656 × 656 cells	2.65	1.7506	1.0004	1.7123		1.147	

In both examples the cell sizes were chosen to align with the edges of the checkers (i.e., the discontinuities of the speed function). On the 11×11 checkerboard, almost all of the HCM trials outperforms FMM and LSM, and most of the FHCM trials are more than twice as fast as LSM and three times faster than FMM while the additional errors are negligible; see Table 2.3.

The 41×41 example is much more difficult for the sweeping algorithms because the number of times the characteristics changes direction increases with the number of checkers. We note that the performance of FMM is only moderately worse here (mostly due to a larger length of level curves and the resulting growth of the "Considered List"). Again, almost all hybrid methods outperform all other methods. The difference is less striking than in the 11×11 example when compared with FMM, but FHCM and FMSM are 4 to 6 times faster than LSM; see Table 2.4.

2.6 Continuous speed functions with a point source

Suppose the speed function is $F \equiv 1$ and the exit set consists of a single point $Q = \{(0.5, 0.5)\}$. In this case the viscosity solution is simply the distance to the center of the unit square. We also note that the causal ordering of cells is clearly available here; as a result, FHCM and FMSM do not introduce any additional errors. The performance data is summarized in Table 2.5. For constant speed functions LSM performs significantly better than FMM on fine meshes (such as this one). The reason why FMSM and FHCM are faster than LSM in some trials is that LSM checks all parts of the domain in each sweep, including non-downwinding or already-computed parts. Additionally LSM must perform a final sweep to check that all gridpoints are locked. All of the hybrid algorithms slow down monotonically as J increases because of the cost of sorting the heap.

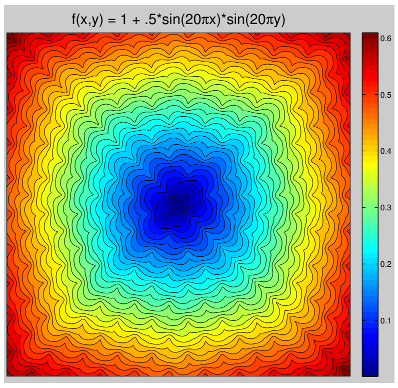
Table 2.5: Performance/convergence results for constant speed function.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	1.0956e-003	6.8382e-004	2.72	2.07	0.83	5

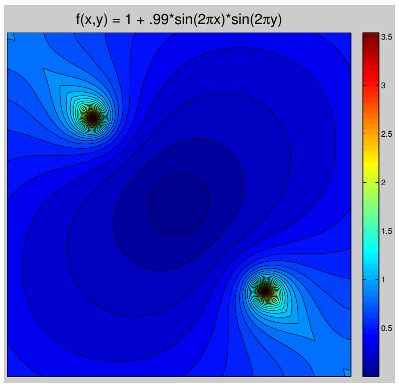
METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	1.05				1.000	3.692	
HCM 44 × 44 cells	1.12				1.000	3.718	
HCM 88 × 88 cells	1.10				1.000	3.733	
HCM 176 × 176 cells	1.14				1.000	3.742	
HCM 352 × 352 cells	1.29				1.000	3.746	
HCM 704 × 704 cells	1.76				1.000	3.748	
FHCM 22 × 22 cells	0.66	1.0000	1.0000	1.0000	1.000	1.025	100.0
FHCM 44 × 44 cells	0.67	1.0000	1.0000	1.0000	1.000	1.006	100.0
FHCM 88 × 88 cells	0.69	1.0000	1.0000	1.0000	1.000	1.002	100.0
FHCM 176 × 176 cells	0.75	1.0000	1.0000	1.0000	1.000	1.000	100.0
FHCM 352 × 352 cells	0.92	1.0000	1.0000	1.0000	1.000	1.000	100.0
FHCM 704 × 704 cells	1.47	1.0000	1.0000	1.0000	1.000	1.000	100.0
FMSM 22 × 22 cells	0.47	1.0000	1.0000	1.0000		1.103	
FMSM 44 × 44 cells	0.47	1.0000	1.0000	1.0000		1.049	
FMSM 88 × 88 cells	0.49	1.0000	1.0000	1.0000		1.024	
FMSM 176 × 176 cells	0.53	1.0000	1.0000	1.0000		1.012	
FMSM 352 × 352 cells	0.67	1.0000	1.0000	1.0000		1.006	
FMSM 704 × 704 cells	1.23	1.0000	1.0000	1.0000		1.003	

Next we consider examples of min-time to the center under two different oscillatory continuous speed functions. For $F(x,y) = 1 + \frac{1}{2}\sin(20\pi x)\sin(20\pi y)$ the level sets of the value function are shown in Figure 2.12A and the performance data is summarized in Table 2.6. For $F(x,y) = 1 + 0.99\sin(2\pi x)\sin(2\pi y)$ the level sets of the value function are shown in Figure 2.12B and the performance data is summarized in Table 2.7.

Note that HCM outperforms Fast Marching on all trials, and outperforms the sweeping methods significantly on the first example (Table 2.6) despite the fact that no special selection of cell boundaries was made. Small changes in the frequency of the speed function did not significantly alter the performance of the hybrid algorithms. In the second example (Table 2.7) most HCM trials were again faster than LSM and FMM. Note that for some cell sizes, both FMSM and FHCM have $R \ll \mathcal{R} =$



A



B

Figure 2.12: Min time to the center under sinusoidal speed functions.

Table 2.6: Performance/convergence results for $F(x, y) = 1 + \frac{1}{2}\sin(20\pi x)\sin(20\pi y)$.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	4.7569e-003	1.9724e-003	3.74	23.7	6.39	24

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	3.61				1.913	10.785	
HCM 44 × 44 cells	2.97				1.446	6.811	
HCM 88 × 88 cells	2.60				1.245	5.201	
HCM 176 × 176 cells	2.40				1.117	4.350	
HCM 352 × 352 cells	2.40				1.047	3.945	
HCM 704 × 704 cells	2.92				1.016	3.788	
FHCM 22 × 22 cells	2.72	5.6062	1.1358	2.0960	4.413	5.310	67.3
FHCM 44 × 44 cells	1.82	3.1094	1.1480	1.0000	1.555	2.132	78.7
FHCM 88 × 88 cells	1.61	1.4025	1.0122	1.0000	1.277	1.575	88.2
FHCM 176 × 176 cells	1.53	1.0560	1.0022	1.0000	1.125	1.262	94.5
FHCM 352 × 352 cells	1.65	1.0226	1.0004	1.0000	1.048	1.106	98.1
FHCM 704 × 704 cells	2.40	1.0037	1.0001	1.0000	1.016	1.035	100.0
FMSM 22 × 22 cells	1.14	10.497	2.4811	2.9653		1.262	
FMSM 44 × 44 cells	1.10	6.0892	1.3657	2.2889		1.200	
FMSM 88 × 88 cells	1.16	4.6801	1.0515	1.9504		1.213	
FMSM 176 × 176 cells	1.18	3.4828	1.0074	1.3705		1.126	
FMSM 352 × 352 cells	1.34	1.5987	1.0007	1.0000		1.067	
FMSM 704 × 704 cells	2.14	1.1262	1.0001	1.0000		1.035	

$\max_j(E_j/e_j)$. Whenever R is close to 1, the rate of convergence of hybrid methods (based on L_∞ errors) is the same as that of FMM and FSM.

Table 2.7: Performance/convergence results for $F(x, y) = 1 + 0.99 \sin(2\pi x) \sin(2\pi y)$.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	2.1793e-002	9.8506e-004	3.69	12.7	2.73	13

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	2.29				1.165	4.651	
HCM 44 × 44 cells	2.15				1.070	4.132	
HCM 88 × 88 cells	2.11				1.034	3.920	
HCM 176 × 176 cells	2.13				1.015	3.811	
HCM 352 × 352 cells	2.26				1.008	3.763	
HCM 704 × 704 cells	2.80				1.002	3.741	
FHCM 22 × 22 cells	1.37	60.848	1.0020	1.0014	1.174	1.409	92.7
FHCM 44 × 44 cells	1.28	4.5786	1.0002	1.0001	1.078	1.185	96.1
FHCM 88 × 88 cells	1.28	1.0224	1.0000	1.0000	1.039	1.086	98.2
FHCM 176 × 176 cells	1.35	1.0019	1.0000	1.0000	1.017	1.039	99.3
FHCM 352 × 352 cells	1.55	1.0003	1.0000	1.0000	1.008	1.018	99.7
FHCM 704 × 704 cells	2.27	1.0001	1.0000	1.0000	1.002	1.006	100.0
FMSM 22 × 22 cells	1.13	1362.4	1.0270	1.0053		1.231	
FMSM 44 × 44 cells	1.06	174.62	1.0054	1.0053		1.116	
FMSM 88 × 88 cells	1.05	38.545	1.0021	1.0046		1.057	
FMSM 176 × 176 cells	1.09	7.1581	1.0006	1.0046		1.029	
FMSM 352 × 352 cells	1.28	1.1687	1.0001	1.0028		1.014	
FMSM 704 × 704 cells	2.08	1.0724	1.0000	1.0000		1.007	

2.7 Performance on coarser grids

Our hybrid methods exploit the fact that there exists h^c small enough so that most cell-boundaries will be either fully inflow or fully outflow and most pairs of cells will not be mutually dependent. But if the original grid X is sufficiently coarse, this may not be possible to achieve since we also need $h^c \geq 2h$ (otherwise FMM is clearly more efficient). In this subsection we return to some of the previous examples but on significantly coarser grids, to test whether the hybrid methods remain competitive with FMM and LSM. The performance data is summarized in Tables 2.8-2.11.

Since M is much smaller here, the $\log M$ term in the complexity of Fast Marching plays less of a role. On most of the examples in this subsection HCM and FHCM are not much faster than Fast Marching or Locking Sweeping. For example, in Table 2.9

Table 2.8: Performance/convergence results for 20 trials of 11 × 11 checkerboard example on a coarse grid.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
176 × 176	2.0986e-002	1.1087e-002	0.82	3.91	0.81	16

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	0.59				1.438	5.134	
HCM 44 × 44 cells	0.59				1.171	4.199	
HCM 88 × 88 cells	0.72				1.041	3.779	
FHCM 22 × 22 cells	0.41	1.0017	1.0000	1.0000	1.440	1.804	88.2
FHCM 44 × 44 cells	0.43	1.0015	1.0000	1.0000	1.171	1.374	97.0
FHCM 88 × 88 cells	0.59	1.0000	1.0000	1.0000	1.041	1.158	100.0
FMSM 22 × 22 cells	0.29	5.1670	1.0770	2.3920		1.269	
FMSM 44 × 44 cells	0.35	2.2742	1.0066	1.3489		1.334	
FMSM 88 × 88 cells	0.53	1.2309	1.0004	1.0040		1.221	

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
352 × 352	1.1470e-002	6.0787e-003	3.52	15.4	3.16	16

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	2.40				1.438	5.302	
HCM 44 × 44 cells	2.25				1.208	4.465	
HCM 88 × 88 cells	2.32				1.059	3.904	
HCM 176 × 176 cells	2.91				1.018	3.757	
FHCM 22 × 22 cells	1.61	1.1194	1.0002	1.0725	1.490	1.936	84.9
FHCM 44 × 44 cells	1.53	1.0434	1.0000	1.0000	1.228	1.508	92.2
FHCM 88 × 88 cells	1.69	1.0745	1.0000	1.0000	1.059	1.190	97.5
FHCM 176 × 176 cells	2.40	1.0273	1.0000	1.0000	1.018	1.086	100.0
FMSM 22 × 22 cells	1.12	10.551	1.1593	4.0315		1.269	
FMSM 44 × 44 cells	1.21	4.7036	1.0252	3.9089		1.334	
FMSM 88 × 88 cells	1.38	4.1945	1.0093	3.9089		1.222	
FMSM 176 × 176 cells	2.12	4.1945	1.0074	3.9089		1.127	

even though the cell boundaries are perfectly aligned with the checker boundaries, both Heap-Cell methods are merely on par with Fast Marching. Note that when h is sufficiently small, their advantage over FMM and LSM is clear (see Table 2.4). FMSM, however, is about twice as fast as the faster of FMM and LSM. In addition, FMSM's error ratios (R, $\mathcal{R}$, and ρ) are smaller here than for the same examples on finer grids in subsections 2.5-2.6.

Table 2.9: Performance/convergence results for 20 trials of 41×41 checkerboard on a coarse grid.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
164 × 164	7.1112e-002	3.8397e-002	1.08	17.9	4.01	44

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 41 × 41 cells	1.13				2.204	7.041	
HCM 82 × 82 cells	1.05				1.261	4.215	
FHCM 41 × 41 cells	0.85	1.0000	1.0000	1.0000	2.204	2.449	92.2
FHCM 82 × 82 cells	0.90	1.0000	1.0000	1.0000	1.261	1.474	100.0
FMSM 41 × 41 cells	0.53	1.4878	1.0850	1.0197		1.163	
FMSM 82 × 82 cells	0.77	1.1277	1.0162	1.0193		1.210	

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
328 × 328	4.0403e-002	2.3205e-002	4.44	73.3	16.6	45

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 41 × 41 cells	5.42				2.873	9.970	
HCM 82 × 82 cells	4.02				1.500	5.104	
HCM 164 × 164 cells	4.19				1.181	4.105	
FHCM 41 × 41 cells	3.65	1.0988	1.0008	1.0679	2.873	3.802	81.6
FHCM 82 × 82 cells	2.90	1.0236	1.0000	1.0000	1.501	1.923	88.0
FHCM 164 × 164 cells	3.55	1.0000	1.0000	1.0000	1.181	1.384	100.0
FMSM 41 × 41 cells	1.88	2.9459	1.4364	1.4668		1.164	
FMSM 82 × 82 cells	2.22	2.3040	1.0533	1.1457		1.211	
FMSM 164 × 164 cells	3.27	1.1540	1.0009	1.0679		1.281	

Remark 6. Since two of the hybrid methods introduce additional errors, an important question is, "Given the total errors resulting from FHCM and FMSM at a given resolution (h, h^c), for which $\bar{h} > h$ would FMM commit similar errors, and how well would FMM perform on that new coarser grid?" For simplicity, assume in the following discussion that the CPU time required by FMM is roughly linear in $M = O(h^{-2})$ and that the resulting L_∞ error is $O(h)$. These are reasonable assumptions for coarse grids; e.g., see Tables 2.8-2.11. For example, if we want to decrease the execution time by a factor of τ^2, then $M \to M/\tau^2$, $h \to \tau h$ (in 2D), and errors would increase by a factor of τ. Such estimates allow for a more accurate performance comparison between FMM and FMSM (or FHCM) based on the ratio R. Dividing the reported FMM time by the value R^2, we will arrive at an estimate for the new FMM time

Table 2.10: Performance/convergence results for 20 trials of $F(x, y) = 1 + \frac{1}{2}\sin(20\pi x)\sin(20\pi y)$ on a coarse grid.

Grid Size	**L_∞ Error**	**L_1 Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
176 × 176	3.6535e-002	1.3374e-002	0.94	8.77	3.08	28

METHOD	**TIME**	**$\mathcal{R}$**	**ρ**	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	0.97				1.773	8.233	
HCM 44 × 44 cells	0.88				1.280	4.992	
HCM 88 × 88 cells	0.87				1.100	3.975	
FHCM 22 × 22 cells	0.66	1.3736	1.0209	1.0000	2.153	2.814	69.3
FHCM 44 × 44 cells	0.60	1.1703	1.0186	1.0000	1.285	1.684	87.7
FHCM 88 × 88 cells	0.71	1.1170	1.0072	1.0000	1.100	1.234	100.0
FMSM 22 × 22 cells	0.38	7.0809	1.2945	1.0359		1.244	
FMSM 44 × 44 cells	0.42	2.2023	1.0402	1.0100		1.197	
FMSM 88 × 88 cells	0.64	1.0945	1.0024	1.0000		1.213	

Grid Size	**L_∞ Error**	**L_1 Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
352 × 352	1.8414e-002	7.0584e-003	3.92	33.7	11.1	27

METHOD	**TIME**	**$\mathcal{R}$**	**ρ**	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	4.43				1.909	9.864	
HCM 44 × 44 cells	3.57				1.403	5.969	
HCM 88 × 88 cells	3.18				1.178	4.493	
HCM 176 × 176 cells	3.45				1.060	3.891	
FHCM 22 × 22 cells	2.89	1.8770	1.0300	1.0202	2.905	3.630	66.2
FHCM 44 × 44 cells	2.29	1.8064	1.0712	1.0000	1.425	1.918	82.2
FHCM 88 × 88 cells	2.23	1.2724	1.0108	1.0000	1.182	1.394	93.7
FHCM 176 × 176 cells	2.84	1.0500	1.0016	1.0000	1.060	1.130	100.0
FMSM 22 × 22 cells	1.44	4.3257	1.4890	1.1939		1.246	
FMSM 44 × 44 cells	1.46	2.2958	1.0975	1.1932		1.197	
FMSM 88 × 88 cells	1.78	1.7082	1.0110	1.0806		1.213	
FMSM 176 × 176 cells	2.57	1.0845	1.0010	1.0000		1.126	

computed on a coarser $\bar{h}$-grid with errors similar to those committed by FMSM on an(h,h^c)-grid.

Among Tables 2.8-2.11, the overall worst-case scenario for FMSM under this analysis is the 11×11 checkerboard example. Using the data in Table 2.8 with $M = 176^2$ and comparing FMM with FMSM at 22^2, 44^2, and 88^2 cells, the new estimated FMM times would be $0.82/(2.392^2) = .343$, $0.82/(1.3489^2) = .608$, and $0.82/(1.004^2) = .817$. Comparing this to 0.29, 0.35, 0.53 reported for FMSM, we see that each of the cell trials still outperforms the corresponding improved time of FMM. Similar conclusions

Table 2.11: Performance/convergence results for 20 trials $F(x,y) = 1 + 0.99\sin(2\pi x)\sin(2\pi y)$ on a coarse grid.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
176 × 176	1.0533e-001	5.6430e-003	0.93	4.00	0.93	13

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	0.74				1.165	4.496	
HCM 44 × 44 cells	0.73				1.085	4.040	
HCM 88 × 88 cells	0.83				1.026	3.790	
FHCM 22 × 22 cells	0.47	1.0952	1.0020	1.0004	1.169	1.388	94.2
FHCM 44 × 44 cells	0.50	1.0200	1.0005	1.0000	1.087	1.173	97.9
FHCM 88 × 88 cells	0.66	1.0045	1.0001	1.0000	1.027	1.051	100.0
FMSM 22 × 22 cells	0.37	1.2819	1.0044	1.0164		1.231	
FMSM 44 × 44 cells	0.41	1.1839	1.0007	1.0053		1.116	
FMSM 88 × 88 cells	0.59	1.0979	1.0001	1.0000		1.057	

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
352 × 352	6.8813e-002	3.1818e-003	3.84	15.9	3.64	13

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	3.00				1.178	4.624	
HCM 44 × 44 cells	2.76				1.076	4.082	
HCM 88 × 88 cells	2.83				1.033	3.853	
HCM 176 × 176 cells	3.29				1.008	3.747	
FHCM 22 × 22 cells	1.82	1.1364	1.0040	1.0004	1.178	1.405	93.2
FHCM 44 × 44 cells	1.71	1.0204	1.0005	1.0000	1.080	1.170	97.7
FHCM 88 × 88 cells	1.98	1.0034	1.0001	1.0000	1.034	1.071	99.2
FHCM 176 × 176 cells	2.69	1.0006	1.0000	1.0000	1.008	1.022	100.0
FMSM 22 × 22 cells	1.44	2.3482	1.0080	1.0074		1.231	
FMSM 44 × 44 cells	1.42	1.5167	1.0014	1.0037		1.116	
FMSM 88 × 88 cells	1.61	1.1989	1.0004	1.0034		1.057	
FMSM 176 × 176 cells	2.44	1.0953	1.0001	1.0015		1.028	

are reached when this analysis is performed using error ratios in L_1 norms.

Remark 7. We could perform a similar comparison between FMSM and sweeping methods, but the latter allow for yet another speed up technique: the sweeping can be stopped before the full convergence to the solution of system (1.4). In fact, in many implementations of Fast Sweeping, the method terminates when the changes in grid values due to the most recent sweep fall below some positive threshold κ; e.g.; see [40]. Similarly to FHCM and FMSM, this results in additional errors, and it is useful to consider both these errors and the corresponding savings in computational

Sweep #	**Max Change**	**% GPs changing**	$\boldsymbol{\mathcal{R}}$	$\boldsymbol{\rho}$	**R**
1	1.00e+008	26.22	-	-	-
2	1.000e+008	31.856	-	-	-
3	1.000e+008	58.247	44.595	1.7709	4.8179
4	2.7622e-001	44.4527	1.4685	1.1027	1.2445
5	6.3846e-003	41.5341	1.4224	1.0888	1.1995
6	5.9641e-003	41.1957	1.4195	1.0759	1.1995
7	5.9641e-003	41.0730	1.3832	1.0631	1.1951
8	5.4993e-003	40.1919	1.3331	1.0509	1.1562
9	4.9918e-003	37.0650	1.3243	1.0440	1.1205
10	4.9918e-003	36.6337	1.3230	1.0377	1.1205
11	4.9918e-003	36.3995	1.2881	1.0314	1.1191
12	4.7740e-003	34.6743	1.2492	1.0255	1.0854
13	4.5076e-003	31.3318	1.2403	1.0218	1.0532
14	4.5076e-003	30.8150	1.2400	1.0185	1.0520
15	4.5076e-003	30.4767	1.2098	1.0152	1.0511
16	4.1600e-003	28.0934	1.1820	1.0121	1.0270
17	3.6304e-003	22.6502	1.1646	1.0102	1.0004
18	3.6304e-003	21.8062	1.1644	1.0085	1.0000
19	3.6304e-003	21.2374	1.1467	1.0068	1.0000
20	3.2984e-003	19.1404	1.1268	1.0052	1.0000
21	2.7917e-003	14.3367	1.1079	1.0043	1.0000
22	2.7142e-003	13.6340	1.1079	1.0035	1.0000
23	2.7142e-003	13.2250	1.0951	1.0027	1.0000
24	2.4311e-003	11.6746	1.0812	1.0020	1.0000
25	2.1725e-003	8.4659	1.0637	1.0015	1.0000
26	1.8533e-003	7.9677	1.0630	1.0012	1.0000
27	1.8533e-003	7.6852	1.0546	1.0009	1.0000
28	1.7075e-003	6.6664	1.0457	1.0006	1.0000
29	1.5049e-003	4.8000	1.0365	1.0004	1.0000
30	1.1216e-003	4.4653	1.0303	1.0003	1.0000
31	1.1216e-003	4.2646	1.0257	1.0002	1.0000
32	1.0109e-003	3.5656	1.0209	1.0001	1.0000
33	8.5675e-004	2.2754	1.0153	1.0001	1.0000
34	4.8751e-004	2.0300	1.0110	1.0001	1.0000
35	4.8751e-004	1.8813	1.0087	1.0000	1.0000
36	4.2582e-004	1.4314	1.0068	1.0000	1.0000
37	3.4338e-004	0.7064	1.0043	1.0000	1.0000
38	1.1188e-004	0.5689	1.0025	1.0000	1.0000
39	1.1188e-004	0.4871	1.0015	1.0000	1.0000
40	8.9968e-005	0.2863	1.0011	1.0000	1.0000
41	6.8284e-005	0.0632	1.0006	1.0000	1.0000
42	2.4066e-005	0.0297	1.0002	1.0000	1.0000
43	1.0931e-005	0.0112	1.0000	1.0000	1.0000
44	0.0000e+000	0.0000	1.0000	1.0000	1.0000

A

Sweep #	**Max Change**	**% GPs changing**	$\boldsymbol{\mathcal{R}}$	$\boldsymbol{\rho}$	**R**
1	1.0e+008	25.2	-	-	-
2	1.000e+008	34.249	-	-	-
3	1.000e+008	62.372	48.051	9.2339	30.026
4	3.621e-001	49.221	12.002	4.1935	7.7797
5	1.0709e-002	43.0590	11.194	3.9168	7.7098
6	1.0252e-002	42.1586	10.474	3.6528	7.2822
7	1.0252e-002	42.0001	10.269	3.3925	7.2771
8	1.0229e-002	39.8694	9.6885	3.1431	6.9538
9	1.0207e-002	34.3652	9.6713	2.9458	6.9386
10	1.0207e-002	33.2951	9.4074	2.7562	6.5653
11	1.0207e-002	33.1453	9.0914	2.5686	6.5623
12	1.0185e-002	31.2973	8.6218	2.3892	6.1648
13	1.0165e-002	26.4975	8.5771	2.2483	6.1607
14	1.0165e-002	25.5465	8.3781	2.1135	5.8497
15	1.0165e-002	25.3961	8.0972	1.9804	5.8487
16	1.0145e-002	23.7761	7.5600	1.8536	5.4607
17	1.0127e-002	19.6460	7.5550	1.7561	5.4571
18	1.0127e-002	18.8095	7.2488	1.6635	5.0667
19	1.0127e-002	18.6819	7.0133	1.5722	5.0658
20	1.0108e-002	17.2760	6.5857	1.4861	4.7569
21	1.0092e-002	13.8028	6.5691	1.4221	4.7449
22	1.0092e-002	13.0992	6.2438	1.3619	4.3682
23	1.0092e-002	12.9746	6.0465	1.3027	4.3674
24	1.0075e-002	11.8224	5.5031	1.2475	3.9749
25	1.0060e-002	9.0073	5.4990	1.2088	3.9720
26	1.0060e-002	8.4400	5.2424	1.1730	3.6712
27	1.0060e-002	8.3202	5.0816	1.1380	3.6705
28	1.0045e-002	7.3932	4.5433	1.1059	3.2817
29	1.0031e-002	5.2311	4.5402	1.0854	3.2794
30	1.0031e-002	4.8007	4.1140	1.0670	2.8875
31	1.0031e-002	4.7054	3.9971	1.0491	2.8871
32	1.0018e-002	4.0108	3.5893	1.0334	2.5926
33	1.0005e-002	2.5072	3.5711	1.0249	2.5795
34	1.0005e-002	2.2144	3.1264	1.0177	2.2008
35	1.0005e-002	2.1276	3.0465	1.0109	2.2005
36	9.9928e-003	1.6782	2.4976	1.0053	1.8040
37	9.9809e-003	0.8296	2.4942	1.0034	1.8016
38	9.9809e-003	0.6789	2.1526	1.0020	1.5223
39	9.9809e-003	0.6047	2.1076	1.0009	1.5223
40	9.9619e-003	0.3888	1.5638	1.0002	1.1295
41	5.0711e-003	0.1151	1.5638	1.0001	1.1295
42	4.9343e-003	0.0698	1.1631	1.0000	1.0000
43	1.4668e-003	0.0338	1.0000	1.0000	1.0000
44	0.0000e+000	0.0000	1.0000	1.0000	1.0000

B

Table 2.12: Maximum change of V for the sweeping methods for the 41×41 checkerboard example on the 164×164 grid (A) and 1312×1312 grid (B).

time. To the best of our knowledge, this issue has not been analyzed so far. The practical implementations of FSM and LSM typically select κ heuristically or make it proportional to the grid-size h. It is usually claimed that the number of sweeps necessary for convergence is h-independent [70]. Tables 2.9 and 2.10 seems to show that the number of sweeps-to-convergence (i.e., for $\kappa = 0$) depends on h. We believe this is due to both the fact the viscosity solution is revealed in more detial as h decreases and that the location of gridpoints relative to shocklines is h-dependent.

