Fine-tuning an OpenMP-based TVD–Hopmoc method using Intel® Parallel Studio XE tools on Intel® Xeon® Architectures

Frederico L. Cabral[1], Carla Osthoff[1], Roberto P. Souto[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,rpsouto,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 paper is concerned with parallelizing the TVD–Hopmoc method for numerical time integration of evolutionary differential equations. Using Intel® Parallel Studio XE tools, we studied three OpenMP implementations of the TVD–Hopmoc method (naive, CoP and EWS-Sync), with executions performed on Intel® Xeon® Many Integrated Core Architecture and Scalable processor. Our implementation, named EWS-Sync, defines an array that represents threads and the scheme consists of synchronizing only adjacent threads. Moreover, this approach reduces the OpenMP scheduling time by employing an explicit work-sharing strategy. Instead of permitting the OpenMP API to perform thread scheduling implicitly, this implementation of the 1-D TVD-Hopmoc method partitions among threads the array that represents the computational mesh of the numerical method. Thereby, this scheme diminishes the OpenMP spin time by avoiding barriers using an explicit synchronization mechanism where a thread only waits for its two adjacent threads. Numerical simulations show that this approach achieves promising performance gains in shared memory for multi-core and many-core environments.

Keywords: OpenMP · Xeon Phi · High performance computing · Parallel processing · Advection–diffusion equation · Thread synchronization.

1 Introduction

Investigations in transport phenomena are crucial in several scientific and engineering problems. For example, in environment or reactive fluid flow problems, a fluid transports and dissolves contaminant or chemical species. Specifically,

the numerical solution of the advection–diffusion transport equation arises from various important applications in engineering, chemistry, and physics. Relevant examples of its use are found in geophysical flows, such as meteorology and oceanography, as well as in the transport of contaminants in air, groundwater, rivers, and lagoons, oil reservoir flow, aerodynamics, astrophysics, biomedical applications, in the modeling of semiconductors, and so forth. Consequently, modeling the transport equation is an expressive subject in numerical mathematics because it has connections with a wide range of scientific and engineering fields [1].

The Hopmoc method (see [2] and references therein) is a spatially decoupled alternating direction procedure for solving advection–diffusion equations. It was designed to be executed in parallel architectures (see [3] and references therein). Specifically, this method decouples the set of unknowns 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. This method combines the time derivative and the advection term as a directional derivative. Thus, it performs time steps in the flow direction along characteristics of the velocity field of the fluid. We consider here the advection–diffusion equation in the form

$$u_t + v u_x = d u_{xx}, \tag{1}$$

with appropriate initial and boundary conditions, where v and d are constant positive velocity and diffusivity, respectively, $0 \leq x \leq 1$ 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 θ_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 linear interpolation technique [2].

Discretization of the advective term in transport equations is frequently afflicted with severe complications. To avoid spurious numerical oscillations, Harten [4] introduced the concepts of Total Variation Diminishing (TVD) techniques and flux limiter, which provide monotonicity-preserving properties and stable higher-order accurate solutions of advection-diffusion equations. The original Hopmoc method employs an interpolation technique to determine the value in the foot of the characteristic line. This inherently introduces numerical errors to the solution. A recent work [5] integrated the Hopmoc method with a TVD scheme with the objective to deal with this restriction. We referred this new approach as TVD–Hopmoc method [5].

We evaluated a naive OpenMP implementation of the TVD–Hopmoc method under the Intel® Parallel Studio XE software for Intel's Haswell/Broadwell ar-

chitectures. This product showed us that the main problems in the performance of a naive OpenMP TVD–Hopmoc method was the use of the implicit OpenMP scheduling and synchronization mechanisms. Hence, a previous publication [3] employed alternative strategies to these naive OpenMP scheduling and synchronization strategies. Our earlier approach reduced the number of loops in relation to the original algorithm from four to two loops, improving the performance of the algorithm in approximately 50%. It used a strategy where a single loop combines the explicit operators, and another loop joins the implicit operators employed in the TVD–Hopmoc method. We named it as Cluster of Points (CoP). However, further investigations revealed that this chunking strategy still presents an unreasonable spin time due to the OpenMP implicit barrier constructs.

