Universität des Saarlandes

Fachrichtung 6.1 – Mathematik

Preprint Nr. 250

A Domain-Decomposition-Free Parallelisation of the Fast Marching Method

Michael Breuß, Emiliano Cristiani,
Pascal Gwosdek and Oliver Vogel

Saarbrücken 2009

Fachrichtung 6.1 – Mathematik
Universität des Saarlandes

Preprint No. 250
submitted: October 8, 2009

A Domain-Decomposition-Free Parallelisation of the Fast Marching Method

Michael Breuß
Saarland University
Department of Mathematics
P.O. Box 15 11 50
66041 Saarbrücken
Germany
breuss@mia.uni-saarland.de

Emiliano Cristiani
CEMSAC
Università di Salerno
Fisciano (SA), Italy
and IAC-CNR, Rome, Italy
emiliano.cristiani@gmail.com

Pascal Gwosdek
Saarland University
Department of Mathematics
P.O. Box 15 11 50
66041 Saarbrücken
Germany
gwosdek@mia.uni-saarland.de

Oliver Vogel
Saarland University
Department of Mathematics
P.O. Box 15 11 50
66041 Saarbrücken
Germany
vogel@mia.uni-saarland.de

Edited by
FR 6.1 – Mathematik
Universität des Saarlandes
Postfach 15 11 50
66041 Saarbrücken
Germany

Fax: + 49 681 302 4443
e-Mail: preprint@math.uni-sb.de
WWW: http://www.math.uni-sb.de/

Abstract

The Fast Marching Method (FMM) is an efficient technique to numerically solve the eikonal equation. The parallelisation of the FMM is not easy because of its intrinsic sequential nature. In this paper we propose a novel approach to parallelise the FMM which is *not* based on a classic domain-decomposition procedure. Compared to other techniques in the field, our method is much simpler to implement and initialise. By numerical experiments we show that it gives in addition a slightly better computational speed-up. Making use of an example from the field of computer vision, we verify that the favourable properties of our approach are useful for real-world applications.

1 Introduction

The Fast Marching Method (FMM) is an efficient technique to solve numerically the eikonal equation

$$\begin{cases} f(x)|\nabla u(x)| = 1\,, & x \in \mathbb{R}^d \backslash \Gamma_0 \\ u(x) = u_0(x)\,, & x \in \Gamma_0\,, \end{cases} \tag{1}$$

where $f(x) > 0$ is a given Lipschitz continuous function, and Γ_0 is a $(d-1)$-dimensional manifold in $\mathbb{R}^d$. Equation (1) is well-posed in the framework of *viscosity solutions* [2]. The unique viscosity solution u of (1) is in general not differentiable, even if $f \in C^1(\mathbb{R}^d)$ and Γ_0 is smooth.
Eikonal and eikonal-type equations appear in a number of different application fields [23], such as computer vision, image processing, optics, geoscience, and medical image analysis. In some cases, the approximation of the solution must be carried out on very large grids, requiring a significant computational time. For instance, this is the case in applications in computer vision such as shape from shading [18], or in image processing tasks as e.g. inpainting [26] on real-size digital images. Although the FMM is already in its basic version rather fast – much faster than a classic iterative algorithm where all the nodes are visited iteratively in a predefined update order – solving equation (1) on large grids in real-time is still out of reach. In order to obtain a significant potential speed-up of the algorithm, an interesting option is to parallelise it. In this context, let us mention that dual- and quad-core processors are by now quite common, so that there is a demand for an easy-to-use, non-massive parallelisation procedure. In this paper, we address this need. We propose a novel parallel algorithm for FMM, both numerically fast and easy to implement.

The FMM. The FMM was introduced in [22] (see also [27]), and it is based on the classic Dijkstra's shortest path algorithm for graphs [12]. A complete proof of convergence can be found in [6, 8], together with the right assumptions on the set-up that lets the method work in practice. We now briefly recall for the reader's convenience the principle of the FMM, cf. [22] for details.

We restrict the discussion to the case $d = 2$ to avoid cumbersome notations. We first introduce a bounded computational domain $\Omega \supset \Gamma_0$, discretised by a regular grid $G = \{(x_i, y_j);\, i = 1, \ldots, N_x;\, j = 1, \ldots, N_y\}$. Every cell is a square of side length Δx. We denote by u_{ij} the approximation of the solution u at (x_i, y_j). Equation (1) is discretised by means of the usual *upwind* first-order finite difference approximation [21]

$$\begin{aligned} &\left(\max\{\max\{D_{ij}^{-x}u, 0\}, -\min\{D_{ij}^{+x}u, 0\}\}\right)^2 \\ &\quad + \left(\max\{\max\{D_{ij}^{-y}u, 0\}, -\min\{D_{ij}^{+y}u, 0\}\}\right)^2 \;=\; f_{i,j}^{-2}\,, \end{aligned} \tag{2}$$

with $D_{ij}^{-x}u := \frac{u_{i,j}-u_{i-1,j}}{\Delta x}$, $D_{ij}^{+x}u := \frac{u_{i+1,j}-u_{i,j}}{\Delta x}$, and an analogous definition of $D_{ij}^{-y}u$ and $D_{ij}^{+y}u$.

It can be shown that an iterative fixed point algorithm based on this discretisation converges for any initial guess – in a large number of iterations – to the viscosity solution [21]. The idea behind the FMM is to introduce an *ordering* in the selection of the grid nodes in such a way that convergence is reached in just one iteration over the grid.

The basic FMM realises this as follows. During the computation, the grid G is always partitioned in the three sets *accepted*, *trial* and *far*. The *accepted* nodes are those where the solution has been already computed; an accepted value does not change anymore. The *trial* nodes are the nodes where the computation actually takes place. Their value can still change as long as they are labelled as *trial*. Finally, the *far* nodes are the remaining nodes where an approximate solution has not been computed yet.

For initialisation, the nodes adjacent to Γ_0 are labelled as *accepted* and their value is set to 0. All the values at non-*accepted* nodes adjacent to an *accepted* node are computed solving (2), and these nodes are labelled as *trial*. All the remaining nodes are labelled as *far* and their value is set to infinity, or just to a very large value. At every step of the algorithm, the *trial* node with the minimal value is labelled as *accepted*, and all its *far* neighbours are labelled as *trial*. Only the non-*accepted* neighbours of the last *accepted* node need to be (re-)computed using (2). The accept-the-minimum-rule is crucial, and it is based on the fact that a value can not be affected by other values larger than itself. The principle behind the latter property is called *causality principle*. The algorithm ends when all the nodes are *accepted*.

An important detail is that equation (2) can have two solutions for u_{ij}. In this case the largest computed value must be chosen, see [8, 22].
Since its introduction, the FMM has been the subject of many papers, leading to a number of improvements. Let us mention for example the works [7, 17, 34] where the authors proposed modifications to speed up the method and drop the computational complexity from $O(N \log N)$ to the optimal $O(N)$. The papers [4, 8, 11, 13] deal with modifications leading to a higher accuracy of the approximate solution. It was also proved that the FMM is not limited to equation (1), but can be used to solve other eikonal-type equations. This fact was first stated in general terms in [24], and then investigated in more detail in the recent paper [1]. Finally, several extensions of the FMM to more general Hamilton-Jacobi equations were proposed, see for example [5, 9, 10, 20, 24, 32].

Related work. Regarding parallelisation techniques for the FMM the literature is quite scarce. This is due to the fact that the FMM works in a highly sequential way. Indeed, only one node per iteration becomes *accepted*, and nodes must be computed in an increasing order so that the causality principle is respected. Up to now, there are only three notable works concerned with the parallelisation of the FMM. The paper [14] proposes a parallel algorithm based on a domain-decomposition method, and in the PhD thesis [28] that method is modified in some technical details without leading to a substantial improvement. The work [33] is instead focused on a parallel implementation using graphics processing units (GPUs). However, while the algorithm presented in that paper is somewhat similar to the FMM, it is specifically taylored to geodesic distance computation. With respect to our work, the method from [14] is important, since the FMM as the algorithmical basis in addition to the assumed underlying computer architecture are identical to our setting.
Let us remark that there exist also other parallelisation strategies for solving the eikonal equation (1). One of these approaches is based on the Fast Sweeping Method [35] for which a parallelisation was proposed in [36]. Another one is the Fast Iterative Method [15, 16] which was parallelised on GPUs as described in the same papers.

