# An OpenMP implementation of the TVD–Hopmoc method based on a synchronization mechanism using locks between adjacent threads on Xeon Phi™ accelerators

Frederico L. Cabral[1], Carla Osthoff[1], Gabriel P. Costa[1], Sanderson L. Gonzaga de Oliveira[2], Diego Brandão[3], and Mauricio Kischinhevsky[4]

[1] Laboratório Nacional de Computação Científica - LNCC
{fcabral,osthoff,gcosta}@lncc.br
[2] Universidade Federal de Lavras - UFLA
sanderson@edcc.ufla.br
[3] Centro Federal de Educação Tecnológica Celso Suckow da Fonseca - CEFET-RJ
diego.brandao@eic.cefet-rj.br
[4] Universidade Federal Fluminense - UFF
kisch@ic.uff.br

**Abstract.** This work focuses on the study of the 1–D TVD–Hopmoc method executed in shared memory manycore environments. In particular, this paper studies barrier costs on Intel® Xeon Phi™ (KNC and KNL) accelerators when using the OpenMP standard. This paper employs an explicit synchronization mechanism to reduce spin and thread scheduling times in an OpenMP implementation of the 1–D TVD–Hopmoc method. Basically, we define an array that represents threads and the new scheme consists of synchronizing only adjacent threads. Moreover, the new approach reduces the OpenMP scheduling time by employing an explicit work-sharing strategy. In the beginning of the process, the array that represents the computational mesh of the numerical method is partitioned among threads, instead of permitting the OpenMP API to perform this task. Thereby, the new scheme diminishes the OpenMP spin time by avoiding OpenMP barriers using an explicit synchronization mechanism where a thread only waits for its two adjacent threads. The results of the new approach is compared with a basic parallel implementation of the 1–D TVD–Hopmoc method. Specifically, numerical simulations shows that the new approach achieves promising performance gains in shared memory manycore environments for an OpenMP implementation of the 1–D TVD–Hopmoc method.

## 1 Introduction

Over the last decades, since both the demand for faster computation and shared memory multicore and manycore architectures became available in large scale, an important issue has emerged: how can one obtain a speedup proportional to the number of physical cores available in a parallel architecture? Specific

ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_67

programming languages, libraries, and programming models has been proposed to assist programmers and researches to surmount this challenge. Among them, the OpenMP library [2] is one of the best-known standards nowadays.

Since new machines with faster clock speed are facing a certain limit because overheat and electromagnetic interference, technologies have been proposed with the objective of improving the processing capacity in computations. Multithreading, distributed processing, manycore architectures (e.g. General Purpose Graphical Processing Units (GPGPUs)), and Many Integrated Core (MIC) accelerators (such as Intel® Xeon Phi™) are examples of technologies employed to boost a computer performance. Even with these recent technologies, it is still a challenge to obtain a speedup proportional to the number of available cores in a parallel implementation due to the hardware complexity in such systems. To overcome this particular issue, the OpenMP standard offers a simple manner to convert a serial code to a parallel implementation for shared memory multicore and manycore architectures. Although a parallel implementation with the support of this API is easily reached, a basic (or naive) OpenMP implementation in general does not attain straightforwardly the expected speedup in an application. Furthermore, despite being easily implemented, the most common resources available in the OpenMP standard may generate high scheduling and synchronization running times.

