An Improved OpenMP Implementation of the TVD–Hopmoc Method Based on a Cluster of Points

Frederico Cabral[1], Carla Osthoff[1], Roberto Pinto Souto[1], Gabriel P. Costa[1], Sanderson L. Gonzaga de Oliveira[2], Diego N. Brandão[3(✉)], and Mauricio Kischinhevsky[4]

[1] LNCC, Petrópolis, RJ, Brazil
{fcabral,osthoff,rpsouto,gcosta}@lncc.br
[2] Universidade Federal de Lavras, Lavras, MG, Brazil
sanderson@dcc.ufla.br
[3] CEFET/RJ, Rio de Janeiro, RJ, Brazil
diego.brandao@eic.cefet-rj.br
[4] Universidade Federal Fluminense, Niterói, RJ, Brazil
kisch@ic.uff.br

Abstract. This paper concentrates on an OpenMP implementation of the TVD–Hopmoc method with executions performed on Intel® Many Integrated Core and Xeon® Scalable Processor architectures. Specifically, this paper evaluates an improved OpenMP implementation of the TVD–Hopmoc method based on a cluster of points when applied to the convection–diffusion equation in 1–D. Aiming at avoiding fine-grained parallelism employed in a basic OpenMP implementation of the TVD–Hopmoc method, this approach groups variables (located at stencil points) to be calculated simultaneously in parallel instead of calculating them individually. Numerical experiments performed on Intel® Many Integrated Core and Scalable Processor architectures show that the improved OpenMP implementation of the TVD–Hopmoc method based on a cluster of points provides further worthwhile gains when compared both with our previous implementation based only on parallel chunk loops and a basic OpenMP implementation of this method.

Keywords: High perfomance computing · Parallel processing · Convection–diffusion equation

1 Introduction

The Hopmoc method was designed to solve parabolic convective–dominated problems in parallel architectures (see [2] and references therein). It devises a spatial partition of stencil points that allows how to minimize communication among threads. The Hopmoc method subdivides the set of unknowns into two subsets. Thus, the Hopmoc method divides each time step into two time semi–steps. The unknowns containing in these subsets are calculated alternately by

H. Senger et al. (Eds.): VECPAR 2018, LNCS 11333, pp. 132–145, 2019.
https://doi.org/10.1007/978-3-030-15996-2_10

explicit and implicit approaches during the first and second time semi–steps, respectively. The Hopmoc method evaluates time semi–steps along characteristic lines using a Semi-Lagrangian approach taking into account concepts of the Modified Method of Characteristics [11].

Discretization of the convective term in transport equations is frequently afflicted with severe complications. To avoid spurious numerical oscillations, Harten [3] introduced the concepts of Total Variation Diminishing (TVD) techniques and flux–limiter formulations, which are stable higher–order accurate solutions of convection–diffusion equations. The TVD–Hopmoc [4] uses a flux-limiter formulation to calculate the value at the foot of the characteristic line based on the Lax–Wendroff scheme and increase the accuracy of the original Hopmoc method [4,8].

We evaluated a simple (or naive) OpenMP-based TVD–Hopmoc method. This implementation was analyzed using the Intel® Parallel Studio XE software development product so that we followed its recommendations and proposed our previous OpenMP-based TVD–Hopmoc method [8]. A further analysis using this software led us to improve our earlier implementation. This paper presents this enhanced OpenMP implementation and compares its results both with the naive and our initial version of the TVD–Hopmoc method [8]. We investigate here an approach that groups variables (located at stencil points) to be calculated concomitantly in parallel instead of calculating them separately.

Various publications have been proposing to improve performance on Intel Xeon Phi accelerators. These problem-solving techniques have been trying to handle the challenge presented in this architecture to achieve linear speedups, principally in OpenMP implementations. For example, Ma et al. [5] proposed strategies to optimize the OpenMP implicit barrier constructs. These authors revealed how to remove the OpenMP inherent barrier constructs when there is no data dependence. Their second strategy uses a busy-waiting synchronization. Their optimized OpenMP implementation obtained better results than the basic OpenMP strategies. Caballero et al. [6] introduced a tree-based barrier that uses cache locality along with SIMD instructions. Their approach achieved a speedup of up to 2.84x over the basic OpenMP barrier in the EPCC barrier micro-benchmark. Cabral et al. [7] evaluated the original Hopmoc method in different parallel programming paradigms. The authors, however, did not perform the implementations on Intel® Xeon® Phi accelerators. A previous publication [8] showed that a simple OpenMP implementation of the TVD-Hopmoc method suffers from high load imbalance caused by the fine-grained parallelism used inherently by the OpenMP standard. This implementation employed a parallel chunk loop strategy to avoid the fine-grained parallelism, which improved its performance in approximately 50%.

The remainder of this paper is structured as follows. Section 2 presents the TVD–Hopmoc method in details. Section 3 explains a simple OpenMP implementation of the TVD–Hopmoc method. Section 4 describes an improved OpenMP-based TVD–Hopmoc method. Finally, Sect. 6 addresses the conclusion and discusses future directions in this work.