For $\kappa > 0$, the more relevant questions are:

1. How well do the changes in the most recent sweep represent the additional errors, which would result if we were to stop the sweeping?
2. Is the number of sweeps (needed for a fixed $\kappa > 0$) h-independent?
3. Supposing the additional ("early-termination") errors could be estimated, would the number of required sweeps be h-independent?
4. Supposing FSM or LSM were run for as many sweeps as necessary to make the additional errors approximately the same as those introduced by FMSM or FHCM, would the resulting computational costs be less than those of hybrid methods?

To answer these questions for one specific (41×41 checkerboard) example, we have run both sweeping methods on 164^2 and 1312^2 grids. In table 2.12 we report the L_∞ change in grid values, the percentage of gridpoints changing, and potential early-termination errors ($\boldsymbol{\mathcal{R}}$, $\boldsymbol{\rho}$, and $\mathbf{R}$) after each sweep. At least for this particular example:

1. The answer to Question 1 is inconclusive, though the max changes are clearly correlated with $\mathbf{R}$ and ρ.

2. The answer to Question 2 is negative; moreover, after the same number of sweeps, the max changes on the 1312^2 grid are clearly larger than on the 164^2 grid.

3. The answer to Question 3 is negative; e.g., $\mathbf{R}$ reduces below 1.1 after only 12 sweeps on the 164^2 grid, but the same reduction on the 1312^2 grid requires 42 sweeps.

4. To answer the last question, we note that for this example FHCM produces very small additional errors, while FMSM results in $\mathbf{R} = 1.0197$ and $\mathbf{R} = 1.0193$ (on the 164^2 grid with 21^2 and 42^2 cells, respectively; see Table 2.9). As Table 2.12A shows, 16 sweeps would be needed for FSM or LSM to produce the same $\mathbf{R}$ values on this grid. Our computational experiment shows that FSM and LSM times for these 16 sweeps are 6.62 and 2.91 seconds respectively (note that this is the total time for 20 trials, similar to the times reported in Table 2.9). Thus, FMSM is still more than 3.5 times faster than the early-terminated LSM and more than 8 times faster than the early-terminated FSM. For the 1312^2 example, we see that the error ratios take longer to converge to 1 for the sweeping methods (Table 2.12B). The FMSM $\mathbf{R}$ values of $\{3.3991, 1.7662, 1.7123\}$ (from Table 2.4, for the different cell sizes) correspond $\{28, 37, 37\}$ sweeps in Table 2.12B. The experimentally measured early-terminated execution times for FSM and LSM are $\{36.85, 48.77, 48.77\}$ seconds and $\{7.40, 11.68, 11.68\}$ seconds respectively. Again, FMSM still holds a large advantage (more than 4 times faster than LSM and more than 18 times faster than FSM). We note that for both the 164^2 and 1312^2 cases, the early-terminated FSM time was linear in the number of sweeps, while LSM did not receive as much of a speed boost; this is natural since the percentage of gridpoints changing in the omitted "later iterations" is low, and the LSM's computational cost is largely dependent on the number of unlocked gridpoints in each sweep.

2.8 Continuous speed functions with general homogeneous boundary conditions

Next we return to speed functions $F(x, y) = 1 + 0.99 \sin(2\pi x) \sin(2\pi y)$ and $F(x, y) = 1 + \frac{1}{2} \sin(20\pi x) \sin(20\pi y)$, but this time with zero boundary conditions on the entire boundary of the square. The performance data is summarized in Tables 2.13 and 2.14.

Remark 8. Our current implementation of FMSM treats the coarse gridpoints nearest to the boundary as *Accepted* in the initialization. If there is more than one coarse gridpoint in the exit set, as in the following examples, care must be taken when ranking the "acceptance order" of these coarse gridpoints. While in the case of single-point exit sets it is safe to assign a zero value to these coarse gridpoints, for general boundary conditions we compute the values by a one-sided update from the cell center to the nearest point on the boundary. In addition, our FMSM implementation iterates FSM to convergence on all cells containing parts of Q before determining the sweeping directions for any other cells.

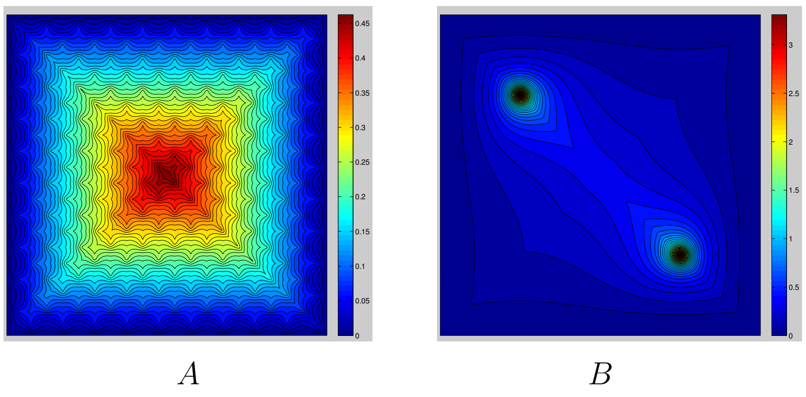


Figure 2.13: Min time to $\partial\Omega$ under two sinusoidal speed functions.

Table 2.13: Performance/convergence results for $F(x,y) = 1 + \frac{1}{2}\sin(20\pi x)\sin(20\pi y)$ with $Q = \partial\Omega$.

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	1.3670e-003	3.7171e-004	3.89	24.3	6.62	24

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	3.48				1.853	10.273	
HCM 44 × 44 cells	2.92				1.470	6.811	
HCM 88 × 88 cells	2.53				1.195	4.987	
HCM 176 × 176 cells	2.35				1.098	4.301	
HCM 352 × 352 cells	2.37				1.046	3.951	
HCM 704 × 704 cells	2.91				1.018	3.785	
FHCM 22 × 22 cells	2.60	20660	1.5321	3.2150	2.915	4.498	54.5
FHCM 44 × 44 cells	1.95	62.164	1.2465	1.5447	1.539	2.502	68.0
FHCM 88 × 88 cells	1.66	64.719	1.0187	1.0128	1.223	1.749	83.9
FHCM 176 × 176 cells	1.55	5.7122	1.0032	1.0063	1.102	1.361	92.4
FHCM 352 × 352 cells	1.66	1.1083	1.0007	1.0011	1.047	1.165	97.5
FHCM 704 × 704 cells	2.40	1.0192	1.0001	1.0001	1.018	1.064	100.0
FMSM 22 × 22 cells	1.97	1.6383e+5	7.8665	12.339		2.184	
FMSM 44 × 44 cells	1.67	1.1325e+6	2.6113	4.2370		1.892	
FMSM 88 × 88 cells	1.42	5506.21	1.0388	1.8072		1.527	
FMSM 176 × 176 cells	1.29	859.45	1.0044	1.2609		1.265	
FMSM 352 × 352 cells	1.40	253.58	1.0009	1.0270		1.134	
FMSM 704 × 704 cells	2.17	6.6107	1.0001	1.0000		1.062	

Table 2.14: Performance/convergence results for $F(x,y) = 1 + 0.99\sin(2\pi x)\sin(2\pi y)$ with $Q = \partial\Omega$.

Grid Size	**L_∞ Error**	**L_1 Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	2.2246e-002	2.7572e-004	3.66	8.06	2.58	8

METHOD	**TIME**	**$\mathcal{R}$**	**ρ**	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	2.03				1.176	4.448	
HCM 44 × 44 cells	1.97				1.089	4.021	
HCM 88 × 88 cells	1.93				1.047	3.830	
HCM 176 × 176 cells	1.96				1.020	3.718	
HCM 352 × 352 cells	2.10				1.009	3.670	
HCM 704 × 704 cells	2.74				1.006	3.649	
FHCM 22 × 22 cells	1.51	136.37	1.0001	1.0000	1.176	1.903	93.4
FHCM 44 × 44 cells	1.35	2.4167	1.0000	1.0000	1.091	1.443	99.0
FHCM 88 × 88 cells	1.32	2.4167	1.0000	1.0000	1.048	1.226	99.6
FHCM 176 × 176 cells	1.39	1.6390	1.0000	1.0000	1.020	1.110	99.8
FHCM 352 × 352 cells	1.57	1.0000	1.0000	1.0000	1.009	1.054	99.9
FHCM 704 × 704 cells	2.33	1.0000	1.0000	1.0000	1.006	1.028	100.0
FMSM 22 × 22 cells	1.57	12592	1.0441	1.0000		1.599	
FMSM 44 × 44 cells	1.27	355.53	1.0088	1.0000		1.306	
FMSM 88 × 88 cells	1.15	355.53	1.0055	1.0000		1.157	
FMSM 176 × 176 cells	1.14	303.61	1.0030	1.0000		1.079	
FMSM 352 × 352 cells	1.31	134.60	1.0012	1.0000		1.040	
FMSM 704 × 704 cells	2.11	68.199	1.0004	1.0000		1.014	

CHAPTER 3

PARALLELIZATION

We devote most of this chapter to the development of the parallel Heap-Cell Method (pHCM). Other material includes a new cell value heuristic for HCM and a seismic imaging application. The extensive results section of this chapter shows that pHCM scales well and greatly outperforms prior serial methods and parallel methods for shared-memory architectures. We begin with a literature review of parallel methods for Eikonal equations.

3.1 Prior Parallel Methods

Several interesting approaches have been used to design parallel methods for Eikonal and related PDEs. A careful performance/scalability comparison of *all* such methods would be clearly valuable for practitioners but remains outside of scope of this thesis. Here we give a brief overview of prior approaches primarily to put pHCM in context. In section 3.4 we also use one of them as a benchmark for comparison with our own approach.

Two different parallelizations of FSM were introduced in [71]. The first performs a domain decomposition and uses separate processors to run the serial FSM on each subdomain. Subdomains are pre-assigned to processors and communication takes place along the shared boundaries. The second approach 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 [26] is a more recent parallel sweeping technique (which we call "Detrixhe Fast Sweeping Method" or DFSM) that utilizes the fact that, for the upwind scheme in 3D (eq.(1.4)), gridpoints along certain planar slices through the computational domain do not directly depend on each other. The planes are given by

$$\phi_i i + \phi_j j + \phi_k k = C,$$

for $\phi_i, \phi_j, \phi_k \in \{-1, 1\}$ and $C \in \mathbb{Z}$. The choice of ϕ's determines one of the 2^3 sweeping directions; once the χ'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* [54] 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 [71], 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 2.3. 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 recently used to parallelize the sweeping for more general (anisotropic) problems in [31].

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 preassigned subdomain(s) (e.g., [35]).In the absence of a strictly causal relationship between subdomains, this inevitably leads to erroneous gridpoint values, which can be later fixed by re-running the FMM whenever the subdomains' boundary data changes. One recent method A very recent massively parallel implementation for distributed memory architecture in [28] uses coarse grid computations to find a good subdomain preassignment, attempting to exploit non-strict causality to improve the efficiency; the approach is then re-used recursively to create a multi-level 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 3.1. 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.

Two recent approaches aim to minimize inter-domain communication by creating problem-dependent causal domain decompositions. One [14] decomposes the exit set into P sets, where P is the number of processors, and then each processor runs FMM serially starting from its own piece of the exit set. It is still necessary to reprocess ACCEPTED gridpoints (since subdomain boundaries are only approximately characteristic), and there is no guarantee that loads will be balanced. The other is the "Patchy FMM" developed in [15] 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 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 [64] and the Group Marching Method [41]) without resorting to domain decompositions. But we are not aware of any existing parallel implementations,

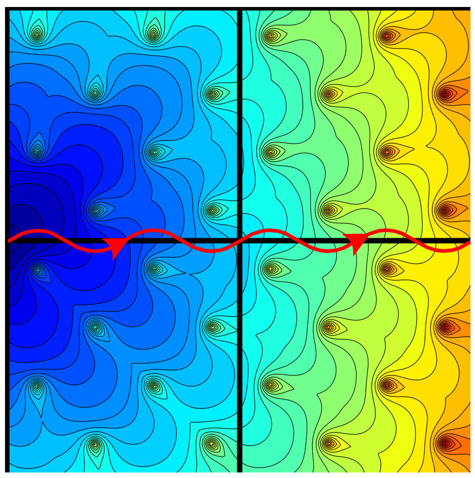


Figure 3.1: Level sets for an Eikonal problem in 2D with cell boundaries in black and a characteristic in red. Since the characteristic repeatedly crosses the subdomain boundary, any method that solves this problem using the given domain decomposition will require a large number of iterations.

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 [11]. 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 [67], 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 [71] and [26].

Another method intended for massively parallel (SIMD GPU) architectures is the "Fast Iterative Method" (FIM) developed in [38]. 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.

3.2 An improvement to HCM: new cell value heuristic

In this chapter, our treatment of the cell value is different from the one in chapter 2 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 equation (3.1). 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. To be precise,

$$\widetilde{V}^c(B) \leftarrow \min_{j \in A_{new}} V(\boldsymbol{x}_j) \qquad V^c(B) \leftarrow \min(V^c(B), \widetilde{V}^c(B)) \tag{3.1}$$

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 8.

We use a natural initialization of cell values before the main loop of the algorithm:

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

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 original cell value (e.g., see section 2.3). A comparison with the original heuristic is also performed in section 3.4.8.

Although this cell value was independently developed, we later found that it appeared in earlier work by Kimmel, et al. [67] in their "Raster scan algorithm for a multi-chart geometry image," which also ranks the charts (cells) on the priority queue according to this heuristic. This method differs from pHCM in several ways, one of which is that pHCM achieves its efficiency when each cell is almost completely inflow or outflow; for complicated speed functions and boundary conditions this may require significantly smaller cell sizes compared to chart sizes used in the implementation in [67]. The similarities and differences between the two methods are described in detail in Remark 4 in the previous chapter.

3.3 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 re-processing, we attempt 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 10, all data is shared unless stated otherwise.

Algorithm 10 Parallel Heap-Cell Method pseudocode.

```
 1: Cell Initialization: same as in HCM (divide cells $Q^c$ evenly among all heaps $L_p$)
 2: Fine Grid Initialization: same as in HCM
 3: $P \leftarrow$ number of threads
 4: $activeCellCount \leftarrow |Q^c|$
 5: PARALLEL SECTION
 6: while $activeCellCount > 0$ do
 7:     while $L_p$ is nonempty do
 8:         Lock heap $L_p$
 9:             Position-lock cell $c$ at the top of $L_p$
10:                 Remove $c$ from $L_p$
11:                 $V^c(c) \leftarrow +\infty$
12:             Position-unlock $c$
13:         Unlock $L_p$
14:         Compute-Lock $c$
15:             Perform modified LSM on $c$ and populate the (local) list $DN$
16:             of *currently downwind* neighboring cells        //see Algorithm 8
17:             Set all preferred sweeping directions of $c$ to FALSE
18:         Compute-Unlock $c$
19:         for each $c_k \in DN$ do
20:             Compute a possible new (local) cell value $\widetilde{V}$ for $c_k$
21:             if $\widetilde{V} < V(c_k)$ then
22:                 Set Cell Value $(c_k, \widetilde{V})$        //see Algorithm 11
23:             end if
24:             if $c_k$ is not on a heap then
25:                 Add Cell $(c_k)$        //see Algorithm 12
26:             end if
27:             Update sweeping directions of $c_k$ based on location of $c$
28:         end for
29:         $activeCellCount$ $--$        (atomic)
30:     end while
31: end while
```

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:

Algorithm 11 Set Cell Value $(c_k, \widetilde{V})$.

```
1: $success \leftarrow$ FALSE
2: while $success ==$ FALSE do
3:     if $c_k$ is not on a heap then
4:         Position-lock $c_k$
5:         if $c_k$ is still not on a heap then
6:             $V(c_k) \leftarrow \min(\widetilde{V}, V(c_k))$
7:             $success \leftarrow$ TRUE
8:         end if
9:         Position-unlock $c_k$
10:    else
11:        $j \leftarrow$ index of the heap of $c_k$
12:        Lock $L_j$
13:        Position-lock $c_k$
14:        if $c_k$ is still on $L_j$ then
15:            $V(c_k) \leftarrow \min(\widetilde{V}, V(c_k))$
16:            Heap-sort $L_j$
17:            $success \leftarrow$ TRUE
18:        end if
19:        Position-unlock $c_k$
20:        Unlock $L_j$
21:    end if
22: end while
```

- 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.
- There are two individual cell operations that must be **individually** serialized: 1) the movement of a cell onto/ off / within a heap and 2) the update of gridpoint values within that cell. However, **together** both can safely be 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

Algorithm 12 Add Cell (c_k).

```
1: $j \leftarrow$ index of heap with fewest elements (no locking; counts may be outdated
   during search);
2: $testCount \leftarrow 0$
3: while Lock $L_{(j+testCount)\%P}$ can not be immediately obtained do
4:     $testCount$++
5: end while
6:
7: Position-Lock $c_k$
8: if $c_k$ is still not on a heap then
9:     Add $c_k$ onto $L_{(j+testCount)\%P}$
10:    $activeCellCount$ ++          (atomic)
11: end if
12: Position-Unlock $c_k$
13: Unlock $L_{(j+testCount)\%P}$
```

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 12. 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 10 (rather than around line 10) to prevent other threads from quitting prematurely.
- The cell update (lines 15-17 of Algorithm 10) is exactly the same sweeping procedure as in HCM. Just as in HCM, any other method that solves system (1.4) 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.

3.3.1 Efficiency and data race conditions

There is always a delicate trade-off 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 10 are executed in the order 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 multi-threaded environment this can lead to unexpected results[1].

Consider first 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 a border gridpoint by Eq. (1.4) requires reading information in a neighboring cell. But it is easy to see that the monotonicty 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 2.6). 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).

[1]Indeed, in our implementation it was actually necessary to explicitly prevent such reordering of certain lines of code (using Open MP's "flush" pragma).

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 8 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 8 makes this scenario impossible, since setting a gridpoint inactive is immediately *followed* by the re-computation of that gridpoint's value.

Finally, there is an one additional design choice we have made that causes a race condition at the cell-level when setting the cell's preferred sweeping 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.

3.4 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 3.4.1 and 3.4.2 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 chapter 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 8 gridpoints closest to the center. Since J values are also even, the set Q^c contains 8 cells in all of the examples.

The speed functions are:

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. Figure 3.2 shows various level sets of examples 2 and 3.

Experimental setup and implementation details

All experiments in this section (except for those in subsection 3.4.5) 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 processers 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 (except for those in subsection 3.4.4) 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 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 methods' performance/scaling to a parallelization of the sweeping methods. Our implementation largely follows the method described in [26], but with two exceptions:

- Detrixhe et. al. 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 implementation of DFSM and DLSM use the default Open MP static loop scheduling ("omp for") to divide the work amongst threads instead of the manual load balancing procedure described in [26].

In all iterative methods, the sweeps were continued as long as some gridpoints received updated values; in subsection 3.4.3 we separately investigate the performance improvements due to an "early termination". In subsection 3.4.4 we explore the influence of memory footprint by storing/computing values in *single* precision. In subsection 3.4.5 we provide additional benchmarking results on a different shared memory architecture. Subsections 3.4.6 and 3.4.7 contain results for additional examples (with piecewise-constant F). Finally, in subsection 3.4.8 we provide data for performance with a different cell value heuristic.

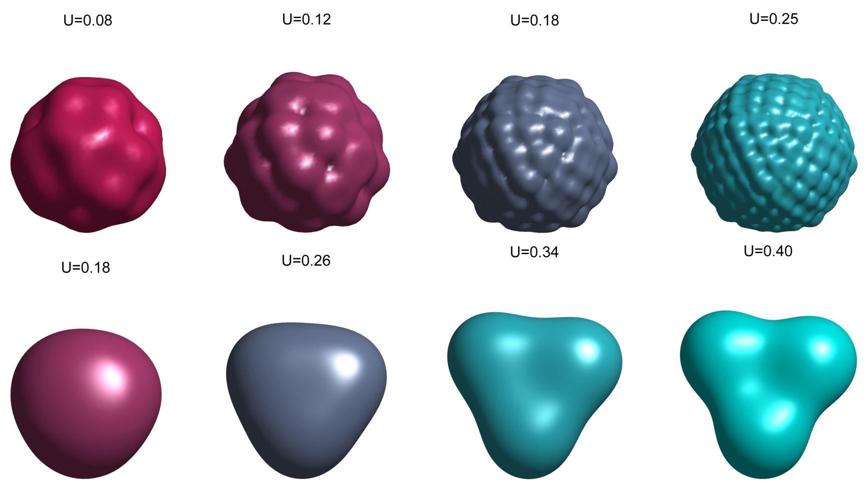


Figure 3.2: Some level sets of the value functions of Example 2 (row 1) and Example 3 (row 2). Not to scale.

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 2/4/8/16/32 gridpoints per cell side. "HCMr" and "pHCMr" in the legends mean HCM and pHCM with $J = M/r^3$. On each test problem the performance of pHCM depends on 3 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 3.3, 3.4, 3.6, and 3.7 are organized so that columns present different examples and rows give different comparison metrics. Figure 3.3 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 3.4, $\boldsymbol{M}$ **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),

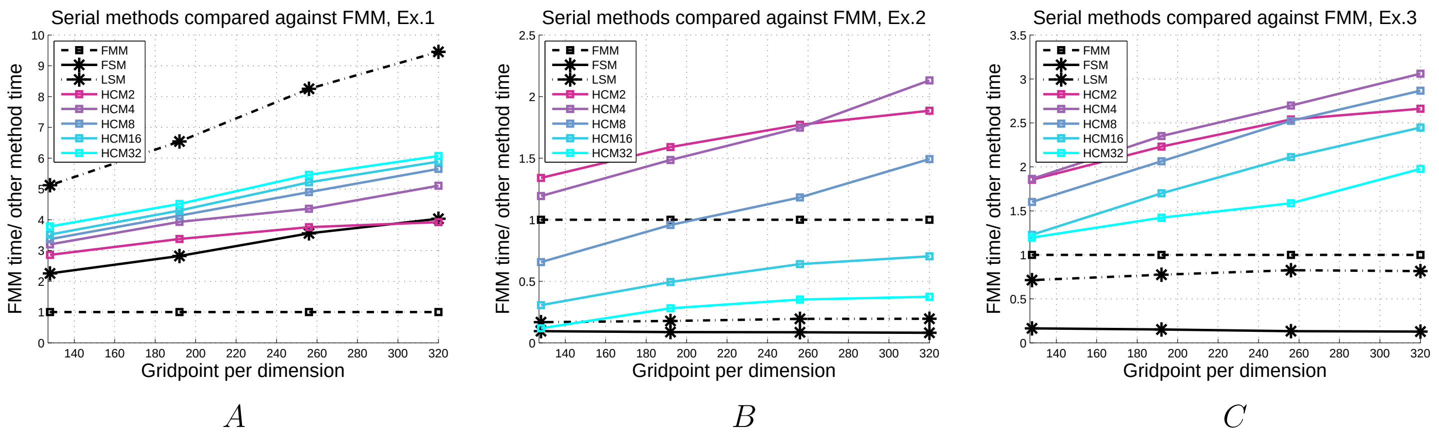


Figure 3.3: Performance of the serial methods for different M. The first chart has $F \equiv 1$, the second has $F = 1 + .5\sin(20\pi x)\sin(20\pi y)\sin(20\pi z)$, and the third has $F = 1 + .99\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$. The data is given as a ratio of FMM's CPU time to the times of all other method.

and (LSM time / DLSM time). The second row of Fig. 3.4 provides the performance comparison of all parallel methods. The growth of parallel overhead and the change in total work (as functions of P) are presented for each pHCMr in Figure 3.6. Plots similar to Figure 3.4 but computed for $M = 128^3$ are presented in subsection 3.4.2.

Main observations:

1. LSM significantly outperforms FMM on example 1 (Fig. 3.3A) and its advantage grows with M. FMM greatly outperforms LSM on example 2 (Fig. 3.3B) for all values of M. Their performance is more comparable on the third example (Fig. 3.3C).

