Shared-Memory Parallelization of the Semi-Ordered Fast Iterative Method

Josef Weinbub [1]
weinbub@iue.tuwien.ac.at

Florian Dang [2,3]
fdang@aneo.fr

Tor Gillberg [4,5]
torgi@simula.no

Siegfried Selberherr [1]
selberherr@iue.tuwien.ac.at

ABSTRACT

The semi-ordered fast iterative method is used to compute a monotone front propagation of anisotropic nature by solving the eikonal equation. Compared to established iterative methods, such as the fast iterative method, the semi-ordered fast iterative method (SOFI) offers increased stability for variations in the front velocity. So far, the method has only been investigated in a serial, two-dimensional context; in this paper we investigate a parallel implementation of SOFI (using OpenMP) and evaluate the method for three-dimensional real-world type problems. We discuss the parallel algorithm and compare its performance and its computed solutions with an OpenMP-powered fast iterative method. Different speed functions together with varying problem sizes are used to investigate the impact of the computational load. Although the semi-ordered fast iterative method is inferior to the fast iterative method with respect to parallel efficiency, we show that its execution performance is significantly faster.

Author Keywords
Semi-ordered fast iterative method; fast iterative method; eikonal equation; front propagation, OpenMP

ACM Classification Keywords
G.1.0 NUMERICAL ANALYSIS: General—Parallel algorithms

INTRODUCTION

Simulating an expanding front is a fundamental step in many computational science and engineering applications, such as image segmentation [5], brain connectivity mapping [12], medical tomography [11], seismic wave propagation [13], geological folds [8], semiconductor process simulation [18], or computational geometry [15].

[1]Institute for Microelectronics, TU Wien, Gußhausstraße 27-29/E360, 1040 Wien, Austria
[2]ANEO, 122 Ave du General Leclerc, 92100 Boulogne Billancourt, France
[3]Laboratoire PRiSM, Université de Versailles, 45 Ave des Etats-Unis, 78035 Batiment Descartes, France
[4]Bank of America Merrill Lynch, 2 King Edward St, EC1A 1HQ London, England
[5]Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway

In general, an expanding front originating from a start position Γ is described by its first time of arrival T to the points of a domain Ω. This problem can be described by solving the eikonal equation [14], which for n spatial dimensions reads:

$$\begin{aligned} \|\nabla T(\mathbf{x})\|_2 &= f(\mathbf{x}) \quad \mathbf{x} \in \Omega \subset \mathbb{R}^n \\ T(\mathbf{x}) &= g(\mathbf{x}) \quad \mathbf{x} \in \Gamma \subset \Omega \end{aligned}$$

T is the unknown solution (i.e. first time of arrival), f is an inverse velocity field (i.e. $f(\mathbf{x}) = 1/F(\mathbf{x})$), and g are boundary conditions for Γ. Generally speaking, isosurfaces to the solution represent the position of the front at a given time, and can thus be regarded as the geodesic distance relative to Γ. If the velocity $F = 1$, the solution $T(\mathbf{x})$ represents the minimal Euclidian distance from Γ to $\mathbf{x}$.

The most widely used method for solving the eikonal equation is the fast marching method [14]. This method is inherently sequential and attempts to parallelize it have been unsatisfactory [11]. Other approaches for solving the eikonal equation include the fast sweeping method (FSM) [19][20] and the fast iterative method (FIM) [10]. Both methods support parallel execution; FIM supports fine-grained parallelism, thus inherently offering greater potential for parallelism over the entire spectrum than the coarse-grained parallelism of FSM. FIM was originally implemented for parallel execution on Cartesian meshes and later extended to triangular surface meshes [6]. FIM relies on a modification of a label correction scheme coupled with an iterative procedure for the mesh point update. The inherent high degree of parallelism is due to the ability of processing all nodes of an active list (i.e. narrow band) in parallel, thus efficiently supporting a single instruction, multiple data parallel execution model. Therefore, FIM is suitable for implementations on highly parallel accelerators, such as graphics adapters [10][11]. Although FIM has been primarily investigated regarding fine-grained parallelism on accelerators, investigations on shared-memory approaches have also been conducted [3][4][18].

Although FIM provides superior parallel performance to other available methods (in most cases), its performance is problem dependent. Complex speed functions tend to increase significantly the solution time.

To overcome this shortcoming, the semi-ordered fast iterative (SOFI) method has been developed [7]; SOFI is based on both the FIM as well as on the Two-Queue method [1]. SOFI enforces an ordering to get the iterative behavior closer to front tracking methods, i.e., fast marching and wavefront tracking methods, in turn offering an increased stability, when faced with intricate speed functions. Front tracking methods inherently favor sequential execution, therefore parallel scalability is expected to be inferior to that of the FIM. Rather than computing all *active* nodes in parallel (as in FIM), the SOFI method pauses some of the awaiting updates according to a cutoff criterion based on statistical in-situ analysis of the solution values. Therefore, the computational resources are not fully used, limiting the potential for parallel speedup relative to the FIM. However, SOFI offers excellent performance for two-dimensional, sequential problems. In turn, the Two-Queue method also pauses nodes to get a partially ordered technique, but it is only applicable to isotropic problem formulations, whereas the SOFI method supports also anisotropic problems.