2 The TVD–Hopmoc Method

We describe below the Hopmoc method in details (see [2] and references therein). Consider the one–dimensional convection–diffusion equation in the form

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

with adequate initial and boundary conditions, where v represents a constant positive velocity, d is a positive constant of diffusivity, and $0 \leq x \leq 1$. In Eq. (1), u_t refers to the time derivative and not u evaluated at the discrete time step t. Nevertheless, we abuse the notation and now use t to denote a discrete time step so that $0 \leq t \leq T$, for T time steps.

Consider also a conventional finite–difference discretization for this equation, with $\Delta t = u^{t+1} - u^t$, and $\delta t = \frac{\Delta t}{2} = u^{t+\frac{1}{2}} - u^t$ represents a time semi–step of the method. A characteristic line permits to obtain $\overline{u}\left(\overline{x}_i^{t+\frac{1}{2}}\right)$ and $\overline{\overline{u}}\left(\overline{\overline{x}}_i^{t}\right)$ in the previous two time semi–steps, for $\overline{x}_i^{t+\frac{1}{2}} = x_i - v \cdot \delta t$ and $\overline{\overline{x}}_i^{t} = x_i - 2v \cdot \delta t$, respectively, and the variable values $\overline{\overline{u}}\left(\overline{\overline{x}}_i^{t}\right)$ are obtained using an interpolation technique [2], as described below. For clarity, a variable in a previous time semi–step and in a previous time step are written as $\overline{u}_i^{t+\frac{1}{2}} = u\left(\overline{x}_i^{t+\frac{1}{2}}\right)$ and $\overline{\overline{u}}_i^{t} = u\left(\overline{\overline{x}}_i^{t}\right)$, respectively. Additionally, $\overline{u}_i^{t+\frac{1}{2}}$ is the variable in the previous time semi–step at the foot of the characteristic line originated at x_i^{t+1}. Moreover, we use a uniform spatial discretization so that $\Delta x = x_{i+1} - x_i$ [2].

We use a three–point finite–difference scheme in the discretization of diffusive terms and $\overline{\overline{u}}_i^{t+1}$ is a numerical approximation of u in $\left(x_i, u^{t+1}\right)$ when t is even. Using the finite-difference operator $L_h\left(u_i^t\right) = d\frac{u_{i-1}^t - 2u_i^t + u_{i+1}^t}{\Delta x^2}$, the consecutive time semi–steps of the Hopmoc method can be written as $\overline{u}_i^{t+\frac{1}{2}} = \overline{\overline{u}}_i^{t} + \delta t\left(\theta_i^t L_h \overline{\overline{u}}_i^{t} + \theta_i^{t+1} L_h \overline{u}_i^{t+\frac{1}{2}}\right)$ or $u_i^{t+1} = \overline{u}_i^{t+\frac{1}{2}} + \delta t\left(\theta_i^t L_h \overline{u}_i^{t+\frac{1}{2}} + \theta_i^{t+1} L_h u_i^{t+1}\right)$, for $\theta_i^t = 1\ (= 0)$ if $t + i$ is even (odd).

The discretization of the convective term demands to calculate the values of the concentration at midpoints of the sides of each grid interval. The method obtains these values using a TVD scheme [4,8].

Our numerical simulations were performed 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), $\Delta t = 10^{-7}$, Δx is set as 10^{-5}, 10^{-6}, and 10^{-7} (i.e., 10^5, 10^6, and 10^7 stencil points, respectively), and T is established as 10^4, 10^5, and 10^6.

3 A Basic OpenMP Implementation of the TVD–Hopmoc Method

This section outlines a simple (or naive) OpenMP-based TVD–Hopmoc method. We implemented the algorithms using the C++ programming language. Specifically, we used the icpc (Intel C++ compiler) version 2018, with the optimization

flags -xHost and -O3. Additionally, in Xeon Scalable Processors, the -qopt-zmm-usage=high flag was used.

In short, this basic OpenMP implementation (i.e., using the OpenMP *parallel for* directive) consists of the "main" time loop that performs two steps. Firstly, the method computes the MMOC step, where the TVD Van Leer flux–limiter scheme is implemented. Secondly, it executes the first and second (explicit and implicit) semi–steps.

The simplistic approach to parallelize the TVD–Hopmoc method in executions on Intel® Many Integrated Core Architecture is to insert OpenMP directives in specific loops of the code, for example, in loops that solve the explicit and implicit operators. Algorithm 1 shows a fragment of pseudo-code that delineates how this naive implementation performs a time step of the Hopmoc method. This fragment of pseudo-code shows four for loops that calculate the two time semi–steps of the algorithm using alternately explicit and implicit approaches. The first and second (third and fourth) for loops calculate unknowns $\overline{u}_i^{t+\frac{1}{2}}$ $\left(u_i^{t+1}\right)$ using explicit and implicit approaches in the first (second) time semi–step. In this naive implementation, we observed no improvement when using other forms of OpenMP thread schedulings, such as OpenMP dynamic and guided schedulings. The ANNOTATE_ITERATION_TASK macro instructs the Intel® Advisor shared memory threading assistance tool that these loops must be analyzed to generate the performance estimates.