2. The performance ranking among serial HCMr methods is problem-dependent (Fig. 3.3A-3.3C).

3. Figures 3.4D-3.4F demonstrate that pHCM has a large advantage over all serial methods for most r and P combinations. On the three examples with $M = 320^3$, the median performance for pHCM8 on 32 threads was between 34 and 84 times

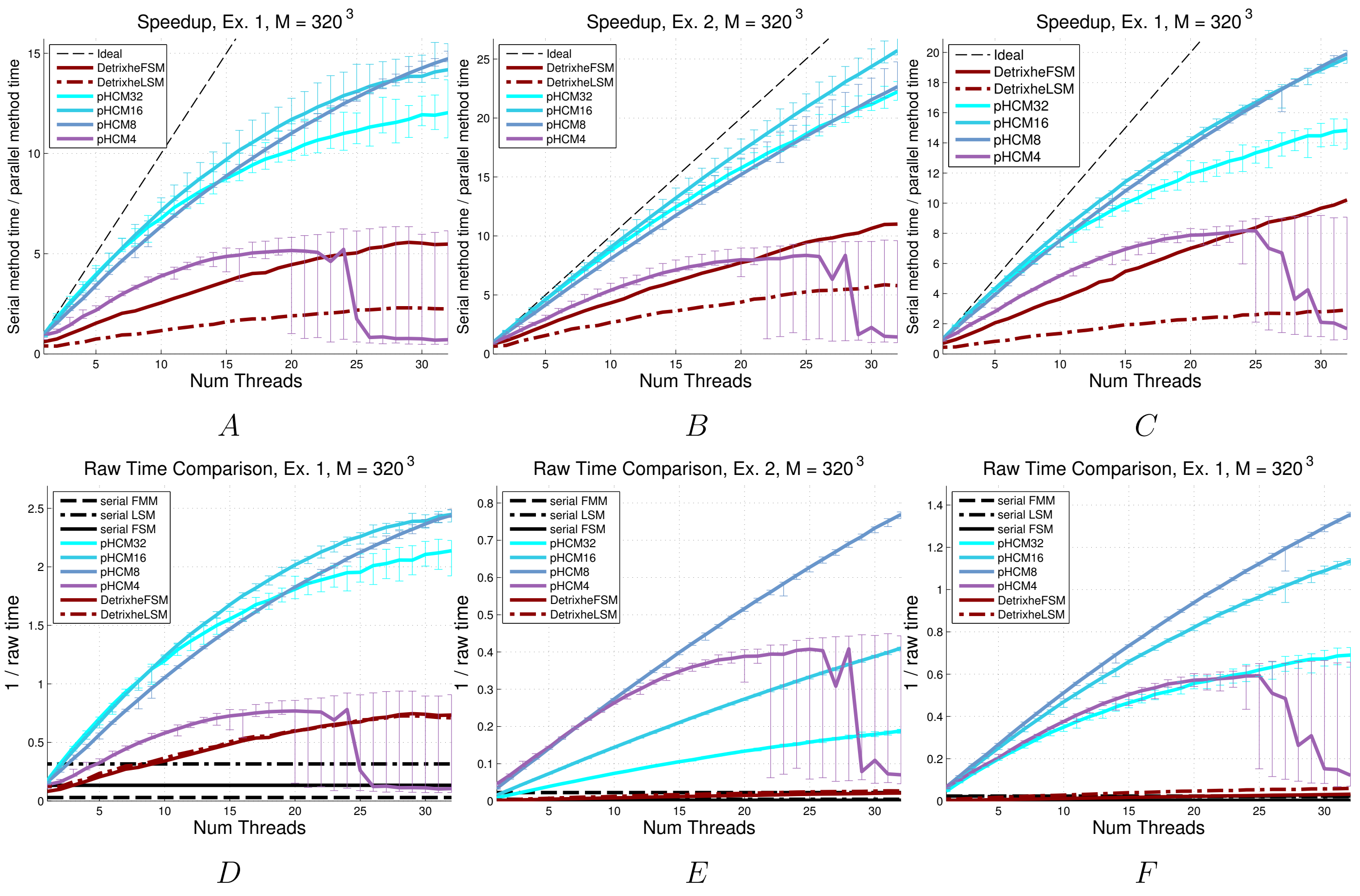


Figure 3.4: Scaling and performance for pHCM at $M = 320^3$. The first column has $F \equiv 1$, the second has $F = 1 + .5\sin(20\pi x)\sin(20\pi y)\sin(20\pi z)$, and the third has $F = 1 + .99\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$.

faster than FMM, between 7.7 and 166 times faster than LSM, and between 18.4 and 436 times faster than FSM.

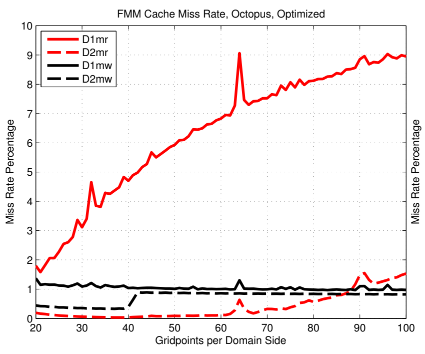
4. Generally, the pHCM speedup over HCM is greater when there is more work per cell. We see in Figures 3.4A-3.4C that the experiments with higher gridpoints-per-cell number r exhibit better parallelization, and the speedup of pHCM4 is always the worst.

5. In Figure 3.4 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.

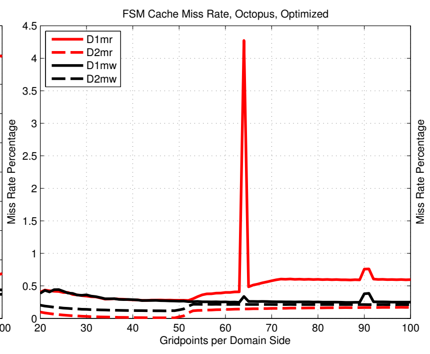
6. Based on Figure 3.4, for most r values pHCM scales much better than DFSM/DLSM. Since DFSM is a synchronous parallel algorithm, it comes as no surprise that using the Locking Sweeping does not boost performance significantly – LSM only reduces the amount of work performed by a subset of the threads. Better scaling in DLSM would likely be achieved if it were possible to apply a special load balancing procedure based on the set of currently "active" gridpoints.

3.4.1 Further comments on performance of serial methods

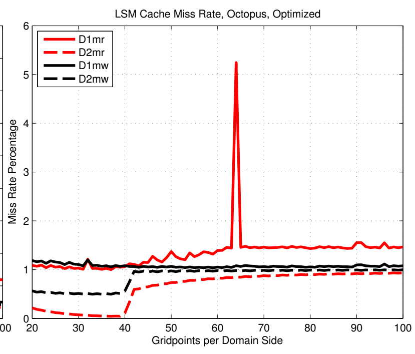
1. *Tradeoffs between FMM and LSM.* It is well known that Marching and Sweeping methods are each advantageous on their own subsets of Eikonal problems. The exact delineation remains a matter of debate. The readers can find careful comparative studies in [34, 37] and partly in [16]. In each example (Figs. 3.3A-3.3C) 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. Solving equation (1.4) 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-



A



B



C

Figure 3.5: Cache miss rates for FMM, FSM, and LSM as M increases. "D1mr" means the rate of data misreads at the highest level (L1) cache. "D2mr" means the rate of data misreads at the **lowest level** (L2 or L3) cache. The black curves are rates of data mis-writes.

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., [46]), 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. On top of this, 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 compared to the sweeping methods; see Figure 3.5.

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 2.2.2) 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 3.1, 3.2, and 3.3 ("Heap Maintenance %" means the percentage of execution time spent outside of sweeping cells). Turning to individual examples:

   (a) Ex.1: HCM with larger cell sizes performs better. See Figure 3.3A and Table 3.1. 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 3.1: 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 example 1, and the sweeping methods are the slowest. See Figure 3.3B and Table 3.2.

   (c) Ex. 3: Figure 3.3C and Table 3.3 show that the performance among the HCMr methods is qualitatively different from the previous examples. A weakly causal ordering already exists here for moderately-sized cells.

Table 3.2: 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 3.3: 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

3.4.2 Detailed performance analysis of parallel methods

Two key factors that affect the speedup of parallel methods are the amount of parallel overhead (contention, inter-thread 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 3.1, 3.2, and 3.3.

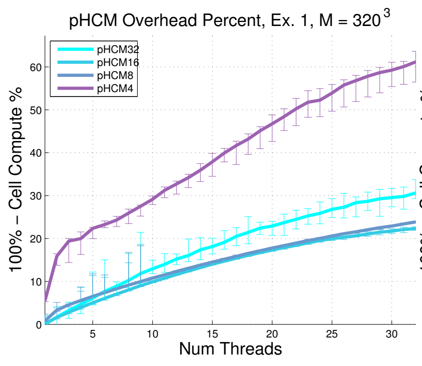
We define:

• AvS $= \sum_{p=0}^{P-1}$ (Total number of sweeps performed by processor p) $/J$.

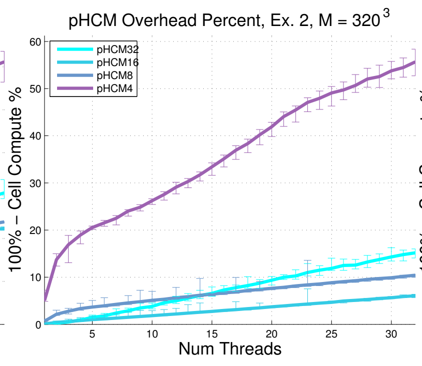
• Cell Comp % = percent of total time spent on sweeping cells alone.

• Overhead % = 100% - Cell Comp %, i.e., percent of total time spent beyond sweeping cells.

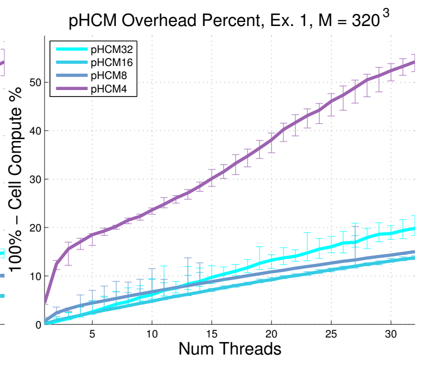
1. *Effects of P on overhead.* As P increases, contention and network communication increase. If more threads are used for a given cell discretization, it is more likely for a processor $\hat{p}$ to wait to obtain a lock (e.g., as in line 8 of algorithm



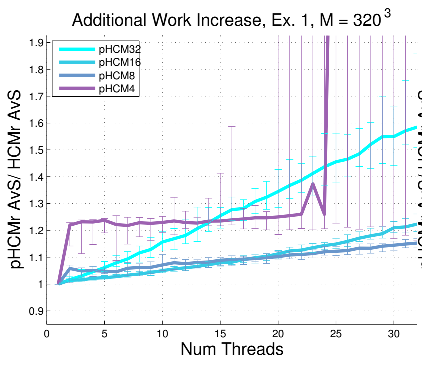
A



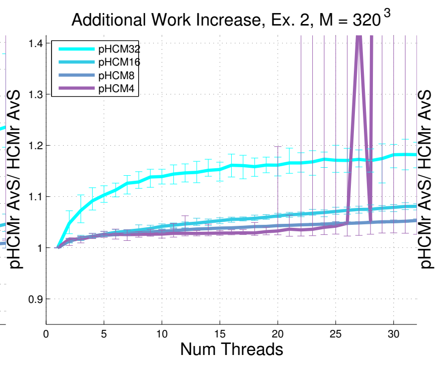
B



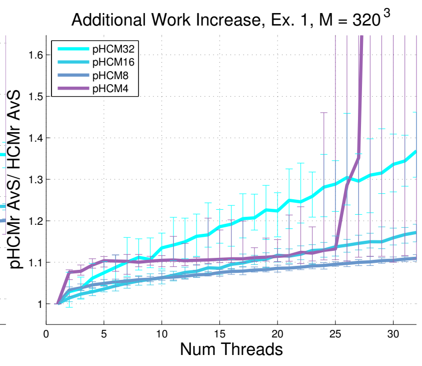
C



D



E



F

Figure 3.6: Overhead percentages and additional work in pHCMr for different P for the three examples, with $M = 320^3$. In figures 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.

10).

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 large amount of time outside the main loop just waiting for work. A good illustration of this is the pHCM32 curve in Fig. 3.7A and 3.7C. Since here $M = 128^3$, the cell decomposition for pHCM32 is only 4 cells per domain side; the scaling plateaus at a low number of threads.

3. *Effect of a strong causal structure.* The order of processing the cells is different for pHCM and HCM. On Ex. 1 (Figure 3.3A) there is a strict causal relationship

among cells, resulting in exactly 1 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.

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 2.6, suppose 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 [19]. This operation is orders of magnitude slower than simply updating a cached value without communication.

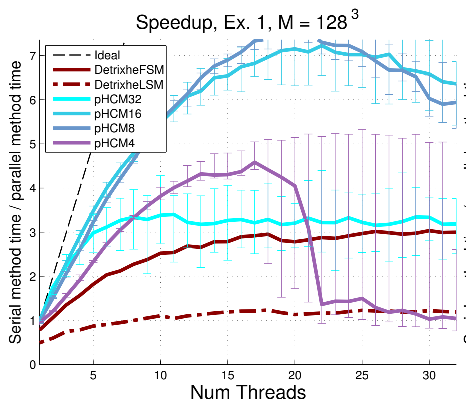
5. *Robustness of pHCM.* There is a possibility of the total amount of work increasing significantly if processor speeds vary. Suppose processor $\hat{p}$ is slow or has become slow and is processing a high-priority cell A. The other fast processors will not be able to do useful work on cells downwind from A. What's 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 Fig. 3.6D - 3.6F. The non-robust 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 here) 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 3.7 present the same information as in Figure 3.4, but for $M = 128^3$. The speedup of the parallel methods here is expectedly worse than for $M = 320^3$.

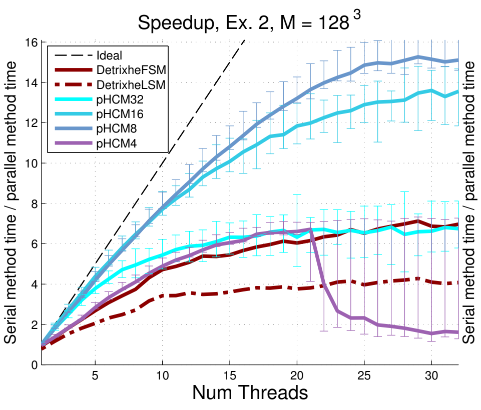
7. *Possible decrease in work.* The total amount of work performed by pHCM may also actually *decrease* compared to HCM in cases where the cell heuristic poorly predicts the dependency structure of the cells. See subsection 3.4.8.

8. *Parallel Sweeping.* As reported in [26], the algorithmic complexity of Detrixhe Sweeping is constant in the number of threads; for DFSM and DLSM, charts like $3.6D$-$3.6F$ 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 scalability is also sensitive to hardware properties of the specific platform; see also subsection 3.4.5. We note that the authors of [26] have also implemented their method in lower-level memory languages (MPI, CUDA) to alleviate this sensitivity.

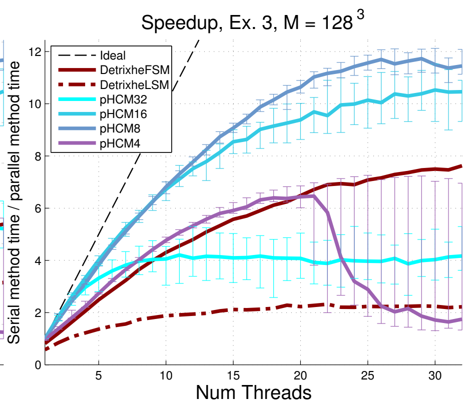
Choosing the *optimal* cell decomposition for a given problem and grid resolution remains a difficult problem even for the serial HCM. But luckily, as shown in Fig. 3.3 and in [16], a wide range of medium-sized cells exhibits good serial performance **and** parallelizes sufficiently well (Figures 3.4 and 3.7). 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.



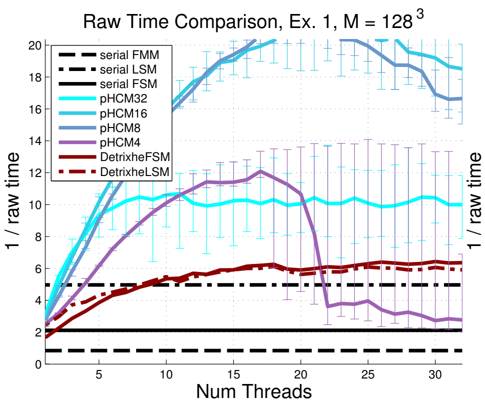
A



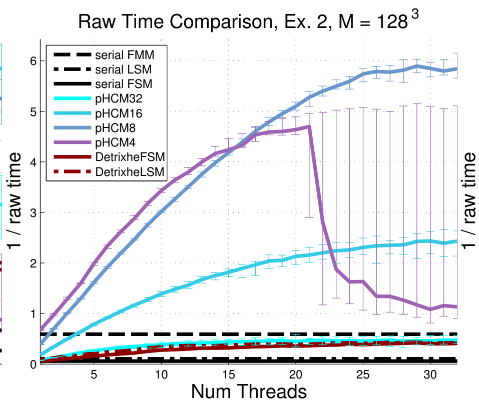
B



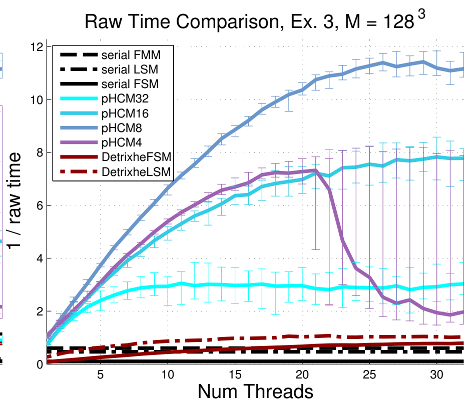
C



D



E



F

Figure 3.7: 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 $F = 1 + .99\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$.

3.4.3 Performance with "early sweep terminations"

All sweeping methods can be accelerated by stopping the iterations once 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 than the true solution of the discretized system (1.4). 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., [70]). 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.

All results reported in previous subsections were obtained with $\kappa = 0$, but on a computer with finite precision the iterations stop when the gridpoint value changes fall below the machine epsilon. I.e., for "double precision" computations this is equivalent to using $\kappa = 2^{-52} \approx 2.2 \times 10^{-16}$.

Here we repeat the same 3 examples but with $\kappa = 10^{-8}$ to force an early sweeping termination, keeping all other parameters the same as in subsections 3.4.1-3.4.2. As expected, this modification results in faster termination for FSM, LSM, DFSM, and DLSM (see Figure 3.8). For a fair comparison, in HCM/pHCM we now terminate the sweeping within a cell when the maximum change in a gridpoint's value is less than κ. We also add an additional condition on line 11 of Algorithm 8: if a gridpoint value changes by less than κ, then the procedure on line 12 will not be executed (i.e., the adjacent cell will not be marked for update). For most r values and on most examples, the number of "updates per gridpoint" done by HCMr decreases when $\kappa = 10^{-8}$ – yielding the expected decrease in CPU times. However, we have also observed a surprising (and as of now unexplained) work *increase* for HCM32 on Example 2 with $M = 320^3$.

For the parallel methods, the scaling is about the same (e.g., Figures 3.8D and 3.8F) or slightly worse (e.g. Figure 3.8E) than it was before with $\kappa = 0$. For pHCM this is not surprising, since there is effectively less work per cell. However, for most r values, the improvement in HCM still results in faster pHCM execution times (compared to those in Figure 3.4).

An experimental study of additional errors due to early termination can be found in section 2.3.

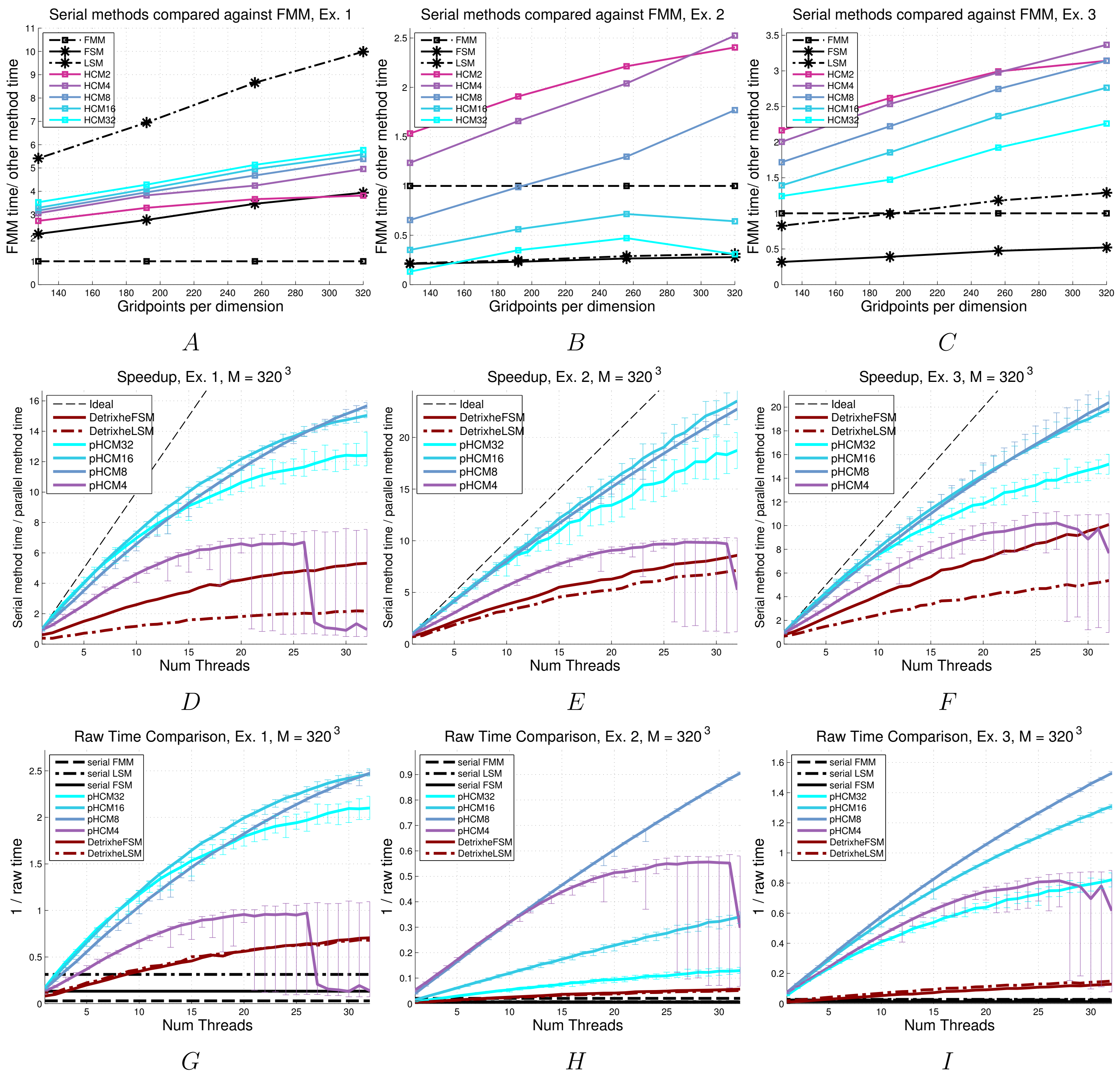


Figure 3.8: Early termination testing (subsection 3.4.3). Top row: performance of serial methods for different M; compare with Figure 3.3. Two bottom rows: scaling and performance for pHCM at $M = 320^3$; compare with Figure 3.4.

3.4.4 Performance with “single precision” data

In this subsection we repeat the same three experiments but storing/computing the numerical solution in single precision. This implementation uses “float” instead of “double” variables throughout the C++ code. The results are presented in Figure 3.9.

We would expect that in single precision a smaller data footprint would have advantages for high-level cache operations and scaling. This is mostly true, as illustrated best for DFSM and pHCM on Example 3 (Figure 3.9F). It is also natural to expect that switching to single precision should substantially decrease the total number of needed iterations to convergence, because the iterations stop when the maximum change in values is less than machine epsilon (i.e., we are effectively using $\kappa = 2^{-23} \approx 1.2 \times 10^{-7}$). Tables 3.5 and 3.6 are a side-by-side comparison of sweeping-convergence data for Example 2 with $M = 64^3$ under single and double precision. Based on Table 3.6, it is natural to expect that sweeping in single precision should converge in about 33 sweeps. Table 3.5 shows that this is **not** the case: 53 sweeps are in fact required for convergence. The reason for this discrepancy is that intermediate computations are also conducted in single precision. In fact, Table 3.4 shows that on Ex. 3 with $M = 320^3$, the number of sweeps to convergence is actually higher in single than in double precision. This helps explain the downward-sloping LSM curve in Figure 3.9C.

We note that Table 3.4 also shows a growth in the number of iterations-to-convergence with M for the sweeping methods on examples 2 and 3 in either single or double precision.

Table 3.4: Number of sweeps for different values of M in double and single precision.

		64^3	128^3	192^3	256^3	320^3
Ex. 1	double	9	9	9	9	9
	single	9	9	9	9	9
Ex. 2	double	69	99	131	164	191
	single	53	88	116	144	173
Ex. 3	double	42	58	77	107	121
	single	36	56	89	97	129

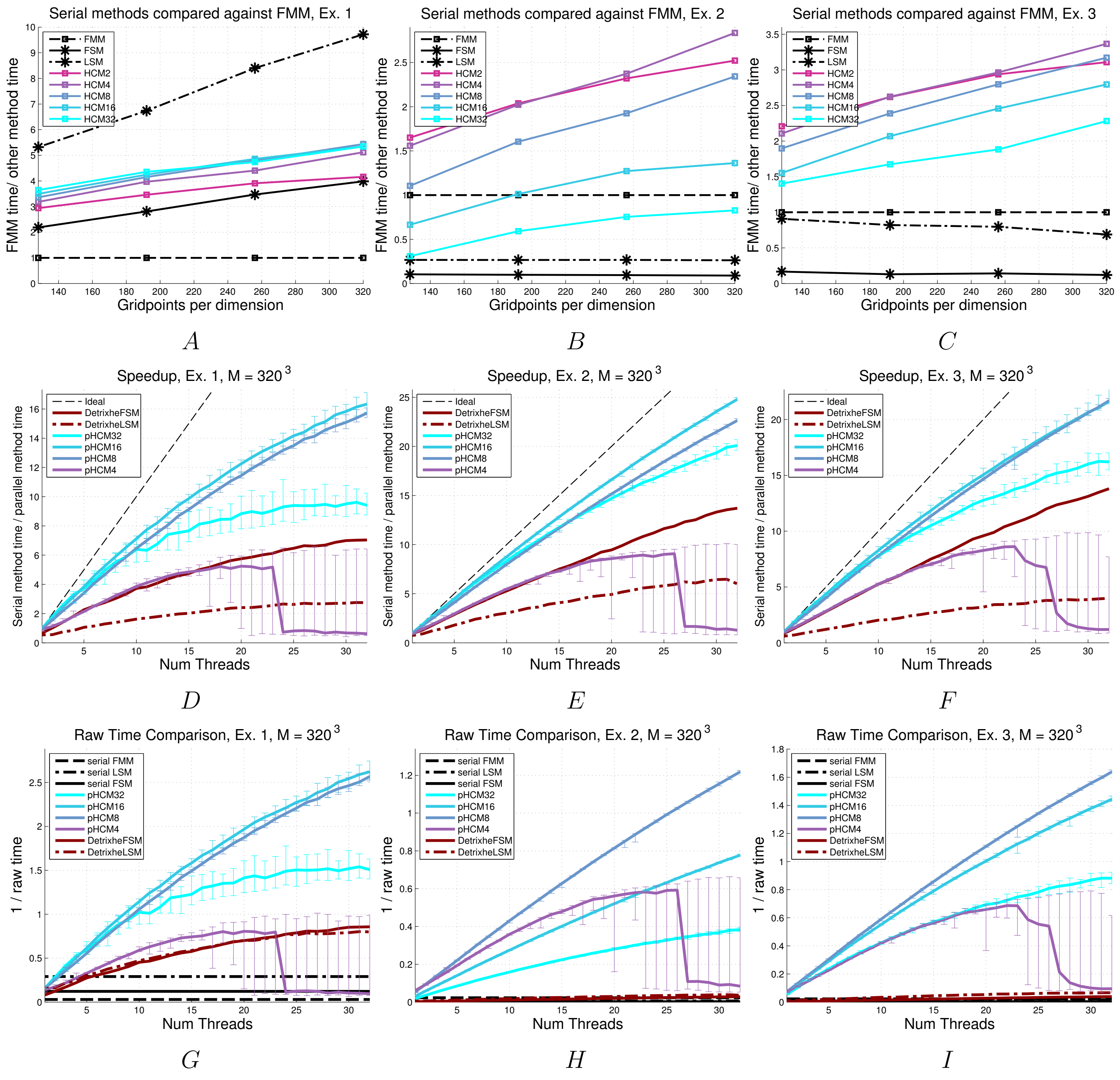


Figure 3.9: Single precision testing (subsection 3.4.4). Top row: performance of serial methods for different M; compare with Figure 3.3. Two bottom rows: scaling and performance for pHCM at $M = 320^3$; compare with Figure 3.4.

