Multi-level parallel upwind finite difference scheme for anisotropic front propagation

Florian Dang[1 2] and Nahid Emad[2 3]

[1] Silkan, Meudon-La-Forêt, France
[2] Université de Versailles, Laboratoire PRiSM, 78035 Cedex Versailles, France
[3] Maison de la Simulation, CNRS, Bâtiment 565 - Digiteo, Gif-sur-Yvette, France

Abstract. Considering a cartesian grid we reinterpret Dijkstra's famous algorithm as an upwind finite difference scheme. The scheme is able to compute approximations of static Hamilton-Jacobi PDEs which arise in front propagation. The method which manage the update of the grid points has the particularity to highly fit parallel purpose. We develop a multi-level parallel strategy which can target a wide range of parallel architectures. We show the efficiency of our approach with several numerical examples, in two and three dimensions and we compare to recent state of the art.

1 Introduction

Hamilton-Jacobi equations arise in large range of applications including chemistry [4], image segmentation [5], robotics [6, 7], etc... Static Hamilton-Jacobi equations take the following form :

$$H(x, \nabla u(x)) = F, x \in \Omega \subset \mathbb{R}^n, F > 0 \tag{1}$$

where H is a Lipschitz-continuous Hamiltonian and Ω an open subset and F represent a slowness field. We use a first-order upwind finite difference scheme to approximate the above equation (1). In the two dimensional case, this leads to :

$$\max(D^x_- u_{i,j}, -D^x_+ .u_{i,j}, 0)^2 + \max(D^y_- u_{i,j}, -D^y_+ u_{i,j}, 0)^2 = F_{i,j} \tag{2}$$

where $D^x_- u_{i,j} = \frac{u_{i,j} - u_{i-1,j}}{h}$ and $D^x_+ .u = \frac{u_{i+1,j} - u_{i,j}}{h}$ with h the grid spacing. The forward and backward operators D^y_- and D^y_+ work similarly.

Several numerical methods have been developed to compute the solutions of equation (1) where a particular form is the famous Eeikonal equation. Rouy and Tourin [11] propose to use an iterative method with an upwind discretization scheme (such as eq. (2) when computing ∇u. Nowadays, two faster methods are prefered by computing scientists : the fast marching method (FMM) proposed by Sethian [12] and the fast sweeping method [14] proposed by Zhao. These methods also use an upwind scheme but the points are updated differently in a specific order. For isotropic front propagation the reader can refer to ordered upwind methods by Sethian and Vladimirsky [13]. The fast marching method (FMM) is a one-pass method based on a a narrow band management. Narrow band points contain grid points where the computation will take place. They are neighbors of accepted points where the solution is already known. Points in the narrow band are updated in the order of propagating solutions. The fast sweeping method (FSM) alternates sweep ordering of Gauss-Seidel iterations on the whole grid until convergence. Parallel works on these methods are quite scarce due to the fact that these methods are not easily parallelizable. The FMM has been parallelized using classical and adaptative domain-decomposition methods in [1] and [8]. Zhao already propose a parallel FSM for its own method in [15] and recently, a different parallel FSM has been proposed in [3]. We propose in this paper a multi-level parallel strategy using a different way of managing the upwind scheme with the fast iterative method [10, 9] firstly designed for GPGPU. We propose a new version of the algorithm compliant with shared-memory architectures, we also propose a coarse-grained parallel strategy and we show experiments in several contexts.

2 Managing the upwind scheme for multi-level parallel purpose

The fast marching method ordering recquires a sorting algorithm which is in $O(log(N))$ where N is the number of grid points. The FMM has therefore a complexity of $O(N.log(N))$. Recent state of the art on parallel FMM indicates that the method is not efficient (see [1, 8]) compared to parallel FSM. Indeed, sorting the narrow band heap structure used in the FMM hinders performance. Jeong and Whitaker observe that we have to design fast parallel algorithms for solving the eikonal equation on parallel architectures. Thus, the authors have developped the FIM firstly for GPGPU purpose [9, 10]. The fast iterative method (FIM) is based on the Rouy-Tourin method [11] and the fast marching method. Sequential FIM is generally faster than sequential FMM or FSM. The reader is invited to refer to [10] for more details on the FIM. Briefly, the FIM narrow band (called the active list by the authors) is wider and contains more points compared to the FMM narrow band so that there is no need for sorting at every iteration. In consequence multiple points can be updated simultaneously. We show in this paper that the FIM also scales well on other parallel architectures.