The TVD–Hopmoc method computes the solution of an advection–diffusion equation in such a way that a particular thread needs information only from its adjacent threads so that an implicit barrier is unnecessary. Consequently, we replaced the OpenMP implicit barrier by an explicit lock mechanism, in which a synchronization point occurs between adjacent threads, i.e., each thread waits only for two adjacent threads to reach the same synchronization point. The strategy employed is a simple lock mechanism. Using an array of booleans, a thread sets (releases) an entry location in this array and, hence, informs its adjacent threads that the data cannot (can) be used by them. We referred this strategy as explicit work sharing with explicit synchronization (EWS-Sync) [6].

This paper evaluates the EWS-Sync implementation of the 1-D TVD–Hopmoc method when executed on Intel® Xeon Phi™ Knights-Corner and Knights Landing accelerators. Additionally, this paper shows simulations performed on an Intel® Xeon® Scalable Processor. We compare this implementation with the CoP approach [3]. We evaluate both approaches along with three thread binding policies: balanced, compact, and scatter policies.

Section 2 discusses state-of-the-art approaches in load balancing when using the OpenMP standard. Section 3 discusses a naive OpenMP implementation of the TVD–Hopmoc method. Section 4 shows results of the CoP OpenMP-based implementation of the TVD–Hopmoc method [3]. Section 5 presents the EWS-Sync strategy. Section 6 shows the experimental results that compare the new approach with a naive and CoP OpenMP–based TVD–Hopmoc methods. Finally, Section 7 addresses the conclusions and discusses the next steps in this investigation.

2 Related work

Practitioners have been using two scheduling paradigms to address the problem of scheduling multi-threaded computations: work sharing and work stealing. In the work-stealing strategy, underutilized processors attempt to “steal” threads from other processors. The work-stealing idea dates back at least as far as a work proposed by Burton and Sleep [7]. These authors presented a model for concurrently executing process trees, which provided a basis for matching the generation of new tasks to the available resources [7]. They also presented an

interpretation of a topology for the support of virtual process trees on a physical network. These authors point out that the benefits of the work-stealing paradigm to reduce space and communication in a parallel context. Afterward, many researchers have implemented variants on this strategy. Blumofe and Leiserson [8] analyzed a work-stealing algorithm for scheduling "fully strict" (well-structured) multi-threaded computations.

In the work sharing paradigm, whenever a processor generates new threads, the scheduler attempts to migrate some of them to other processors with the purpose of distributing the work to underutilized processors. Intuitively, the migration of threads occurs less frequently when employing a work-stealing approach than using a work-sharing strategy. When many processors have tasks to be done, a work-stealing scheduler does not migrate threads among processors, but a work-sharing scheduler always migrates threads among processors.

Penna *et al.* [9] proposed a workload-aware loop scheduling strategy for irregular parallel loops in which iterations are independent. These authors applied their scheme in a large-scale NUMA machine using a synthetic kernel.

Various researchers have been proposing strategies 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.* [10] proposed strategies to optimize the OpenMP implicit barrier constructs. These authors revealed how to remove the OpenMP implicit 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.* [11] 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.* [12] evaluated the original Hopmoc method in different parallel programming paradigms; however this appraisal was not performed on Intel® Xeon Phi™ accelerators.

A previous publication [3] 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 with the objective of avoiding the fine-grained parallelism, which improved the performance of the implementation in approximately 50%. Another previous work [6] used an explicit work-sharing strategy in conjunction with a new synchronization approach based on a lock array and reached promising results both in multi-core and many-core environments.

3 A naive OpenMP implementation of the TVD–Hopmoc method

This section describes a naive OpenMP implementation of the TVD–Hopmoc method. This naive OpenMP implementation consists of the main time loop that carries out two steps: (*i*) compute the MMOC step, which runs the TVD scheme; (*ii*) compute the first and second (explicit and implicit) semi-steps.

We analyzed a naive OpenMP method (i.e., using the OpenMP *parallel for* directive) under the Intel® Advisor shared memory threading assistance tool. Algorithm 1 shows a fragment of a pseudo-code that is used to obtain the suitability analysis carried out by this shared memory threading assistance tool. This fragment of pseudo-code shows an OpenMP parallel region comprised of a time loop of the TVD–Hopmoc method. This while loop is identified as a parallel region to be examined by the Intel® Advisor shared memory threading assistance tool.

```
1 begin
2 |  #pragma omp parallel;
3 |  {;
4 |  while (t < T) do
5 |  |  [...];
6 |  end
7 |  };
8 end
```

Algorithm 1: A fragment of pseudo-code outlining how to obtain the suitability analysis performed by the Intel® Advisor shared memory threading assistance tool.