Our contribution. In this paper we propose a new parallelisation approach for the FMM which is not based on a domain decomposition technique. It is much easier to implement than the method proposed in [14]. The main idea employed by us is to split the set Γ_0 between the processors since the very beginning of the computation. Then the subsets of Γ_0 resulting from this splitting are used by separate processors as starting points for computing independent evolutions. Thereby, the process interaction is realised by a

relatively simple procedure constructed to satisfy the causality principle. By numerical tests, we validate the usefulness of the new approach. It turns out to be computationally competitive to the domain decomposition technique employed in [14], in some tests it is even more efficient.

The contents. The paper is organised as follows. In Section 2 we recall the domain-decomposition method proposed in [14]. In Section 3 our method is proposed, recalling the theoretical properties of equation (1) which let the algorithm converge to the viscosity solution. We present in Section 4 numerical tests comparing our method with that in [14]. In Section 5 we elaborate on a real-world application from the field of computer vision. The paper is finished by a summary and conclusion.

2 Domain-decomposition methods

In this section we recall the method introduced in [14]. For convenience, in the following we will refer to this method by the acronym DDM.

The set-up. At the beginning of the computation, the domain Ω is divided in P sub-domains, P being the number of processes we can run in parallel. Let us focus on two neighbouring sub-domains $\mathcal{D}_m$ and $\mathcal{D}_{m+1}$. Each domain $\mathcal{D}_m$ is extended by *ghost nodes* in the normal direction to the boundary, see Fig.1. Ghost nodes are shared by neighbouring sub-domains and allow communication between corresponding processes.

Process assignment. After the usual initialisation of the sequential FMM, each sub-domain is assigned to one process. Every process associated to a sub-domain creates his own *trial* region, and starts working independently of the others.
Of course, depending on the domain decomposition and the shape/position of Γ_0, it is possible that only one sub-domain contains Γ_0, or that just a few sub-domains contain a significant part of Γ_0. Thus, it may happen that only one process is assigned to a major part of the computation.

Communication at overlapping sub-domains. Every time a process updates a ghost node, the information is communicated to the sub-domain the node is shared with. In this way the information flowing along characteristics moves from a sub-domain to another, until the domain Ω is fully covered.
The delicate and crucial point of the DDM is to define a correct modification of a process running in a sub-domain in the case that information enters it via a ghost node. Because of the relevance of this issue, we now give more details on it.

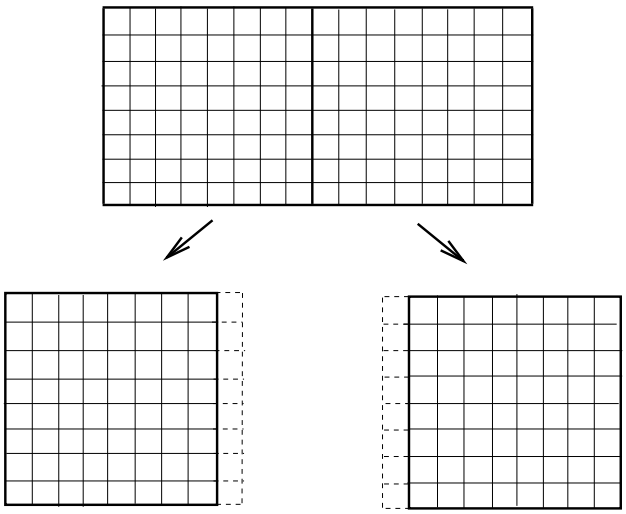

Figure 1: Classic domain decomposition with overlapping regions to allow communication between processes. The ghost nodes are indicated by dashed lines.

Let us assume that a ghost node of the sub-domain p is updated by another process by the value $\bar{u}$. We also denote by u^p_{min} the smallest value in the local *trial* zone. If $\bar{u} \geq u^p_{min}$, the ghost node is simply labelled as *trial* so that information can start propagating from that node. On the contrary, if $\bar{u} \leq u^p_{min}$, *all* the values greater than $\bar{u}$ might be wrong, since they could depend on $\bar{u}$. Hence, to allow for a consistent algorithm it is necessary to *rollback* to the status *trial* all the nodes whose value are greater than $\bar{u}$, in such a way that they can be computed again. Of course, it may be necessary to perform the rollback operation for many times, even for the same nodes, depending of Γ_0 and f.
Let us remark that also the Fast Iterative Method [15, 16] includes the rollback procedure, where it allows to "reactivate" nodes already *accepted*.

Discussion of the rollback. The rollback procedure affects a lot the efficiency of the algorithm and must be restricted as much as possible. Besides the fact that it enables the re-computation of an *accepted* node, it is required to visit all the nodes of a sub-domain to find the nodes to rollback.
We define the *rollback factor* RF as in [14], i.e.

$$RF := \frac{\text{total number of rollback operations}}{\text{total number of nodes}}. \tag{3}$$

The number RF greatly depends on the choice of the domain decomposition. In order to shed some light on the effect of the latter, let us consider the

following toy example:

$$\Omega = [-2,2]^2\,, \quad f(x) \equiv 1\,, \quad \Gamma_0 = [-2,2] \times \{-2\}\,, \quad u_0(x,y) = \frac{x+3}{2}. \quad (4)$$

The solution u of the equation and its level sets are depicted in Fig. 2. In

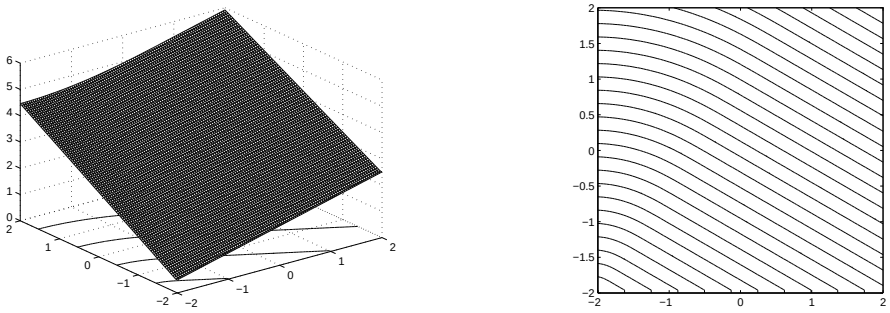

Figure 2: Solution of (1)-(4). Function u (left) and its level sets (right).

the eikonal equation characteristic lines are orthogonal to the level sets of the solution. Since information propagates along characteristics, it is easy to investigate the effect of different domain decompositions. If all the sub-

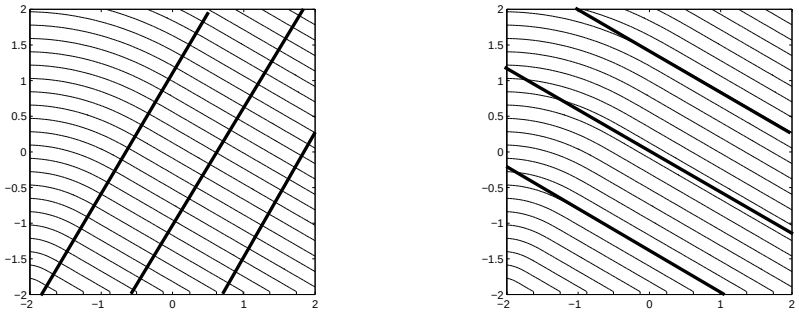

Figure 3: Different domain decompositions for (4), $P = 4$. Optimal (left), worst (right).

domain boundaries are parallel to the characteristics, the decomposition is optimal in the sense that no information does cross these boundaries. As a consequence, RF is very small or zero, this situation is depicted in Fig.3 (left).