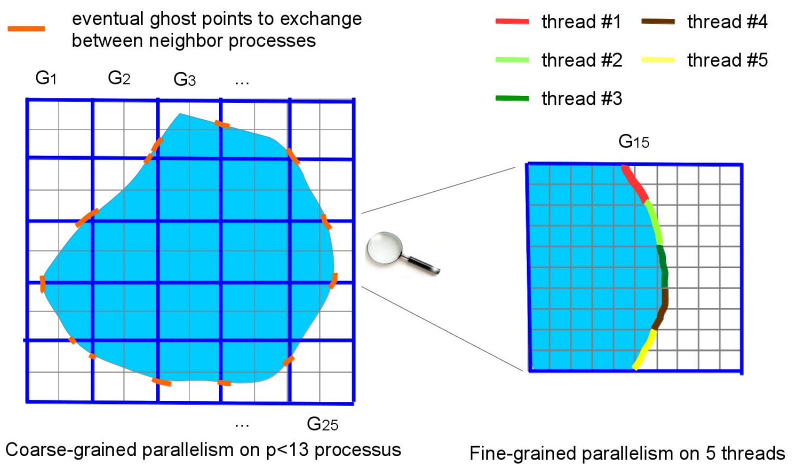
2.1 Narrow band partitioning

Jeong and Whitaker GPU parallelization on the FIM is based on a block-based update scheme which can be compared to a domain decomposition. Synchronization issues are managed by reduction calls at every iteration. Using this strategy on other parallel systems than GPUs-like ones is likely to give poor performance since the synchronization calls among every threads would be too costly and would not be recovered by the computing process. We propose to use a different strategy based on an narrow band partitioning which target shared-memory architectures such as multi-core processing. We can refer to our first fine-grained parallel model in [2]. We propose detailed improvements in this paper. We limit the use of any grid flags compared to the FMM and FIM original algorithms. We manage to make points updating independent in every partitioned narrow band. We propose the following fine-grained parallel FIM.

Algorithm 1: Fine-grained parallel fast iterative method algorithm

```
Function Initialization()
    $\forall x \in \Omega, u(x) = +\infty$
    $\forall x \in InitialFront, u(x) \leftarrow 0$ and add neighbor vertices of $x$ in $NB$
Function Main loop()
    Clear $NewNB$
    while $NB \neq \emptyset$ do
        $NewU = u$
        foreach $x \in NB$ in parallel do
            $p_{outer} \leftarrow u(x)$
            $q_{outer} \leftarrow$solution of eq. 2 at $x$
            if $|p_{outer} - q_{outer}| < \epsilon$ then
                foreach neighbor $x_{neighbor}$ of $x$ do
                    if $x_{neighbor} \notin NB$ then
                        $p \leftarrow u(x_{neighbor})$
                        $q \leftarrow$ solution of eq. 2 at $x_{neighbor}$
                        if $q < p$ then
                            $u(x_{neighbor}) \leftarrow q$
                            add $x_{neighbor}$ to $NewNB$ /* critical section  */
            else
                if $q_{outer} < NewU(x)$ then
                    add $x_{neighbor}$ to $NewNB$ /* critical section        */
                $NewU(x) = q_{outer}$
        $NB = NewNB$
        $u = NewU$
```

Note that the method has to use temporary arrays for the grid point values and the narrow band values in order to update points indepedently. This does not increase much cost, since we can swap the original and temporary array at the end of an iteration (since the temporary array will be erased). In our model, unlike in [1], we do not have to be concerned about having to look for efficient way to split the grid at the initialization. The parallelization is straightforward to implement. The partitioning and load-balancing on the narrow band can be managed by a shared memory multiprocessing programming API such as OpenMP. We do not have to be concerned about rollback operations. Critical sections are necessary and placed at the lowest level possible. They do not hinder performance as we can see in section 3.