This work investigates the SOFI method for three dimensional problems of varying sizes and complexity based on an OpenMP parallelization. A short overview of the original SOFI method is provided, followed by a discussion of our parallel SOFI algorithm and a set of benchmarks which are used to assess the parallel execution performance of SOFI relative to a reference FIM implementation.

THE SEMI-ORDERED FAST ITERATIVE METHOD

Let X denote the set of nodes on which we want to compute the time of arrival in a solution list T, and assume that the initial distance is known at the nodes $\Gamma \subset X$. Initially, we assume that the front does not reach any nodes which are not initialised, i.e., $T(x) = \infty, \forall x \in X \setminus \Gamma$. The current front is described by the active list, aL, containing source points. In the initialisation step, all initialised nodes are added to aL. The list of nodes, containing nodes which are ahead of the source points, the paused list, pL, is initially empty but will eventually contain paused nodes, i.e., nodes that at some point will be used as source points. All nodes in aL are used as source points to evolve the front.

The solution is constructed in a semi-ordered fashion using a cutoff parameter av, which depends on the average solution m_{k} (k refers to the iteration counter) of the nodes in pL. Assume that node x_n receives a new solution value that is smaller than the old value, $t_{new} < T(x_n)$ and that in addition $t_{new} \leq av$ then x_n is added to the end of aL, and thus used as a source point. If instead $t_{new} > av$, we postpone its function as a source by adding x_n to pL. When there are no source points left, no nodes in X can get a lower arrival time. Figure 1 schematically depicts the evolution of the aL and pL container.

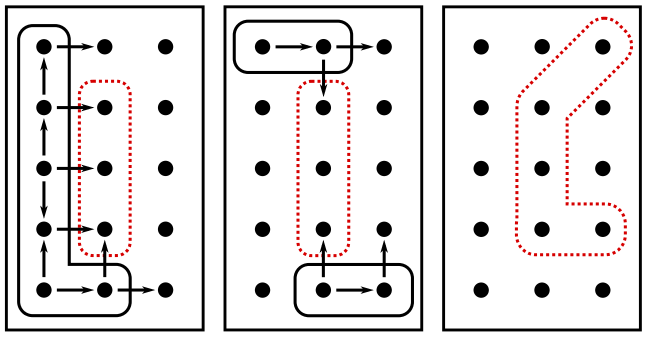

Figure 1: The evolution of the active list (black, solid boxes) and paused list (red, dashed boxes) for different iteration steps (from left to right) is shown schematically. Arrows point from nodes in aL to nodes being re-evaluated. Nodes in the active list are gradually updated until the active list is empty. The nodes in the paused list have a higher solution value than the applied cutoff value.

PARALLEL ALGORITHM

The parallel algorithm follows the original SOFI algorithm [7] to a large extent, albeit offering additional handling of shared-memory parallel programming aspects and an advanced cutoff method suitable for three-dimensional problems. We first introduce the required general data objects, discuss the initialization step and the actual parallel algorithm, and finally conclude with an analysis of the developed semi-automatic cutoff criterion.

Additionally to the originally used algorithm entities, for our parallel approach we propose a temporary aL_{temp} container to avoid expensive deletion processes of aL during the compute-intensive iterations. Also, we use a counter (aL-swaps) to track the number of swaps between aL and aL_{temp}. An essential aspect of the algorithm is the determination whether a node has been already added to aL or pL. To avoid an expensive lookup step, which would require finding the node in question within aL or pL, we use a tag-based system. To that end, we employ the aL_{tags} and pL_{tags} data structures, which provide us with element-based tag lookup for the expense of additional memory overhead. The coefficients c, $relax$, m_{k}, and $m_{\mathrm{k}-1}$ are required for our improved automatic cutoff computation, which will be explained later on.

The initialization of the parallel SOFI algorithm sets all coefficients to zero (e.g. av, aL-swaps, t_{sum}), and the solution field is preloaded with an arbitrarily high number (e.g. 10^{12}). However, the solution of each source point is initialized with zero, whereas the source nodes themselves (representing the input of the algorithm) are added to the active list aL. This is different from FIM, where neighbours of source nodes (rather than source nodes) are added to the active list.

Upon convergence of the algorithm, the result of the algorithm (i.e. the signed distance field) is stored in the solution list T.