Algorithm 2 shows a fragment of pseudo-code that performs a time step of the Hopmoc method in this naive implementation. 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.

A naive approach to parallelize the TVD–Hopmoc method inserts OpenMP directives in each loop that solves: (1) the total diminishing variation scheme; (2) explicit operators; (3) implicit operators. We conducted experiments using the OpenMP static, dynamic, and guided scheduling directives. However, we observed poor performance for static scheduling and that dynamic and guided scheduling directives decrease even more the performance of the algorithm. The Intel® Thread Advisor revealed that even with most of the code vectorized, the estimated gain when using the OpenMP API is limited. The reason for this is because the calculations in the method use very fine granularity to take full advantage of parallelism techniques and HPC capabilities.

We conducted experiments with a naive OpenMP implementation of the TVD–Hopmoc method performed on a machine containing an Intel® Xeon®

```
1  begin
2      #pragma omp for;
3      for (i ← 1; i ≤ n − 1; i ← i + 1) do
           // compute the MMOC step
4      end

5      #pragma omp for;
6      for (i ← 1; i ≤ n − 1; i ← i + 2) do
           // compute the first explicit time semi-step
7      end

8      #pragma omp for;
9      for (i ← 1; i ≤ n − 1; i ← i + 2) do
           // compute the first implicit time semi-step
10     end

11     #pragma omp for;
12     for (i ← 1; i ≤ n − 1; i ← i + 2) do
           // compute the second explicit time semi-step
13     end

14     #pragma omp for;
15     for (i ← 1; i ≤ n − 1; i ← i + 2) do
           // compute the second implicit time semi-step
16     end
17 end
```

Algorithm 2: A time step comprised of four for loops that iterate the first and second time semi-steps of a naive OpenMP–based TVD–Hopmoc method using alternately explicit and implicit approaches.

CPU E5-2698 v3 @ 2.30GHz composed of 32 physical cores. This naive OpenMP implementation of the TVD–Hopmoc method obtained an inefficient performance in a multi-core environment for $\Delta x = 10^{-5}$ (i.e., a mesh composed of 10^5 stencil points). The left side of Figure 1 shows the results of an experiment performed with the support of the Intel® Advisor Advanced Hotspot Analysis shared memory threading assistance tool. It shows the high spin (imbalance or serial spin) and overhead (scheduling) times caused by the implicit OpenMP scheduling mechanism. The left side of Figure 1 shows a high clock ticks per Instructions Retired (CPI) rate obtained by the naive OpenMP implementation of the TVD–Hopmoc method. In general, the CPI rate is the first metric to observe when verifying the performance of an application during tuning effort. Specifically, CPI event ratio is one of the first performance metrics analyzed to study a hardware event-based sampling collection [13]. This ratio is determined by dividing the number of continued processor cycles (clock ticks) by the number of instructions retired. The CPI value of an application is an indication of how much latency influenced its execution. A high CPI value means more latency, on average, during runtime, i.e., the application took more clock ticks for

an instruction to retire [13]. Generally, the code, the processor, and the system configuration determine the CPI rate of a workload, and 0.75 (4) is a reasonable (high) value for this ratio [13].

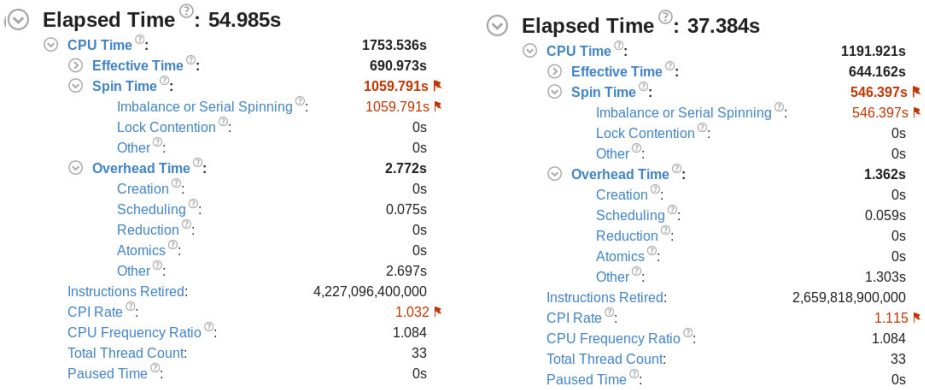

Fig. 1. Executions times obtained by a naive OpenMP-based method on left and the CoP implementation of the TVD–Hopmoc method [3] on right when studied with the support of the Intel® Advisor shared memory threading assistance tool. Spin and overhead times add to the idle CPU usage value.