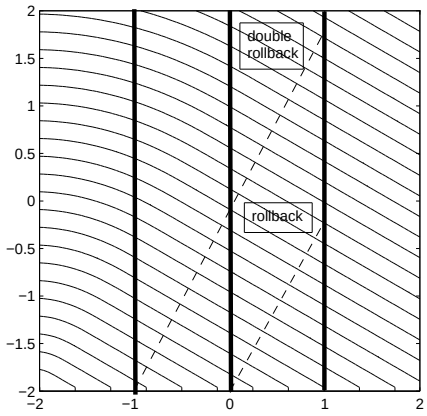

Figure 4: A generic domain decomposition for the problem (4), $P = 4$. Here the rollback appears.

On the contrary, if all the sub-domain boundaries are orthogonal to the characteristics as in Fig.3 (right), the computation becomes serial in practice, no matter how P is chosen.
For decompositions like displayed in Fig.4, the number RF appears and it could be large. In such a case, the number RF grows as P increases.
We will see in the next section that our method overcomes the difficulties in the choice of the decomposition, and it greatly simplifies the rollback procedure although it does not eliminate it completely.

3 A domain-decomposition-free method

In this section we detail our method. We start with the theoretical foundations it is based on.

3.1 Theoretical bases

It is well known that the solution u of the eikonal equation has two important physical interpretations [23]:

1. The t-level set of u

$$\Gamma_t = \{x \in \mathbb{R}^d : u(x) = t\}$$

represents a front (interface) at time t propagating in its normal direction with velocity $f(x)$, starting from the initial configuration Γ_0.

2. $u(x)$ is the minimal time to reach a target $\Gamma_0 \in \mathbb{R}^d$ when moving from the point x with velocity f, where any direction is allowed. The optimal trajectories that are the paths of such movements correspond to the characteristic curves of the equation. These coincide with the gradient lines, i.e. the curves orthogonal to the level sets of u.

In order to realise the new parallelisation idea, we start by partitioning the set Γ_0 in $P > 1$ subsets. We define a corresponding family of sets $\{\Gamma_0^p\}_{p=1,\dots,P}$, such that

$$\Gamma_0^p \subset \Gamma_0 \quad \forall p = 1, \dots, P\,, \qquad \bigcup_{p=1}^{P} \Gamma_0^p = \Gamma_0\,, \quad \text{and} \quad \bigcap_{p=1}^{P} \Gamma_0^p = \emptyset. \tag{5}$$

For any specific $p = 1, \dots, P$, this set-up leads to the sub-problem

$$\begin{cases} f(x)|\nabla u^p(x)| = 1\,, & x \in \Omega \backslash \Gamma_0^p \\ u^p(x) = u_0^p(x)\,, & x \in \Gamma_0^p\,, \end{cases} \tag{6}$$

where $u_0^p := u_0|_{\Gamma_0^p}$. Following the minimal arrival time interpretation, it is easy to see that the solution u of equation (1) can be obtained by combining the solutions of the sub-problems (6),

$$u(x) := \min_{p=1\dots,P} u^p(x) \qquad \text{for any } x \in \Omega. \tag{7}$$

3.2 The algorithm

The algorithm stems on the theoretical considerations described above. First, every subset Γ_0^p, $p \in \{1, \dots, P\}$, is assigned to the p-th thread. Then every thread solves independently of the others the corresponding sub-problem (6) computing an approximation of the solution u^p via the classic FMM. The solution of the main problem is then given by (7).

Thread interaction. To make the method efficient, we have to avoid to multiply by P *(i)* the equations to be solved, and *(ii)* the memory that needs to be allocated. To do so, we let all threads have access to a shared memory area where the matrix containing the values $\{u_{ij}\}$ is allocated. Whenever a thread is ready to write the value u_{ij}^p in the common matrix, *two rules* are followed:

R1. If some thread $q \neq p$ already wrote a value $u_{ij}^q \leq u_{ij}^p$ in a previous iteration, the thread p refrains from writing its value, and does not enlarge the *trial* zone from the node (i, j).

R2. If some thread $q \neq p$ already wrote a value $u_{ij}^q > u_{ij}^p$ in a previous iteration, or if the initial guess still holds at the node (i, j), the thread p substitutes the current value with the new one. It enlarges the *trial* zone, thereby ignoring the fact the thread q labelled that node as *accepted*.

Rule R1 is crucial to avoid unnecessary computations: In a completely automatic way, every thread stops the others at the boundary of its own domain of competence.
Since multiple threads work concurrently on a common array of results in shared memory, two or more of them might access one resource at the same time. This can not only invalidate the result in this specific point, but it can even have an immediate effect on a larger region: If reading, comparing, and updating one cell is not one atomic operation, a thread might continue its evolution to a region which is already optimal. Then, it may deteriorate the existing results.
Thus, such critical sections need to be mutual exclusive, which is solved by one *mutex flag* for any data element in the solution. This flag can be locked by one thread before it reads the current value at a point, and is unlocked after the new value has been written. In the mean time, no other thread is allowed to modify this cell, though we still allow pure read access. By use of this construction, all threads can thus work on a well-defined data set just like they would do in the single-threaded case. Since no direct thread interaction takes place, it is even algorithmically irrelevant if there are other threads running, and how many of them.
For any thread, the computation ends when all *narrow bands* are empty. The algorithm ends when all threads came to an end. The shared array then contains the desired solution.

Details on the management of *accepted, trial* and *far* labels. Different to the sequential algorithm described in the introduction, our algorithm needs a different understanding of which nodes are *far*, which are in the *trial* band, and which are *accepted*: The threads may disagree about the status of certain nodes, since the labeling always only applies to one particular wave front computed by one individual thread. Consequently, the labeling information is kept thread-local.
As it turns out, the *far* state hereby takes on a more restrictive role than in the original setting: A node is *far* – in the sense of a specific thread of interest – with respect to the new meaning,

1. if it is *far* by the old definition, and

2. if the particular thread can reach it.

The second point relates to the situation that a thread may stop the propagation of its wave front because another thread already provided a better result. In such cases, the particular thread adapts to the global notion of acceptance. Unreachable nodes at which another thread is better can still be implemented to be flagged as *far*, while in practice they are implicitly treated as accepted. This slight simplification saves complex algorithmic operations. If the second condition does not hold but the first one does, the state of the particular node is comparable to an *accepted* state in the standard FMM.
By this proceeding, the local wave propagation is steered by the preliminary solution, which acts in favour of a globally optimal result. In this respect, concrete runs of our algorithm might differ in their extent and run-time if threads do not run perfectly synchronously on the observed hardware machine.

Initialisation. In order to ensure a fair load balancing among the threads, and thus a fast convergence of the global algorithm, we need to choose the sets Γ_0^p in such a way that the emerging wave fronts ideally cover nearly the same portions of the computational domain. Obviously, this problem is intimately linked to the unknown solution and cannot be decided beforehand. We deal with this situation by taking clusters of neighbouring points within Γ_0 assigning them to the same subset Γ_0^p.
This is achieved by applying a hierarchical domain decomposition scheme as it is known from the construction of kd-trees [3]. For the description of the procedure, we consider a 2-D setting with $x = (x_1, x_2)^\top$:

1. Let $o := 0$, $n := P$. Furthermore, let $\Gamma_0^0 := \Gamma_0$.

2. First, we compute the medians m_{x_1} and m_{x_2} of coordinates of all points $\gamma \in \Gamma_0^o$ in x_1 and in x_2 direction, respectively.

3. Let $(c_{x_1}, c_{x_2})^\top$ be the center of the current domain. The new direction to split in is then given by

$$i := \underset{i \in \{x_1, x_2\}}{\operatorname{argmin}} |c_i - m_i| \,. \tag{8}$$

4. We now split at m_i in direction i, meaning all points $\gamma \in \Gamma_0^o$ are moved into $\Gamma_0^{o+n/2}$, if $\gamma_i > m_i$, and stay in Γ_0^o otherwise.

5. If $n > 2$, we proceed recursively with

   - $o \leftarrow o, \qquad n \leftarrow n/2, \quad \Gamma_0^o \leftarrow \Gamma_0^o,$ and with
   - $o \leftarrow o + n/2, n \leftarrow n/2, \quad \Gamma_0^o \leftarrow \Gamma_0^{o+n/2},$

by going back to Step 2 in both cases.