eventual ghost points to exchange between neighbor processes
G1
G2
G3
...
...
G25
Coarse-grained parallelism on p<13 processus
thread #1
thread #2
thread #3
thread #4
thread #5
G15
Fine-grained parallelism on 5 threads

Fig. 1. Multi-level parallelism

2.2 Master-worker model

Classic or adaptive domain decompositions [1, 8] offer a limited parallel scalability because of rollbacks operations. We propose in algorithm 2 a coarse-grained parallelism which distribute the work among processes based on a master-worker model.

Algorithm 2: Distributed fast iterative method Main loop

```
Process $P_0$ master()
    $P_0$ sends subgrid $G_{k\in[1;p-1]}$ to compute to worker process $P_{i\in[1;p-1]}$
    while $k < nblocks$ i.e. there are still subdomains not computed yet do
        $P_0$ sends subgrid $G_k$ to compute to available $P_i$
        Increment $k$
Processes $P_{i\in[1;p-1]}$ worker()
    $P_i$ computes work $G_k$
    Main loop() of algorithm 1 where $NB_i$ and $u_i$ are local to the
    process $P_i$.
    if vertices in $NewNB_i$ are in ghost zones then
        $P_i$ sends $NewNB_i$ ghost vertices to corresponding neighbor
        process $P_{i-1} or P_{i+1}$
```

Let p be the number of processes, P_0 the master process and $P_{i\in[1;p-1]}$ a worker process. The grid G is divided into $nblocks$ $G_{k\in[0;nblocks]}$ where $nblocks >$

$2(p-1)$ for load-balancing. Indeed, a grid can have few points to update compared to another one. As soon as a job finishes one worker process, the master process immediatly gives back another job. Advantage of this strategy is that we can minimize blocking and global communications, permits to overlap communications with computations. Drawbacks are that hiding communications especially when exchanging ghost points is hard to manage and can become cumbersome. We illustrate the previous parallel levels on figure 1 where we can see how hybrid parallel computations can occur.

3 Experiments

We propose several case tests running in different dimensions for the fine-grained parallel approach. We run the tests using OpenMP at the HPC@LR Montpellier Resource Center which is a shared-memory system composed of 2 processors Intel Xeon X5650 at 2.66 GHz with 6 physical cores each. Hyper-threading at HPC@LR is deactivated hence 12 logical cores in total are available. The results obtained in parallel are the same as the sequential simulations with the same number of iterations achieved.

3.1 Center, wall and random test

We show parallel scalability on three different cases which can be compared with results in [1], including a center test, a wall test with two fronts and a random seed test composed of 32 initial fronts. The domain is $\Omega = [-5.0, 5.0]^2$ and the grid size is composed $N = 4$ million points. We consider monotone propagation (slowness field is constant with $F = 1$). The first test "center test" is composed of one single circle initial front at the center. The second test "wall test" illustrates the algorithm behaviour with an obstacle. The last test "random test" is more complex, simulates a more realistic environment composed of 32 random seed initial fronts (fig 2).