The left side of Figure 2 shows a CPU usage histogram extracted from the Intel® Advisor shared memory threading assistance tool. This figure reveals that the naive OpenMP implementation of the TVD–Hopmoc method computes a small number of threads simultaneously. In particular, this implementation used on average 12 cores simultaneously (in a machine composed of 32 cores).

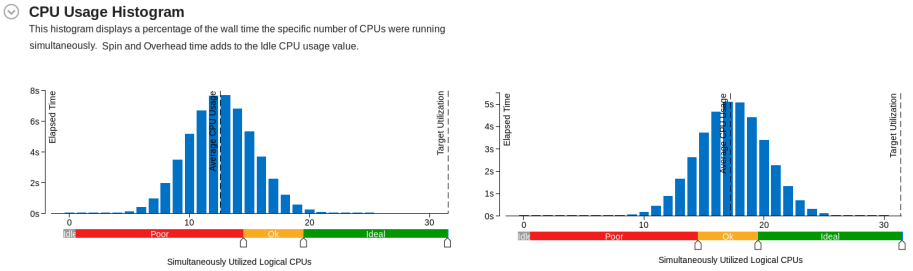

Fig. 2. CPU usage histograms generated in an execution of the naive method on left and in an execution of the CoP implementation of the TVD–Hopmoc method [3] on right. These histograms display a percentage of the wall time, i.e., the specific number of cores that were used simultaneously.

Figure 3 shows the speedup obtained by the naive OpenMP implementation of the TVD–Hopmoc method. This figure shows that the maximum speedup (10) obtained with this implementation is reached when using 15 cores (in a machine with 32 cores).

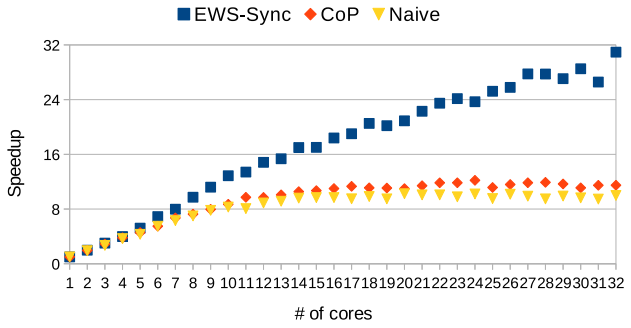

Fig. 3. Speedups of the naive, CoP, and EWS-Sync implementations of the TVD–Hopmoc method applied to the advection–diffusion equation (1) for a Gaussian pulse with amplitude 1.0 and Δx set as 10^{-5} (i.e., a mesh composed of 10^5 stencil points), and $T = 10^6$.

4 The CoP OpenMP-based implementation of the TVD–Hopmoc method

As mentioned, we performed an analysis with the support of the Intel® Thread Advisor. It revealed that even with the code mostly vectorized, the OpenMP API strongly limits the gains because of the fine-grained parallelism used inherently by the OpenMP standard in each loop. This analysis led us to a version in which a single loop joins the explicit operators and a single loop combines the implicit operators in the TVD–Hopmoc method [3]. We named this strategy as Cluster of Points (CoP). This strategy reduced the number of loops used in the original algorithm from four to two loops. It improved the performance of the method in approximately 50%.

A further investigation revealed that this chunking strategy still presented unreasonable spin time due to the OpenMP implicit barrier constructs. The right side of Figure 1 shows the clock ticks per Instructions Retired (CPI) rate obtained by the CoP OpenMP-based implementation of the TVD–Hopmoc method. The right side of Figure 2 shows a CPU usage histogram extracted from the Intel® Advisor shared memory threading assistance tool. This figure reveals that the CoP OpenMP-based implementation of the TVD–Hopmoc method uses 17 threads simultaneously (in a machine with 32 cores).