When this recursive algorithm comes to an end, we have Γ_0^p defined for all $p \in \{1, \ldots, P\}$, which concludes the initialisation phase.

Discussion of load balancing. As already indicated, a crucial task to obtain efficient parallel computations is to assure that every thread has the same computational load. This is one of the main issues in the domain-decomposition-based methods as the one discussed in the previous section, since it is expected that some threads are completely idle during the computation. Especially, this is the case when the front did not yet enter their region of competence during the computation.
In our method the load is perfectly balanced at the beginning of the computation, since the set Γ_0 is split in P equal parts. It is possible that after a while some thread ends its job. This can be the case when the part of the front of its competence region reaches the boundary of the computational domain Ω, or when it hits an existing wave front of another set by which it is overruled. In this case, it is a possible option to re-divide all the narrow band nodes in P equal parts, re-balancing the jobs. It should be noted that this procedure is very costly, and it leads to a gain in CPU time only if the runtime benefit gained exceeds the cost of the reordering step noticeably. In practice, this should be done only rarely, for very large problems and when many threads are idle. In the experiments presented in this paper we omit this option completely.

Differences with DDM. Our method differs from DDM in many respects. In the proposed method, the rollback procedure is substituted by rule R2, which allow to recompute a node already *accepted*. The procedure here is completely automatic and does not require to find in advance the nodes to be recomputed, nor store their labels at each iteration.
Let us consider again the test problem (1)-(4). In the case of four processors, the natural decomposition is the one depicted in Fig. 5. As it can be easily argued, re-computation of nodes is limited to a neighbourhood on the dashed lines, since every thread stops the others by means of the values wrote by itself, so that the processes run side by side approximately along the dashed lines, preventing double computations. In addition, every node is recomputed at most two times.
Load balance can still be a problem. As mentioned before, the load balance is optimal at the beginning of the computation (for *any choice* of the decomposition of Γ_0), but afterwords the unbalance becomes larger.

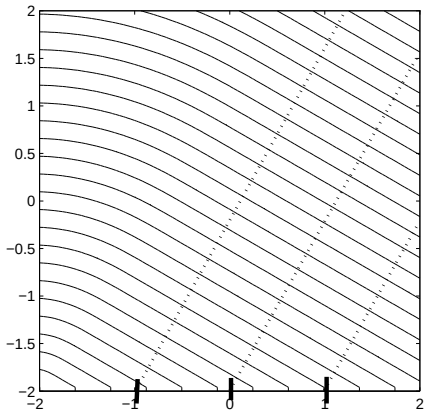

Figure 5: Region of competence for equation (1)-(4), $P = 4$

4 Numerical tests with comparisons

In this section we present some academic numerical tests. We show the efficiency of the proposed method by comparing it with the domain-decomposition method from [14] in terms of CPU time, speed-up and rollback operations. Unless stated otherwise in the specific experiment, we choose $f \equiv 1$, $u_0 \equiv 0$, and $\Omega = [-10, 10] \times [-5, 5]$. The computational domain is given by 2048×1024 nodes for these experiments.

Real Architecture. Our experiments were run on an octo-socket dual-core AMD Opteron setup clocked at 2.8 GHz. These 16 cores possess 1 MB of L2 cache, each, and share a total amount of 32 GB of RAM.
As it turns out, our algorithm scales very well on this machine using up to eight threads, but shows only little or no improvement when increasing the number to 16 threads. The reason for this behaviour can in our opinion be found in the way these threads are scheduled onto the hardware. As long as only one core per processor is occupied, the memory bandwidth suffices to provide fast updates. For two threads per processor, however, data cannot be loaded and stored quickly enough to keep the caches valid. In turn, much higher memory latencies occur, which slow down our approach significantly. With this respect, our results presented in this section are not directly comparable to the results by Herrmann [14], since the latter do not seem to be conducted on a real architecture: The author does not name a specific processor type and does not specify the available memory bandwidth and hierarchy, either. Since his results do not show any hardware-specific over-

heads which can hardly be achieved for real machines, we believe he used a simulated parallel architecture in which he only considered parallelisation overheads, but no secondary run-time effects resulting therein.

Comments on the experiments. We begin by considering the experiment which has the most relevant implications for practical computations, where the seed points are chosen randomly. Here, the benefit of our approach is also most evident. After that, we also consider a variety of specific and partly rather extreme set-ups that highlight properties of the considered parallel approaches, and also some differences between them.

4.1 Random Points

In this case, we randomly chose 32 seed points in the domain, i.e. Γ_0 is given by 32 random points. This is very likely the most relevant test case when it comes to real applications of the parallelisation schemes, since in reality one hardly experiences nicely distributed seeds. Table 1 and Figure 6 show the speedups of this experiment. The CPU time on a single thread was around 15 seconds. Like described in [14], the splitting is alternatingly performed in both directions, beginning with the x_1-direction.

Table 1: Speedups for the random points test

Method	Splitting	Threads	Speedup factor
Herrmann	Alternating	2	1.99
Herrmann	Alternating	4	3.84
Herrmann	Alternating	8	5.71
Herrmann	Alternating	16	7.10
Proposed	-	2	2.06
Proposed	-	4	3.14
Proposed	-	8	5.87
Proposed	-	16	8.65

As we can observe, we obtain nice speedups for both parallelisation techniques, with the proposed method having a clear advantage with a growing number of threads. Even for 16 threads we still obtain a remarkable speedup here, since threads can run independent for a relatively long time.
The rollback factors are also very reasonable in this case as we show in Table 2. Anyway, even though the rollback for our algorithm is still higher than for Herrmann's method, the proposed method is competitive and even faster in many cases.

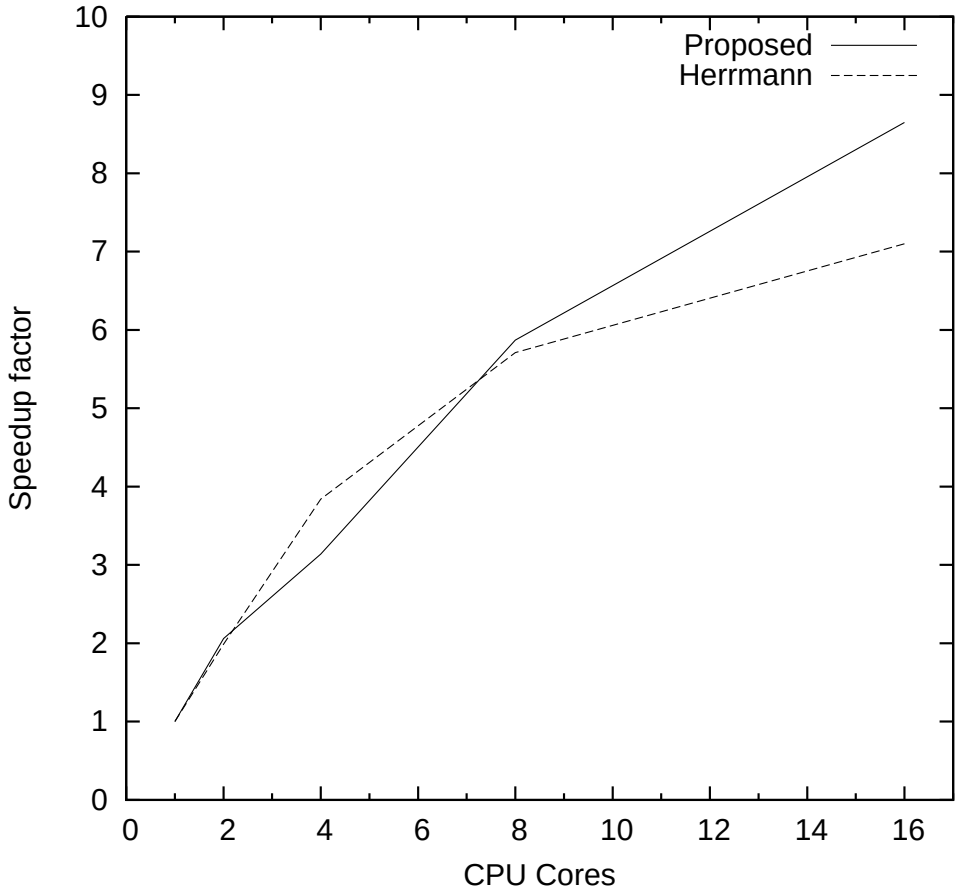

Figure 6: Speedups for the random points experiment.