Table 3.5: Single precision

sweep #	**max change**	**% grid changing**
1	1e+09	15
2	1e+09	23.6
3	1e+09	43.6
4	1e+09	42.8
5	1e+09	75.4
6	0.258	74
7	0.242	72.1
8	0.156	73.4
9	0.00248	69.7
10	0.00155	69
11	0.00213	67.6
12	0.00151	67.9
13	0.00151	63.7
14	0.00147	60.8
15	0.00111	57.5
16	0.000641	55.3
17	0.000216	52.7
18	0.000104	48.7
19	0.000165	44.2
20	9.66e-05	40.5
21	0.0001	37.8
22	8.12e-05	31.4
23	5.51e-05	27.4
24	2.31e-05	22.8
25	6.74e-06	20.8
26	3.58e-06	16.9
27	4.71e-06	15.1
28	2.86e-06	12.6
29	3.28e-06	11.9
30	2.86e-06	8.41
31	2.8e-06	7.51
32	2.74e-06	6.24
33	2.44e-06	4.48
34	2.26e-06	3.54
35	1.85e-06	3.15
36	2.15e-06	2.4
37	2.38e-06	2.07
38	1.67e-06	1.14
39	1.67e-06	1.04
40	1.61e-06	0.668
41	1.79e-06	0.552
42	1.19e-06	0.335
43	1.73e-06	0.367
44	1.43e-06	0.146
45	1.13e-06	0.053
46	9.54e-07	0.0202
47	1.13e-06	0.0153
48	8.34e-07	0.013
49	8.94e-07	0.00687
50	4.17e-07	0.00305
51	9.54e-07	0.00305
52	2.98e-07	0.00114
53	0	0

Table 3.6: Double precision

sweep #	**max change**	**% grid changing**
1	1e+09	15
2	1e+09	28.7
3	1e+09	54.5
4	1e+09	56.3
5	1e+09	87.1
6	0.258	88.3
7	0.242	94.3
8	0.156	98
9	0.00248	86.9
10	0.00155	88.2
11	0.00213	93.7
12	0.00151	96.7
13	0.00151	85.3
14	0.00147	86.3
15	0.00111	90.8
16	0.00064	94.7
17	0.000217	84
18	0.000105	84.4
19	0.000165	86.9
20	9.68e-05	89.6
21	0.0001	79.3
22	8.11e-05	79.6
23	5.57e-05	80.1
24	2.25e-05	82.4
25	7.04e-06	73.9
26	2.74e-06	72.7
27	4.11e-06	70.7
28	1.52e-06	70.3
29	1.67e-06	62.4
30	8.78e-07	59.5
31	4.82e-07	55.8
32	1.82e-07	52.9
33	5.14e-08	47.8
34	1.7e-08	44.2
35	1.7e-08	40.6
36	5.07e-09	36.4
37	6.38e-09	32
38	1.19e-09	26.9
39	6.36e-10	23.4
40	3.96e-10	19.8
41	8.61e-11	17.1
42	2.3e-11	13.7
43	1.24e-11	12.3
44	7.12e-12	10.5
45	5.87e-12	8.93
46	6.39e-13	5.8
47	2.77e-13	5.2
48	2.26e-13	4.34
49	4.12e-14	3.01
50	1.11e-14	2.07
51	7.22e-15	1.83
52	3.77e-15	1.51
53	3.66e-15	1.43
...	...	...
68	5.55e-16	0.000381
69	0	0

3.4.5 Performance on a different computer architecture

The performance/scaling of parallel methods is often strongly affected by hardware features of a particular shared memory implementation. All parallel methods considered here scale better when the ratio of memory bandwidth to CPU speed is higher. In addition, the scaling is affected by the network topology of the cores. Stampede has "dual eight-core sockets," so communication between processors is necessarily slower when $P > 16$.

To explore the influence of these features, we repeat our main three examples on a different platform ("Octopus"): a computer with 8 Dual Core AMD Opteron 880 microprocessors running at 2.4 GHz, with 128 GB total RAM under the Scientific Linux v5.1 operating system. We have implemented all methods in C++ and compiled with the `-O2` level of optimization using the g++ compiler v4.2.1. The scaling was tested on up to 16 threads. All other experimental settings are exactly the same as described for "Stampede" at the beginning of section 2.3. The results are reported in Figure 3.10.

While the main conclusions are the same as in subsections 3.4.1-3.4.2, this change in hardware architecture yields noticeably different relative performance even for serial methods. We observe that FMM seems to benefit more from larger cache sizes than FSM and LSM do; thus, on Octopus the sweeping methods appear more competitive on large grids than in the previous tests on Stampede. The HCM2, whose algorithmic behavior is similar to FMM, is also less advantageous on Octopus, while HCM16 and HCM32 (whose computational cost is dominated by cell-sweeping) appear to be more advantageous here for large grids.

As for scaling (Figures 3.10D - 3.10F), all parallel methods seem to do much better on Octopus than on Stampede, even when only the first 16 threads are accounted for on Stampede. For example, on Octopus the pHCM8 median scaling curve has approximate slopes of .6, .92, and .83 on the three examples, while on Stampede the slopes up to $P = 16$ are approximately .5, .8, and .73. For pHCM4 on Octopus, the slopes are approximately .33, .73, and .67 (making pHCM4 very competitive on Octopus), while on Stampede the slopes up to $P = 16$ are only .27, .43, and .43. The scaling for DFSM not only improves on Octopus, but the slope of the scaling curve appears to be higher when the number of threads exceeds 8.

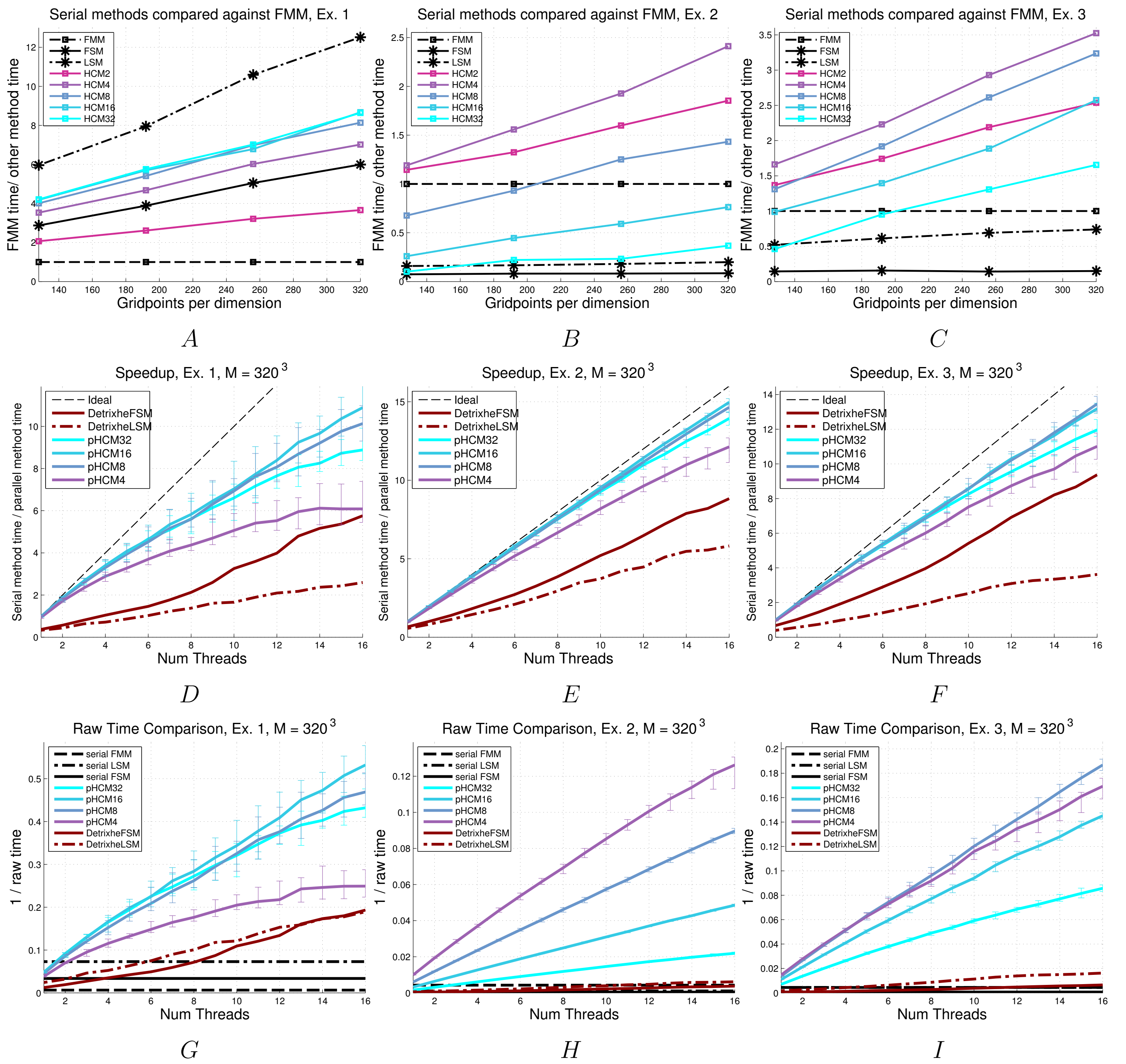


Figure 3.10: "Octopus" testing (subsection 3.4.5). Top row: performance of serial methods for different M; compare with Figure 3.3. Two bottom rows: scaling and performance for pHCM at $M = 320^3$; compare with Figure 3.4.

3.4.6 Additional examples: checkerboard speed functions

We consider two additional examples with periodic piecewise constant speed functions, which generalize the 2D checkerboard test problems of [16, 17]. These examples arise in the numerical computation of effective Hamiltonians in highly oscillatory problems; see also [48]. The goal is to determine the **homogenized speed profile**, i.e., the shape of the set of points that can be reached in a fixed amount of time when the number of checkers per domain side K goes to $+\infty$. Thus, each level set of the value function is simply the homogenized speed profile with perturbations of size $1/K$ superimposed on it.

Figure 3.11 has been obtained by taking $K = 11$ and $m = 1/h = 256^3$. Since we are interested in levels sets that are farther away from the origin, the rightmost plot is the most meaningful. It is also insightful to look at a larger checkerboard, such as

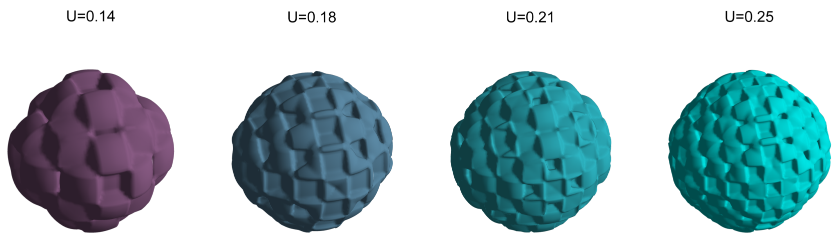


Figure 3.11: Some level sets of the value function of the 3D checkerboard example with $K = 11$

Figure 3.12 with $K = 41$. If one were to conjecture the homogenized speed profile

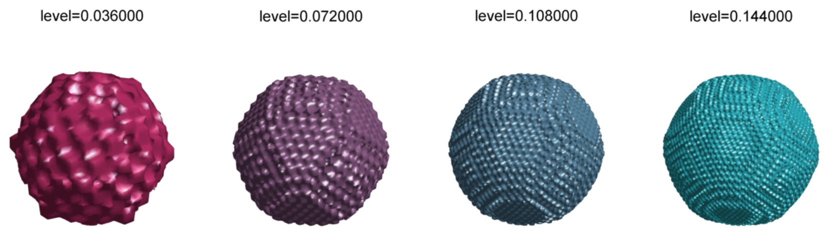


Figure 3.12: Some level sets of the value function of the 3D checkerboard example with $K = 41$. The speed ratio for these is 5:1, not 2:1.

based on Figure 3.12, it might be a truncated octahedron (see Figure 3.13). However, Figure 3.12 can be misleading because the number of gridpoints per checker side is on

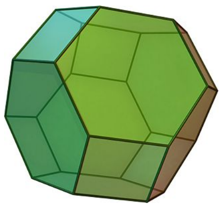


Figure 3.13: A truncated octahedron (from Wikipedia).

average only about 6 (versus 23 when $K = 11$), which produces numerical artifacts. A more careful analytical evaluation suggests that the homogenized speed profile is the convex hull of three circles with centers at the origin but each lying in a different coordinate plane (illustrated in Figure 3.14), which is closer to the rightmost level set of Figure 3.11. Note that while the pictures in Figure 3.12 have more checkers, the

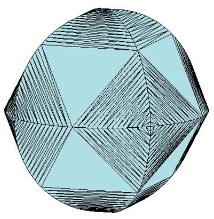


Figure 3.14: The convex hull of three circles (from Alex Vladimirsky).

ones in Figure 3.11 have more gridpoints per checker.

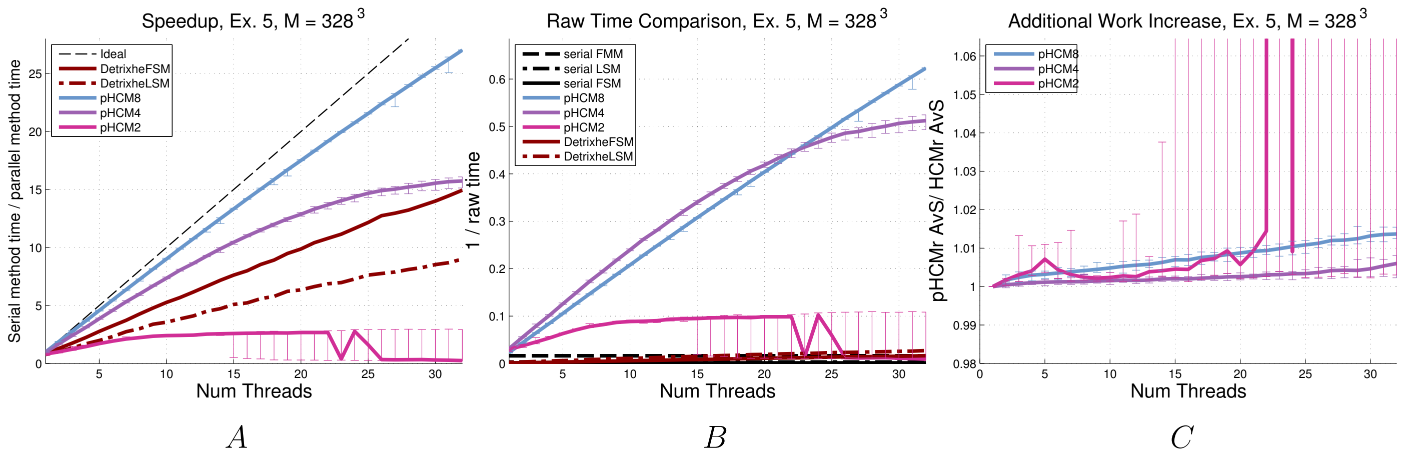


Figure 3.15: 3D Checkerboard example with $K = 41$ (subsection 3.4.6).

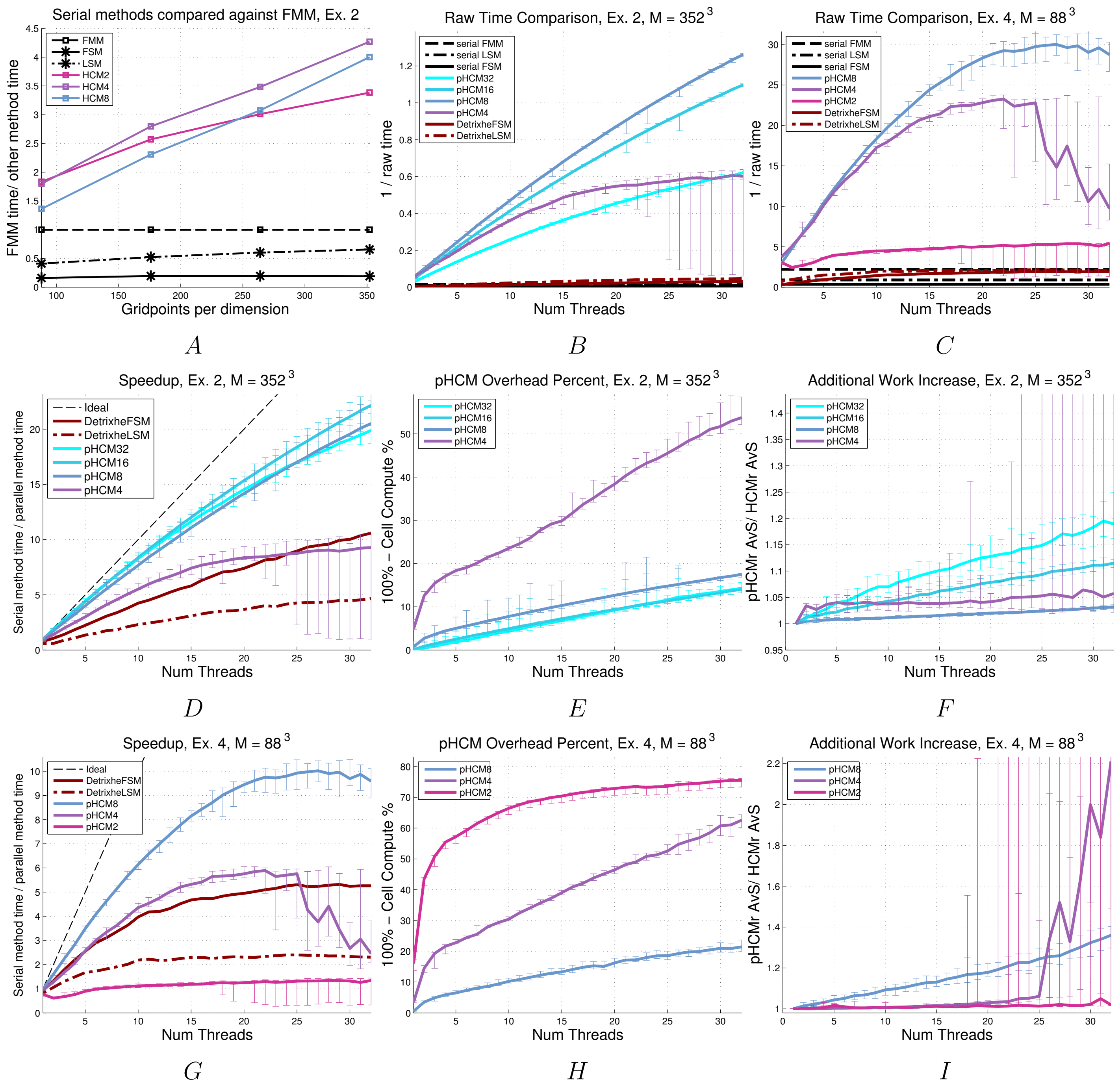


Figure 3.16: 3D Checkerboard example with $K = 11$ (subsection 3.4.6). Chart A is a comparison of serial methods for different M; compare with Figure 3.3. Scaling/performance for parallel methods with $M = 352^3$ is shown in charts B and D; compare with Figure 3.4. Parallel overhead and additional work with $M = 352^3$ are shown in charts E and F; compare with Figure 3.6. The same information for a coarser grid with $M = 88^3$ is shown in charts C and $G - I$.

Suppose that the unit cube $\overline{\Omega}$ is split into K^3 smaller cubes (or "3D checkers") of edge length $1/K$. Suppose these smaller cubes are divided into two types ("black" and "white") so that no two cubes of the same type have a face in common. The speed function F is defined to be 2 on black cubes and 1 on white cubes[2]. The exit set Q again consists of a single point in the center of $\overline{\Omega}$ and, given the even number of gridpoints, the set Q' consists of 8 gridpoints.

We conducted experiments on two different 3D checkerboards, with $K = 11$ and $K = 41$. The respective performance/scaling results are summarized in Figures 3.16 and 3.15. As observed in [16], HCM performs very well on problems where the discontinuities of the speed function align with cell boundaries. The scaling trends for $K = 11$ are most similar to those observed in Example 2, where the speed function is also highly oscillatory. For $K = 41$, the speedup for pHCM4 is surprisingly large and stable.

[2]We can also take $F = 2$ on the boundary of the cubes. Computationally, the issue does not arise since our gridsizes are selected to ensure that each gridpoint is in the interior of either black or white cube.

3.4.7 Additional examples: maze speed functions

Suppose the domain contains four concentric spherical "barriers" of thickness t that have openings on alternating sides. Specifically, $\overline{\Omega} = [-1,1]^3$, $Q = \{0,0,0\}$, and $F = 1$ outside the set of (slowly permeable) barriers and .001 inside, with the barriers described as follows:

$$A_1 = \{\boldsymbol{x} | .3 < |\boldsymbol{x}| < .3 + t\} \backslash \big(\{x^2 + y^2 < w\} \cap \{z < 0\}\big)$$
$$A_2 = \{\boldsymbol{x} | .5 < |\boldsymbol{x}| < .5 + t\} \backslash \big(\{x^2 + y^2 < w\} \cap \{z > 0\}\big)$$
$$A_3 = \{\boldsymbol{x} | .7 < |\boldsymbol{x}| < .7 + t\} \backslash \big(\{x^2 + y^2 < w\} \cap \{z < 0\}\big)$$
$$A_4 = \{\boldsymbol{x} | .9 < |\boldsymbol{x}| < .9 + t\} \backslash \big(\{x^2 + y^2 < w\} \cap \{z > 0\}\big)$$

where $t = 1/12$ and $w = 1/10$. This is a modified version of an example from [26], where the barriers considered were impermeable (i.e., with $F = 0$). Unlike the checkerboard examples, here the discontinuities of the speed function **do not** align with the cell boundaries in any special way. In that sense, this problem is also analogous to the second "comb maze" example from section 2.4.

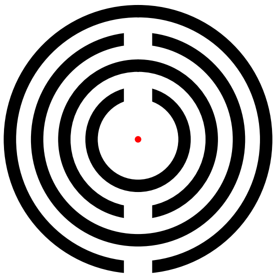


Figure 3.17: A cross section of the speed function for the Permeable Shell Maze example. $F = .001$ in the barriers (black) and $F = 1$ outside.

First, Figure 3.19A shows HCMr is very effective for each r. One of the drawbacks of the original version of HCM [16] was precisely the slow convergence on problems of this type. The greatly improved performance shown here is due to the use of the

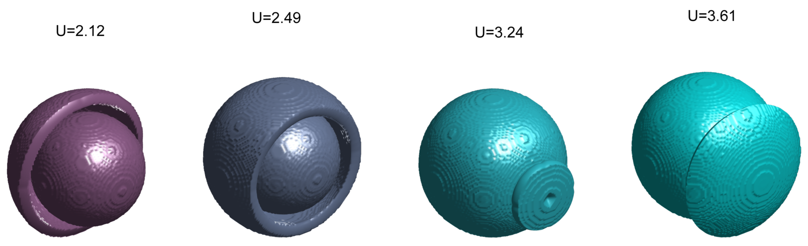


Figure 3.18: Some level sets of the value function of the Permeable Shell Maze

new cell value heuristic (equation (3.1)).

The pHCM's speedup (Fig. 3.19 *B*), on the other hand, is significantly lower here (while for DFSM the speedup here is still typical). We believe this is due to certain level sets of the value function getting "pinched" at the locations where there is a hole in one of the barriers. If the ordering of non-barrier cells is strictly causal, this means that, at several stages of the algorithm, there is only one cell upon which all still-to-be-computed cells depend. (For example, since $w = .1$, in pHCM16 at most one cell will fit through the hole in each barrier.) Furthermore, as mentioned in section 3.4.2, pHCM sees an increase in work over HCM for problems with a strictly causal cell ordering. However, due to the large-enough advantage that HCM holds over other serial methods, the performance of pHCM is still significantly better than that of DFSM/DLSM; see Fig. 3.19 *C*.

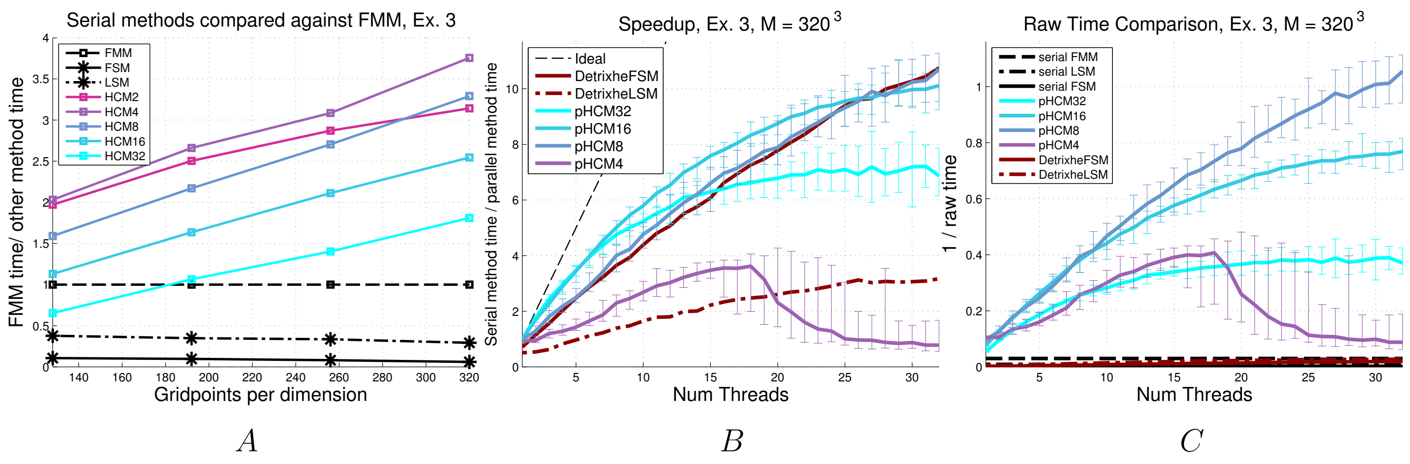


Figure 3.19: Permeable Shell Maze example: serial M-scaling comparison (A), parallel scaling at $M = 320^3$ (B), and comparison of all methods at $M = 320^3$ (C).