Algorithm 2 shows a fragment of a pseudo-code that is used to obtain the suitability analysis carried out by the Intel® Advisor shared memory threading assistance tool. This fragment of pseudo-code shows an OpenMP parallel region composed of a loop that iterates the time steps of the TVD–Hopmoc method. Thus, this while loop is identified as a parallel region to be analyzed by the Intel® Advisor shared memory threading assistance tool.

We executed experiments with this simple OpenMP-based TVD–Hopmoc method performed on a machine containing an Intel® Xeon™ CPU E5-2698 v3 @ 2.30 GHz composed of 32 physical cores. To evaluate our source code, we analyzed it using the Intel® Advisor shared memory threading assistance tool. This analysis revealed that even with most of the implementation vectorized, the gains using the OpenMP standard is limited. The reason for this is because the calculations in the method are implemented using an approach with very fine granularity to take full advantage of parallelism and HPC capabilities.

Figure 1 exhibits the results of an experiment performed with the support of the Intel® VTune™ Amplifier performance profiler. This figure shows that this simple OpenMP implementation obtains an inefficient performance in a multicore environment in a simulation with $T = 10^6$ and $\Delta x = 10^{-5}$, i.e., in a mesh composed of 10^5 stencil points. Specifically, Fig. 1 reveals that the execution of this implementation obtained high spin (imbalance or serial spinning) time caused by the use of the implicit OpenMP strategies. Additionally, Fig. 1 exhibits a high clock ticks per Instructions Retired (CPI) rate (1.301) achieved by the basic OpenMP-based TVD–Hopmoc method.

Algorithm 1. A time step composed of four for loops that iterate the first and second time semi–steps of a naive OpenMP-based TVD–Hopmoc method.

1: #pragma omp for
{First time semi–step of the Hopmoc method, where $\alpha = d * \frac{\delta t}{(\delta x)^2}$}
2: **for** $i \leftarrow head + 1$; $i \leq n - 2$; $i \leftarrow i + 2$ **do**
3: ANNOTATE_ITERATION_TASK (loop_HOP_EXP_1);
$\overline{u}_i^{t+\frac{1}{2}} \leftarrow \alpha \cdot \left(\overline{\overline{u}}_{i-1}^{t} + \overline{\overline{u}}_{i+1}^{t}\right) + (1 - 2\alpha)\overline{\overline{u}}_i^{t}$;
4: **end for**
{First time semi–step of the Hopmoc method using an implicit approach}
5: #pragma omp single
6: $head \leftarrow (head + 1)\%2$;
7: #pragma omp for
8: **for** $i \leftarrow head + 1$; $i \leq n - 2$; $i \leftarrow i + 2$ **do**
9: ANNOTATE_ITERATION_TASK (loop_HOP_IMP_1);
$\overline{u}_i^{t+\frac{1}{2}} \leftarrow \frac{\overline{\overline{u}}_i^{t} + \alpha \cdot \left(\overline{u}_{i-1}^{t+\frac{1}{2}} + \overline{u}_{i+1}^{t+\frac{1}{2}}\right)}{1+2\alpha}$;
10: **end for**
{Second time semi–step of the Hopmoc method using an explicit approach}
11: #pragma omp single
12: $head \leftarrow (head + 1)\%2$;
13: #pragma omp for
14: **for** $i \leftarrow head + 1$; $i \leq n - 2$; $i \leftarrow i + 2$ **do**
15: ANNOTATE_ITERATION_TASK (loop_HOP_EXP_2);
$u_i^{t+1} \leftarrow \alpha \cdot \left(\overline{u}_{i-1}^{t+\frac{1}{2}} + \overline{u}_{i+1}^{t+\frac{1}{2}}\right) + (1 - 2\alpha) \cdot \overline{u}_i^{t+\frac{1}{2}}$;
16: **end for**
{Second time semi–step of the Hopmoc method using an implicit approach}
17: #pragma omp single
18: $head \leftarrow (head + 1)\%2$;
19: #pragma omp for
20: **for** $i \leftarrow head + 1$; $i \leq n - 2$; $i \leftarrow i + 2$ **do**
21: ANNOTATE_ITERATION_TASK (loop_HOP_IMP_2);
$u_i^{t+1} \leftarrow \frac{\overline{u}_i^{t+\frac{1}{2}} + \alpha \cdot \left(u_{i-1}^{t+1} + u_{i+1}^{t+1}\right)}{1+2\alpha}$;
22: **end for**

Figure 2 displays a CPU usage histogram extracted from Advanced Hotspots Analysis performed by the Intel® VTune™ Amplifier performance profiler. This figure shows that the basic OpenMP-based TVD–Hopmoc method uses a small number of cores concurrently. In particular, this implementation used on average 19 cores at the same time (in a machine composed of 32 cores).