This can easily be explained as follows. Because of the regular splitting, most threads are idle for a considerable amount of time in Herrmann's case, while in our case all threads are busy all the time. Finding ways to reduce the rollback in the proposed parallelisation technique would allow for even better speedups.
We finish the discussion of this experiment with the conclusion that for this test case which gives the most important clue concerning practical applications, our scheme performs competitive, and in three of four test runs shown in Table 1 even better than Herrmann's method.

Table 2: Rollback factors for the random points test

Method	Splitting	Threads	Rollback factor
Herrmann	Alternating	2	0.016
Herrmann	Alternating	4	0.012
Herrmann	Alternating	8	0.033
Herrmann	Alternating	16	0.044
Proposed	-	2	0.226
Proposed	-	4	0.339
Proposed	-	8	0.497
Proposed	-	16	0.568

4.2 Wall test

In this test, we solve the Eikonal equation with $\Gamma_0 = [-10, -9.99] \times [-5 \times 5]$. (we initialised the leftmost node in every row with the value zero). Table 3 shows the speedup factors for each of the methods. Computation time on one thread was 9 seconds for this experiment.

Table 3: Speedups for the wall test

Method	Splitting	Threads	Speedup factor
Herrmann	Vertical	2	1.00
Herrmann	Vertical	4	1.00
Herrmann	Vertical	8	0.99
Herrmann	Vertical	16	0.91
Herrmann	Horizontal	2	1.93
Herrmann	Horizontal	4	3.07
Herrmann	Horizontal	8	5.02
Herrmann	Horizontal	16	5.143
Proposed	-	2	1.88
Proposed	-	4	3.63
Proposed	-	8	4.40
Proposed	-	16	3.24

Because we know in advance about the characteristics of the solution, we did the splitting for Herrmann's method only in horizontal or vertical direction, in contrast to the original proposal. By this, we show that the splitting direction has a significant impact on the performance in practical computations. Splitting in the wrong direction results in no speedup at all, since

the method runs practically sequentially. The horizontal splitting is optimal for Herrmann's method. Figure 7 shows the speedup factors represented graphically.

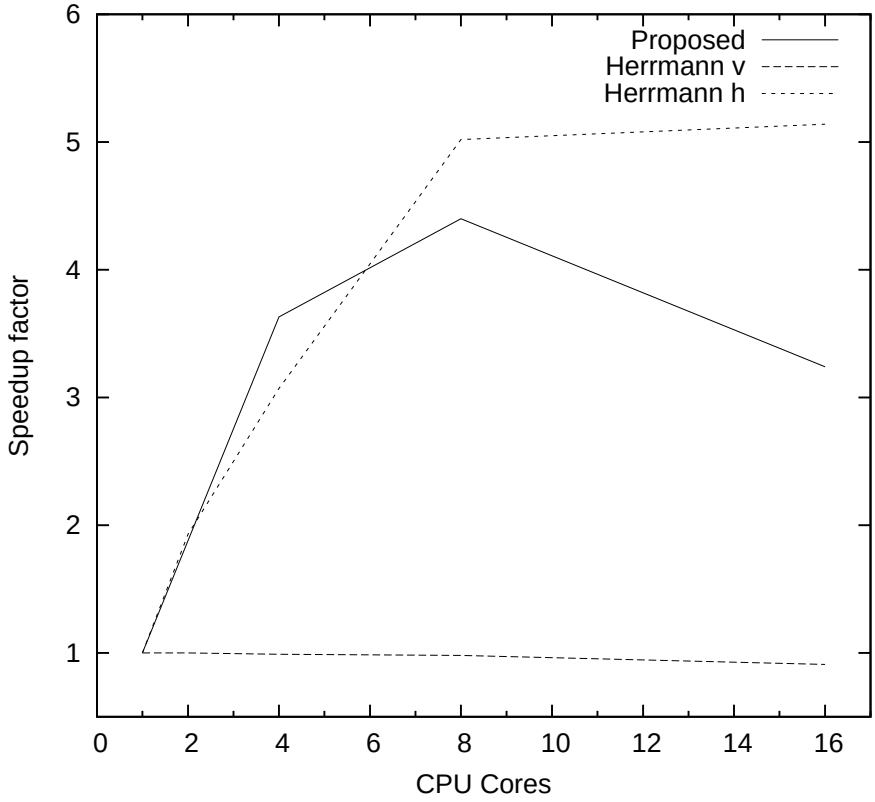

Figure 7: Speedups for the wall experiment.

As we can see, we obtain in the total slightly better speedups than Herrmann's method. Only for 16 threads, the computation time declines due to heavy caching overhead, which is caused by the architecture on which we ran the experiments and cannot be avoided on real hardware like this.
Table 4 shows the rollback factor caused by the parallelisation techniques. The rollback factor is the amount of pixels updated by more than one thread. As an example, if we use two threads on 100 pixels, and 30 of the pixels are updated by both threads, then we have a rollback factor of 0.3.
Note that since we chose an extreme example with expected wave propagations that follow thin but long line-shaped areas, a high parallelisation overhead is necessary along the boundaries between threads. This is even more severe when threads run asynchronously. This can particularly been

observed for 16 threads, when high memory latencies caused by much higher transfer rates require threads to wait for each other: Both our method as well as the reference method show a clear degradation in the speedup factor. In the proposed method, it can happen that one thread takes over the area of another thread, which in turn terminates its computation; cf. the paragraph on load balancing in Section 3.2. Then our method performs even worse using 16 threads than it does for eight. This is also reflected in the rollback numbers, see Table 4. We will see later, however, that such extreme situations as artificially constructed here are rare for real applications and much higher speedups are possible.

Table 4: Rollback factors for the wall test

Method	Splitting	Threads	Rollback factor
Herrmann	Vertical	2	0.00013
Herrmann	Vertical	4	0.00036
Herrmann	Vertical	8	0.00072
Herrmann	Vertical	16	0.00450
Herrmann	Horizontal	2	0
Herrmann	Horizontal	4	0
Herrmann	Horizontal	8	0.00025
Herrmann	Horizontal	16	0.00030
Proposed	-	2	0.00025
Proposed	-	4	0.00191
Proposed	-	8	0.06077
Proposed	-	16	0.24381

As we can observe, for the non-optimal splitting, Herrmann's method has virtually no overhead. This is not surprising, since the computation is running more or less sequential. For the optimal splitting some overhead occurs, but it is still very small. For the proposed method, we get higher rollback factors, but still very small taking into account the image size. In addition, the threads in our method tend to run a little more synchronously on this architecture, so that this slightly higher overhead is not an issue.
Because of the very small overheads here, we can conclude that in this test case, both domain splitting approaches are practically perfect, with overhead in computation time only caused by architecture-specific issues. Also, our dynamic splitting of the domains proves to give competitive speedups.

4.3 Slope Wall Test

In this case, we use a quadratic input image of size 2000×2000 where the pixel width was assumed to be 0.001 and 0 being in the centre of the image, i.e. $\Omega = [-2, 2] \times [-2, 2]$. We initialised the leftmost quarter of the image with the x_2-coordinate in the image domain. On the remainder of the image, we solved the Eikonal equation as before. The speedups for this experiment can be found in Table 5 and in Figure 8. We only considered a generic horizontal splitting direction for Herrmann's method here. This is the best of the simple, axis-aligned splitting settings by Herrmann in this test case.