High Performance Computing (HPC) is a practice widely used in computational simulations of several real-world phenomena. Numerical methods have been designed to maximize the computational capacity provided by a HPC environment. In particular, the Hopmoc method (see [3] and references therein) is a spatially decoupled alternating direction procedure for solving convection–diffusion equations. It was designed to be executed in parallel architectures (see [1] and references therein). To provide more specific detail, the unknowns are spatially decoupled, permitting message-passing minimization among threads. Specifically, the set of unknowns is decoupled into two subsets. These two subsets are calculated alternately by explicit and implicit approaches. In particular, the use of two explicit and implicit semi-steps avoids the use of a linear system solver. Moreover, this method employs a strategy based on tracking values along characteristic lines during time stepping. The two semi-steps are performed along characteristic lines by a Semi-Lagrangian scheme following concepts of the Modified Method of Characteristics. The time derivative and the advection term are combined as a direction derivative. Thus, time steps are performed in the flow direction along characteristics of the velocity field of the fluid. The Hopmoc method is a direct method in the sense that the cost per time step is previously known. To determine the value in the foot of the characteristic line, the original method uses an interpolation technique, that introduces inherent numerical errors. To overcome this limitation, the Hopmoc method was combined with a Total Variation Diminishing (TVD) scheme. This new approach, called TVD–Hopmoc, employs a flux limiter to determine the value in the foot of the characteristic line based on the Lax-Wendroff scheme [1].



ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_67

We studied a basic OpenMP implementation of the TVD–Hopmoc method under the Intel® Parallel Studio XE software for Intel's Haswell/Broadwell architectures. This product showed us that the main problem in the performance of a simple OpenMP–based TVD–Hopmoc method was the use of the basic OpenMP scheduling and synchronization mechanisms. Thus, this paper employs alternative strategies to these basic OpenMP strategies. Specifically, this work uses an explicit work-sharing (EWS) strategy. The array that denotes the computational mesh of the 1–D TVD–Hopmoc method is explicitly partitioned among threads. In addition, to avoid implicit barrier costs imposed by the OpenMP standard, this paper employs an explicit synchronization mechanism to guarantee that only threads with real data dependencies participate in the synchronization. Additionally, we compare our approach along with three thread binding policies (balanced, compact, and scatter).

This paper is organized as follows. Section 2 outlines the TVD–Hopmoc method and shows how the experiments were conducted in this work. Section 3 presents the EWS strategy employed here along with our strategy to synchronize adjacent threads. Section 4 shows the experimental results that compares the new approach with a naive OpenMP–based TVD–Hopmoc method. Finally, section 5 addresses the conclusions and discusses the next steps in this investigation.

## 2 The TVD–Hopmoc method and description of the tests

Consider the advection–diffusion equation in the form

$$u_t + vu_x = du_{xx}, \tag{1}$$

with adequate initial and boundary conditions, where $v$ is a constant positive velocity, $d$ is the constant positive diffusivity, $0 \leq x \leq 1$ and $0 \leq t \leq T$, for $T$ time steps. Applying the Hopmoc method to equation (1) yields $\overline{u}_i^{t+\frac{1}{2}} = \overline{\overline{u}}_i^t + \delta t \left[\theta_i^t L_h\left(\overline{\overline{u}}_i^t\right) + \theta_i^{t+1} L_h\left(\overline{u}_i^{t+\frac{1}{2}}\right)\right]$ and $u_i^{t+1} = \overline{u}_i^{t+\frac{1}{2}} + \delta t \left[\theta_i^t L_h\left(\overline{u}_i^{t+\frac{1}{2}}\right) + \theta_i^{t+1} L_h\left(u_i^{t+1}\right)\right]$, where $\theta_i^t$ is 1 (0) if $t+i$ is even (odd), $L_h\left(u_i^t\right) = d\frac{u_{i-1}^t - 2u_i^t + u_{i+1}^t}{\Delta x^2}$ is a finite-difference operator, $\overline{u}_i^{t+\frac{1}{2}}$ and $u_i^{t+1}$ are consecutive time semi-steps, and the value of the concentration in $\overline{\overline{u}}_i^t$ is obtained by a TVD scheme to determine $\overline{\overline{u}}_{i+1}^t = u_i^t - c\left(u_i^t - u_{i-1}^t\right)\left[1 - \frac{(1-c)\phi_{i-1}}{2} + \frac{(1-c)\phi_i}{2r}\right]$, where $r = \frac{u_i^t - u_{i-1}^t}{u_{i+1}^t - u_i^t}$ [1]. The Van Leer flux limiter [4] was employed in the numerical simulations performed in this present work. This scheme delivered better results when compared with other techniques [1]. Our numerical simulations were carried out for a Gaussian pulse with amplitude 1.0, whose initial center location is 0.2, with velocity $v = 1$ and diffusion coefficient $d = \frac{2}{Re} = 10^{-3}$ (where $Re$ stands for Reynolds number), $\Delta x = 10^{-5}$ and $\Delta x = 10^{-6}$ (i.e. $10^5$ and $10^6$ stencil points, respectively), and $T$ is established as $10^4$, $10^5$, and $10^6$.



ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_67

## 3 EWS and synchronization strategies combined with an explicit lock mechanism

The main idea of our OpenMP–based TVD–Hopmoc method is to avoid the extra time caused by scheduling threads and the implicit barriers after a load sharing construct employed in the OpenMP standard. We deal with thread imbalance by partitioning explicitly the array into the team of threads. Our implementation of the TVD–Hopmoc method defines a static array that represents the unknowns. Thus, a permanent partition of this array is established for each thread in the beginning of the code, i.e. no changes are made to this partition during the execution of the process. Since the mesh is static, thread scheduling is performed only at the beginning of the execution. In addition, since each thread in the team has its required data, we do not need to use the OpenMP *parallel for* directive in our code. Thus, our OpenMP implementation of the TVD–Hopmoc method was designed to minimize synchronization in a way that a particular thread needs information only from its adjacent threads so that an implicit barrier is unnecessary. Thereby, the mechanism applied here involves a synchronization of adjacent threads, i.e. each thread waits only for two adjacent threads to reach the same synchronization point.

The strategy employed here is based on a simple lock mechanism. Using an array of booleans, a thread sets (releases) an entry in this array and, hence, informs its adjacent threads that the data cannot (can) be used by them. We will refer this approach as EWS-adSync implementation.

## 4 Experimental results

This section shows the results of the new and basic approaches aforementioned in executions performed on a machine containing an Intel® Xeon Phi™ Knights-Corner (KNC) accelerator 5110P, 8GB DDR5, 1.053 GHz, 60 cores, 4 threads per core (in Section 4.1), and on a machine containing an Intel® Xeon Phi™ CPU 7250 @ 1.40GHz, 68 cores, 4 threads per core (in Section 4.2). We evaluate the EWS-adSync implementation of the TVD–Hopmoc method along with three thread binding policies: balanced, compact, and scatter policies.

### 4.1 Experiments performed on a Xeon Phi™ (Knight's Corner) accelerator

This section shows experiments performed on an Intel® Xeon Phi™ (KNC) accelerator composed of 240 threads. Figure 1 shows that our OpenMP–based TVD–Hopmoc method using the EWS strategy in conjunction with synchronizing adjacent threads along with compact (comp.) thread binding policy (pol.) yielded a speedup of approximately 140x (using 240 threads) in a simulation with $\Delta x = 10^{-6}$ (i.e. a mesh composed of $10^6$ stencil points) and $T = 10^5$. This implementation used in conjunction with the scatter (sct.) policy delivered a speedup of 136x (using 238 threads) in a simulation with the same settings.



ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_67

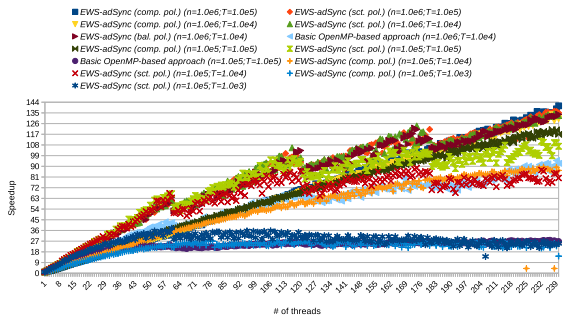


**Fig. 1.** Speedups obtained by two OpenMP implementations of the TVD–Hopmoc method applied to the advection–diffusion equation (1) for a Gaussian pulse with amplitude 1.0 in executions performed on a Xeon Phi$^{TM}$ (KNC) accelerator. A basic (ou naive) implementation was employed in a simulation with $T = 10^5$ and $\Delta x = 10^{-5}$ (i.e. the mesh is composed of $10^5$ stencil points). Our OpenMP–based TVD–Hopmoc method (EWS-adSync implementation) was employed with three different thread binding policies (balanced, compact, and scatter policies) in simulations with $\Delta x$ set as $10^{-5}$ and $10^6$ (i.e. meshes composed of $10^5$ and $10^6$ stencil points, resp.) and $T$ specified as $10^5$, $10^4$, and $10^3$.

The EWS-adSync implementation alongside scatter, balanced (bal.), and compact scatter policies respectively achieved speedups of 134.2x, 133.8x, and 133.5x in simulations with $\Delta x = 10^{-6}$ and $T = 10^4$. These implementations dominated the basic OpenMP–based TVD-Hopmoc method, which obtained a speedup of 28x.

The EWS-adSync implementation alongside the compact policy (with speedup of 121x) achieved better results than the scatter policy (with speedup of 111x) in a simulation with $\Delta x = 10^{-5}$ and $T = 10^5$. Both implementations dominated the basic OpenMP–based TVD-Hopmoc method, which obtained a speedup of 95x. The EWS-adSync implementation alongside the compact policy (with speedup of 90x) reached slightly better results than the scatter policy (with speedup of 88x) in a simulation with $\Delta x = 10^{-5}$ and $T = 10^4$. On the other hand, the EWS-adSync implementation in conjunction with the scatter policy (with speedup of 38x) yielded better results than the compact policy (with speedup of 27x) in a simulation with $\Delta x = 10^{-5}$ and $T = 10^3$. Figure 1 shows that in general the OpenMP–based TVD–Hopmoc method obtains better speedups when setting a larger number of iterations $T$ since it takes advantage of a higher cache hit rate.

### 4.2 Experiments performed on a Xeon Phi$^{TM}$ (Knights Landing) accelerator

This section shows experiments performed on an Intel® Xeon Phi$^{TM}$ (KNL) accelerator composed of 272 threads. Figure 2 shows that our OpenMP–based



ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_67

TVD–Hopmoc method using the EWS strategy in conjunction with synchronizing adjacent threads along with scatter thread binding policy yielded a speedup of approximately 91x (using 135 threads) in a simulation with $\Delta x = 10^{-5}$ (i.e. a mesh composed of $10^5$ stencil points) and $T = 10^6$. This implementation used in conjunction with scatter and compact policies delivered respectively speedups of 85x and 77x (using 135 and 271 threads, respectively) with the same settings. Similarly, the EWS-adSync implementation alongside balanced policy achieved better results (with speedup of 84x) than in conjunction with both scatter (with speedup of 80x) and compact (with speedup of 69x) policies in simulations with $\Delta x = 10^{-5}$ and $T = 10^5$.

The EWS-adSync implementation alongside the scatter policy achieved similar results to the compact policy (both with speedup of 17x) in a simulation with $\Delta x = 10^{-6}$ and $T = 10^5$. It seems that the simulation with $\Delta x = 10^{-6}$ obtained a higher cache miss rate than the other experiments.