Algorithm 1 introduces the actual parallel SOFI algorithm. The main parallel loop is - as with the FIM - processing the active list aL. We use a *guided* scheduling method, as it has shown to be the best performing scheduling procedure, due to the irregular workload inside the parallel loop demanding a dynamic load balancing. The tag system ensures that the same nodes are not added to aL/pL again during an iteration (Lines 3,9,10,19,20,42). However, nodes might be reprocessed later on during a subsequent iteration, such is the general procedure of iterative methods. The use of write guards in form of atomic locks has been minimized to three spots (Line 10,20,25). The neighbor (nb) iteration is required to generate the required 7-point stencil (Line 4), which is used to discretize the eikonal equation's differential operator in three dimensions (Line 6). Parallel write access to the pL and aL_{temp} data structures has been realized via thread-exclusive containers (Lines 13,21), which - although requiring a serial merging step at the end of the parallel for loop (Line 29,41) - scales better for increasing thread numbers than guarding central data structures with additional critical sections. A similar technique is used for the cutoff procedure's essential coefficients t_{sum} and t_{sqsum} (Lines 11,12,15,16). The thread-exclusive aL_{temp} and pL containers are merged into their global counterparts in serial *merge* steps (Lines 29,41), and are reset (i.e. the arrays are cleared) to prepare for subsequent iterations.

For the SOFI method to perform well, the algorithm for computing the cutoff coefficient av is vital, as av controls the assignment of a node to either aL or pL (Line 8,18). The cutoff level enforces an ordering of the nodes to be updated, in order to reduce the number of iterations needed. When too many nodes are activated (i.e. added to aL), the number of computations is high and the numerical solvers are slow. Similarly, if too many nodes are paused (i.e. added to pL), too few nodes are computed, as the ordering is too strict. Empirical investigations have shown best performance when approximately 80% of the nodes are activated [1][7].

The original method used for computing av is based on the average solution value of the paused nodes [7]. However, this approach does not perform well for general problems in three dimensions. The ordering enforced from this simple method tends to be too weak, since too many nodes are activated by being put into aL. When that happens, the additional cost of ordering computations outgrows its benefits. Therefore, we use a different method to compute the cutoff av, being based not only on the average solution value m_k but also on the standard deviation σ of the nodes in pL.

Algorithm 1 SOFI Parallel Algorithm

```
1: while $aL \neq \emptyset$ do
2:   for all $x \in aL$ #parallel, guided do
3:     $aL_{\text{tags}}(x) = 0$ #guard
4:     for all $x_{\text{nb}}$ of $x$ #edge-connected do
5:       if $T(x) < T(x_{\text{nb}})$ #downwind condition then
6:         $t_{\text{new}} = SolveEikonal(x_{\text{nb}})$
7:         if $t_{\text{new}} < T(x_{\text{nb}})$ then
8:           if $t_{\text{new}} > av$ then
9:             if $pL_{\text{tags}}(x_{\text{nb}}) == 0$ then
10:              $pL_{\text{tags}}(x_{\text{nb}}) = 1$ #guard
11:              $t_{\text{sum}} += t_{\text{new}}$
12:              $t_{\text{sqsum}} += t_{\text{new}} \cdot t_{\text{new}}$
13:              Add $x_{\text{nb}}$ to $pL$
14:            else
15:              $t_{\text{sum}} += t_{\text{new}} - T(x_{\text{nb}})$
16:              $t_{\text{sqsum}} += t_{\text{new}} \cdot t_{\text{new}} - T(x_{\text{nb}}) \cdot T(x_{\text{nb}})$
17:            end if
18:          else
19:            if $aL_{\text{tags}}(x_{\text{nb}}) == 0$ then
20:              $aL_{\text{tags}}(x_{\text{nb}}) = 1$ #guard
21:              Add $x_{\text{nb}}$ to $aL_{\text{temp}}$
22:            end if
23:          end if
24:        end if
25:        $T(x_{\text{nb}}) = Min(T(x_{\text{nb}}), t_{\text{new}})$ #guard
26:      end if
27:    end for
28:  end for
29:  Merge ($aL_{\text{temp}}$); Swap ($aL, aL_{\text{temp}}$); Reset ($aL_{\text{temp}}$)
30:  if $aL$-swaps $> \sqrt[n]{Size(X)}/10$ then
31:    $av = 0.0$
32:  end if
33:  if $av > m_{\text{k}}$ then
34:    if $pL$-ratio $< 0.5\%$ and $aL$-swaps $> 5$ then
35:      $c = 0.8c$; $av = m_{\text{k}} - relax$
36:    else if $pL$-ratio $> 99\%$ and $aL$-swaps $< 5$ then
37:      $c = 2.0c$; $av = m_{\text{k}} - relax$
38:    end if
39:  end if
40:  if $aL == \emptyset$ then
41:    Merge ($pL$); Swap ($aL, pL$);
42:    Swap ($aL_{\text{tags}}, pL_{\text{tags}}$); Reset ($pL$)
43:    $m_{\text{k}} = \sum t_{\text{sum}} / Size(aL)$
44:    $\sigma = \sqrt{\sum t_{\text{sqsum}} / Size(aL)}$
45:    $relax = c(2(m_{\text{k}} - m_{\text{k-1}}) + \sigma)^2 / 6\sigma^2$
46:    $av = m_{\text{k}} + relax$; $m_{\text{k-1}} = m_{\text{k}}$
47:    $t_{\text{sum}} = t_{\text{sqsum}} = 0.0$
48:  end if
49: end while
```