3.4.8 Other cell values

Interestingly, pHCM seems less influenced by the particular choice of cell value heuristic than the serial HCM. As noted in section 3.4.2, if the cell value is a very good predictor of information flow, pHCM will usually see an increase in the total amount of work by not being able to process cells exactly in their causal ordering. However, pHCM can also partially mitigate the effect of poor cell values; instead of the cell with the lowest value always being processed, we can think of pHCM as simultaneously processing cells in the lowest range of values. If it is always the case that the true "most upwind" cell has a value in that range, then pHCM will need fewer heap removals than HCM. Furthermore, neighboring cells that are simultaneously processed may be able to resolve their interdependencies, which would also reduce the total number of heap removals and the number of sweeps per cell (see Figure 3.21A).

We have tested both HCM and pHCM with several other cell value heuristics, including the one from the section 2.2.2. We rewrite it here for convenience in Figure 3.20 and equation (3.2), supposing A and B are two adjacent cells, with A currently processed. As before, we define $A_{new} \subset N(B) \bigcap A$ as the set of newly updated inflowing gridpoints of A along the relevant cell border (colored in blue in Figure 3.20).

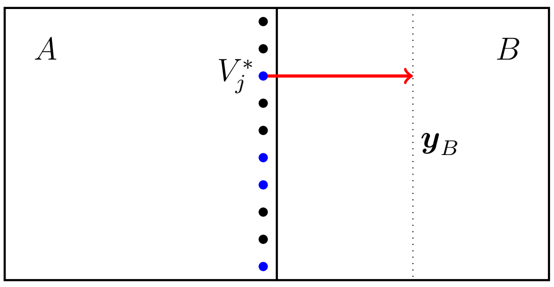


Figure 3.20: When cell A tags B as downwind, the value computed for B is an approximation to the value of a point along a center axis of B; see equation (3.2).

$$V_{max} \leftarrow \max_{i \in A_{new}} V(\boldsymbol{x}_i) \qquad D \leftarrow \frac{h^c + h}{2}$$

$$\widetilde{V}^c(B) \leftarrow V_{max} + \frac{D}{F(\boldsymbol{y})} \tag{3.2}$$

$$V^c(B) \leftarrow \min(V^c(B), \widetilde{V}^c(B))$$

See Figure 3.20 for a geometric interpretation. For consistency with [16], we tested this heuristic *without* resetting cell values to $+\infty$ each time a cell is processed (see line 5 of Algorithm 7 and line 11 in Algorithm 10). We observed that

- For serial methods, formula (3.1) results in better performance than formula (3.2) if r is large.
- For smaller r the median raw time and scaling are better when using (3.2).
- For parallel methods, (3.2) leads to improved scaling for larger cells. E.g., Figure 3.21A illustrates how pHCM32 performs noticeably *less* work (measured in terms of AvS) than HCM32, though the raw time actually increases compared to heuristic (3.1).

However, the main motivation for using the new cell heuristic (3.1) is that formula (2.2) leads to very bad performance on problems where discontinuities in the speed function are not aligned with cell boundaries. E.g., for the example of subsection 3.4.7 with $M = 64^3$, HCM8 yields 20.4 average sweeps per cell with formula (3.1) compared to 8366 average sweeps per cell with formula (3.2). Further evidence is demonstrated by re-running the 2D comb maze examples from the previous chapter using the original and improved cell values.

- The performance comparison on Stampede versus that on the older machine is qualitatively similar.
- Unlike before, FHCM and HCM perform very well on the 8-wall comb maze (where the discontinuities of the maze are misaligned with the cell boundaries) using the new cell value (3.1).
- As before, FHCM and HCM still perform very well on the 4-wall comb maze (where the discontinuities of the maze align with the cell boundaries) using the new cell value.

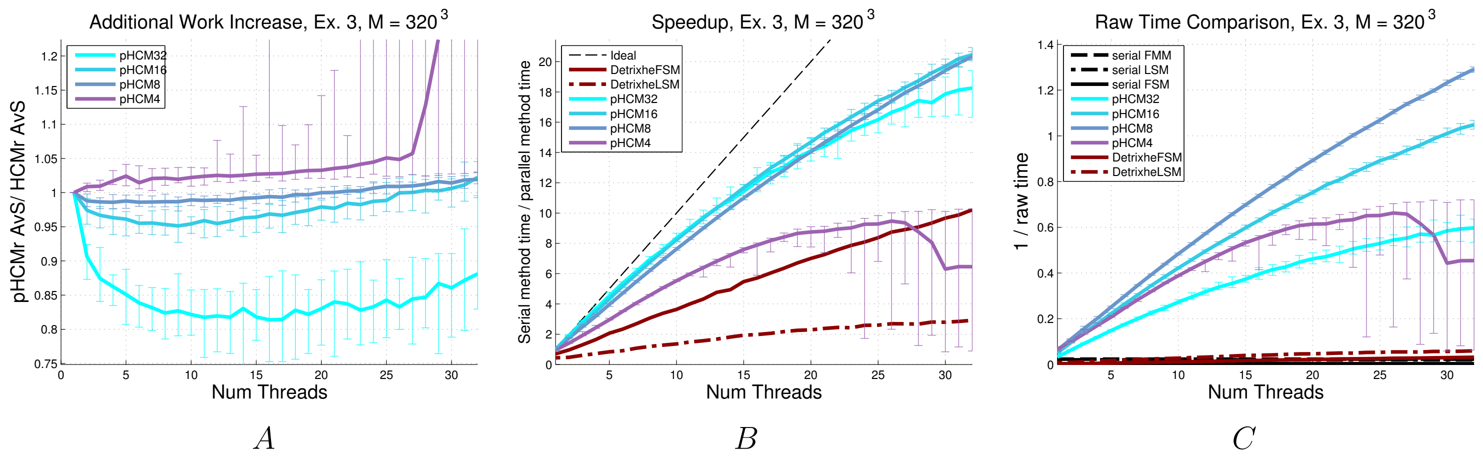


Figure 3.21: An example of pHCM performing less work than HCM for the cell value given by equation (3.2) on example 3. Compare with Figures 3.4C, 3.4F and 3.6F, and note the difference in scaling in pHCM32.

Table 3.7: Performance/convergence results for a 4 wall comb maze example using the **original** cell value (3.2) (tested on Stampede).

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	5.9449e-02	1.4210e-02	0.57	0.46	0.19	12

METHOD	**TIME**	$\boldsymbol{\mathcal{R}}$	$\boldsymbol{\rho}$	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	0.12				1.151	3.971	
HCM 44 × 44 cells	0.11				1.078	3.724	
HCM 88 × 88 cells	0.12				1.042	3.598	
HCM 176 × 176 cells	0.12				1.019	3.516	
HCM 352 × 352 cells	0.14				1.010	3.482	
HCM 704 × 704 cells	0.24				1.003	3.454	
FHCM 22 × 22 cells	0.08	1.0460	1.0000	1.0000	1.151	1.618	85.5
FHCM 44 × 44 cells	0.08	1.0191	1.0000	1.0000	1.078	1.310	92.6
FHCM 88 × 88 cells	0.08	1.0085	1.0000	1.0000	1.042	1.160	96.1
FHCM 176 × 176 cells	0.08	1.0073	1.0000	1.0000	1.019	1.078	98.3
FHCM 352 × 352 cells	0.10	1.0002	1.0000	1.0000	1.010	1.042	99.3
FHCM 704 × 704 cells	0.20	1.0000	1.0000	1.0000	1.003	1.017	100.0
FMSM 22 × 22 cells	0.06	1.1659	1.0000	1.0000		1.436	
FMSM 44 × 44 cells	0.05	1.0706	1.0000	1.0018		1.218	
FMSM 88 × 88 cells	0.04	1.0821	1.0000	1.0018		1.110	
FMSM 176 × 176 cells	0.05	1.0468	1.0000	1.0008		1.055	
FMSM 352 × 352 cells	0.07	1.0378	1.0000	1.0004		1.028	
FMSM 704 × 704 cells	0.16	1.0064	1.0000	1.0001		1.014	

Table 3.8: Performance/convergence results for a 4 wall comb maze example using the **improved** cell value (3.1) (tested on Stampede).

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	5.9449e-02	1.4210e-02	0.30	0.41	0.19	12

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	0.11				1.167	3.996	
HCM 44 × 44 cells	0.12				1.086	3.742	
HCM 88 × 88 cells	0.12				1.043	3.600	
HCM 176 × 176 cells	0.12				1.019	3.517	
HCM 352 × 352 cells	0.13				1.008	3.476	
HCM 704 × 704 cells	0.21				1.003	3.454	
FHCM 22 × 22 cells	0.08	1.0000	1.0000	1.0000	1.167	1.626	85.7
FHCM 44 × 44 cells	0.09	1.0129	1.0000	1.0000	1.086	1.319	92.6
FHCM 88 × 88 cells	0.09	1.0130	1.0000	1.0000	1.043	1.159	96.1
FHCM 176 × 176 cells	0.08	1.0021	1.0000	1.0000	1.019	1.078	98.3
FHCM 352 × 352 cells	0.09	1.0020	1.0000	1.0000	1.008	1.038	99.3
FHCM 704 × 704 cells	0.18	1.0000	1.0000	1.0000	1.003	1.017	100.0
FMSM 22 × 22 cells	0.05	1.1659	1.0000	1.0000		1.436	
FMSM 44 × 44 cells	0.05	1.0706	1.0000	1.0018		1.218	
FMSM 88 × 88 cells	0.05	1.0821	1.0000	1.0018		1.110	
FMSM 176 × 176 cells	0.05	1.0468	1.0000	1.0008		1.055	
FMSM 352 × 352 cells	0.06	1.0378	1.0000	1.0004		1.028	
FMSM 704 × 704 cells	0.15	1.0064	1.0000	1.0001		1.014	

Table 3.9: Performance/convergence results for an 8 wall comb maze example using the **original** cell value (3.2) (tested on Stampede).

Grid Size	$\boldsymbol{L_\infty}$ **Error**	$\boldsymbol{L_1}$ **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408×1408	6.5644e-02	1.6865e-02	0.45	1.04	0.34	20

METHOD	**TIME**	$\boldsymbol{\mathcal{R}}$	$\boldsymbol{\rho}$	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22×22 cells	0.25				2.866	9.461	
HCM 44×44 cells	0.68				6.995	22.852	
HCM 88×88 cells	1.18				12.541	41.085	
HCM 176×176 cells	0.63				5.669	18.949	
HCM 352×352 cells	0.33				2.466	8.350	
HCM 704×704 cells	0.24				1.027	3.560	
FHCM 22×22 cells	0.21	1.4247	1.0000	1.0000	2.930	3.779	89.8
FHCM 44×44 cells	0.50	1.2133	1.0000	1.0000	7.092	7.814	96.1
FHCM 88×88 cells	0.83	1.0634	1.0000	1.0000	12.742	13.107	98.9
FHCM 176×176 cells	0.42	1.1962	1.0000	1.0000	5.737	5.891	99.1
FHCM 352×352 cells	0.24	1.0095	1.0000	1.0000	2.476	2.552	99.1
FHCM 704×704 cells	0.22	1.0000	1.0000	1.0000	1.027	1.056	100.0
FMSM 22×22 cells	0.08	604.49	5.0344	35.126		1.783	
FMSM 44×44 cells	0.06	228.29	3.1529	19.442		1.385	
FMSM 88×88 cells	0.06	313.01	2.7666	6.4608		1.195	
FMSM 176×176 cells	0.06	381.98	1.7374	5.5944		1.097	
FMSM 352×352 cells	0.07	45.397	1.1718	2.0506		1.049	
FMSM 704×704 cells	0.18	23.303	1.1738	1.3536		1.024	

Table 3.10: Performance/convergence results for an 8 wall comb maze example using the **improved** cell value (3.1) (tested on Stampede).

Grid Size	L_∞ **Error**	L_1 **Error**	**FMM Time**	**FSM Time**	**LSM Time**	**# Sweeps**
1408 × 1408	6.5644e-02	1.6865e-02	0.31	0.84	0.32	20

METHOD	**TIME**	$\mathcal{R}$	ρ	**R**	**AvHR**	**AvS**	**Mon %**
HCM 22 × 22 cells	0.12				1.333	4.713	
HCM 44 × 44 cells	0.12				1.162	4.050	
HCM 88 × 88 cells	0.12				1.078	3.756	
HCM 176 × 176 cells	0.12				1.037	3.610	
HCM 352 × 352 cells	0.13				1.015	3.533	
HCM 704 × 704 cells	0.21				1.005	3.494	
FHCM 22 × 22 cells	0.09	1.4253	1.0000	1.0000	1.349	2.260	75.3
FHCM 44 × 44 cells	0.10	1.4253	1.0000	1.0000	1.169	1.761	80.2
FHCM 88 × 88 cells	0.08	1.4253	1.0000	1.0000	1.080	1.390	89.6
FHCM 176 × 176 cells	0.08	1.2636	1.0000	1.0000	1.038	1.187	95.2
FHCM 352 × 352 cells	0.10	1.9138	1.0000	1.0000	1.016	1.087	97.9
FHCM 704 × 704 cells	0.19	1.5700	1.0000	1.0000	1.005	1.034	100.0
FMSM 22 × 22 cells	0.08	604.49	5.0344	35.126		1.783	
FMSM 44 × 44 cells	0.06	228.29	3.1529	19.442		1.385	
FMSM 88 × 88 cells	0.05	313.01	2.7666	6.4608		1.195	
FMSM 176 × 176 cells	0.06	381.98	1.7374	5.5944		1.097	
FMSM 352 × 352 cells	0.07	45.397	1.1718	2.0506		1.049	
FMSM 704 × 704 cells	0.16	23.303	1.1738	1.3536		1.024	

Tables 3.7 – 3.10 provide additional evidence that

- HCM and FHCM with the improved cell value do no worse than when using original cell value in 2D on problems where the discontinuities of the speed function align with the cell boundaries

- HCM and FHCM with improved cell value perform significantly better in 2D on problems where the discontinuities of the speed function do not align with the cell boundaries.

3.4.9 An application from geophysics

A common application of the Eikonal equation is to solve inverse problems from geophysics; that is, given the traveltimes of waves at certain locations, what is the speed function? Tackling this problem requires a fast forward solver. Here we apply HCM/pHCM to a 3D model commonly used for benchmarking methods used in seismic imaging. This model was produced by the Society of Exploration Geophysicists (SEG) and the European Association of Geoscientists and Engineers (EAGE). We note that it is a common benchmark beyond Eikonal solvers, but here we compare only the same methods described in the rest of this chapter.

Some of the difficulties with the SEG/EAGE salt model are:

- large dataset- $208 \times 672 \times 672 = 93929472 \approx 10^8$ gridpoints
- actual data without subsampling or smoothing
- irregularly-shaped discontinuities on several different scales

Figure 3.22 shows some level sets of F for this problem. The main feature is that there is an irregularly-shaped salt pocket in the center of the domain where the speed is approximately 4 times faster than on the rest of the domain.

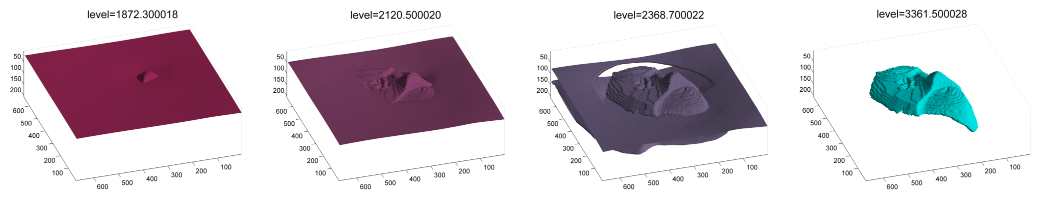


Figure 3.22: Various level sets of the speed function of the SEG/ EAGE salt model

Table 3.11: Timing data for serial methods with Q as a single point in the center of the salt pocket and as a corner of the domain outside the salt.

	FMM	LSM	LSM early	HCM8	HCM16
Center	142	260	152	23.2	26.8
Corner	125	718	204	16.6	22.2

Table 3.12: Timing data for the parallel methods with $P = 16$ and Q as a single point in the center of the salt pocket and as a corner of the domain outside the salt.

	DFSM	DFSM early	DLSM	DLSM early	pHCM8	pHCM16
Center	126	29.2	69.0	32.7	1.55	1.70
Corner	271	31.4	174	36.6	1.26	1.73

HCM does very well, and the pHCM scaling is nearly ideal. The performance of pHCM8 compared to the other methods is 90-99 times faster than FMM and 98-162 times faster than LSM with an early sweep termination threshold of $\kappa = 10^{-8}$ (HCM/pHCM still used $\kappa = 0$).

CHAPTER 4

ERROR ANALYSIS OF APPROXIMATE METHODS FOR ADVECTION EQUATIONS WITH CONSTANT COEFFICIENTS

Until now we were considering the Eikonal equation discretized on grid X over the domain $\bar{\Omega}$. The viscosity solution u corresponded to the min time for reaching $Q \subset \partial\Omega$ from $\boldsymbol{x}$, where the minimization was performed over all $\bar{\Omega}$-constrained trajectories. On the numerical level, the grid function U was computed by specifying the boundary conditions on $Q \cap X$ and solving the discrete system

$$\left(\max\left(D_{ij}^{-x}U,\, -D_{ij}^{+x}U,\, 0\right)\right)^2 \;+\; \left(\max\left(D_{ij}^{-y}U,\, -D_{ij}^{+y}U,\, 0\right)\right)^2 \;=\; \frac{1}{F_{ij}^2} \tag{4.1}$$

on $X \backslash Q$. To interpret (4.1) on $X \cap (\partial\Omega \backslash Q)$, the missing gridpoints outside of $\bar{\Omega}$, were assumed to have the values of $U = +\infty$ as boundary conditions.

Several other numerical methods for the Eikonal PDE [20, 15] rely on solving (4.1) on a subset of the grid $\hat{X} \subset X$ also containing $Q \cap X$. We will call that numerical solution $\hat{U}$. For a specific gridpoint $S \in \hat{X}$, $\hat{X}$ generally does not contain the entire dependency graph $G(S)$ (defined in section 1.3 and reintroduced section 4.1). As a result, $\hat{U}$ is typically larger than U, and deriving estimates and/or bounds on $\hat{U}(S) - U(S)$ is an important practical question, particularly if considered under grid refinement as $h \to 0$. See section 4.6 and the Figures 4.15 and 4.16 for examples.

A related and slightly more general question is the effect of specifying additional/artificial boundary conditions q_i for all $\boldsymbol{x}_i$ in the set $\Xi = \{\boldsymbol{x} \in X \backslash \hat{X} | N(\boldsymbol{x}) \cap \hat{X} \neq \emptyset\}$. We will use $\bar{U}$ to refer to the latter numerical solution on $\hat{X}$ (implicitly dependent on the particular q_i's chosen on Ξ.) The monotonicity of (4.1) ensures that $\bar{U} \geq U$ on $\hat{X}$ as long as $q_i \geq U_i$ for all $\boldsymbol{x}_i \in \Xi$. Similarly, $\hat{U} \geq \bar{U}$ on $\hat{X}$, with the equality guaranteed if we choose all $q_i = +\infty$.

It is easy to show the maximum principle: $|\bar{U} - U|$ on $\hat{X}$ is bounded by its maximum value on Ξ. However, sharper upper bounds suitable for studying the behavior as $h \to 0$ remain a challenge for the general Eikonal case. In this chapter we study these issues for the (time-implicit) upwind discretization of a much simpler linear advection-reaction PDE with constant coefficients.

$$\begin{aligned} w_t + a w_x &= b \qquad (x,t) \in (0,\infty)^2 \\ w &= q \qquad x = 0 \text{ or } t = 0 \end{aligned} \tag{4.2}$$

To mimic the Eikonal equations with a similar characteristic structure, we choose the constants $a = \dfrac{\beta}{1-\beta}$ and $b = \dfrac{\sqrt{1 + (\frac{\beta}{1-\beta})^2}}{F}$ for $0 < \beta < 1$ and $F > 0$.

An asymptotic estimate for $|\hat{W} - W|$ is derived in sections 4.2 and 4.3, and an upper bound is proven for a specific $\hat{X}$ in section 4.4. We then discuss connections between the Eikonal and advection in section 4.5 and numerical methods that solve (4.1) on $\hat{X}$ in section 4.6.

4.1 Discretization, dependency digraph, and sensitivities

We make the time axis vertical and the space axis horizontal. We take $\Delta t = \Delta x = h$ and denote $\boldsymbol{x} = (x,t)$. We denote the value of a gridpoint $\boldsymbol{x}_i^n = (ih, nh)$ as W_i^n, or simply W where not ambiguous. We denote the horizontal and vertical upwind neighbors as $\boldsymbol{x}_H = \boldsymbol{x}_{i-1}^n$ and $\boldsymbol{x}_V = \boldsymbol{x}_i^{n-1}$, with values W_H and W_V respectively. Since the goal is to mimic the upwind discretization of the Eikonal PDE, we use a time-implicit upwind discretization as in [66] instead of usual upwind finite differences for advection equations:

$$\frac{W - W_V}{\Delta t} + \frac{\beta(W - W_H)}{(1-\beta)\Delta x} = \frac{\sqrt{1 + (\frac{\beta}{1-\beta})^2}}{F}$$

Thus, the numerical update at a gridpoint is:

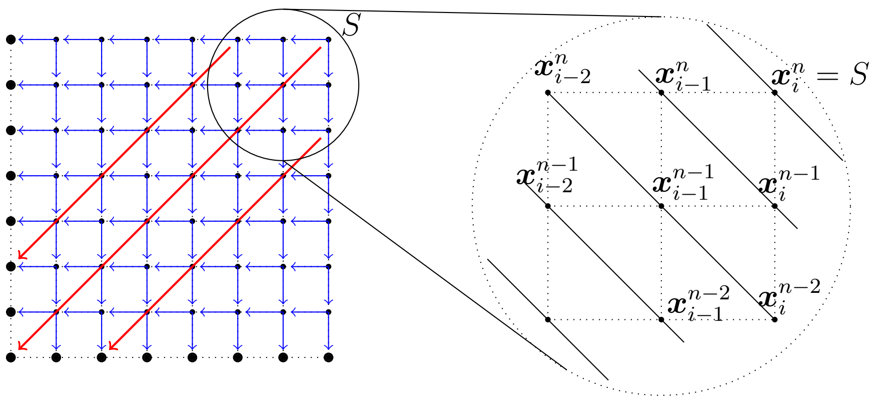
$$W_i^n = \mathcal{H}(W_H, W_V) = \begin{cases} \beta W_H + (1-\beta)W_V + h\frac{\sqrt{\beta^2 + (1-\beta)^2}}{F} & \boldsymbol{x} \in X \backslash Q \\ q_i^n & \boldsymbol{x} \in Q \end{cases} \tag{4.3}$$

where $Q = \{x = 0\} \cup \{t = 0\}$.

We emphasize that all analysis from this section up through section 4.4 is only for the advection PDE with constant coefficients (4.2). In future work we intend to use our analysis of (4.3) as motivation for studying similar questions for the upwind scheme for Eikonal equations. More details about the connection between the two schemes are given in section 4.5.

Suppose system (4.3) has been solved. Because of the special structure of (4.3) (in particular that $0 < \beta < 1$), each interior $\boldsymbol{x}_i$ depended on exactly two of its neighboring gridpoints (the one to the south and the one to the west). As in sections 1.3 and 2.2, the dependency digraph G is defined as the graph containing vertices $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M$ and directed edges from $\boldsymbol{x}_i$ to $\boldsymbol{x}_j$ indicating that W_j was needed to compute W_i. We refer to $G(\boldsymbol{x})$ as the subgraph of G containing only the computational domain of dependence of the gridpoint $\boldsymbol{x}$. The dependency graph of a gridpoint S is illustrated in Figure 4.1.

Let $\bar{W}(S)$ be the solution of (4.3) on a new domain $\hat{X} \subset X$ with additional

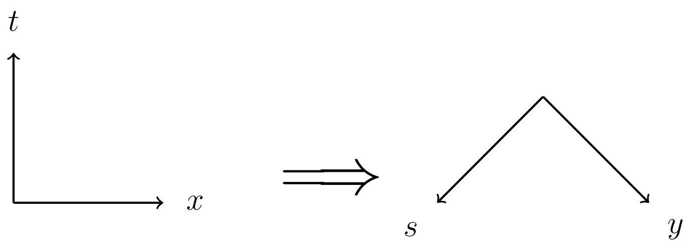


S
$\boldsymbol{x}_{i-2}^{n}$
$\boldsymbol{x}_{i-1}^{n}$
$\boldsymbol{x}_{i}^{n} = S$
$\boldsymbol{x}_{i-2}^{n-1}$
$\boldsymbol{x}_{i-1}^{n-1}$
$\boldsymbol{x}_{i}^{n-1}$
$\boldsymbol{x}_{i-1}^{n-2}$
$\boldsymbol{x}_{i}^{n-2}$

Figure 4.1: Since the update of each gridpoint comes from the southwest quadrant, the dependency digraph of S is a rotation of the grid. Characteristics are drawn in red and boundary conditions are drawn in black.

boundary conditions specified in a set Ξ. Suppose $\hat{X}$ is the same as X but with the single additional boundary condition $\Xi = \{\boldsymbol{x}_H\}$. Since $\mathcal{H}$ is a linear function of the neighboring values, it is clear that if $q_H = W_H + \epsilon$, then $\bar{W} = W + \frac{\partial \mathcal{H}}{\partial q_H}\epsilon = W + \beta\epsilon$. Similarly, if $\Xi = \{\boldsymbol{x}_V\}$ and $q_V = W_V + \epsilon$, then $\bar{W} = W + (1-\beta)\epsilon$.

To begin analyzing the sensitivity of $W(S)$ to a change in a value farther down $G(S)$, it will be convenient to relabel gridpoints based on their location within $G(S)$, with S relabeled as $\boldsymbol{x}_0^0$. In this setting, $(x,t) \to (y,s)$, with the new time axis pointing towards the southwest and the new space axis pointing towards the southeast:



We refer to a gridpoint's "level" (superscript ν) as **half** the number of transitions it is away from S. These dependency graph coordinates are illustrated in Figure 4.2.