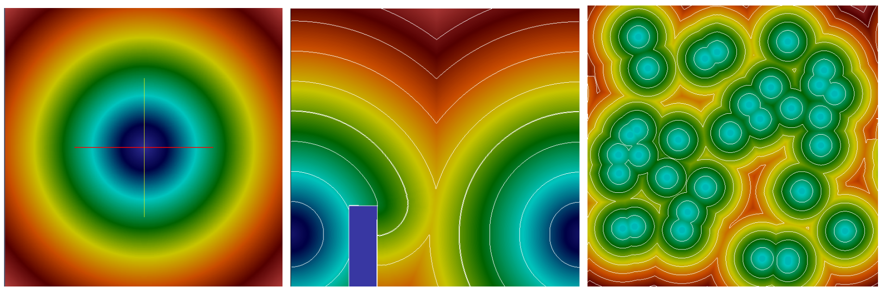


Fig. 2. Three different tests : center, wall and random

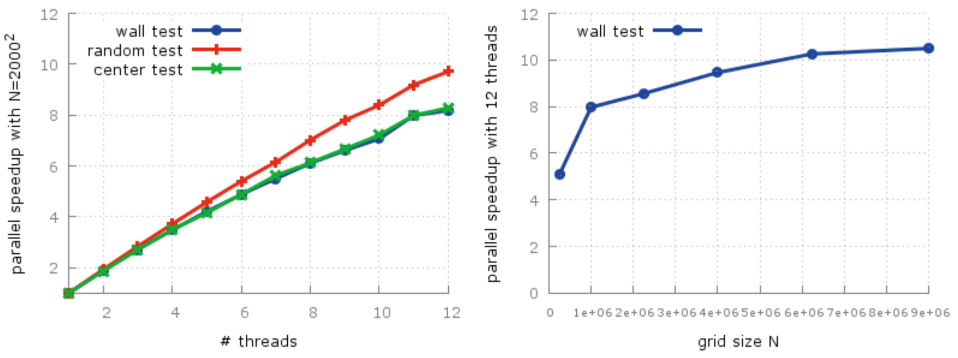


Fig. 3. Parallel speedup and parallel datasize scalability for 2D tests

Figures 3 show that the parallel model is scalable with a smooth speedup even for the center and wall tests where the narrow band is small. When the narrow band is wider, we obtain better results as shown with the random test with an efficiency above 0.8. Therefore real applications should also scale well. We also observe that the parallel speedup on 12 threads increase with the size grid. Therefore, large scale applications should benefit from this.

3.2 Three dimensional case

In three dimensions the proposed algorithm stays the same, only the upwind scheme has to be modified in order to manage neighbors for another dimension. The simulation (fig. 4) reproduces the center test in the domain $\Omega = [-5.0, 5.0]^3$.

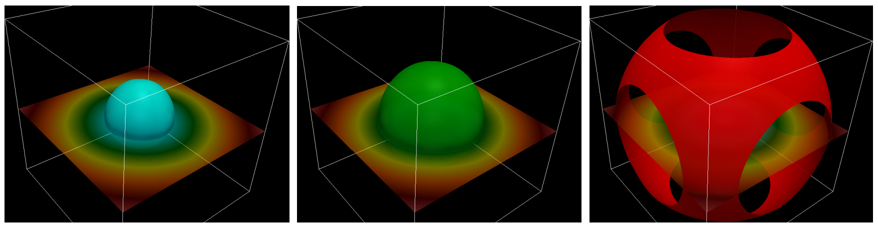


Fig. 4. 3D center test taken at geodesic distance 2.0, 3.0 and 6.0

The results show that the parallel model scale very well in 3D. Efficiency is above 0.8 and can even reach 0.9 for large data size. Increasing the size of the problem tends to improve the parallel scalability as in 2D. We can remark a little scalability drop for $N = 100^3$ which can can be neglected since the efficiency loss is minor (still above 0.8). This behaviour may be explained with partionned narrow band size which do not fit caches in the best conditions.

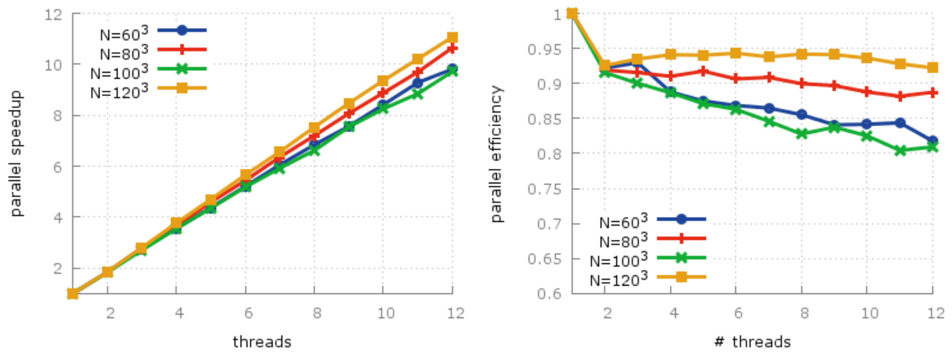


Fig. 5. Parallel speedup and efficiency for the 3D center test on different data size