Table 5: Speedups for the slope wall test

Method	Splitting	Threads	Speedup factor
Herrmann	Horizontal	2	1.84
Herrmann	Horizontal	4	3.14
Herrmann	Horizontal	8	3.11
Herrmann	Horizontal	16	2.99
Proposed	-	2	1.94
Proposed	-	4	2.35
Proposed	-	8	4.37
Proposed	-	16	6.84

For Herrmann's method, we obtain slightly worse speedups here than in the non-slope version of this test, see Section 4.2, while we obtain similar speedups for the proposed method. This can be explained by looking at the rollback factors, which are shown in Table 6. Compared to the test in Section 4.2, we observe larger rollback factors for Herrmann's method here, and similar ones for the proposed method. With around 15 seconds for a single thread, the run times are significantly larger than for the non-slope version of this test.

4.4 Circle in the centre

In this test, Γ_0 is a circle at the centre of the image domain. This is identical to the test scenario described by Herrmann in [14]. As we will see, our experiment indicates that the tests in [14] have presumably been done on simulated hardware, or under exclusion of hardware-specific characteristics. Computation time on one CPU was around 10 seconds for this experiment. We show the speedups of both methods in Table 7 as well as graphically in Figure 9. Up to 4 threads, Herrmann's method is again practically optimal.

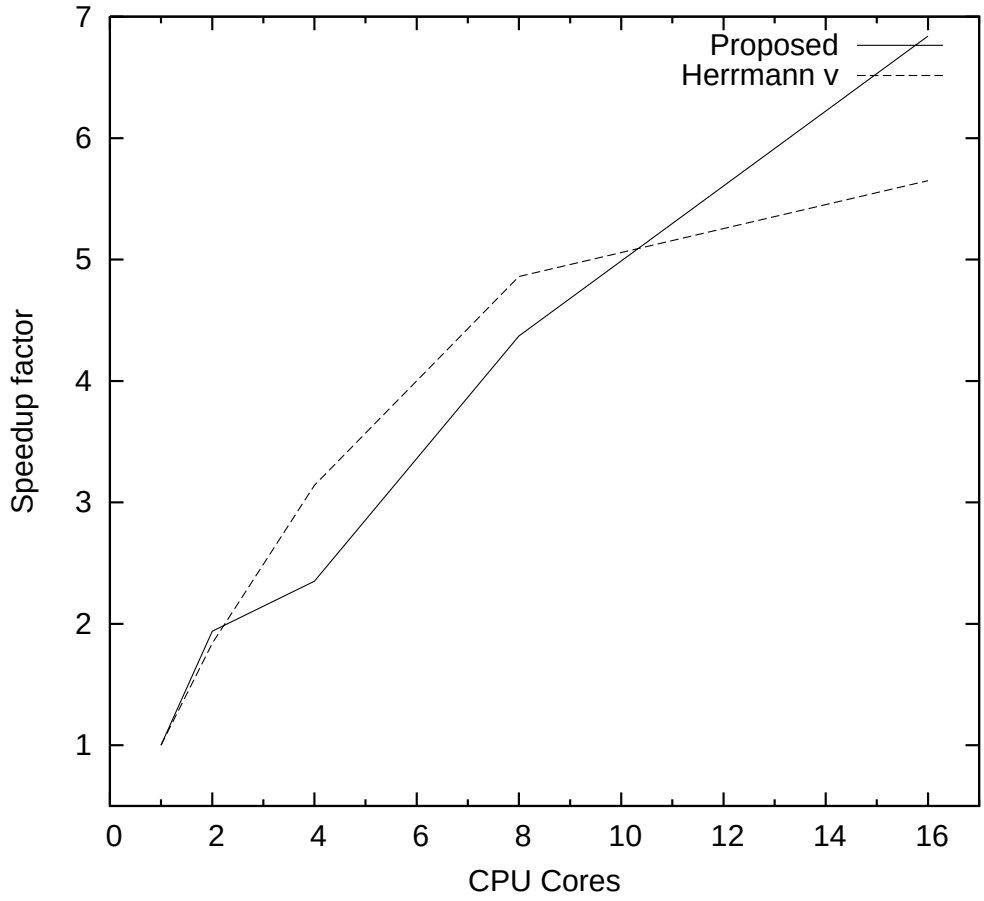

Figure 8: Speedups for the slope wall experiment.

Note that the speedup is not exactly 4, but quite a bit lower, which is caused by architecture-based as well as by synchronisation-related overheads. For only two threads, this overhead is so small that we indeed obtain a perfect speedup here. For our method, the speedups at two and four threads are worse. Since on real hardware, threads cannot be perfectly balanced, our method does not guarantee for each thread computing exactly half (or a quarter for four threads) of the domain. Instead, some regions might be a bit larger, causing a worse speedup. However, the parallelisation potential of Herrmann's method mostly stops at four cores here, since the additional split will create additional threads which are idle most of the time. Our method, however, allows for splitting in more than four sections right from the beginning, which allows to beat Herrmann's method at eight threads.

Table 6: Rollback factors for the slope wall test

Method	Splitting	Threads	Rollback factor
Herrmann	Horizontal	2	0.001
Herrmann	Horizontal	4	0.002
Herrmann	Horizontal	8	0.004
Herrmann	Horizontal	16	0.009
Proposed	-	2	0.0285
Proposed	-	4	0.065
Proposed	-	8	0.087
Proposed	-	16	0.138

The results at 16 threads are, as before, suffering from architecture-specific problems.

Table 7: Speedups for the circle in the centre test

Method	Splitting	Threads	Speedup factor
Herrmann	Alternating	2	2.01
Herrmann	Alternating	4	3.11
Herrmann	Alternating	8	3.43
Herrmann	Alternating	16	4.72
Proposed	-	2	1.39
Proposed	-	4	2.22
Proposed	-	8	3.64
Proposed	-	16	3.89

4.5 Circle in the Corner

This experiment is similar to the test before, but this time we put a quarter of a circle in a corner of the image. Table 8 and Figure 10 show the speedups of the different methods. As we can see, the speedups of the proposed method are generally better than the ones of Herrmann's method here. This is caused by Herrmann's algorithm running sequentially in the beginning, and only splitting into several threads when reaching the computational domain of another thread. In contrast, our method splits the work to different threads right from the beginning, which makes it clearly superior in this case. CPU time on just one thread was around 9 seconds.

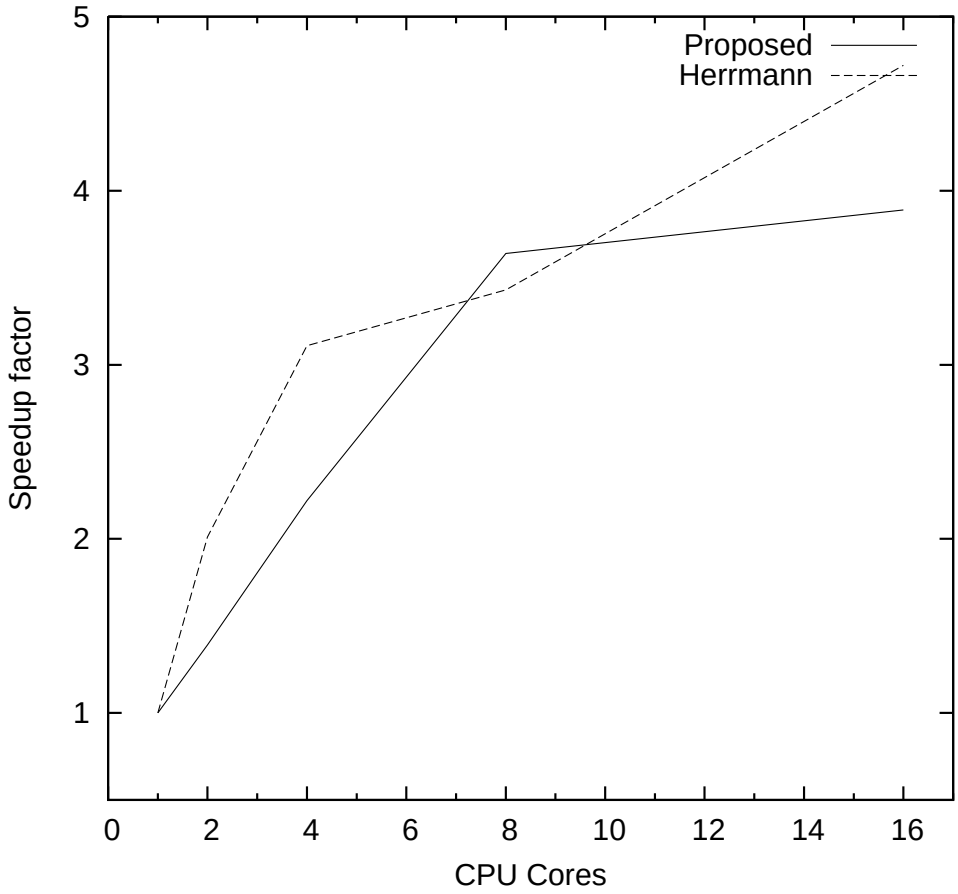

Figure 9: Speedups for the circle in the centre experiment.