Figure 3a shows a screen captured from the suitability report performed by the Intel® Advisor shared memory threading assistance tool. This figure shows that the naive OpenMP-based TVD–Hopmoc method suffers from high load imbalance and reaches 100% of runtime overhead. To provide specific detail, a suitability analysis performed by the Intel® Advisor shared memory threading assistance tool indicated that the simple OpenMP-based TVD–Hopmoc method

Algorithm 2. Pseudo-code outlining how to obtain the suitability analysis performed by the Intel® Advisor shared memory threading assistance tool.

```
1: t_beg ← omp_get_wtime();
2: #pragma omp parallel
3: {
4: ANNOTATE_SITE_BEGIN(time_loop);
5: while (t < T) do
6:    [...]
7: end while
8: ANNOTATE_SITE_END();
9: }
10: t_end ← omp_get_wtime();
```

Fig. 1. Execution time obtained by a basic OpenMP-based TVD–Hopmoc method when analyzed with the support of the Intel® VTune™ Amplifier performance profiler.

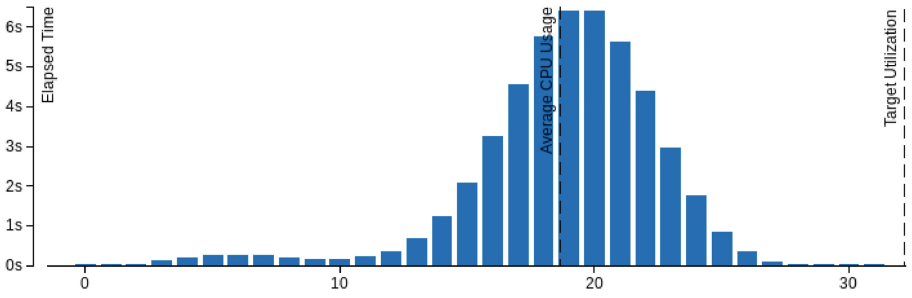


Fig. 2. CPU usage histogram produced with the results of an execution of the basic OpenMP-based TVD–Hopmoc method. This histogram was taken from Advanced Hotspots Analysis performed by the Intel® VTune™ Amplifier performance profiler. It displays a percentage of the wall time.

presents 75% of load imbalance and high runtime overhead, including high thread scheduling time, due to fine-grained parallelism employed in this implementation. Additionally, there is no scalability gain when executing this implementation. Intel® Advisor shared memory threading assistance tool advises that this implementation is too fine-grain, and it is not adequate for multi-threading. This software suggested to increase task granularity, reduce task overhead, or consider vectorization in this implementation.

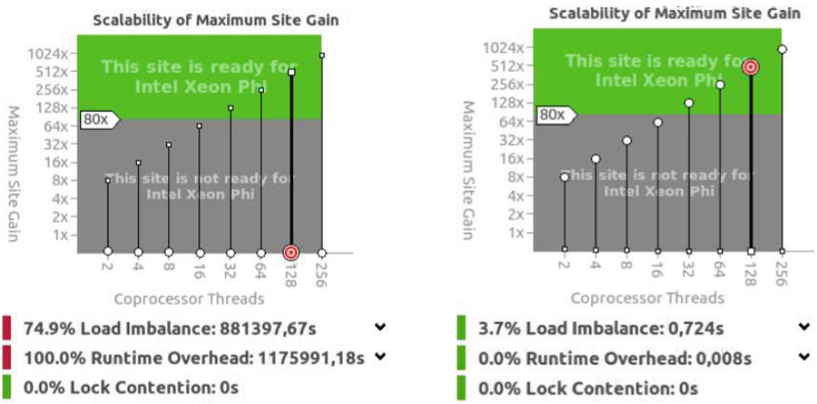


Fig. 3. Intel Advisor's predictions.

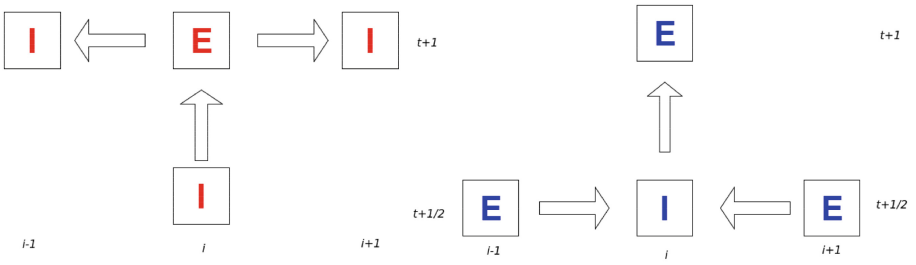
4 An Improved OpenMP Implementation of the TVD–Hopmoc Method Based on a Cluster of Points

This section presents an improved OpenMP implementation of the TVD–Hopmoc method based on a cluster of points. As mentioned, we analyzed a simple OpenMP-based TVD–Hopmoc method with the support of the Intel® Parallel Studio XE software for Intel's Haswell/Broadwell architectures to discover the vulnerabilities of our code and consequently to propose solutions to them.