Assuming a normal distribution of the solution values of nodes in pL, we would activate approximately 84% by assigning a cutoff av of the average plus a standard deviation. However, a large spread (i.e. large σ) within pL indicates that a stricter ordering is needed. We estimate the average shift in cutoff level, by the difference between the current m_k and the previous: $\Delta m_\mathrm{k} = m_\mathrm{k} - m_\mathrm{k-1}$. The original SOFI relaxation method is to have a cutoff level as a *relaxed* average by using $av = m_\mathrm{k} + 1.5\Delta m_\mathrm{k}$. Trying to factor in the indications from σ we have found that the following cutoff performs well (Lines 43-46):

$$av = m_\mathrm{k} + c\frac{(2\Delta m_\mathrm{k} + \sigma)^2}{6\sigma^2}$$

The additional coefficient c and *relax* are used as additional parameters to adjust the cutoff computation, by investigating the pL-ratio, i.e., the number of nodes added to pL relative to the total number of nodes added to pL and aL (Lines 33-39). The used thresholds and coefficients have been shown to work best for the presented examples, but may be adjusted for more realistic devices, hence the designation *semi-automatic*.

Another mechanism to ensure that the computed cutoff level is reasonable is proposed, which is based on monitoring the number of iterations (Lines 30-32). If too many iterations are detected, it is assumed that the cutoff level is not optimal. Therefore, the cutoff procedure is restarted by triggering a recomputation of the cutoff level, increasing the chance of upholding a high convergence rate.

BENCHMARKS

We investigate the performance of our parallel SOFI implementation relative to a reference FIM implementation. Our benchmarks cover different three-dimensional problems with varying problem sizes (100^3 and 200^3 Cartesian cube grids), speed functions, and single/multiple-source configurations (a single center source node versus 100 source nodes spread over the entire simulation domain).

Regarding speed functions, we investigate three different configurations: (1) constant speed (F_const), where for the entire domain $F = 1$ is used; (2) checkerboard speed (F_check), where the computational domain is divided into eleven equally sized cubes in each direction and the velocity is alternated between $F = 1$ to $F = 2$ from cube to cube [2][8]; (3) oscillatory speed (F_osc), where the speed function is modeled by a highly oscillatory continuous speed function, being $F = 1 + \frac{1}{2}sin(20\pi x)sin(20\pi y)sin(20\pi z)$ [2]. The benchmark platform is a dual-socket node with two Intel Xeon E5-2620 (SandyBridge EP) 6-core (2 threads per core) processors with 128 GB of main memory, powered by a 64-bit GNU/Linux and the GNU/GCC compiler 4.9.2. The parallel algorithm introduced in the previous section has been implemented in C++. The presented execution performances are based on the median of five execution timings. The threads have been pinned to the individual physical cores via the likwid [17][16] library to avoid thread-core reassignments, which would otherwise potentially introduce a performance penalty.

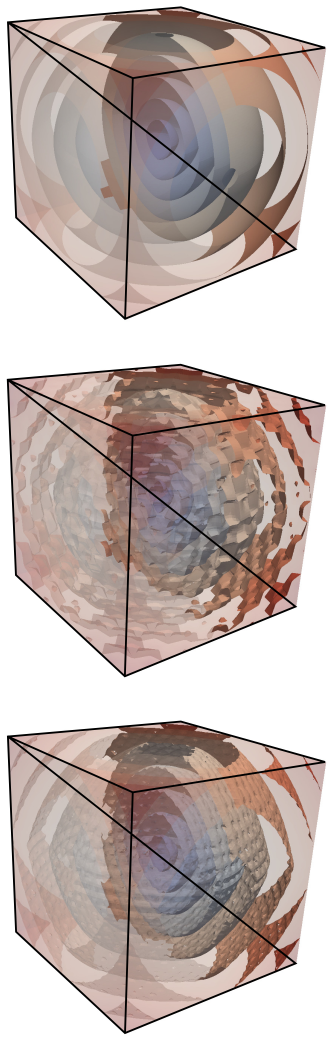
Figure 2: Isosurfaces of the F_const (top), F_check (middle), and F_osc (bottom) solution on a 100^3 domain for a single center source

Figure 2 depicts the isosurfaces of the solutions of the center test configurations for the 100^3 simulation grids, to provide a frame of reference for the benchmark setup and the solutions. The results for the 200^3 are similar, albeit offering an increased resolution. To verify the correctness of the solutions, the FIM and the SOFI method results of the single source problem with constant speed for a 100^3 grid have been compared to an analytic solution given by the Euclidian distance function. The error norms of both methods are the same, being $L_1 = 29 \cdot 10^{-3}$, $L_2 = 10^{-3}$, and $L_\infty = 36 \cdot 10^{-3}$, indicating that the SOFI method computes the same result as the FIM. If indeed the results would be different, the ε used in the FIM's algorithm can prevent full convergence of the algorithm.