5 Application to Shape from Shading

For a real-world application of our method, we consider the Shape-from-Shading-problem. This is a classic task in the field of computer vision. It amounts to recover at hand of one given grey-value image and assumptions on the illumination as well as on the reflectance properties of depicted objects their 3-D shape. While about ten years ago this task seemed to be unsolvable [37], in recent years much progress has been made beginning with the works [19, 25], so that it is now possible to deal with real-world images [31]. As these are usually taken with a digital camera with resolutions of several mega-pixels, the corresponding computational domain is quite large.
A mathematical model for Shape from Shading (SfS) that gives adequate

Table 8: Speedups for the circle in the corner test

Method	Splitting	Threads	Speedup factor
Herrmann	Alternating	2	1.04
Herrmann	Alternating	4	1.67
Herrmann	Alternating	8	1.83
Herrmann	Alternating	16	2.50
Proposed	-	2	1.27
Proposed	-	4	2.32
Proposed	-	8	2.93
Proposed	-	16	2.80

solutions for real-world images is the model of Vogel et al. [29]. The arising task is to solve the space-variant, highly non-linear Hamilton-Jacobi equation

$$J(x)W(x) - k_d I_d \exp(-2v(x)) - \frac{W(x) k_s I_s}{Q(x)} \exp(-2v(x)) \left(\frac{2Q(x)^2}{W(x)^2} - 1 \right)^{\alpha} = 0 \tag{9}$$

where k_a, k_d, k_s, α and f are model parameters, $I(x) = I_a + I_d(x) + I_s(x)$ is identical to the given grey value image, and with the abbreviations

$$Q(x) := \sqrt{\mathsf{f}^2/(|x|^2 + \mathsf{f}^2)}\,, \tag{10}$$

$$J(x) := (I(x) - k_a I_a)\mathsf{f}^2/Q(x)\,, \tag{11}$$

$$W(x) := \sqrt{\mathsf{f}^2 |\nabla v|^2 + (\nabla v \cdot x)^2 + Q(x)^2}\,. \tag{12}$$

The sought unknown depth $u(x)$ is determined via the relation $v(x) := \ln(u(x))$. The partial differential equation (9) is complemented by state constraints boundary conditions.

We apply the method to a real-world image of three chess figures, shown in Figure 11. This is quite a large image of around 8 million pixels. The computation time using a standard SfS method for such an image is several hours, and still more than a minute for fast-marching methods [30].
Figure 12 shows the reconstructions using this model. The accuracy of this reconstruction has been discussed in [30]. In this paper, however, we are mainly interested in the performance of the method under parallelisation.
Table 9 shows the speedup factors of the proposed parallelisation method for 1, 2, and 4 threads on a single Intel Core 2 Quad Q8200, 2.33 GHz, with 2×2 MB L2 Cache and 4 GB RAM. The computation times on just a single thread was about 86.5 seconds for the SfS model. Since the four cores on this machine share caches and a common memory interface, the computation

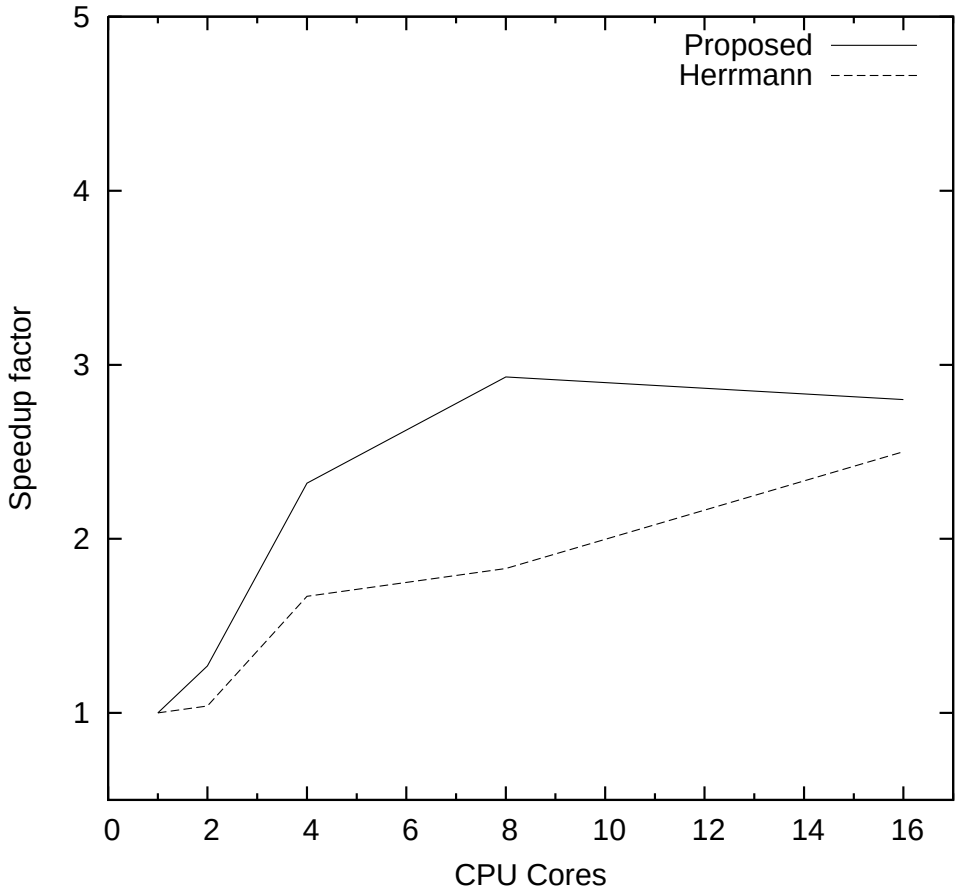

Figure 10: Speedups for the circle in the corner experiment.

times are less meaningful than the ones obtained in the previous section. However, an architecture like this has become very popular in the recent years on home PCs, therefore this experiment has practical relevance.
As we can observe, it is possible to speed the computation up by almost a factor 2.5 on four threads, even on a shared cache architecture. This reduces the computation time to about 35 seconds, which is very impressive for such a large input image.

Figure 11: Real-world chess input image: Rook, knight, and pawn.

Table 9: Speedups for the wall test

Threads	Computation Time	Speedup factor
1	86.532s	1
2	64.133s	1.35
4	35.764s	2.42

6 Summary and Conclusion

In this work we have described a new approach to parallelise the FMM that is especially useful for machines with about 4 cores that are in common use today. The method is much easier to implement than existing approaches, and it even gives better speedups in relevant experimental settings.
Our experiments reveal even better speedups if the arithmetic density of a computational problem, i.e. the ratio of arithmetic operations vs. memory interactions, is higher. In this moment, as observable by the speedup obtained with the already quite complex SfS model, more instructions are performed on a per-thread-basis, i.e. on data that is already cached. Memory bottlenecks are hence less severe since threads are better occupied with available data which in turn grants a better parallelity.

Acknowledgements

Pascal Gwosdek thanks the Cluster of Excellence *Multimodal Computing and Interaction* for funding his work.