Figure 3 shows the speedup obtained by the CoP OpenMP implementation of the TVD–Hopmoc method. This figure shows that the maximum speedup (12) obtained with this implementation is reached when using 24 cores (in a machine with 32 cores).

5 An improved explicit work-sharing approach along with an explicit synchronization (EWS-Sync) strategy

Our implementation determines a static array of booleans that denotes the unknowns. Additionally, our implementation handles thread imbalance by subdividing permanent and explicitly this array into the team of threads. Consequently, this implementation carries out thread scheduling only at the beginning of the execution. Thereby, our implementation of the TVD–Hopmoc method does not use the OpenMP *parallel for* directive because each thread has its data. A thread sets (releases) its associated entry in this array to notify its two adjacent threads that the data cannot (can) be used [6].

The EWS-Sync OpenMP-based implementation of the 1-D TVD–Hopmoc method slightly improves our previous implementation [6]. We removed the *#pragma omp atomic* directive from the code that updates the lock array. This modification improved the results of the EWS-Sync OpenMP-based implementation in more than 40%.

Algorithm 3 shows a fragment of pseudo-code that outlines how we synchronize adjacent threads. Line 10 in this fragment of code shows how we define this array of locks when executing it in a machine with up to 240 threads. Since the first (last) thread have no neighbor to its left (right) side, the first (last) entry of this lock array is unset. In particular, this implementation is a thread-safe code.

Algorithm 3 also describes the explicit synchronization mechanism employed in the TVD–Hopmoc method and how we synchronize adjacent threads. It shows how we replace OpenMP barriers, defining a range from localStart to localEnd variables for each thread in the team.

A few spin time may be desirable instead of increasing thread context switches. High spin time, however, can diminish productive work. The OpenMP barrier directive recognizes a synchronization point at which threads in a parallel code fragment will not run after the OpenMP barrier until all other threads in the team terminate all their tasks in the parallel code fragment. Then, one can use the no-wait clause and include an OpenMP barrier directive outside the loop; but even with these directives, all threads in the team synchronize at the same point.

Figure 3 shows the speedup obtained by the EWS-Sync OpenMP implementation of the TVD–Hopmoc method. This figure shows that the maximum speedup (31) obtained with this implementation is reached when using 32 cores (in a machine composed of 32 cores).

We also performed the experiments with the TVD–Hopmoc method using the EWS-Sync strategies on a machine containing an Intel® Xeon® CPU E5-2698 *v*3 @ 2.30GHz with 32 physical cores. Figure 4 shows the results of an experiment performed with this implementation and the support of the Intel® Advisor Advanced Hotspot Analysis shared memory threading assistance tool. Figure 4 shows that the EWS-Sync implementation obtained lower execution time (458s) than both the naive OpenMP (1754s) and CoP implementations (1192s; see Figure 1) of the TVD–Hopmoc method. Moreover, Figure 4 exhibits that our EWS-Sync implementation obtained lower wall time (16s) than both

```
1  begin
2      tid ← omp_get_thread_num();
3      nt ← omp_get_num_threads();
4      size ← (n−2)/nt;
5      remainder ← (n − 2)%nt;
6      localStart ← tid · size + 1;
7      localEnd ← localStart · size − 1;
8      if (tid = nt − 1) then localEnd ← localEnd + remainder;
9      nid ← tid + 1;
10     boolean lock[242];
11     lock[nid] ← false;
12     #pragma omp master;
13     {
14       lock[0] ← false;
15       lock[nt+1] ← false;
16     }
17     #pragma omp flush (lock) // update lock array
18     [...]
       // lock mechanism: inform the adjacent threads that
       // this thread is performing a task in the shared memory
19     lock[nid] ← true;
20     for (i ← localStart; i ≤ localEnd; i ← i + 2) do
         // some work
21     end
       // release the shared memory to the adjacent threads
22     lock[nid] ← false;
       // verify if the shared memory is
       // locked awaits until it is released
23     while (lock[nid + 1] ∨ lock[nid − 1]) do ;
24     [...]
25 end
```

Algorithm 3: A fragment of pseudo-code that shows the explicit synchronization mechanism employed in the EWS-Sync TVD–Hopmoc method.

the naive OpenMP (55s) and CoP (37s) implementations of this method. The Figure 4 also shows a CPI rate smaller than 0.7 when executing the EWS-Sync implementation, against a CPI rate higher than 1 obtained by both naive and CoP implementations of the TVD–Hopmoc method (see Figure 1).