Figure 3b shows another screen captured from the suitability report performed by the Intel® Advisor shared memory threading assistance tool when enabling task chunking in the naive implementation of the TVD–Hopmoc method (reported in Sect. 3). According to the Intel® Advisor shared memory threading assistance tool, using this technique would reduce load imbalance to 4% with no runtime overhead. Figure 3b also shows that the expected scalability would be high if this approach were employed. Chunking is a strategy that merges several tasks into a single one, i.e., they are executed together as a chunk, with little or no overhead between them. In the case of using parallel chunk loops in the source code, the programmer merges loops inside the analyzed site into

a single one. An implementation based on parallel chunk loops groups some of them to consequently obtain a smaller number of parallel loops. Therefore, the code employs a coarse–grained approach. However, data dependency limits the use of this approach. In the case of the TVD–Hopmoc method, unknowns that are approximated using an implicit approach depend on unknowns that are solved using an explicit approach. Thus, it was not possible to integrally follow this recommendation because of limitations in data dependency to calculate variable values in the TVD–Hopmoc method. Therefore, we adopted an intermediate solution. We referred to it as an implementation based on parallel chunk loops in an earlier version [8].

We conducted then a further data dependency analysis with the objective of building an implementation that groups stencil points to calculate them in parallel. This analysis led us to implement an improved OpenMP-based TVD–Hopmoc method that merges all operators in the Hopmoc method into two loops, instead of using four loops as the original Hopmoc method performs (see Algorithm 1). To provide further details, we analyzed data dependency to guarantee that the loops are suitable for parallelism, i.e., to assure that tasks inside loops are computed concurrently. Each of these two loops executes over a cluster of four grid points. The first cluster of points, called I-cluster, is composed of the two unknowns $\overline{u}_{i-1}^{t+1}$ and $\overline{u}_{i+1}^{t+1}$, which are solved using an implicit approach, and the unknown $\overline{u}_{i}^{t+1}$, which is solved using an explicit approach during the second time semi–step of the TVD–Hopmoc method (see Fig. 4a). Additionally, the unknown $u_i^{t+\frac{1}{2}}$ belongs to this cluster of points. It is solved using an implicit approach in the first time semi–step of the TVD–Hopmoc method.



(a) E-Cluster (b) I-Cluster

Fig. 4. Two strategies for clustering points.

Figure 4b shows the second cluster of points employed in this scheme, called E-cluster. This cluster of points is also composed of four unknowns. The E-cluster is composed of an unknown $\overline{u}_{i}^{t+1}$ that is solved using an explicit approach during the second time semi–step and three unknowns that are solved during the first time semi–step of the TVD–Hopmoc method: $u_i^{t+\frac{1}{2}}$, $u_{i-1}^{t+\frac{1}{2}}$, and $u_{i+1}^{t+\frac{1}{2}}$.

The finite-difference mesh can be composed of these two types of a cluster of points [8]. In particular, this scheme allows that balancing load to be evaluated during runtime so that adjacent clusters of points in a level l can be grouped again to form a larger cluster of points in a level $l+1$.

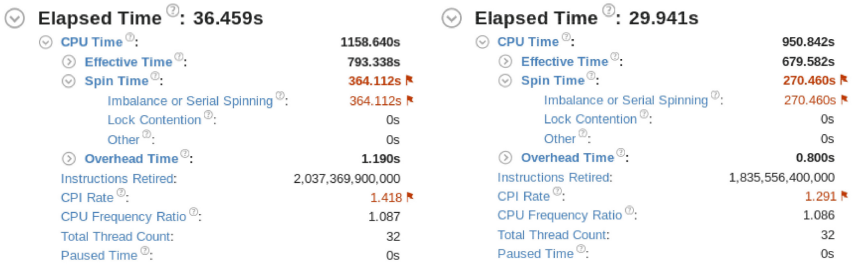
Algorithm 3 shows a fragment of pseudo-code that illustrates the first and second time semi–steps of the Hopmoc method. Specifically, Algorithm 3 displays that the four parallel for loops outlined in Algorithm 1 (from a naive OpenMP implementation) are substituted by two parallel for loops that calculate all E-clusters and I-clusters. Using this scheme to cluster points, our OpenMP-based TVD–Hopmoc method avoids data dependency and increases parallelism.

Algorithm 3. Pseudo-code depicting the first and second time semi–steps of the improved OpenMP-based TVD–Hopmoc method.