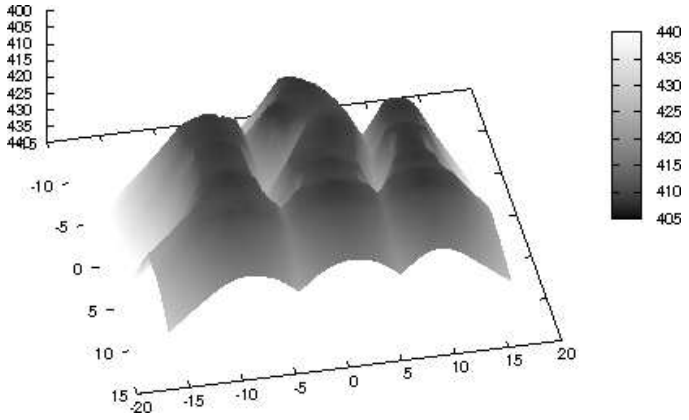

Figure 12: Reconstruction of the chess figures using the parallel fast marching SfS method.

References

[1] K. Alton, I. M. Mitchell, *Fast Marching methods for stationary Hamilton–Jacobi equations with axis-aligned anisotropy*, SIAM J. Numer. Anal., **47** (2008), 363–385.

[2] M. Bardi, I. Capuzzo Dolcetta, *Optimal control and viscosity solutions of Hamilton-Jacobi-Bellman equations*, Birkhäuser, 1997.

[3] J.L. Bentley, *Multidimensional Search Trees Used for Associative Searching*, Communications of the ACM, Vol. 18, No. 9, September 1975

[4] D. L. Chopp, *Some improvements of the fast marching method*, SIAM J. Sci. Comput., **23** (2001), 230–244.

[5] E. Cristiani, *A fast marching method for Hamilton–Jacobi equations modeling monotone front propagations*, J. Sci. Comput., **32** (2009), 189–205.

[6] E. Cristiani, *Fast Marching and semi–Lagrangian methods for Hamilton–Jacobi equations with applications*, Ph.D. Thesis, SAPIENZA – Università di Roma, Rome, Italy. February, 2007. `www.emiliano.cristiani.name/research.htm`

[7] E. Cristiani, M. Falcone, *A characteristics driven Fast Marching method for the eikonal equation*, in K. Kunisch, G. Of, O. Steinbach (eds.), Numerical Mathematics and Advanced Applications (Proceedings of ENU-

MATH 2007, Graz, Austria, September 10-14, 2007), 695–702, Springer Berlin Heidelberg, 2008.

[8] E. Cristiani, M. Falcone, *Fast semi-Lagrangian schemes for the Eikonal equation and applications*, SIAM J. Numer. Anal., **45** (2007), 1979–2011.

[9] E. Carlini, E. Cristiani, N. Forcadel, *A non-monotone Fast Marching scheme for a Hamilton-Jacobi equation modelling dislocation dynamics*, in A. Bermdez de Castro, D. Gmez, P. Quintela, P. Salgado (eds.), Numerical Mathematics and Advanced Applications, Proceedings of ENUMATH 2005 (Santiago de Compostela, Spain, July 2005), 723–731, Springer, Berlin, 2006.

[10] E. Carlini, M. Falcone, N. Forcadel, R. Monneau, *Convergence of a Generalized Fast Marching Method for an eikonal equation with a velocity changing sign*, SIAM J. Numer. Anal., **46** (2008), 2920–2952.

[11] P.-E. Danielsson, Q. Lin, *A modified Fast Marching Method*, in J. Bigun, T. Gustavsson (eds.), Proc. 13th Scandinavian Conf. Image Analysis, 1154–1161, LNCS 2749, Springer, 2003.

[12] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numerische Mathematik, **1** (1959) 269–271.

[13] M. S. Hassouna, A. A. Farag, *Multistencils Fast Marching Methods: A highly accurate solution to the eikonal equation on Cartesian domains*, IEEE Trans. Pattern Anal. Mach. Intell., **29** (2007), 1563–1574.

[14] M. Herrmann, *A domain decomposition parallelization of the Fast Marching Method*, in: Annual Research Briefs-2003, Center for Turbulence Research, Stanford, CA.

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

[16] W.-K. Jeong, R. T. Whitaker, *A fast iterative method for a class of Hamilton-Jacobi equations on parallel systems*, University of Utah, School of Computing, Technical Report UUCS-07-010, 2007.

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

[18] R. Kimmel, J. A. Sethian, *Optimal algorithm for shape from shading and path planning*, Journal of Mathematical Imaging and Vision, **14** (2001), 237–244.

[19] Prados, E., Faugeras, O.: Perspective shape from shading and viscosity solutions. In: Proc Ninth International Conference on Computer Vision. Volume 2., Nice, France, IEEE Computer Society Press (2003) 826–831

[20] E. Prados, S. Soatto, *Fast marching method for generic shape from shading*, in N. Paragios, O. Faugeras, T. Chan, C. Schnörr (eds.), Variational, geometric, and level set methods in computer vision, LNCS 3752, 320–331, Springer, 2005.

[21] E. Rouy, A. Tourin, *A viscosity solutions approach to shape-from-shading*, SIAM Journal of Numerical Analysis, 29(3):867–884, 1992.

[22] J. A. Sethian, *A fast marching level set method for monotonically advancing fronts*, Proc. Natl. Acad. Sci. USA, **93** (1996), 1591–1595.

[23] J. A. Sethian, *Level set methods and Fast Marching methods. Evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*, Cambridge University Press, 1999.

[24] J. A. Sethian, A. Vladimirsky, *Ordered upwind methods for static Hamilton-Jacobi equations: theory and algorithms*, SIAM J. Numer. Anal., **41** (2003), 325–363.

[25] A. Tankus, N. Sochen, Y. Yeshurun, *A new perspective [on] shape-from-shading*. In: Proc. Ninth International Conference on Computer Vision. Volume 2, Nice, France, IEEE Computer Society Press (2003), 862–869.

[26] A. Telea, *An image inpainting technique based on the Fast Marching Method*, Journal of graphics, gpu, and game tools, **9** (2004), 23–34.

[27] J. N. Tsitsiklis, *Efficient algorithms for globally optimal trajectories*, IEEE Tran. Automatic. Control, **40** (1995), 1528–1538.

[28] M. C. Tugurlan, *Fast Marching Methods – Parallel implementation and analysis*, Ph.D. Thesis, Louisiana State University, December 2008.

[29] O. Vogel, M. Breuß, J. Weickert, *Perspective shape from shading with non-Lambertian reflectance*, In G. Rigoll (Ed.): Pattern Recognition. Lecture Notes in Computer Science, Vol. 5096, 517-526, Springer, Berlin, 2008.

[30] O. Vogel, M. Breuß, T. Leichtweis, J. Weickert, *Fast Shape from Shading for Phong-type surfaces*, In X.-C. Tai et al. (Eds.): Scale Space and Variational Methods in Computer Vision. Lecture Notes in Computer Science, Vol. 5567, 733 - 744, Springer, Berlin, 2009.

[31] O. Vogel, L. Valgaerts, M. Breuß, J. Weickert, *Making Shape from Shading work for real-wrold images*, In J. Denzler et al. (Eds.): DAGM 2009. Lecture Notes in Computer Science, Vol. 5748, 191–200, Springer, Berlin Heidelberg, 2009.

[32] A. Vladimirsky, *Static PDEs for time-dependent control problems*, Interfaces and Free Boundaries, **8** (2006), 281–300.

[33] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, R. Kimmel, *Parallel algorithms for approximation of distance maps on parametric surfaces*, ACM Trans. Graph., **27** (2008), 1–16.

[34] L. Yatziv, A. Bartesaghi, G. Sapiro, $O(N)$ *implementation of the Fast Marching algorithm*, Journal of Computational Physics, **212** (2006), 393–399.

[35] H. Zhao, *A fast sweeping method for eikonal equations*, Math. Comp., **74** (2005), 603–627.

[36] H. Zhao, *Parallel implementations of the Fast Sweeping method*, Journal of Computational Mathematics, **25** (2007), 421-429.

[37] R. Zhang, P. S. Tsai, J. E. Cryer, M. Shah, *Shape from shading: A survey*. IEEE Transactions on Pattern Analysis and Machine Intelligence, **21** (1999), 690–706.