Figures 3-5 compare the execution times and the parallel efficiency between our SOFI method and FIM implementation for a 100^3 grid. A logarithmic scaling is used to highlight the variances in the execution time, especially relevant for high thread numbers. The SOFI method outperforms the FIM both for the single and multiple source configurations; for the more important multiple source setups (as these cases resemble real-world applications more closely) and 12 threads, a speed-up factor of 1.5 for F_{const}, 2 for F_{check}, and 1.7 for F_{osc} is achieved. The parallel efficiency of the single source test setups is by far inferior to the FIM, although both methods suffer in general from efficiency limiting factors typical for stencil computations, being cache misses and memory latency [9]. This stems from the fact that the SOFI method inherently does not favor single source problems, as in this case no ordering is needed, thus introducing unnecessary overhead. However, for the more important multiple source cases, the scalability is reasonable: for 12 threads efficiencies of around 60% can be achieved for the highly challenging F_{check} and F_{osc} problems. The results show load balancing problems, which can be identified by the somewhat erratic parallel efficiency behavior. This fact is to be attributed to an unbalanced utilization of the aL and pL containers, triggered by insufficiencies in our automatic cutoff calculation.

Figures 6-8 continue the investigation for an increased computational domain size, being 200^3, which allows to judge the performance under increased load. Again, execution timings show that the SOFI method is faster than FIM. For the multiple-source cases and 12 threads, a speed-up factor of 1.9 for F_{const}, 2 for F_{check}, and 2.6 for F_{osc} is achieved. The parallel efficiency is comparable to the 100^3 grid results, being around 60% for the F_{check} and F_{osc} problems.

Overall, the previously mentioned inferior parallel potential of the SOFI method relative to the FIM is reflected in the results, albeit being still reasonable, especially for more relevant multiple source scenarios. However, the execution time is what matters in real world applications. In this light, the parallel SOFI method is significantly superior to the parallel FIM, underlining the potential of the parallel SOFI method as a compelling alternative for solving the eikonal equation in applied numerical simulations arising from the diverse field of computational science and engineering.

CONCLUSION

An approach for parallelizing the SOFI method via a shared-memory OpenMP technique has been introduced. An alternative cutoff method supporting three-dimensional problems has been discussed, for semi-automatically driving the applied iterative Two-Queue technique. Our parallel SOFI algorithm offers superior execution performance relative to a reference FIM implementation for different speed functions and problem sizes, while offering reasonable parallel efficiency. This work shows the excellent capabilities of the SOFI method for tracking front propagation paving the way for further investigations regarding real-world applications.

ACKNOWLEDGMENTS

This work has been supported by the Austrian Science Fund (FWF) through the grant P23296. The authors thank Joachim Schöberl for providing access to a dual-socket Intel Xeon system for benchmarking purposes.

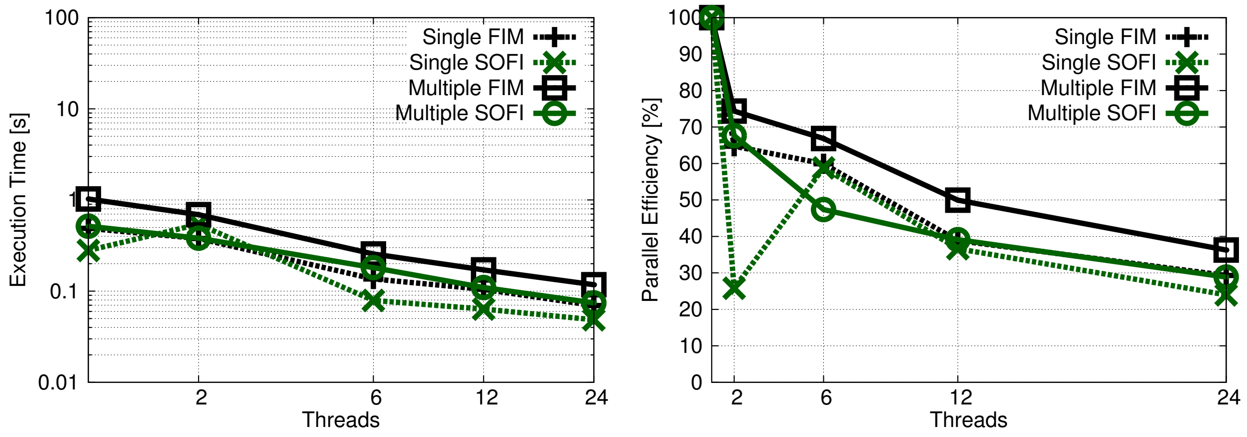

Figure 3: Execution times (left) and parallel efficiencies (right) of the F_{const} problem on a 100^3 domain