Figure 5 shows a CPU usage histogram extracted from the Intel® Advisor shared memory threading assistance tool. This figure shows that the EWS-Sync

Fig. 4. Execution time obtained by an explicit work-sharing OpenMP–based TVD–Hopmoc method when studied with the support of the Intel® Advisor shared memory threading assistance tool. Spin and overhead times add to the idle CPU usage value.

implementation of the TVD–Hopmoc method used approximately 32 threads simultaneously when performed on the machine afore cited. Figure 3 shows that the EWS-Sync implementation obtained a speedup of approximately 31 when set to run with 32 threads in the machine afforested.

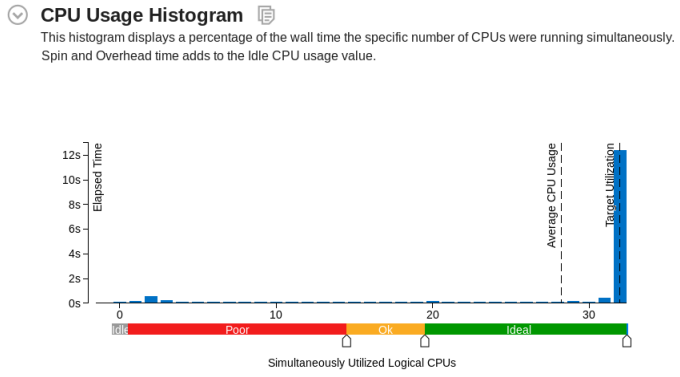

Fig. 5. CPU usage histogram generated in an execution of the EWS-Sync implementation of the TVD–Hopmoc method [6]. Again, this histogram displays a percentage of the wall time, i.e., this implementation simultaneously uses a specific number of cores during its execution.

6 Results and analysis

This section presents the results of the CoP and EWS-Sync approaches in executions performed on Intel® Xeon® architectures. In particular, we evaluate both implementations along with three thread binding policies: balanced, compact, and scatter policies.

This section shows experiments that apply both OpenMP-based implementations of the 1-D TVD–Hopmoc method to the advection–diffusion equation

(1) 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), $\Delta t = 10^{-5}$, $\Delta x = 10^{-5}$ (i.e., 10^5 stencil points), and T is established as 10^6. Specifically, Sections 6.1 and 6.2 present the results of the CoP and EWS-Sync approaches in runs carried out on Intel® Many Integrated Core architectures and a Scalable Processor, respectively.

6.1 Executions performed on Intel® Many Integrated Core architectures

Figure 6 shows the results of the EWS-Sync and CoP approaches in executions performed on a machine containing an Intel® Xeon Phi™ Knights-Corner (KNC) accelerator 5110P 1.053 GHz, with 8GB DDR5 of main memory, composed of 60 cores, with 4 threads per core. This figure exhibits that the EWS-Sync implementation yielded a speedup of approximately 150x (using 239 threads) in this simulation alongside the balanced thread binding policy. Therefore, Figure 6 shows that EWS-Sync implementation dominated the CoP implementation of the TVD-Hopmoc method, which obtained a speedup of approximately 52x.

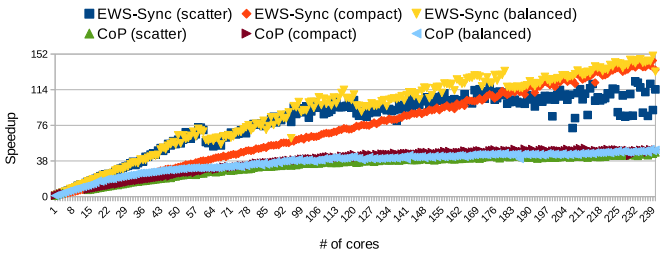

Fig. 6. Speedups obtained by two OpenMP implementations of the 1-D TVD–Hopmoc method in executions performed on an Intel® Xeon Phi™ Knights-Corner accelerator.

Figure 7 shows the results of both EWS-Sync and CoP OpenMP-based implementations in runs carried out on a machine containing an Intel® Xeon® Phi™ Knights Landing (KNL) accelerator CPU 7250 @ 1.40GHz, composed of 68 cores, with 4 threads per core. This figure exhibits that the EWS-Sync approach used in conjunction with the balanced (bal.) thread binding policy delivered a speedup of 132x (using 271 threads) in this simulation. In particular, the EWS-Sync implementation improves our previous version of the CoP method, which obtained a speedup of up to 25x in executions carried out on this Intel® Xeon Phi™ accelerator.