4 Conclusion

We have proposed multi-level parallel strategies when updating upwind scheme using the FIM. Our parallel scalability behaves like the parallel FSM proposed in [3] with an increasing speedup when the context is close to a realistic environment. Our results show a better speedup compared to recent works in [1, 3]. We prove that the FIM is indeed fitted for parallel computing. Our fine-grained parallel model targets shared-memory architectures and our coarse-grained parallel model targets distributed systems. The fine-grained parallel strategy is straightforward to implement with the proposed algorithm without performance loss when running in sequential. We are currently working on combining both models for hybrid computations.

Acknowledgment The authors would like to deeply thank the Montpellier HPC@LR Center for giving us the opportunity to run our simulations.

References

1. Michael Breuss, Emiliano Cristiani, Pascal Gwosdek, and Oliver Vogel. An adaptive domain-decomposition technique for parallelization of the fast marching method. *Appl. Math. Comput*, 118(1):1–206, 2011.
2. Florian Dang, Nahid Emad, and Alexandre Fender. A fine-grained parallel model for the fast iterative method in solving eikonal equations. In *Proceedings of the 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 3PGCIC '13, pages 152–157. IEEE Computer Society, 2013.
3. Miles Detrixhe, Frédéric Gibou, and Chohong Min. A parallel fast sweeping method for the eikonal equation. *Journal of Computational Physics*, 237(0):46–55, March 2013.
4. Bijoy K. Dey and Paul W. Ayers. Computing the chemical reaction path with a ray-based fast marching technique for solving the Hamilton-Jacobi equation in a general coordinate system. *J Math Chem*, 45(4):981–1003, April 2009.

5. Nicolas Forcadel, Carole Guyader, and Christian Gout. Generalized fast marching method: applications to image segmentation. *Numer Algor*, 48(1-3):189–211, July 2008.
6. Santiago Garrido, Luis Moreno, and Pedro U. Lima. Robot formation motion planning using fast marching. *Robotics and Autonomous Systems*, 59(9):675–683, September 2011.
7. Javier V. Gmez, Alejandro Lumbier, Santiago Garrido, and Luis Moreno. Planning robot formations with fast marching square including uncertainty conditions. *Robotics and Autonomous Systems*, 61(2):137–152, February 2013.
8. M. Herrman. A domain decomposition parallelization of the fast marching method and applications to image segmentation. Annual research briefs, Center for Turbulence Research, 2003.
9. Won-Ki Jeong and Ross T. Whitaker. A fast iterative method for a class of Hamilton-Jacobi equations on parallel systems. Technical Report UUCS-07-010, University of Utah, April 2007.
10. Won-Ki Jeong and Ross T. Whitaker. A fast iterative method for eikonal equations. *SIAM J. Sci. Comput. Vol.30 No.5*, pages 2512–2534, 2008.
11. Elisabeth Rouy and Agnès Tourin. A viscosity solutions approach to shape-from-shading. *SIAM J. Numer. Anal.*, 29(3):867–884, June 1992.
12. James A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.
13. James A. Sethian and Alexander Vladimirsky. Ordered upwind methods for static Hamilton-Jacobi equations : Theory and algorithms. *SIAM J. Numer. Anal.*, 41(1):325–363, 2003.
14. Yen-Hsi Tsai, Li-Tien Cheng, Stanley Osher, and Hong-Kai Zhao. Fast sweeping methods for a class of Hamilton-Jacobi-Bellman equations. *SIAM Vol. 41 No.2*, 41(2), 2003.
15. Hongkai Zhao. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics*, 25(4):421–429, 2007.