1: [...]
{Compute all E-clusters}
2: #pragma omp for
3: **for** $i \leftarrow 2$; $i \leq n-3$; $i \leftarrow i+4$ **do**
4: $\quad \overline{u}_{i-1}^{t+\frac{1}{2}} \leftarrow \alpha \cdot \left(\overline{\overline{u}}_{i-2}^{t} + \overline{\overline{u}}_{i}^{t}\right) + (1-2\alpha) \cdot \overline{\overline{u}}_{i-1}^{t}$;
5: $\quad \overline{u}_{i+1}^{t+\frac{1}{2}} \leftarrow \alpha \cdot \left(\overline{\overline{u}}_{i+2}^{t} + \overline{\overline{u}}_{i}^{t}\right) + (1-2\alpha) \cdot \overline{\overline{u}}_{i+1}^{t}$;
6: $\quad \overline{u}_{i}^{t+\frac{1}{2}} \leftarrow \frac{\overline{\overline{u}}_{i}^{t} + \alpha \cdot \overline{u}_{i-1}^{t+\frac{1}{2}} + \alpha \cdot \overline{u}_{i+1}^{t+\frac{1}{2}}}{1+2\alpha}$;
7: $\quad u_{i}^{t+1} \leftarrow \alpha \cdot \left(\overline{u}_{i-1}^{t+\frac{1}{2}} + \overline{u}_{i+1}^{t+\frac{1}{2}}\right) + (1-2\alpha) \cdot \overline{u}_{i}^{t+\frac{1}{2}}$;
8: **end for**
{Compute all I-clusters}
9: #pragma omp for
10: **for** $i \leftarrow 2$; $i \leq n-5$; $i \leftarrow i+4$ **do**
11: $\quad \overline{u}_{i}^{t+\frac{1}{2}} \leftarrow \frac{\overline{\overline{u}}_{i}^{t} + \alpha u_{i-1}^{t+1}}{1+2\alpha}$;
12: $\quad u_{i}^{t+1} \leftarrow \alpha \cdot \left(\overline{u}_{i-1}^{t+\frac{1}{2}} + \overline{u}_{i+1}^{t+\frac{1}{2}}\right) + (1-2\alpha) \cdot \overline{u}_{i}^{t+\frac{1}{2}}$;
13: $\quad u_{i-1}^{t+1} \leftarrow \frac{\overline{u}_{i-1}^{t+\frac{1}{2}} + \alpha \cdot \left(u_{i}^{t+1} + u_{i-2}^{t+1}\right)}{1+2\alpha}$;
14: $\quad u_{i+1}^{t+1} \leftarrow \frac{\overline{u}_{i+1}^{t+\frac{1}{2}} + \alpha \cdot \left(u_{i}^{t+1} + u_{i+2}^{t+1}\right)}{1+2 \cdot \alpha}$;
15: **end for**

Figure 5a exhibits the results of an experiment performed with our previous implementation [8] and the support of the Intel® VTune™ Amplifier performance profiler. In an experiment with $T = 10^6$ and $\Delta x = 10^{-5}$, i.e., in a mesh composed of 10^5 stencil points, this figure displays that our previous OpenMP-based TVD–Hopmoc method [8] obtained lower execution time (1159 s) than the basic OpenMP implementation of this method (1560 s; see Fig. 1). Furthermore, Fig. 5a exhibits that our previous OpenMP-based TVD–Hopmoc method [8] obtained lower wall time (36 s) than the basic OpenMP implementation of this

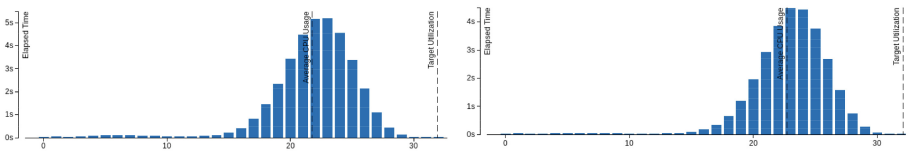
method (49 s). On the other hand, Fig. 5a shows a high Clockticks per Instructions Retired (CPI) rate (1.418) yielded by our previous OpenMP implementation of the TVD–Hopmoc method based uniquely on parallel chunk loops [8].



(a) CoP (b) iCoP

Fig. 5. Execution time obtained by the simple and the improved cluster of points implementations of the TVD–Hopmoc method when analyzed with the support of the Intel® VTune™ Amplifier performance profiler.

Figure 6a displays a CPU usage histogram taken from Advanced Hotspots Analysis performed by the Intel® VTune™ Amplifier performance profiler. This figure displays that our previous method [8] used on average 22 cores concurrently when performed on the machine aforementioned.



(a) CoP (b) iCoP

Fig. 6. CPU usage histogram produced in an execution of simple and the improved OpenMP implementation of the TVD–Hopmoc method based on a cluster of points. This histogram was captured from Advanced Hotspots Analysis produced by the Intel® VTune™ Amplifier performance profiler.

Also with the objective of avoiding fine-grained parallelism, we improved our previous OpenMP-based implementation of the TVD–Hopmoc method [8] by combining two time steps in the same while loop. With this improvement, data are updated straightforwardly to the next iteration instead of performing this explicitly as our previous implementation carried out [8]. Then, to distinguish this implementation from the earlier algorithm [8], we will refer the new version as improved OpenMP implementation of the TVD–Hopmoc based on a cluster of points (iCoP for short). Both codes are thread-safe codes.

According to the Intel® Advisor shared memory threading assistance tool, both implementations present 71% of vectorization efficiency. Our OpenMP-based TVD–Hopmoc method is 50% faster than the naive implementation.