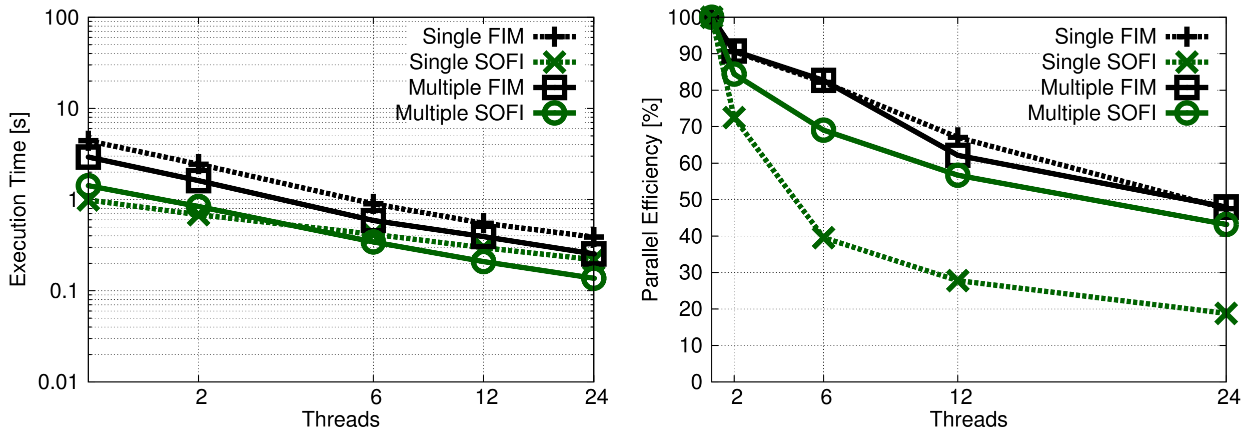

Figure 4: Execution times (left) and parallel efficiencies (right) of the F_{check} problem on a 100^3 domain

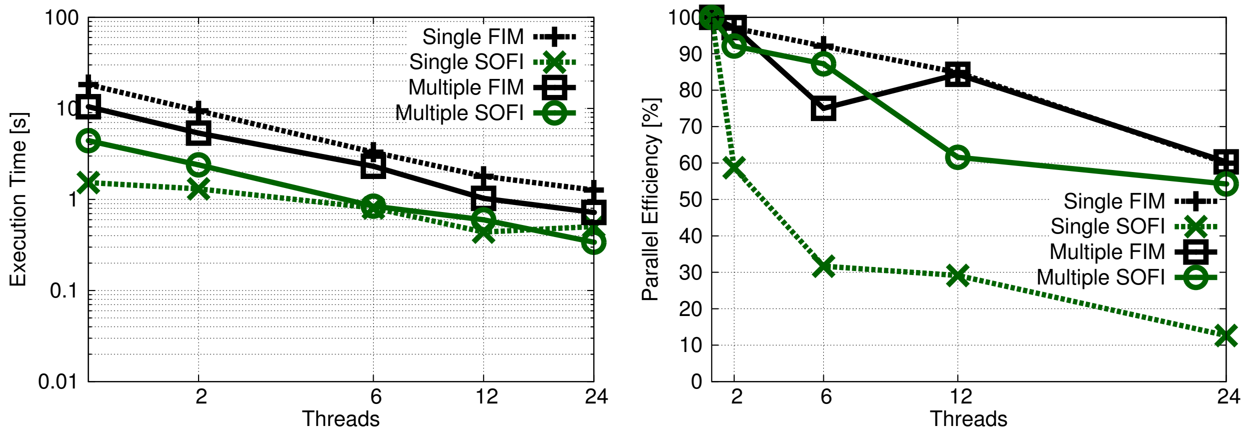

Figure 5: Execution times (left) and parallel efficiencies (right) of the F_{osc} problem on a 100^3 domain