$W(S)$ can be written as a function of the values on a given level (written in

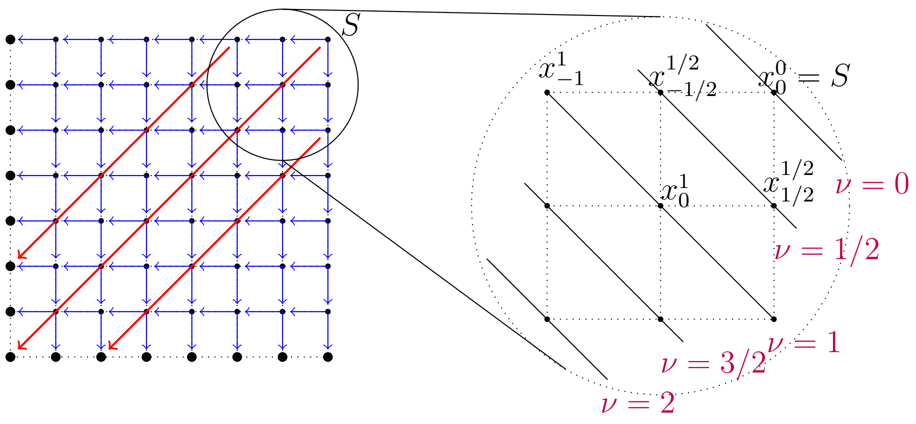


Figure 4.2: Same as Figure 4.1 but with gridpoints rewritten in their $G(S)$-coordinates.

dependency graph coordinates):

$$\begin{aligned} W(S) &= \mathcal{H}(W^{1/2}_{-1/2}, W^{1/2}_{1/2}) \\ &= \mathcal{H}(\mathcal{H}(W^1_{-1}, W^1_0), \mathcal{H}(W^1_0, W^1_1)) \\ &= \mathcal{H}(\mathcal{H}(\mathcal{H}(W^{3/2}_{-3/2}, W^{3/2}_{-1/2}), \mathcal{H}(W^{3/2}_{-1/2}, W^{3/2}_{1/2})), \mathcal{H}(\mathcal{H}(W^{3/2}_{-1/2}, W^{3/2}_{1/2}), \mathcal{H}(W^{3/2}_{1/2}, W^{3/2}_{3/2}))) \\ &= \ldots \end{aligned} \tag{4.4}$$

If $\Xi = \{\boldsymbol{x}^\nu_\iota\}$ and $q^\nu_\iota = W^\nu_\iota + \epsilon$, then $\bar{W}(S) = W(S) + \dfrac{\partial W(S)}{\partial q^\nu_\iota}\epsilon$ by the linearity of $\mathcal{H}$. We thus define $\alpha^\nu_\iota := \dfrac{\partial W(S)}{\partial q^\nu_\iota}$, the sensitivity of $W(S)$ to a change in $W(\boldsymbol{x}^\nu_\iota)$. (The α values are defined at all gridpoints in $X \cup Q$.) For example, $\alpha^1_1 = \dfrac{\partial}{\partial W^1_1}\mathcal{H}(\mathcal{H}(W^1_{-1}, W^1_0), \mathcal{H}(W^1_0, W^1_1))$.

Since $\dfrac{\partial H}{\partial W_H} = \beta$ and $\dfrac{\partial H}{\partial W_V} = 1 - \beta$, by the chain rule a simple recursive formula for α can be derived in terms of the local β-dependencies.

$$\begin{aligned} \alpha^{\nu+1/2}_{\iota-1/2} &= \beta\alpha^\nu_\iota + (1-\beta)\alpha^\nu_{\iota-1} \\ \alpha^{\nu+1/2}_{\iota+1/2} &= \beta\alpha^\nu_{\iota+1} + (1-\beta)\alpha^\nu_\iota \\ \alpha^{\nu+1}_\iota &= \beta\alpha^{\nu+1/2}_{\iota+1/2} + (1-\beta)\alpha^{\nu+1/2}_{\iota-1/2} \end{aligned} \tag{4.5}$$

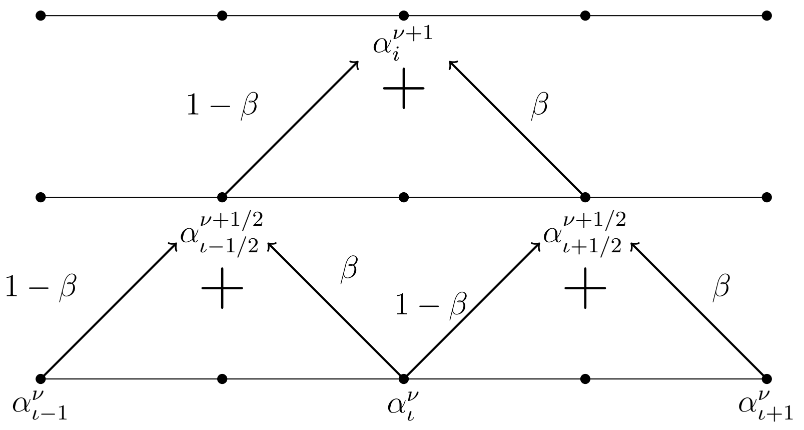


Figure 4.3: Schematic illustrating equation (4.5).

Equation (4.5) can also be written in single-step form:

$$\alpha_\iota^{\nu+1} = \beta^2\alpha_{\iota+1}^{\nu} + 2\beta(1-\beta)\alpha_\iota^{\nu} + (1-\beta)^2\alpha_{\iota-1}^{\nu} \tag{4.6}$$

Now, things get interesting when equation (4.6) is considered as an evolution equation for $\boldsymbol{\alpha}^\nu$, the set of α's on the ν^{th} dependency level. In other words,

$$\boldsymbol{\alpha}^\nu = A\boldsymbol{\alpha}^{\nu-1} = \prod_{j=1}^{\nu} A\boldsymbol{\alpha}^0$$

where A is the (tridiagonal) matrix representing (4.6), and $\boldsymbol{\alpha}^0$ is a unit vector (since $\dfrac{\partial W_0^0}{\partial q_0^0} = 1$). See Figure 4.4 for an illustration.

4.2 An asymptotic estimate for α

In this section we derive a PDE that approximates the behavior of (4.6) by using a technique similar to "modified equations" [47] for backward error analysis of numerical methods.

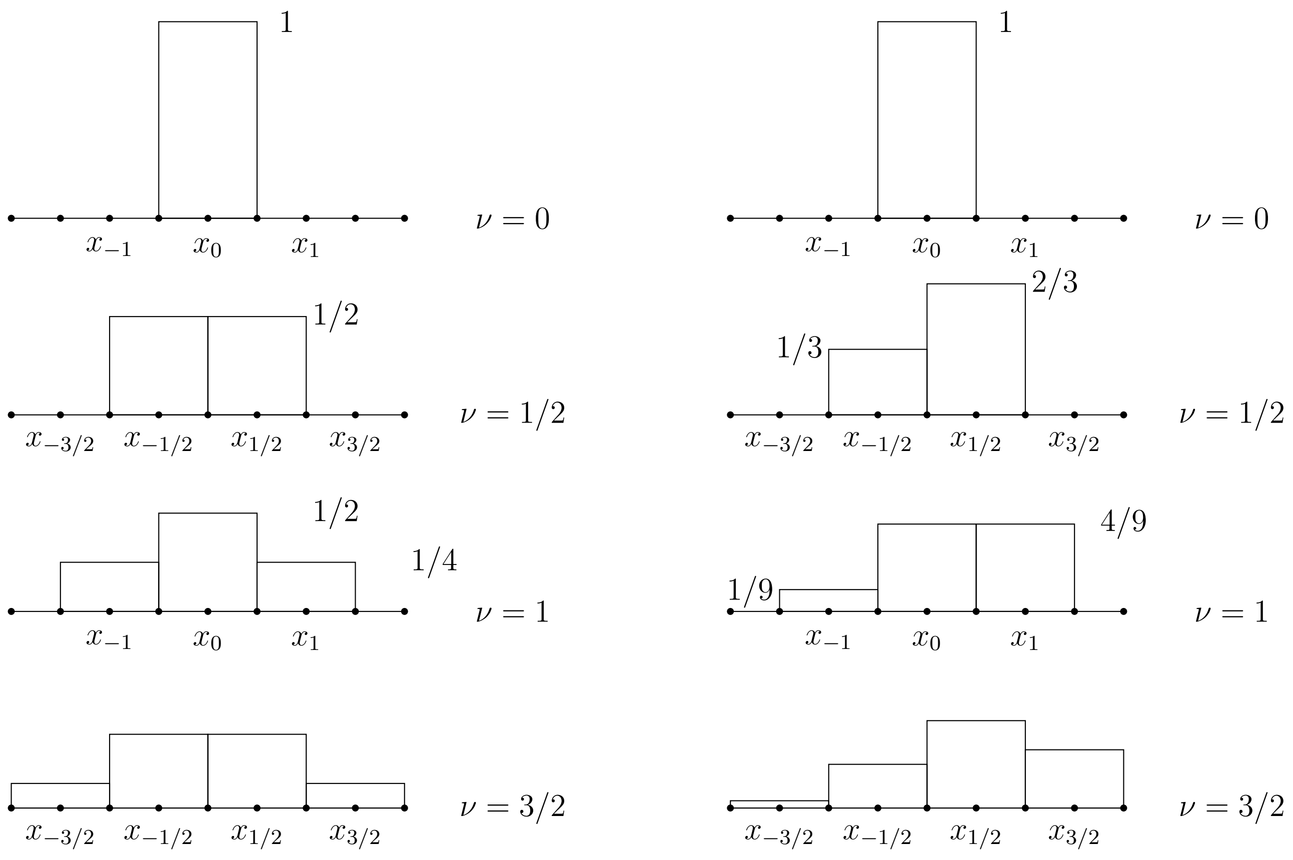


Figure 4.4: Left: evolution of $\boldsymbol{\alpha}^\nu$ for $\beta = 1/2$. Right: evolution of $\boldsymbol{\alpha}^\nu$ for $\beta = 1/3$.

Theorem 9. *Equation* (4.6) *approximates up to* $O(h^2)$ *the PDE*

$$\widetilde{\alpha}_s = (2\beta - 1)\widetilde{\alpha}_y + h\sqrt{2}(\beta(1-\beta))\widetilde{\alpha}_{yy}$$

$$0 < s < \infty \qquad (4.7)$$

$$-\infty < y < \infty$$

with refinement path $\Delta s = \Delta y = h\sqrt{2}$.

Before proving the theorem, note that it illustrates:

1. the stepsizes have a natural correspondence to the grid (e.g., see Figure 4.2), so (y, s) gives a physical location on Ω.
2. the rate of advection, $1 - 2\beta$, is constant. When $\beta = 1/2$ the advection term vanishes, as shown in Figure 4.4.

3. the rate of diffusion, $h\sqrt{2}(\beta(1-\beta))$, decreases with refinement.

Proof. Let $\widetilde{\alpha}(x,t)$ be a smooth polynomial.

$$\widetilde{\alpha}_\iota^\nu := \widetilde{\alpha}(y_\iota, s_\nu)$$

To simplify notation we will use $\widetilde{\alpha}$ as a short form to denote $\widetilde{\alpha}_\iota^\nu$ where not ambiguous. For example,

$$\begin{aligned}\widetilde{\alpha}_\iota^{\nu+1} &= \widetilde{\alpha}(y_\iota, s_\nu + \Delta s)\\ &= \widetilde{\alpha} + \Delta s \widetilde{\alpha}_s + \frac{\Delta s^2}{2}\widetilde{\alpha}_{ss} + O(\Delta s^3)\end{aligned}$$

Plugging in $\widetilde{\alpha}(y,s)$ into (4.6), and Taylor expanding where appropriate:

$$\begin{aligned}&\widetilde{\alpha} + \Delta s \widetilde{\alpha}_s + \frac{\Delta s^2}{2}\widetilde{\alpha}_{ss} + O(\Delta s^3)\\ &= \beta^2(\widetilde{\alpha} + \Delta y \widetilde{\alpha}_y + \frac{\Delta y^2}{2}\widetilde{\alpha}_{yy} + O(\Delta y^3))\\ &+ 2\beta(1-\beta)\widetilde{\alpha}\\ &+ (1-\beta)^2(\widetilde{\alpha} - \Delta y \widetilde{\alpha}_y + \frac{\Delta y^2}{2}\widetilde{\alpha}_{yy} + O(\Delta y^3))\end{aligned}$$

Canceling the zeroth order terms and using $\Delta s = \Delta y = h\sqrt{2}$,

$$\begin{aligned}&\widetilde{\alpha}_s + \frac{h\sqrt{2}}{2}\widetilde{\alpha}_{ss} + O(h^2)\\ &= \beta^2(\widetilde{\alpha}_y + \frac{h\sqrt{2}}{2}\widetilde{\alpha}_{yy} + O(h^2)) + (1-\beta)^2(-\widetilde{\alpha}_y + \frac{h\sqrt{2}}{2}\widetilde{\alpha}_{yy} + O(h^2))\\ &= (2\beta-1)\widetilde{\alpha}_y + (\beta^2 + (1-\beta)^2)\frac{h\sqrt{2}}{2}\widetilde{\alpha}_{yy} + O(h^2)\end{aligned}$$

$$\widetilde{\alpha}_s = -\frac{h\sqrt{2}}{2}\widetilde{\alpha}_{ss} + (2\beta - 1)\widetilde{\alpha}_y + (\beta^2 + (1-\beta)^2)\frac{h\sqrt{2}}{2}\widetilde{\alpha}_{yy} + O(h^2) \tag{4.8}$$

The term $\widetilde{\alpha}_{ss}$ can be thought of as the time derivative of the right hand side of equation (4.8):

$$\begin{aligned}
\widetilde{\alpha}_{ss} &= -\frac{h\sqrt{2}}{2}\widetilde{\alpha}_{sss} + (2\beta - 1)\widetilde{\alpha}_{ys} + (\beta^2 + (1-\beta)^2)\frac{h\sqrt{2}}{2}\widetilde{\alpha}_{yys} + O(h^2) \\
&= (2\beta - 1)\widetilde{\alpha}_{sy} + O(h) \\
&= (2\beta - 1)(-\frac{h\sqrt{2}}{2}\widetilde{\alpha}_{ssy} + (2\beta - 1)\widetilde{\alpha}_{yy} + (\beta^2 + (1-\beta)^2)\frac{h\sqrt{2}}{2}\widetilde{\alpha}_{yyy}) + O(h) \\
&= (2\beta - 1)^2\widetilde{\alpha}_{yy} + O(h)
\end{aligned}$$

So equation (4.8) becomes:

$$\begin{aligned}
\widetilde{\alpha}_s &= -\frac{h\sqrt{2}}{2}(2\beta - 1)^2\widetilde{\alpha}_{yy} + (2\beta - 1)\widetilde{\alpha}_y + (\beta^2 + (1-\beta)^2)\frac{h\sqrt{2}}{2}\widetilde{\alpha}_{yy} + O(h^2) \\
&= (2\beta - 1)\widetilde{\alpha}_y + \frac{h\sqrt{2}}{2}(\beta^2 + (1-\beta)^2 - (2\beta - 1)^2)\widetilde{\alpha}_{yy} + O(h^2) \\
&= (2\beta - 1)\widetilde{\alpha}_y + h\sqrt{2}(\beta(1-\beta))\widetilde{\alpha}_{yy} + O(h^2)
\end{aligned}$$

$\square$

Comparison with the analytical solution: It is well-known (e.g., [29]) that the solution of a linear advection-diffusion PDE with constant coefficients $\widetilde{\alpha}_s + a\widetilde{\alpha}_y = \epsilon\widetilde{\alpha}_{yy}$ on the whole line with $\widetilde{\alpha}(y, 0) = \delta(y)$ (the Dirac delta function) is:

$$\widetilde{\alpha}(y,s) = \frac{1}{\sqrt{4\pi\epsilon s}} \exp(-\frac{(y-\mathrm{a}s)^2}{4\epsilon s}) \tag{4.9}$$

However, the initial condition corresponding to equation (4.6) is not actually $\widetilde{\alpha}(y,0) = \delta(y)$; Figure 4.4 shows that

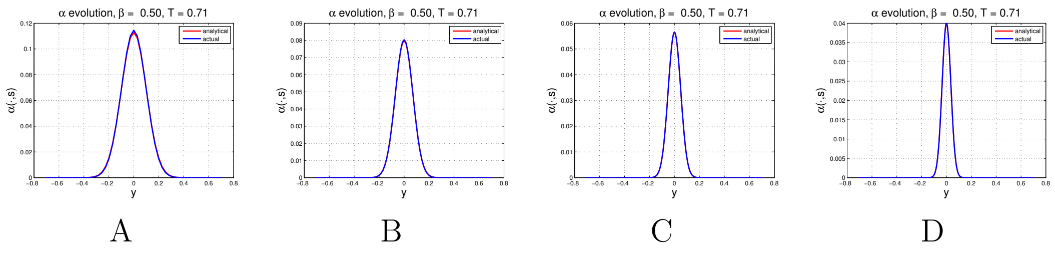
$$\int_{-\infty}^{\infty} \widetilde{\alpha}(y,0)\, dx = h\sqrt{2}$$

instead of 1, resulting in a factor of $h\sqrt{2}$ in the solution of PDE (4.7):

$$\widetilde{\alpha}(y,s) = \frac{\sqrt{2h}}{\sqrt{4\pi s\sqrt{2}(\beta(1-\beta))}} \exp\left(-\frac{(y-(1-2\beta)s)^2}{4h\sqrt{2}(\beta(1-\beta))s}\right) \tag{4.10}$$

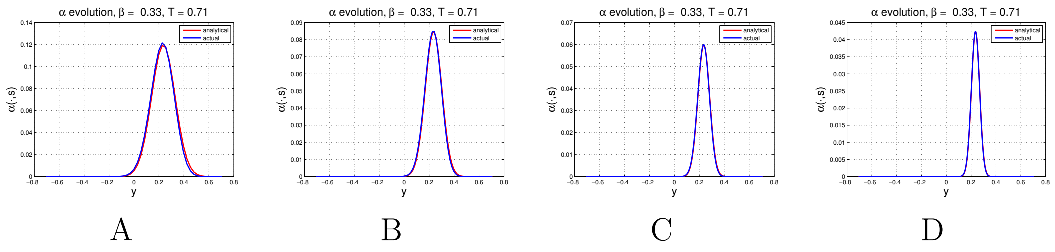
The h in the denominator of the exponent is the reason for the α value of a fixed physical location away from the optimal path decaying exponentially under refinement. Furthermore, for fixed h and y, $\widetilde{\alpha}$ decreases in y monotonically to 0. As for the factor in front of the exponent, the $\sqrt{h}$ in the numerator means that even for $y = (1-2\beta)s$ (i.e., (y,s) on the characteristic going through S), $\widetilde{\alpha}(y,s)$ goes to 0 under refinement, though the rate is much slower than for $y \neq (1-2\beta)s$. Figures 4.5 and 4.6 show a comparison between the analytical solution (4.7) and the "true" α evolution (4.6) for $\beta = 1/2$ and $\beta = 1/3$ at a fixed physical time $s = \sqrt{2}/2$.

Note that the variance of each Gaussian is h-dependent and the drift is β dependent and h-independent. Observe also that the limiting PDE as $h \downarrow 0$ of (4.7) is an advection equation, with information traveling only along the characteristic direction.



h	.02	.01	.005	.0025
L_1	9.3e-5	2.3 e-5	5.8e-6	1.5e-6
order	-	2.02	1.98	1.99
L_∞	5.6e-4	1.99e-4	7.05 e-5	2.49e-5
order	-	1.41	1.41	1.42

Figure 4.5: Comparison between the analytical approximation $\widetilde{\alpha}(\cdot, s)$ (eq. (4.7)) and the true $\boldsymbol{\alpha}^\nu$ evolution at time $s = \sqrt{2}/2$ for different h, and $\beta \equiv 1/2$. The table shows the order of approximation for different h.



h	.02	.01	.005	.0025
L_1	5.07e-4	1.77e-4	6.30e-5	2.22e-5
order	-	1.43	1.40	1.42
L_∞	.0029	.0014	6.98e-4	3.48e-4
order	-	1.04	1.00	1.00

Figure 4.6: Comparison between the analytical approximation $\widetilde{\alpha}(\cdot, s)$ (eq. (4.7)) and the true $\boldsymbol{\alpha}^\nu$ evolution at time $s = \sqrt{2}/2$ for different h, and $\beta \equiv 1/3$. The table shows the order of approximation for different h.

4.3 Using α to bound $E(S)$

As stated earlier, solving system (4.3) on a new domain $\hat{X} \subset X$ with additional boundary conditions $q_i \neq W_i$ in a set Ξ can result in a nonzero error $E(S) = \bar{W}(S) - W(S)$. Suppose α is known $\forall \boldsymbol{x} \in X$. In this section we will show how the α can be used to bound $E(S)$.

In [20], we used a different interpretation of α from what has been presented here. Suppose there is a random walk on $G(S)$ starting from S with local probabilities of transition given by β (to move to the western neighbor) and $1-\beta$ (to move to the southern neighbor). Then α_i can be interpreted as the probability of passing through node $\boldsymbol{x}_i$ on the way to the boundary. The random walk ends when the boundary is reached. As an aside, when the cost per transition is $h\frac{\sqrt{\beta^2+(1-\beta)^2}}{F}$ (from (4.3)) and a price of q_i^n is paid when the boundary node $\boldsymbol{x}_i^n$ is reached, the value $W(S)$ can be interpreted as the expected total cost of the random walk.

We denote

- $\Xi = \{\boldsymbol{x} \in X \backslash \hat{X} | N(\boldsymbol{x}) \cap \hat{X} \neq \emptyset\}$: the set with additional boundary conditions specified: $q_i = W_i + \epsilon_i$ for each $\boldsymbol{x}_i \in \Xi$.
- $\hat{G}(S)$: the dependency digraph of S on $\hat{X} \cup \Xi$.
- $\hat{\alpha}_i$: the sensitivity of $\bar{W}(S)$ to a change in value at $\boldsymbol{x}_i \in \hat{X} \cup \Xi$; i.e., the probability of passing through $\boldsymbol{x}_i$ under the random walk starting from S on $\hat{X}$.
- $\bar{\boldsymbol{W}}^{\nu}$: the set of non-boundary values on level ν (computed from (4.3) on $\hat{X}$).

Additional boundary conditions generally cause changes to the dependency structure of S; Figure 4.7 shows an example. Despite differences between $G(S)$ and $\hat{G}(S)$, note that if $\epsilon_i = 0 \,\forall\, \boldsymbol{x}_i \in \Xi$, then $\bar{W}(S) = W(S)$. One trivial example is if the upwind neighbors of S were both in Ξ, then no other gridpoint's value ($\bar{W}_\iota^{\nu}$ for $\nu \geq 1$) would influence the computation of $W(S)$. When extra boundary conditions are placed on a level ν^*, say, the influence on $\bar{W}(S)$ of values at levels $\nu > \nu^*$ decreases due to $\bar{\boldsymbol{W}}^{\nu}$ no longer influencing $q_\iota^{\nu^*}$. In the stochastic interpretation, $\hat{\alpha}_i \leq \alpha_i$ due to a possible earlier transition into Ξ (at which point the random walk ends).

In section 4.1 we were able to expand $W(S)$ as a function of values on a given

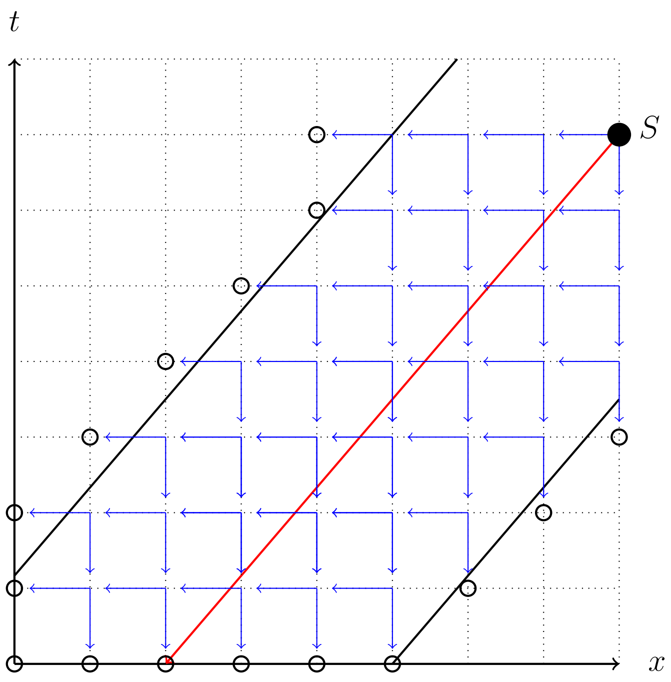


Figure 4.7: An example $\hat{X}$ with relevant boundary conditions circled in black and dependency links drawn in blue.

dependency level (equation (4.4)). For $\bar{W}$, since $\hat{G}(S)$ is different from $G(S)$, an expansion in values on some level ν_j must include all q_ι^ν for $\nu < \nu_j$. We will denote this expansion as $\bar{\mathcal{H}}_{\nu_j}(\boldsymbol{q}^{\nu_1}, ..., \boldsymbol{q}^{\nu_j}; \bar{\boldsymbol{W}}^{\nu_j}) = \bar{W}(S)$, where $\boldsymbol{q}^\nu$ is the set of boundary conditions on level ν.

$\bar{W}(S)$ can also be expanded as a function of boundary values only:

$$\begin{aligned}\bar{W}(S) &= \bar{\mathcal{H}}_{\nu_1}(\boldsymbol{q}^{\nu_1}; \bar{\boldsymbol{W}}^{\nu_1}) \\ &= \bar{\mathcal{H}}_{\nu_j}(\boldsymbol{q}^{\nu_1}, ..., \boldsymbol{q}^{\nu_j}; \bar{\boldsymbol{W}}^{\nu_j}) \\ &= \bar{\mathcal{H}}_{\nu_f}(\boldsymbol{q}^{\nu_1}, ..., \boldsymbol{q}^{\nu_j}, ..., \boldsymbol{q}^{\nu_f})\end{aligned}$$

where ν_f is the final level of $\hat{G}(S)$. Since $\bar{\mathcal{H}}_{\nu_j}$ is still linear and $q_i = W_i + \epsilon_i$ for $\boldsymbol{x}_i \in \Xi \cup Q$,

$$\begin{aligned}
E(S) &= \bar{W}(S) - W(S) \\
&= \bar{\mathcal{H}}_{\nu_j}(\boldsymbol{q}^{\nu_1}, ..., \boldsymbol{q}^{\nu_j}; \bar{\boldsymbol{W}}^{\nu_j}) - W(S) \\
&= \bar{\mathcal{H}}_{\nu_f}(\boldsymbol{q}^{\nu_1}, ..., \boldsymbol{q}^{\nu_j}, ..., \boldsymbol{q}^{\nu_f}) - W(S) \\
&= \bar{\mathcal{H}}_{\nu_f}(\boldsymbol{W}^{\nu_1}, ..., \boldsymbol{W}^{\nu_j}, ..., \boldsymbol{W}^{\nu_f}) + \sum_{\boldsymbol{x}_i \in \Xi \cup Q} \hat{\alpha}_i \epsilon_i - W(S) \\
&= \sum_{\boldsymbol{x}_i \in \Xi \cup Q} \hat{\alpha}_i \epsilon_i \\
&\leq \sum_{\boldsymbol{x}_i \in \Xi \cup Q} \alpha_i \epsilon_i
\end{aligned}$$

4.4 A rigorous bound for a particular boundary value perturbation

In this section we provide a rigorous upper bound on $E(\boldsymbol{x}_i^n)$ for a particular $\hat{X} = \{X \backslash \boldsymbol{x}_0^n\}$, with $\Xi = \{\boldsymbol{x}_0^n\}$ replacing the existing left boundary conditions. We assume that $E_0^n < C \; \forall n$. At the end of the section we compare these bounds against the exact error and an estimate that uses $\widetilde{\alpha}$.