Experiments with the use of our previous implementation [8] and the improved OpenMP implementation of the TVD–Hopmoc method were also performed on a machine containing an Intel® Xeon™ CPU E5−2698 $v3$ @ 2.30 GHz with 32 physical cores. Again, we established $T = 10^6$ and $\Delta x = 10^{-5}$, i.e., the mesh was composed of 10^5 stencil points. Figure 5b shows the results of an experiment performed with the improved OpenMP implementation of the TVD–Hopmoc method based on a cluster of points and the support of the Intel® VTune™ Amplifier performance profiler. This figure shows that the improved OpenMP-based TVD–Hopmoc method obtained lower execution time (951 s) than our previous OpenMP implementation of this method (1159 s; see Fig. 5a). Furthermore, Fig. 5b shows that the improved OpenMP-based TVD–Hopmoc method obtained lower wall time (30 s) than our previous OpenMP implementation [8] of this method (36 s). Furthermore, this figure displays that the improved OpenMP implementation of the TVD–Hopmoc method based on cluster of points reaches a lower Clockticks per Instructions Retired (CPI) rate (1.291) than both the simple OpenMP implementation (1.301) and our previous OpenMP implementation of the TVD–Hopmoc method based exclusively on parallel chunk loops (1.418).

Figure 6b shows a CPU usage histogram generated from the use of the Intel® VTune™ Amplifier performance profiler. This figure shows that the improved OpenMP implementation of the TVD–Hopmoc method [8] used on average 23 cores simultaneously when performed on the machine aforementioned.

5 Experimental Results

The first part of this section shows the results of OpenMP implementations of the TVD–Hopmoc method. Intel's OpenMP implementation specifies environment variables that define the policy of binding OpenMP threads to physical processing units (i.e., cores). Thread affinity can have a considerable impact on the computing time of the application. Although this also depends on the system (machine) topology and operating system, we evaluated our OpenMP implementation of the TVD–Hopmoc method based on a cluster of points along with three thread binding policies, namely balanced, compact, and scatter policies.

Figure 7 shows speedups obtained by those three OpenMP implementations of the TVD–Hopmoc method when applied to the same meshes described earlier in executions performed on a machine containing an Intel® Xeon Phi™ Knights-Corner (KNC) accelerator 5110P, 8 GB DDR5, 1.053 GHz, 60 cores, 4 threads per core. The highest speedup reached by the improved implementation based on cluster of points, implementation based only on parallel chunk loops [8], and simple implementation were 56x, 49x, and 28x (106x, 105x, and 95x) [80x, 72x, and 58x], respectively, when applied to a mesh comprised of 10^5 $\left(10^6\right)$ $\left[10^7\right]$

stencil points and T set as 10^6 (10^5) $[10^4]$. The results of our OpenMP-based TVD–Hopmoc presented in Fig. 7 employed the scatter policy. This binding policy yielded the same maximum speedup as the two other policies evaluated when applied to this machine.

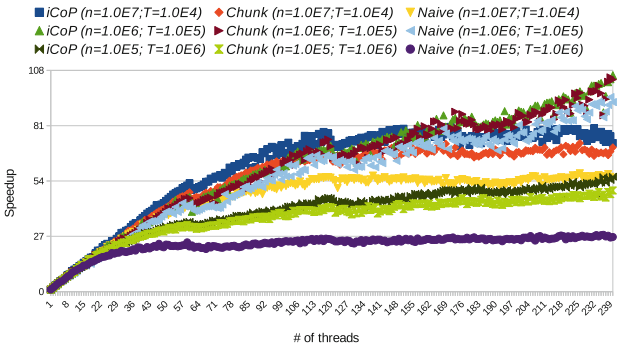


Fig. 7. Speedups obtained by three OpenMP implementations of the TVD–Hopmoc method (iCoP, *chunk* loops [8], and naive implementation) applied to meshes composed of 10^5, 10^6 and 10^7 stencil points and T specified as 10^6, 10^5, and 10^4, respectively, in executions performed on an Intel® Xeon Phi™ KNC accelerator.

Figure 8 shows the results of two OpenMP implementations in runs performed on a machine containing an Intel® Xeon® Platinum 8160 CPU @ 2.10 GHz, with two nodes with 24 cores, 2 threads per core. This figure reveals that our iCoP approach of the TVD–Hopmoc method alongside the balanced binding policy obtained a speedup of approximately 17x (using 95 threads) in this experiment, against a speedup of 11x reached by our previous implementation of the TVD–Hopmoc method [8].

The balanced binding policy distributes threads among physical cores before assigning them to logical cores. Keeping in mind that the system contains two sockets, where each of them holds 24 cores with two threads per core, the balanced (compact) policy assigns threads to both sockets when using more than 24 (48) threads, and consequently, a higher inter-socket communication arises when the system uses this number of threads. Then, when using from 25 to 80 (from 48 to 96) threads, the speedups obtained by the iCoP approach along with the balanced (compact) binding policy significantly vary (see Fig. 8). The same characteristic occurs when employing the CoP approach [8].