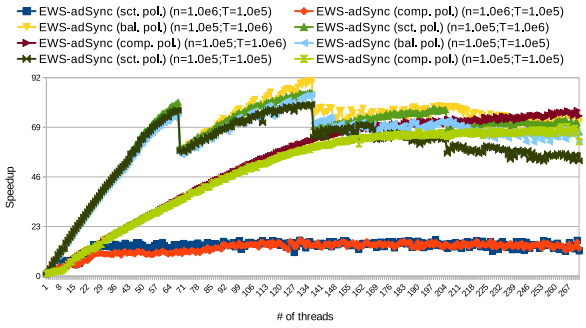


**Fig. 2.** Speedups of two OpenMP implementations of the TVD–Hopmoc method applied to the advection–diffusion equation (1) for a Gaussian pulse with amplitude 1.0 in runs performed on a Xeon Phi$^{TM}$ (KNL) accelerator. Our OpenMP–based TVD–Hopmoc method (EWS-adSync implementation) was employed with three different thread binding policies (balanced, compact, and scatter policies) in simulations with $\Delta x$ established as $10^{-5}$ and $10^6$ (i.e. meshes composed of $10^5$ and $10^5$ stencil points) and $T$ was defined as $10^6$ and $10^5$.

## 5 Conclusions

Based on an explicit work-sharing strategy along with an explicit synchronization mechanism, the approach employed here to implement an OpenMP–based TVD–Hopmoc method attained reasonable speedups in manycore (Xeon Phi$^{TM}$ KNC and KNL accelerators) architectures. In particular, the scheduling time was profoundly reduced by replacing the effort of assigning to threads tasks at runtime by an explicit work-sharing strategy that determines *a priori* the range



ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_67

of the array that represents stencil points where each thread will perform its task. Moreover, using a lock array where an entry denotes a thread, the synchronization time in barriers was substituted by a strategy that only requires that each thread be synchronized with its two adjacent threads. Employing these two strategies, the team of threads presented a reasonable load balancing, where almost the total number of threads available are used simultaneously.

Our OpenMP–based TVD–Hopmoc method along with a compact thread binding policy yielded a speedup of approximately 140x when applied to a mesh composed of $10^6$ stencil points in a simulation performed on an Intel® Xeon Phi™ (Knight's Corner) accelerator composed of 240 threads. Moreover, this parallel TVD–Hopmoc method alongside a balanced thread binding policy reached a speedup of approximately 91x when applied to a mesh composed of $10^5$ stencil points in a simulation performed on an Intel® Xeon Phi™ (Knights Landing) accelerator composed of 135 threads. Furthermore, our OpenMP–based TVD–Hopmoc method in conjunction both with scatter and compact policies achieved a speedup of approximately 17x when applied to a mesh composed of $10^6$ stencil points in a simulation performed on the same machine.

It is observed in Figures 1 and 2 that the speedup of our OpenMP–based TVD–Hopmoc method along with scatter and balanced policies shows four trends. The speedup increases with the use of up to a number of threads that is multiple of the number of cores in the machine. We intend to provide further investigations about this characteristic in future studies.

A next step in this investigation is to implement an OpenMP–based 2–D TVD–Hopmoc method. Even in the 2–D case, we plan to use an array to represent the stencil points so that the approach employed in the 1–D case of the TVD–Hopmoc method is still valid.

## Acknowledgments

This work was developed with the support of CNPq, CAPES, and FAPERJ - Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro. The authors thank the Núcleo de Computação Científica at the UNESP for the use of its computational resources. These resources were partially funded by Intel®.

## References

1. F. Cabral, C. Osthoff, G. Costa, D. N. Brandão, M. Kischinhevsky, and S. L. Gonzaga de Oliveira. Tuning up TVD HOPMOC method on Intel MIC Xeon Phi architectures with Intel Parallel Studio tools. In *Proceedings of the 8th Workshop on Applications for Multi-Core Architectures*, Campinas, SP, Brazil, 2017.
2. L. Dagum and R. Menon. Openmp: An industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.
3. S. R. F. Oliveira, S. L. Gonzaga de Oliveira, and M. Kischinhevsky. Convergence analysis of the Hopmoc method. *Int. J. Comput. Math.*, 86:1375–1393, 2009.
4. B. van Leer. Towards the ultimate conservative difference schemes. *Journal of Computational Physics*, 14:361–370, 1974.



ICCS Camera Ready Version 2018
To cite this paper please use the final published version:
DOI: 10.1007/978-3-319-93713-7_67