The characteristics of the PDE (4.2) are of course straight parallel lines with slope $(1-\beta)/\beta$. $\bar{W}$ only converges to the correct solution $w(x,t)$ in the region beyond the sector $\{x \leq (\frac{\beta}{1-\beta})t\}$, as pictured in Fig. 4.8. I.e., $\bar{W}$ will converge only at $\boldsymbol{x}$ whose characteristics reach $t = 0$, avoiding the perturbed boundary conditions. Our goal is to show that for (x^*, t^*) outside this sector, the error decreases exponentially under refinement, and that the rate of convergence improves the smaller β is.

While the error equation clearly satisfies a recursive relationship ($E_i^n = \beta E_{i-1}^n + (1-\beta)E_i^{n-1}$), an explicit error formula is readily available by again considering a

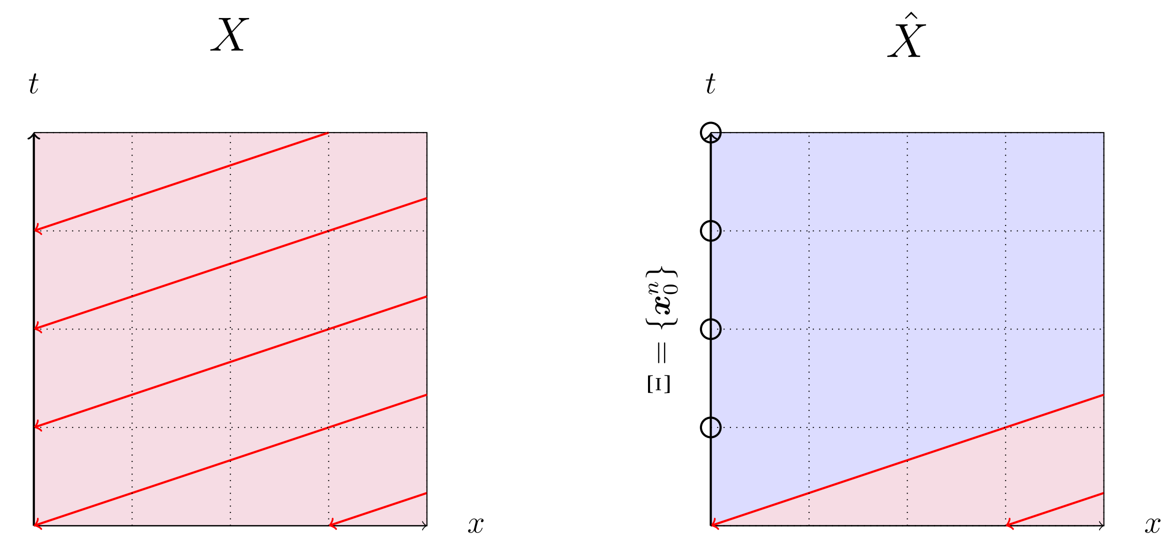


Figure 4.8: Left: characteristics of the solution of equation (4.2) on the original domain X. Right: $\bar{W}$ converges to the correct solution only in the red shaded region.

random walk starting from $\boldsymbol{x}_i^n$. Each step of this walk is a Bernoulli trial, with β being the probability of moving left, $1-\beta$ the probability of moving down, and α_j^k is the probability of passing through $\boldsymbol{x}_j^k$ on the way to the boundary. As described in section 4.3, by the linearity of $\mathcal{H}$, the α's also give an explicit error formula:

$$E_i^n = \sum_{j=1}^{n} E_0^j \alpha_0^j \leq C \sum_{j=1}^{n} \alpha_0^j,$$

where $C = \max_j\{E_0^j\}$. The sum $\sum_{j=1}^{n} \alpha_0^j$ is the probability of reaching the left edge.

Now, consider the probability of reaching *or passing through* the left edge in exactly $N = i+n-1$ steps. In other words, the random walk is now allowed to continue beyond the domain after one of the boundaries is reached, as in Figure 4.9. This is equivalent to the probability of reaching the left edge under the α-random walk, because exactly one of the two edges is guaranteed to be reached in N steps. Furthermore, once the walk reaches the left edge, there is no possibility of moving to the line $t = 0$ in the remaining steps. If Y denotes this random variable, the probability of moving j steps to the left in N trials is given by $\Pr_{N,\beta}(Y = j) = \beta^j (1-\beta)^{N-j} \binom{N}{j}$[1].

[1]We note as an aside that this formulation is exactly the "problem of points" [50] discussed by

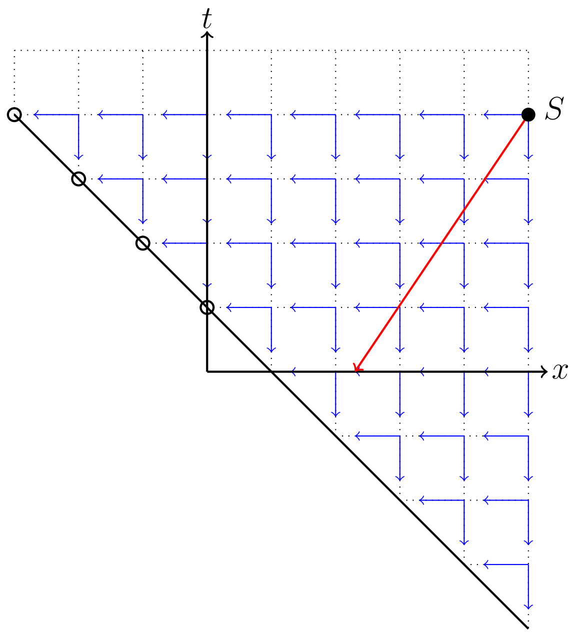


Figure 4.9: The random walk with exactly N steps terminates on the diagonal pictured. The probability of landing on one of the circled nodes is $\sum_{j=1}^{N} \alpha_0^j$. The characteristic through S is drawn in red.

The probability of passing through or terminating on the left boundary is then

$$\Pr_{N,\beta}(Y \geq i) = \sum_{j=i}^{N} \beta^j (1-\beta)^{N-j} \binom{N}{j},$$

and

$$E_i^n \leq C \Pr_{N,\beta}(Y \geq i). \tag{4.11}$$

We are interested in bounding $\Pr_{N,\beta}(Y \geq i)$ when $i \geq \dfrac{n\beta}{1-\beta}$, i.e., at the gridpoints whose characteristic reaches the boundary conditions at $n = 0$. It will be easier to analyze $\Pr_{N+1,\beta}(Y \geq i)$, i.e., the random walk with one more step. Clearly

$$\Pr_{N,\beta}(Y \geq i) \leq \Pr_{N+1,\beta}(Y \geq i).$$

Fermat and Pascal. The historical context was that two teams have contributed equally to a pot in a winner-takes-all game. External circumstances then cause the game to end before either team has reached the number of points necessary to win. If team A needed i more points to win and team B needed n more points, how should the pot be divided fairly?

Since Y is a binomial random variable, the mean μ is generally $\beta(i+n)$. When $i = \dfrac{n\beta}{1-\beta}$, the mean μ is the origin, and the condition $i \geq \dfrac{n\beta}{1-\beta}$ implies $i \geq \mu$:

$$\begin{aligned}
&(1-\beta)i \geq \beta n \\
&i \geq \beta i + \beta n \\
&i \geq \beta(n+i) = \mu.
\end{aligned}$$

Hoeffding's inequality [36] can now be used based on *how* far i is from μ:

$$\Pr_{N+1,\beta}(Y \geq \mu + \epsilon(N+1)) \leq e^{-2\epsilon^2(N+1)} \tag{4.12}$$

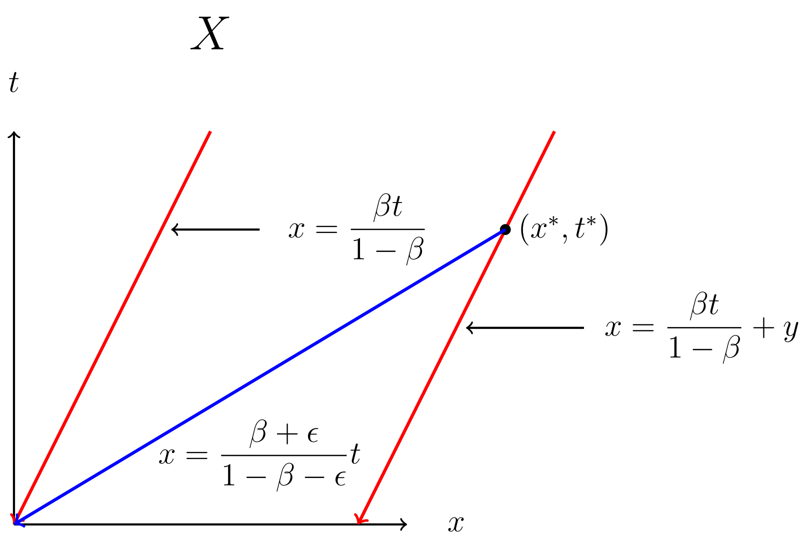
for $\epsilon > 0$. Equating $\Pr_{N+1,\beta}(Y \geq \mu + \epsilon(N+1))$ with $\Pr_{N+1,\beta}(Y \geq i)$ and solving for ϵ yields

$$\epsilon = \frac{x^*(1-\beta) - \beta t^*}{x^* + t^*},$$

so ϵ is independent of h. Geometrically, $\mu + \epsilon N = \beta N + \epsilon N = (\beta+\epsilon)N$ is the mean of a new problem with $\beta \to \beta + \epsilon$ whose characteristic goes through the origin. Figure 4.10 shows an example. Since $N = i + n - 1$, the Hoeffding bound is

$$H(x^*, t^*, \beta, h) := \exp\left(-2\epsilon^2\left(\frac{x^* + t^*}{h}\right)\right)$$

Figures 4.11 and 4.12 are log plots of the exact error, the upper bound H, and an $\widetilde{\alpha}$-based estimate from section 4.2 (using the appropriate change of coordinates). Note that $\widetilde{\alpha}$ is neither an underestimate nor an overestimate on α, since both $\int_{-\infty}^{\infty} \widetilde{\alpha}(y,s)\,\mathrm{d}y = h\sqrt{2}$ and $\int_{-\infty}^{\infty} \alpha(y,s)\,\mathrm{d}y = h\sqrt{2}$ (when α's are thought of as step functions in the spirit of Figure 4.4)



X
t
$x = \frac{\beta t}{1-\beta}$
(x^*, t^*)
$x = \frac{\beta t}{1-\beta} + y$
$x = \frac{\beta+\epsilon}{1-\beta-\epsilon}t$
x

Figure 4.10: The red lines are characteristics of the original problem. The value y in the rightmost equation is $x^* - \frac{\beta t^*}{1-\beta}$. The blue line is a characteristic passing through (x^*, t^*) of a problem with $\beta + \epsilon$ replacing β (with $\epsilon > 0$).

4.5 The connection between the advection upwind scheme and the Eikonal upwind scheme

We first recall notation from Chapter 1 for the update function related to the scheme (4.1). Suppose the value at a gridpoint $\boldsymbol{x}$ is U, its speed F, and the values at its eastern, northern, western, and southern neighbors U_E, U_N, U_W, and U_S respectively. Let $U_H := \min\{U_W, U_E\}$ and $U_V := \min\{U_N, U_S\}$. The "two-sided update" corresponding to equation (4.1) is:

$$\left(\frac{U - U_H}{h}\right)^2 + \left(\frac{U - U_V}{h}\right)^2 = \frac{1}{F^2}. \tag{4.13}$$

In parallel to our development for W, we write U as an explicit function of its upwind neighboring values:

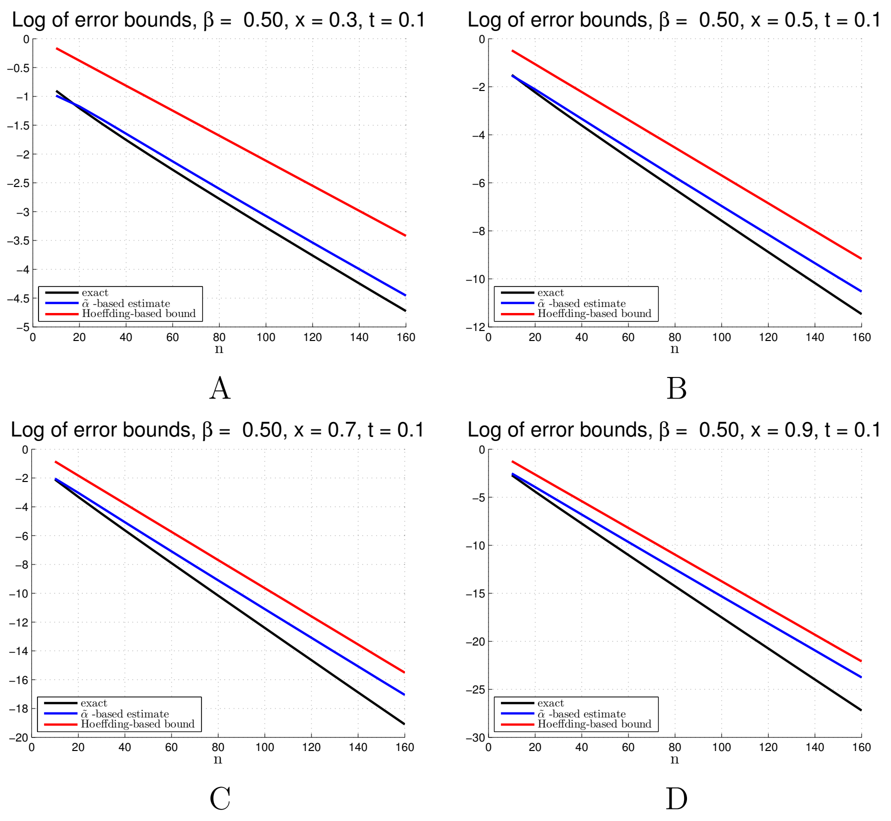


Figure 4.11: Comparison of the various bounds/estimates under refinement for different frozen x with $\beta = .5$, $t = .1$, and $E_1^n \equiv 1$ on $\Omega = [0,1]^2$.

$$U = \mathcal{G}(U_H, U_V) := \frac{U_H + U_V}{2} + \frac{1}{2}\sqrt{\frac{2h^2}{F^2} - (U_H - U_V)^2} \tag{4.14}$$

This solution is only accepted if $U \geq \max\{U_H, U_V\}$. This is called the "upwind condition." It can be shown that

$$\mathcal{G}(U_H, U_V) \geq \max\{U_H, U_V\} \iff |U_H - U_V| \leq \frac{h}{F}$$

If the upwind condition fails, we instead use a "one-sided update":

$$\mathcal{G}(U_H, U_V) = \min\{U_H, U_V\} + \frac{h}{F}$$

Suppose that $\boldsymbol{x}$'s quadrant-of-update is unique and that its upwind neighbors U_H

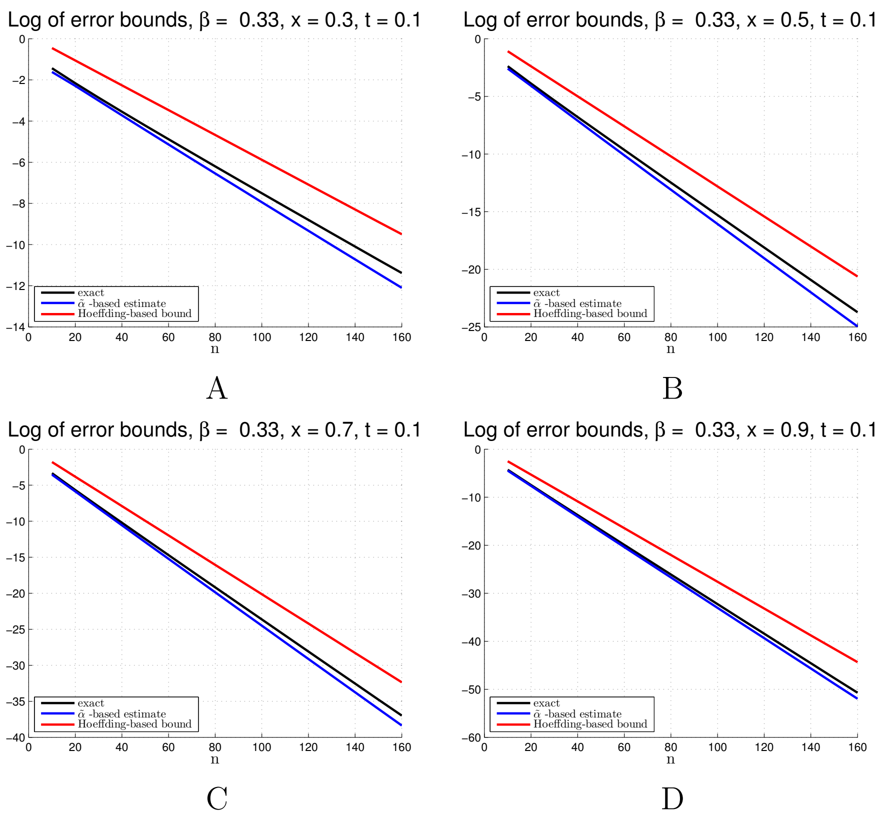


Figure 4.12: Comparison of the various bounds/estimates as functions of h for different frozen x with $\beta = 1/3$, $t = .1$, and $E_1^n \equiv 1$ on $\Omega = [0,1]^2$.

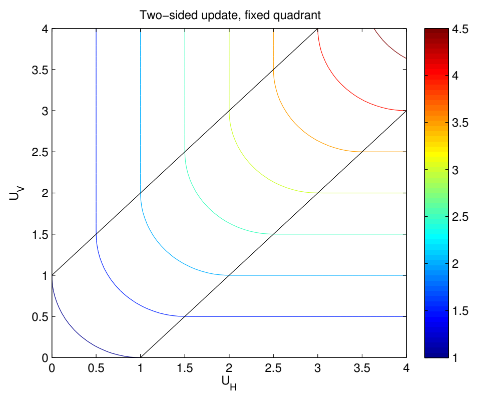


Figure 4.13: Level sets of the update from one quadrant (equation (4.14))

and U_V are replaced by boundary values q_H and q_V. If $q_H = U_H$ and $q_V = U_V$, then clearly $\mathcal{G}(q_H, q_V) = U$. But if instead we force a tiny error, $q_H = U_H + \epsilon$, while holding $q_V = U_V$, then $\mathcal{G}(q_H, q_V) - U \approx \epsilon \dfrac{\partial \mathcal{G}}{\partial q_H}(U_H, U_V)$. We thus define the local dependencies for $\mathcal{G}$ as we did earlier for $\mathcal{H}$:

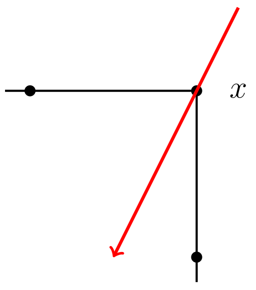


Figure 4.14: If the characteristic passing through $\boldsymbol{x}$ leans more towards the horizontal neighbor, β is closer to 1. If it leans more towards the vertical neighbor, β is closer to 0.

$$\begin{cases} \beta_H := \dfrac{\partial \mathcal{G}}{\partial q_H}(U_H, U_V) & = \dfrac{1}{2} + \dfrac{U_V - U_H}{2\sqrt{\frac{2h^2}{F^2} - (U_V - U_H)^2}} \\ \beta_V := \dfrac{\partial \mathcal{G}}{\partial q_V}(U_H, U_V) & = \dfrac{1}{2} - \dfrac{U_V - U_H}{2\sqrt{\frac{2h^2}{F^2} - (U_V - U_H)^2}}. \end{cases} \tag{4.15}$$

It is easy to show by the monotonicity of $\mathcal{G}$ that

- $0 \le \beta_H, \beta_V \le 1$
- $\beta_H + \beta_V = 1.$

This allows us to simply use $\beta := \beta_H$ and $1 - \beta$ as before. Figure 4.14 illustrates how β relates to the local characteristic direction: the optimal direction of motion is attracted towards the smaller of $\{U_H, U_V\}$.

Recall that our advection equation (4.2) was defined in terms of some constant β. If the optimal direction of motion at each gridpoint is constant, then the discretizations of the Eikonal and advection equations are equivalent; the connection is most clearly illustrated by writing (4.14) in an alternate form (the "semi-Lagrangian" update):

$$\mathcal{G}(U_H, U_V) = \min_{\gamma \in [0,1]} \left\{ \frac{|\gamma \boldsymbol{x}_H + (1-\gamma)\boldsymbol{x}_V - \boldsymbol{x}|}{F} + \gamma U_H + (1-\gamma) U_V \right\} \tag{4.16}$$

It can be shown that formula (4.14) is equivalent to (4.16) (e.g., [60]). Using this fact, a simple calculation shows that β is the minimizer of (4.16) (i.e., formula (4.14) is recovered). Equation (4.16) shows that, as in Figure 4.14, the vector $[\pm\beta, \pm(1-\beta)]$ can be interpreted as the unit 1-norm scaled optimal direction of motion at $\boldsymbol{x}$, where the signs are determined by the direction of update. Equation (4.16) also illustrates that once the optimal controls are known, the update at a gridpoint is linear. Note that if β and F were constants, equation (4.16) is the same as the original upwind scheme for the linear advection equation:

$$\begin{aligned}\mathcal{H}(W_H, W_V) &= \frac{|\beta\boldsymbol{x}_H + (1-\beta)\boldsymbol{x}_V - \boldsymbol{x}|}{F} + \beta W_H + (1-\beta)W_V \\ &= \beta W_H + (1-\beta)W_V + h\frac{\sqrt{\beta^2 + (1-\beta)^2}}{F}\end{aligned} \tag{4.17}$$

This connection makes it easy to compare U with W (and $\bar{U}$ with $\bar{W}$, etc). Listed again for convenience:

- U: the solution of the original Eikonal equation (4.1) on X.
- $\bar{U}$: the solution of system (4.1) with extra boundary conditions $q_i \geq U_i$ on Ξ.
- $\hat{U}$: the solution of system (4.1) with extra boundary conditions $q_i = +\infty$ on Ξ.

Assuming the solution U is known everywhere, the quadrants-of-update were unique, and the optimal policy β_i^n has been computed at each $\boldsymbol{x}_i^n$ by (4.15), we can consider the following analogous linear advection schemes for solving the Eikonal equation:

- W: the solution of (4.17) on X (with coefficients no longer constant).
- $\bar{W}$: the solution of (4.17) on $\hat{X}$ with extra boundary conditions $q_i \geq U_i$ on Ξ. The controls β_i^n may be suboptimal directions of motion when $q_i \neq U_i$.

By equation (4.16), clearly $W = U$ and $\bar{W} \geq \bar{U}$. Thus $E_U(\boldsymbol{x}) := \bar{U}(\boldsymbol{x}) - U(\boldsymbol{x}) \leq E_W(\boldsymbol{x}) := \bar{W}(\boldsymbol{x}) - W(\boldsymbol{x}) = \bar{W}(\boldsymbol{x}) - U(\boldsymbol{x})$. Our previous bounds on W held when β was constant. These bounds would apply to Eikonal equations in very special cases where the characteristics are straight and parallel.

Two attributes of W were crucial to derive estimates for α: 1) we needed the levels of $G(S)$ to be clearly defined so that we could write down an evolution equation for α, and 2) we needed β constant to get an asymptotic analytical formula for α. In the future it may be possible to handle variable β using concentration inequalities on more general $\hat{X}$ than the one in section 4.4.

4.6 Motivation: numerical methods that solve Eikonal equations on $\hat{X}$

Several efficient methods for Eikonal equations introduced in Chapter 1 and section and 3.1 effectively solve system (4.1) on some $\hat{X}$. One is the parallel Patchy FMM [15]. This method uses interpolated values of a coarse precomputation to create a domain decomposition with subdomain boundaries being approximately characteristic. The advantage of this approach is that the resulting subdomains are approximately invariant to each other. Thus, computations can be done on each of them with no communication across boundaries, allowing for efficient parallelization. However, since both the boundary values as well as the boundaries themselves are computed using only coarse information, this additional boundary error propagates into the interiors of the subdomains.

One application of the analysis in section 4.4 is to bound the error of Patchy FMM on a problem with parallel characteristics pointing toward the southwest. Suppose

$\Omega = \{t > 0\}$, $Q = \{t = 0\}$, there are two processors, and one subdomain is quadrant I and the other is quadrant II, with $\{x = 0\}$ being the subdomain boundary created by Patchy FMM. In quadrant II there will be no errors, since the domain of dependence at each gridpoint terminates on the line $t = 0$, where the original boundary conditions are specified. In quadrant I, errors will be small (as quantified by equation (4.12), the Hoeffding bound) if $x = 0$ is approximately characteristic; in our terminology this occurs when β is small.

Another efficient approximate method is the AA* [20] for recovering a **single** optimal path from a source gridpoint S to a target gridpoint T. This method achieves its efficiency by eliminating gridpoints from X that are far from the optimal path. Figures 4.15 and 4.17 show examples with different AA*-restricted domains, and Figures 4.16 and 4.18 show the associated relative error at S.