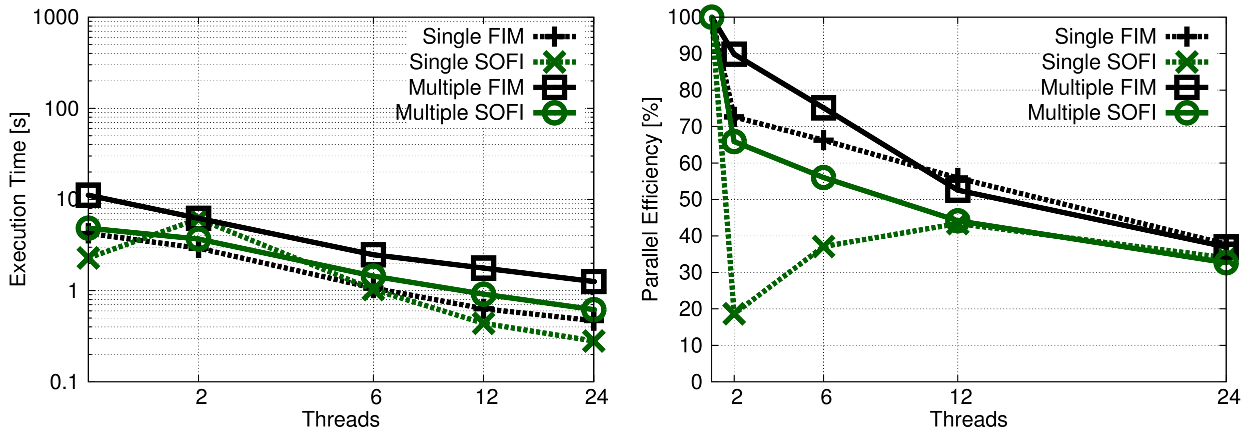

Figure 6: Execution times (left) and parallel efficiencies (right) of the F_{const} problem on a 200^3 domain

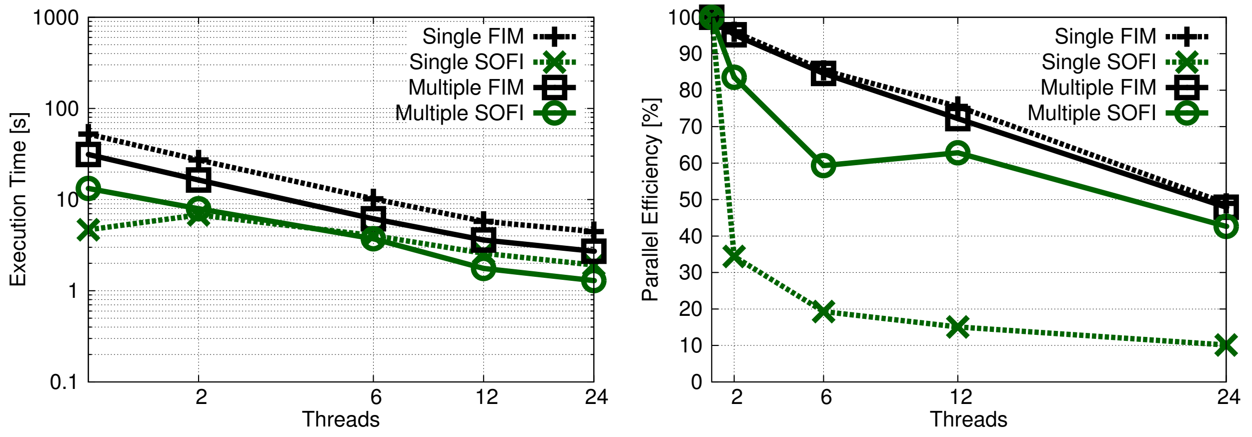

Figure 7: Execution times (left) and parallel efficiencies (right) of the F_{check} problem on a 200^3 domain

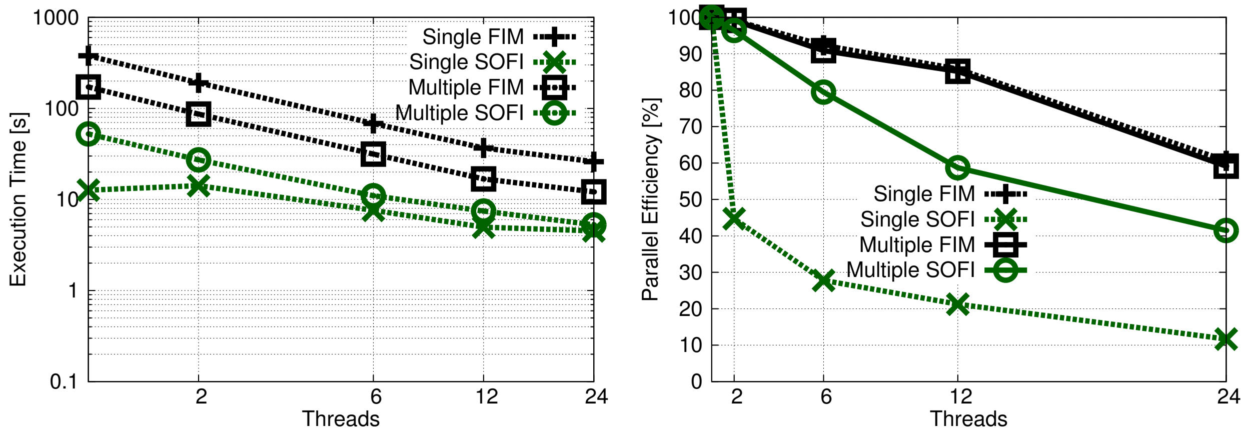

Figure 8: Execution times (left) and parallel efficiencies (right) of the F_{osc} problem on a 200^3 domain

REFERENCES

1. Bak, S., McLaughlin, J., and Renzi, D. Some Improvements for the Fast Sweeping Method. *SIAM Journal on Scientific Computing 32*, 5 (2010), 2853–2874. DOI: 10.1137/090749645.