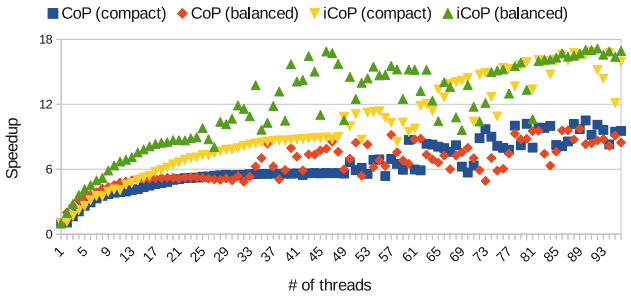


Fig. 8. Speedups of two OpenMP implementations of the 1-D TVD–Hopmoc method applied to meshes composed of 10^6 stencil points and T established as 10^5, in runs performed on an Intel® Xeon® Scalable Processor architecture.

6 Conclusions and Future Directions

This work used Intel® Parallel Studio XE tools to analyze a simple OpenMP-based TVD–Hopmoc method in simulations performed on a machine containing an Intel® Xeon™ CPU E5-2698 v3 @ 2.30 GHz composed of 32 physical cores. As a result, this paper proposed an improved OpenMP implementation of this method based on a cluster of points that obtained a speedup up to 106x (against a speedup of 95x of a naive OpenMP implementation) when applied to a mesh composed of 10^6 stencil points in runs carried out on an Intel® Xeon Phi™ KNC accelerator. Our improved OpenMP implementation of the method achieved a speedup up to 17x (against a speedup of 11x obtained by our previous OpenMP implementation) when applied to a mesh composed of 10^6 stencil points in runs performed on an Intel® Xeon® Scalable Processor. In particular, this work shows how to improve a parallel algorithm based on parallel loops that present high load imbalance by employing a strategy based on parallel chunk loops when analyzing data dependence.

The speedup obtained by our improved method based on a cluster of points was smaller when setting a small number of time steps T than establishing a large number of iterations T (see Fig. 7). In future studies, we intend to investigate the reasons for high cache miss rates and consequently lower speedups in simulations establishing a small number of time steps. Moreover, new studies will be carried out to obtain an implementation that boosts data locality and, hence, generates high cache hit rates. Additionally, we also plan to investigate a coarse–grained implementation that employs an explicit synchronization mechanism capable of providing high speedups in an OpenMP-based TVD–Hopmoc method, even in simulations with a small number of time steps.

Acknowledgement. The Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) supported this work. We would like to thank the Núcleo de Computação Científica at Universidade Estadual Paulista (NCC/UNESP) for letting us execute our simulations

on its heterogeneous multi-core cluster. These resources were partially funded by Intel® through the projects entitled Intel Parallel Computing Center, Modern Code Partner, and Intel/Unesp Center of Excellence in Machine Learning.

References

1. Holstad, A.: The Koren upwind scheme for variable gridsize. Appl. Num. Math. **37**, 459–487 (2001)
2. Oliveira, S.R.F., de Oliveira, S.L.G., Kischinhevsky, M.: Convergence analysis of the Hopmoc method. Int. J. Comput. Math. **86**, 1375–1393 (2009)
3. Harten, A.: High resolution schemes for hyperbolic conservation laws. J. Comput. Phys. **49**, 357–393 (1983)
4. Brandão, D.N., Gonzaga de Oliveira, S.L., Kischinhevsky, M., Osthoff, C., Cabral, F.: A total variation diminishing hopmoc scheme for numerical time integration of evolutionary differential equations. In: Gervasi, O., et al. (eds.) ICCSA 2018. LNCS, vol. 10960, pp. 53–66. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95162-1_4
5. Ma, H., Zhao, R., Gao, X., Zhang, Y.: Barrier optimization for OpenMP program. In: Proceedings of 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking, Parallel and Distributed Computing, pp. 495–500 (2009)
6. Caballero, D., Duran, A., Martorell, X.: An OpenMP* barrier using SIMD Instructions for Intel® Xeon Phi™ coprocessor. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 99–113. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_8
7. Cabral, F.L., Osthoff, C., Kischinhevsky, M., Brandão, D.: Hybrid MPI/OpenMP/OpenACC implementations for the solution of convection diffusion equations with HOPMOC Method. In: Proceedings of 14th International Conference on Computational Science and Its Applications (ICCSA), pp. 196–199 (2014)
8. Cabral, F.L., Osthoff, C., Costa, G.P., Brandão, D., de Oliveira, S.L.G.: Tuning up TVD HOPMOC method on Intel MIC Xeon Phi architectures with Intel Parallel Studio Tools. In: Proceedings of the International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), pp. 19–23 (2017)
9. Gourlay, A.R., McKee, S.: The construction of Hopscotch methods for parabolic and elliptic equations in two space dimensions with mixed derivative. J. Comput. Appl. Math. **3**, 201–206 (1977)
10. van Leer, B.: Towards the ultimate conservative difference schemes. J. Comput. Phys. 361–370 (1974)
11. Douglas Jr., J., Russel, T.F.: Numerical methods for convection-dominated diffusion problems based on combining the method of characteristics with finite element or finite difference procedures. SIAM J. Num. Anal. **19**, 871–885 (1982)