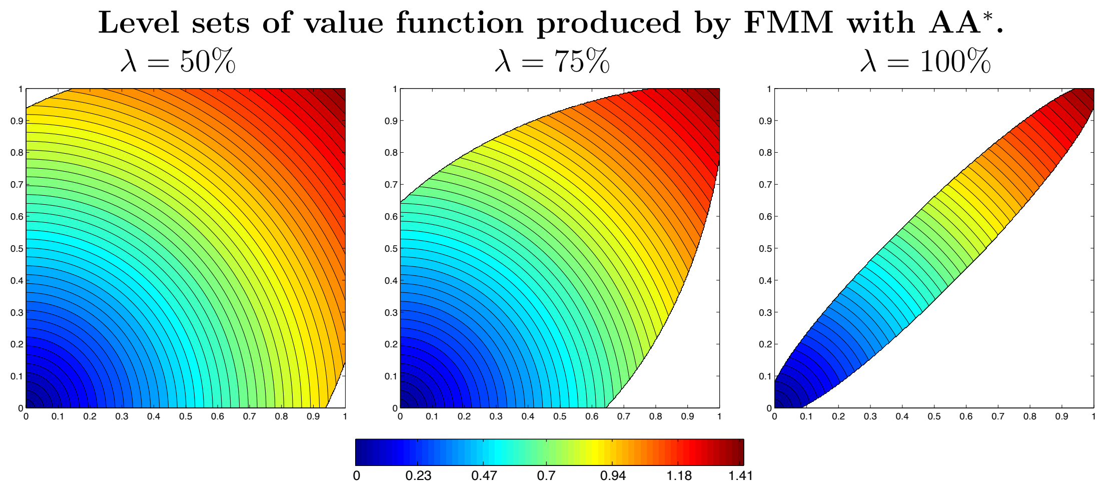


Figure 4.15: Level sets for $\bar{U}$ with $F \equiv 1$ solved using the AA* domain restriction method from [20]. λ is a domain restriction parameter. Courtesy of Zachary Clawson.

One way to approach the error analysis is to define α values for Eikonal equations (system (4.1)) where the quadrant-of-update is unique. However, a recursive formula like (4.6) remains unavailable except in special cases. Suppose a value U_i^n is replaced by a boundary value q_i^n; we define $\alpha_i^n := \dfrac{\partial U(S)}{\partial q_i^n}$. In [20] Alex Vladimirsky showed

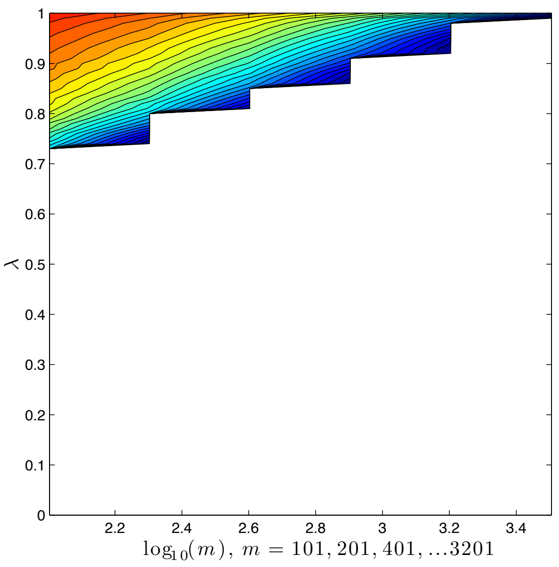


Figure 4.16: $\log_{10}$ of the relative error at S of FMM/AA* for the Eikonal equation with $F \equiv 1$ for different domain restriction parameters and different m = # gridpoints per domain side. Courtesy of Zachary Clawson.

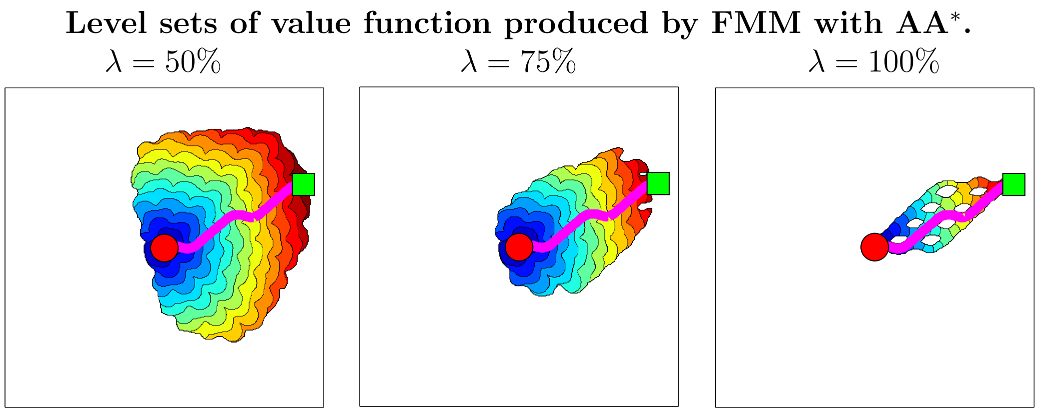


Figure 4.17: Level sets for $\bar{U}$ with $F = 1 + .5\sin(20\pi x)\sin(20\pi y)$ solved using the AA* method from [20]. λ is a domain restriction parameter. Courtesy of Zachary Clawson.

that

$$E(S) \leq C \sum_{\boldsymbol{x}_i \in \Xi} \alpha_i, \tag{4.18}$$

where C is an upper bound on $\hat{U}(S)$. Since the α_i are not actually available without the solution over all of X, this is only useful if their qualitative behavior is known. We conjecture that for the upwind Eikonal scheme with small h,

$$\alpha_i \approx \exp\left(\frac{-\rho d(\boldsymbol{x}_i)^2}{h}\right) \tag{4.19}$$

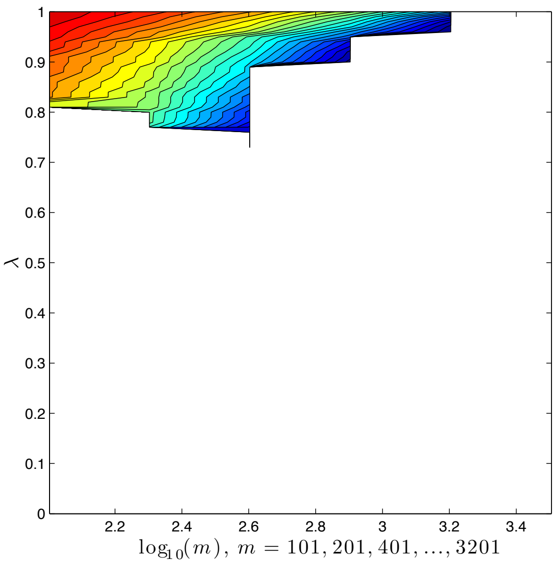


Figure 4.18: $\log_{10}$ of the relative error at S of FMM/AA* for the Eikonal equation with $F \equiv 1 + .5\sin(20\pi x)\sin(20\pi y)$ for different domain restriction parameters and different $m = \#$ gridpoints per domain side. Courtesy of Zachary Clawson.

where $d(\boldsymbol{x}_i)$ is $\boldsymbol{x}_i$'s distance to the optimal path and ρ is a positive constant. Our preliminary numerical experiments seem to confirm (4.19); see Figure 4.19. The idea for exponential decay of the α's was inspired by both the analytical estimate $\widetilde{\alpha}$ from section 4.2 and the Hoeffding bound derived in section 4.4.

Using this, Alex Vladimirsky proved that $\hat{U}$ converges to U even as $\hat{X}$ shrinks under refinement:

Theorem 10. *Let $\{X^h\}$ be a family of Cartesian grids on Ω with grid size $h = 1/(m-1)$ such that both S and T are gridpoints for all m. Define $\hat{X}^h = \{\boldsymbol{x} \in X^h \mid d(\boldsymbol{x}) < r\}$, where $r = O(h^\mu)$, for some $\mu \in [0, \frac{1}{2})$. Let U^h and $\hat{U}^h$ be numerical solutions of the system (4.1) on X^h and $\hat{X}^h$ respectively. If (4.19) holds, then $\left(\hat{U}^h(S) - U(S)\right) \to 0$ as $h \to 0$.*

Figures 4.16 and 4.18 show clear numerical evidence of this theorem (m is the number of gridpoints per domain edge). They show also that for most values of λ, $E(S)$ decays exponentially in h (i.e., the spacing between contours is approximately

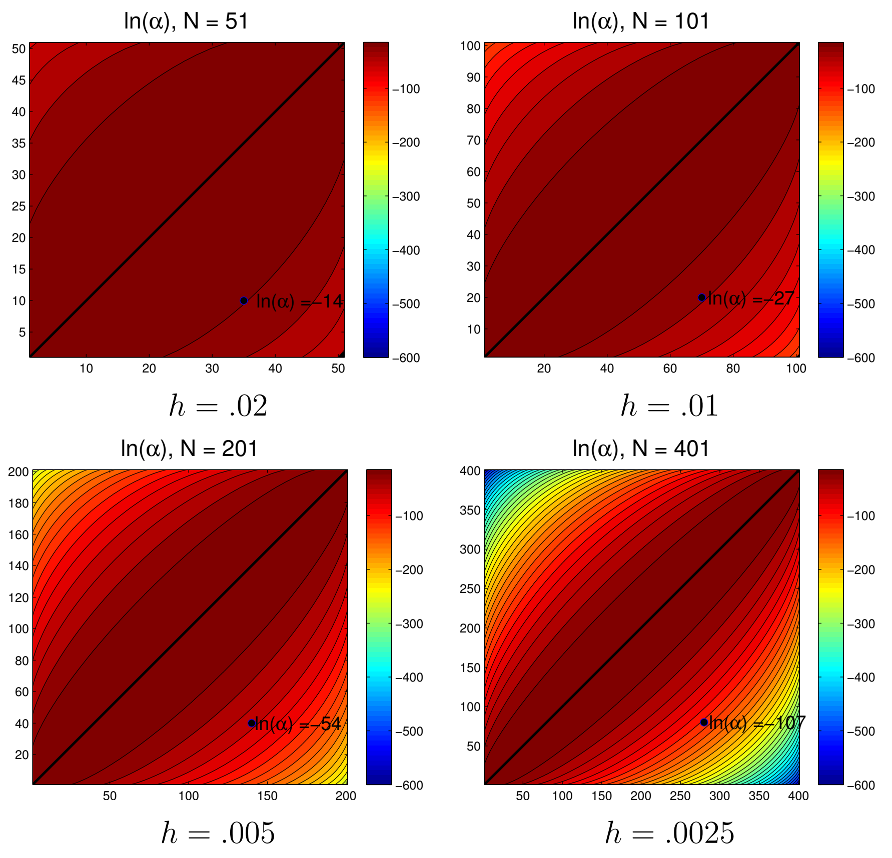


Figure 4.19: Level sets of $\ln(\alpha)$ on X under refinement for an Eikonal equation with $F \equiv 1$. Each plot shows $\ln(\alpha(.7, .2))$ (chosen arbitrarily) in the lower right, which is evidence that the factor ρ in conjecture (4.19) is independent of h.

the same).

CHAPTER 5

CONCLUSIONS

We introduced three new efficient hybrid methods for Eikonal equations, a scalable parallelization of one of them, and showed preliminary error analysis for methods that introduce boundary data pollution or rely on domain restriction techniques.

For the hybrid methods, using a splitting of the domain into a number of cells, they employ sweeping methods on individual cells with the order of cell-processing and the direction of sweeps determined by a marching-like procedure on a coarser scale. Such techniques may introduce additional errors to attain higher computational efficiency. Of these new methods FMSM is generally the fastest and perhaps easiest to implement, while FHCM introduces smaller additional errors, and HCM is usually the slowest of the three but provably converges to exact solutions. The numerical evidence presented in section 2.3 strongly suggests that

- when h and h^c are sufficiently small, additional errors introduced by FMSM and FHCM are negligible compared to those already present due to discretization;
- for the right (h, h^c)-combinations, all three of the hybrid algorithms significantly outperform popular prior fast methods (FMM, FSM, and LSM).

We also introduced a new parallel algorithm for the Eikonal equation based on HCM, a fast two-scale serial solver. The numerical experiments in section 3.4 demonstrate 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.

All of the examples considered here used predetermined uniform cell sizes. From a practitioner's point of view, the value of the proposed methods will greatly increase once we develop bounds and estimates for the additional errors in both FMSM and FHCM with different (h, h^c)-combinations. Similarly, for pHCM, given fixed P and M, what value of J will result in the optimal performance? While the numerical experiments suggest an answer, rigorously addressing it will clearly still be useful. Estimates of the computational costs of all the hybrid methods on a given cell decomposition would also be very useful.

The benchmarking and design of pHCM 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 [67]) 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 thesis.

The performance analysis in section 3.4 suggests a number of possible pHCM improvements. A smarter memory allocation 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 FSM/LSM). The methods of [26] 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 are likely to result in even better pHCM scalability. These include discretizations of anisotropic Hamilton-Jacobi equations, stochastic optimal control problems, and differential games. We expect this to also be the case for extensions of other parallel Eikonal solvers (e.g., DFSM/DLSM).

For each hybrid method we intend to automate the choice of cell-sizes based on the speed function and user specified error tolerances (and for pHCM possibly the computer architecture), and further relax the requirement that all cells need to be uniform. A generalization of this approach to cell-subdivision of unstructured meshes will also be valuable. Most importantly, we hope that the hybrid ideas can serve as a basis for causal domain decomposition and efficient two-scale methods for other static nonlinear PDEs.

In Chapter 4 we showed that given the α-values, $\sum \alpha_i \epsilon_i$ bounds the error of domain restriction for upwind finite difference schemes for advection PDEs with constant coefficients. In these simple cases the characteristics are straight and parallel, and equation (4.7) estimates the α-values as the solution of a linear advection-diffusion equation. An obviously useful next step is to generalize the α-value approach to Eikonal equations and advection equations with variable coefficients. In the case of variable β that does not change signs, the Hoeffding or other concentration inequalities (e.g., [7]) again may apply. Applying this analysis to higher-dimensional problems or non-Cartesian grids is more challenging, since the dependency graphs of these problems have more complicated topologies.

We also hope that a similar analysis can be useful for domain restriction techniques of other finite difference schemes with acyclic dependency graphs. These include explicit methods for parabolic equations, static problems for which there are single-pass algorithms, and directed flows on graphs. We conjecture that domain restriction is generally safer for less dissipative schemes, such as a semi-Lagrangian method for Eikonal equations on a triangulated mesh with a bound $\theta^* << 90^\circ$ on the maximum mesh angle.

BIBLIOGRAPHY

[1] Ahuja, R.K., Magnanti, T.L., & Orlin, J.B., *Network Flows*, Prentice Hall, Upper Saddle River, NJ, 1993.

[2] K. Alton & I. M. Mitchell, *Fast Marching Methods for Stationary Hamilton-Jacobi Equations with Axis-Aligned Anisotropy*, SIAM J. Numer. Anal., 47:1, pp. 363–385, 2008.

[3] S. Bak, J. McLaughlin, and D. Renzi, *Some improvements for the fast sweeping method*, SIAM J. Sci. Comp., Vol 32, No. 5, pp.2853-2874, 2010.

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

[5] G. Barles and P. E. Souganidis, *Convergence of approximation schemes for fully nonlinear second order equations*, Asymptot. Anal., 4:271-283, 1991.

[6] Bellman, R., *Dynamic Programming*, Princeton Univ. Press, Princeton, NJ, 1957.

[7] Bernstein,S.N., *On a modification of Chebyshev's inequality and of the error formula of Laplace*, Ann. Sci. Inst. Savantes Ukraine, Sect. Math. 1(1924), 38-49. (Russian)

[8] Bertsekas, D. P., *A Simple and Fast Label Correcting Algorithm for Shortest Paths*, Networks, Vol. 23, pp. 703-709, 1993.

[9] Bertsekas, D.P., *Network optimization: continuous & discrete models*, Athena Scientific, Boston, MA, 1998.

[10] Bertsekas, D.P., *Dynamic Programming and Optimal Control*, 2nd Edition, Volumes I and II, Athena Scientific, Boston, MA, 2001.

[11] Bertsekas, D. P., Guerriero, F., and Musmanno, R., *Parallel Asynchronous Label Correcting Methods for Shortest Paths*, J. of Optimization Theory and Applications, Vol. 88, pp. 297-320, 1996.

[12] F. Bornemann and C. Rasch, *Finite-element Discretization of Static Hamilton-Jacobi Equations based on a Local Variational Principle*, Computing and Visualization in Science, 9(2), pp.57-69, 2006.

[13] Boué, M. & Dupuis, P., *Markov chain approximations for deterministic control problems with affine dynamics and quadratic cost in the control*, SIAM J. Numer. Anal., 36:3, pp.667-695, 1999.

[14] Breuß, M., Cristiani, E., Gwosdek, P., Vogel, O., *An adaptive domain decomposition technique for parallelization of the fast marching method*, Elsevier Applied Mathematics and Computation, 218, pp. 32-44, 2011.

[15] Cacace, S., Cristiani, E., Falcone, M., Picarelli, A. *A patchy Dynamic Programming scheme for a class of Hamilton-Jacobi-Bellman equations*, submitted to SIAM J. Sci. Comp.

[16] A. Chacon and A. Vladimirsky, *Fast two-scale methods for Eikonal equations*, SIAM J. Sci. Comp., Vol. 33, no.3, pp. A547-A578, 2012.

[17] A. Chacon and A. Vladimirsky, *Fast two-scale methods for Eikonal equa-*

tions, Technical Report; available from `http://www.math.cornell.edu/~vlad/papers/fmsm_full_version.pdf`

[18] A. Chacon and A. Vladimirsky, *A parallel Heap-Cell Method for Eikonal equations*, expanded Technical Report; available from `http://arxiv.org/abs/1306.4743`

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

[20] Clawson, Z., Vladimirsky, A., Chacon, A., *Causal Domain Restriction Techniques for Eikonal Equations*, Preprint available from `http://arxiv.org/abs/1309.2884`

[21] M.G. Crandall, L.C. Evans, & P-L.Lions, *Some Properties of Viscosity Solutions of Hamilton-Jacobi Equations*, Tran. AMS, 282 (1984), pp. 487–502.

[22] Crandall, M.G. & Lions, P-L., *Viscosity Solutions of Hamilton-Jacobi Equations*, Tran. AMS, 277, pp. 1-43, 1983.

[23] E. Cristiani and M. Falcone, *A Characteristics Driven Fast Marching Method for the Eikonal Equation*, in "Numerical Mathematics and Advanced Applications", pp. 695-702, Proceedings of ENUMATH 2007, Graz, Austria, September 2007.

[24] Danielsson, P.-E., *Euclidean Distance Mapping*, Computer Graphics and Image Processing, 14, pp.227–248, 1980.

[25] M. Detrixhe, F. Gibou, and C. Min, *A parallel fast sweeping method for the Eikonal equation*, Journal of Computational Physics, v.237, pp.46-55, 2013.

[26] R. Dial, *Algorithm 360: Shortest path forest with topological ordering*, Comm. ACM, pp. 632–633, 1969.

[27] E.W. Dijkstra, *A Note on Two Problems in Connection with Graphs*, Numerische Mathematik, 1 (1959), pp. 269–271.

[28] Donatelli, J. and Sethian, J. *A massively parallel multilevel Fast Marching Method framework*, unpublished Technical Report, 2012.

[29] L.C. Evans, *Partial Differential Equations*, American Mathematical Society, 2002.

[30] M. Falcone, *The Minimum Time Problem and Its Applications to Front Propagation*, in "Motion by Mean Curvature and Related Topics", Proceedings of the International Conference at Trento, 1992, Walter de Gruyter, New York, 1994.

[31] T. Gillberg, M. Sourouri, and X. Cai, *A new parallel 3D front propagation algorithm for fast simulation of geological folds*, Procedia Computer Science, 9, pp. 947955, 2012.

[32] Glover, F., Glover, R., and Klingman, D., *The Threshold Shortest Path Algorithm*, Math. Programming Studies, Vol. 26, pp. 12-37, 1986.

[33] Gonzales, R. & Rofman, E., *On Deterministic Control Problems: an Approximate Procedure for the Optimal Cost, I, the Stationary Problem*, SIAM J. Control Optim., 23, 2, pp. 242-266, 1985.

[34] Gremaud, P.A. & Kuster, C.M., *Computational Study of Fast Methods for the*

Eikonal Equation, SIAM J. Sc. Comp., 27, pp.1803-1816, 2006.

[35] Herrmann, M., *A domain decomposition parallelization of the fast marching method*, Annual Research Briefs, Center for Turbulence Research, Stanford, CA, USA, 2003.

[36] Hoeffding, W., *Probability inequalities for sums of random variables*, Journal of the American Statistical Association, Vol. 58, pp. 13-30, 1963.

[37] S.-R. Hysing and S. Turek, *The Eikonal equation: Numerical efficiency vs. algorithmic complexity on quadrilateral grids*, In Proceedings of Algoritmy 2005, pp.22-31, 2005.

[38] W.-K. Jeong and R. T. Whitaker, *A Fast Iterative Method for Eikonal Equations*, SIAM J. Sci. Comput., 30:5, pp. 2512-2534, 2008.

[39] Kao, C.Y., Osher, S., & Qian, J., *Lax-Friedrichs sweeping scheme for static Hamilton-Jacobi equations*, J. Comput. Phys., 196:1, pp.367–391, 2004.

[40] Kao, C.Y., Osher, S., & Tsai, Y.H., *Fast Sweeping Methods for static Hamilton-Jacobi equations*, SIAM J. Numer. Analy., 42: 2612–2632, 2005.

[41] Kim, S., *An O(N) level set method for eikonal equations*, SIAM J. Sci. Comput., 22, pp. 2178-2193, 2001.

[42] Kimmel, R. & Sethian, J.A., *Fast Marching Methods on Triangulated Domains*, Proc. Nat. Acad. Sci., 95, pp. 8341-8435, 1998.

[43] H.J. Kushner & P.G. Dupuis, *Numerical Methods for Stochastic Control Problems in Continuous Time*, Academic Press, New York, 1992.

[44] F. Li, C.-W. Shu, Y.-T. Zhang and H.-K. Zhao, *A second order DGM based fast sweeping method for Eikonal equations*, Journal of Computational Physics, v.227, pp.8191-8208, 2008.

[45] S. Luo, Y. Yu, H.-K. Zhao, *A new approximation for effective Hamiltonians for homogenization of a class of Hamilton-Jacobi equations*, preprint (2009).

[46] Mellor-Crummey, J., Whalley, D., and Kennedy, K. *Convergent Difference Schemes for Nonlinear Elliptic and Parabolic Equations:*, International J. of Parallel Programming, Vol. 29, No. 3, pp.217-247, 2001.

[47] K.W. Morton and D.F. Mayers, *Numerical Solution of Partial Differential Equations, 2nd Ed*, Cambridge University Press, Cambridge, UK, 2005.

[48] A.M. Oberman, R. Takei, and A. Vladimirsky, *Homogenization of metric Hamilton-Jacobi equations*, Multiscale Modeling and Simulation, 8/1, pp. 269-295, 2009.

[49] Pape, U., *Implementation and Efficiency of Moore - Algorithms for the Shortest Path Problem*, Math. Programming, Vol. 7, pp. 212-222, 1974.

[50] Pascal, letter to Fermat, quoted in *Games, Gods, and Gambling*, Griffin Press, P.239, 1962.

[51] L. C. Polymenakos, D. P. Bertsekas, and J. N. Tsitsiklis, *Implementation of Efficient Algorithms for Globally Optimal Trajectories*, IEEE Transactions on Automatic Control, 43(2), pp. 278–283, 1998.

[52] C. Rasch and T. Satzger, *Remarks on the $O(N)$ Implementation of the Fast Marching Method*, preprint, 2007. http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0703082

[53] Rouy, E. & Tourin, A., *A Viscosity Solutions Approach to Shape-From-Shading*, SIAM J. Num. Anal., 29, 3, pp. 867-884, 1992.

[54] Saad, Y., *Iterative Methods for Sparse Linear Systems*, 2nd ed., Society for Industrial and Applied Mathematics, 2003.

[55] J.A. Sethian, *A Fast Marching Level Set Method for Monotonically Advancing Fronts*, Proc. Nat. Acad. Sci., 93, 4, pp. 1591–1595, February 1996.

[56] J.A. Sethian, *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision and Materials Sciences*, Cambridge University Press, 1996.

[57] Sethian, J.A., *Fast Marching Methods*, SIAM Review, Vol. 41, No. 2, pp. 199-235, 1999.

[58] J.A. Sethian & A. Vladimirsky, *Fast Methods for the Eikonal and Related Hamilton–Jacobi Equations on Unstructured Meshes*, Proc. Nat. Acad. Sci., 97, 11 (2000), pp. 5699–5703.

[59] J.A. Sethian & A. Vladimirsky, *Ordered Upwind Methods for Static Hamilton-Jacobi Equations*, Proc. Nat. Acad. Sci., 98, 20 (2001), pp. 11069–11074.

[60] J.A. Sethian & A. Vladimirsky, *Ordered Upwind Methods for Static Hamilton-Jacobi Equations: Theory & Algorithms*, SIAM J. on Numerical Analysis 41, 1 (2003), pp. 325-363.

[61] Sethian, J.A. & Vladimirsky, A., *Ordered Upwind Methods for Hybrid Control*, 5th International Workshop, HSCC 2002, Stanford, CA, USA, March 25-27, 2002, Proceedings (LNCS 2289).

[62] Tsai, Y.-H.R., Cheng, L.-T., Osher, S., & Zhao, H.-K., *Fast sweeping algorithms for a class of Hamilton-Jacobi equations*, SIAM J. Numer. Anal., 41:2, pp.659-672, 2003.

[63] J.N. Tsitsiklis, *Efficient algorithms for globally optimal trajectories*, Proceedings, IEEE 33rd Conference on Decision and Control, pp. 1368–1373, Lake Buena Vista, Florida, December 1994.

[64] J.N. Tsitsiklis, *Efficient Algorithms for Globally Optimal Trajectories*, IEEE Tran. Automatic Control, 40 (1995), pp. 1528–1538.

[65] A. Vladimirsky, *Label-setting methods for Multimode Stochastic Shortest Path problems on graphs*, Mathematics of Operations Research 33(4), pp. 821-838, 2008.

[66] A. Vladimirsky and C. Zheng, *A fast implicit method for time-dependent Hamilton-Jacobi PDEs*, http://arxiv.org/pdf/1306.3506v1.pdf

[67] O. Weber, Y. Devir, A. Bronstein, M. Bronstein, R. Kimmel *Parallel algorithms for the approximation of distance maps on parametric surfaces*, ACM Transactions on Graphics, 27(4), 2008.

[68] L. Yatziv, A. Bartesaghi, & G. Sapiro, $O(N)$ *implementation of the fast marching algorithm*, J. Comput. Phys. 212 (2006), no. 2, 393-399.

[69] Y.-T. Zhang, S. Chen, F. Li, H.-K. Zhao and C.-W. Shu, *Uniformly accurate discontinuous Galerkin fast sweeping methods for Eikonal equations*, preprint, 2009.

[70] Zhao, H.K., *Fast Sweeping Method for Eikonal Equations*, Math. Comp., 74, pp. 603-627, 2005.

[71] Zhao, H.K., *Parallel Implementations of the Fast Sweeping Method*, J. Comput. Math. 25, pp. 421-429, 2007.