2. Chacon, A., and Vladimirsky, A. Fast Two-Scale Methods for Eikonal Equations. *SIAM Journal on Scientific Computing 34*, 2 (2012), A547–A578. DOI: 10.1137/10080909X.

3. Dang, F., and Emad, N. Fast Iterative Method in Solving Eikonal Equations: A Multi-level Parallel Approach. *Procedia Computer Science 29* (2014), 1859–1869. DOI: 10.1016/j.procs.2014.05.170.

4. Dang, F., Emad, N., and Fender, A. A Fine-Grained Parallel Model for the Fast Iterative Method in Solving Eikonal Equations. In *Proceedings of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)* (2013), 152–157. DOI: 10.1109/3PGCIC.2013.29.

5. Forcadel, N., Le Guyader, C., and Gout, C. Generalized Fast Marching Method: Applications to Image Segmentation. *Numerical Algorithms 48*, 1-3 (2008), 189–211. DOI: 10.1007/s11075-008-9183-x.

6. Fu, Z., Jeong, W. K., Pan, Y., Kirby, R., and Whitaker, R. T. A Fast Iterative Method for Solving the Eikonal Equation on Triangulated Surfaces. *SIAM Journal on Scientific Computing 33*, 5 (2011), 2468–2488. DOI: 10.1137/100788951.

7. Gillberg, T. A Semi-Ordered Fast Iterative Method (SOFI) for Monotone Front Propagation in Simulations of Geological Folding. In *Proceedings of the International Congress on Modelling and Simulation (MODSIM)* (2011), 631–647.

8. Gillberg, T., Bruaset, A. M., Hjelle, Ø., and Sourouri, M. Parallel Solutions of Static Hamilton-Jacobi Equations for Simulations of Geological Folds. *Journal of Mathematics in Industry 4*, 10 (2014), 1–31. DOI: 10.1186/2190-5983-4-10.

9. Hager, G., and Wellein, G. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010. ISBN: 978-1439811924.

10. Jeong, W. K., and Whitaker, R. T. A Fast Iterative Method for Eikonal Equations. *SIAM Journal on Scientific Computing 30*, 5 (2008), 2512–2534. DOI: 10.1137/060670298.

11. Li, S., Mueller, K., Jackowski, M., Dione, D., and Staib, L. Physical-Space Refraction-Corrected Transmission Ultrasound Computed Tomography Made Computationally Practical. In *Lecture Notes in Computer Science*, vol. 5242. 2008, 280–288. DOI: 10.1007/978-3-540-85990-1_34.

12. Prados, E., Soatto, S., Lenglet, C., Pons, J.-P., Wotawa, N., Deriche, R., and Faugeras, O. Control Theory and Fast Marching Techniques for Brain Connectivity Mapping. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1 (2006), 1076–1083. DOI: 10.1109/CVPR.2006.89.

13. Rawlinson, N., and Sambridge, M. Multiple Reflection and Transmission Phases in Complex Layered Media Using a Multistage Fast Marching Method. *Geophysics 69*, 5 (2004), 1338–1350. DOI: 10.1190/1.1801950.

14. Sethian, J. A. A Fast Marching Level Set Method for Monotonically Advancing Fronts. *Proceedings of the National Academy of Sciences 93*, 4 (1996), 1591–1595.

15. Sethian, J. A. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999. ISBN: 978-0521645577.

16. Treibig, J., et al. LIKWID - Webpage, 2014. `https://code.google.com/p/likwid/`.

17. Treibig, J., Hager, G., and Wellein, G. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW)* (2010), 207–216. DOI: 10.1109/ICPPW.2010.38.

18. Weinbub, J., and Hössinger, A. Accelerated Redistancing for Level Set-Based Process Simulations with the Fast Iterative Method. *Journal of Computational Electronics 13*, 4 (2014), 877–884. DOI: 10.1007/s10825-014- 0604-x.

19. Zhao, H. A Fast Sweeping Method for Eikonal Equations. *Mathematics of Computation 74*, 250 (2005), 603–627. DOI: 10.1090/S0025-5718-04-01678-3.

20. Zhao, H. Parallel Implementations of the Fast Sweeping Method. *Journal of Computational Mathematics 25*, 4 (2007), 421–429.