Figures 6 and 7 shows four discontinuities when using both scatter and balanced thread binding policies because of the increased communication among cores when using a larger number of threads. The compact binding policy allocates software threads on the same core, generating some overload on it. On

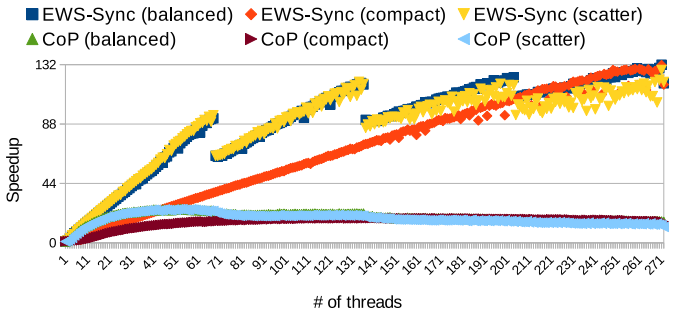

Fig. 7. Speedups of two OpenMP-based implementations of the 1-D TVD–Hopmoc method in runs performed on an Intel® Xeon Phi™ Knights Landing accelerator.

the other hand, it reduces traffic along the interconnection bus. This characteristic does not appear when executing the CoP strategy because, based on implicit OpenMP barriers, the high spin time overcomes the time spent in the interconnection among cores and sockets (see Figures 6, 7, and 8).

6.2 Executions performed on an Intel® Scalable Processor

Figure 8 shows the results of both OpenMP-based implementations in runs performed on a machine containing two nodes of an Intel® Xeon® Platinum 8160 CPU @ 2.10GHz, where each node is composed of 24 cores, with 2 threads per core. This figure reveals that the EWS-Sync approach of the TVD–Hopmoc method alongside the balanced thread binding policy obtained a speedup of approximately 55x (using 95 threads) in this experiment, against a speedup of 11x reached by the CoP implementation of the TVD–Hopmoc method.

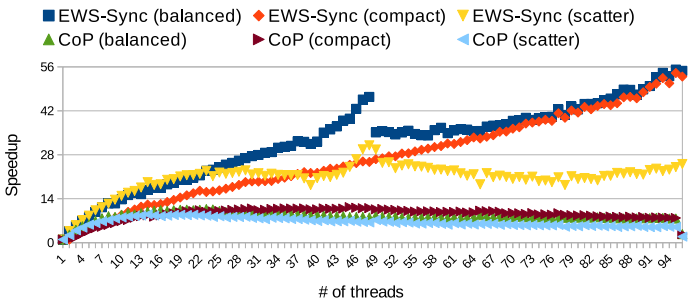

Fig. 8. Speedups of two OpenMP-based implementations of the 1-D TVD–Hopmoc method in runs performed on an Intel® Xeon® Scalable Processor.

In simulations performed on the Skylake and KNC architectures, the use of the EWS-Sync approach along with both compact and balanced binding policies obtain better speedups than the scatter binding policy. In simulations performed on the Skylake architecture, this is due to the inter-socket communication.

Since the Skylake architecture contains two sockets connected by a bus, the scatter binding policy is the worst way to distribute threads because of the

increased traffic alongside the bus. The compact binding policy distributes software threats to hardware threads in such a way that every two threads occupy a single physical core. This thread binding policy overloads some cores even when others threads are available. For this reason, the speedup is small when using a small number of threads.

The balanced binding policy distributes the threads inside a single socket before assigning them to the second socket with the difference that it assigns software threads to physical cores as long as they are available in any socket. The balanced thread binding policy reached the best results among the binding policies evaluated here. Figure 8 shows a discontinuity when the number of threads goes from 48 to 49. The reason for this is because the communication between the sockets appears and therefore some overhead is introduced.

7 Conclusion

This paper shows an OpenMP–based 1-D TVD-Hopmoc method that improves our previous implementations [3,6]. Our implementation employs an explicit work-sharing approach alongside a specific synchronization mechanism. The strategies used here to implement our OpenMP–based TVD–Hopmoc method achieved reasonable speedups in both multi-core and manycore architectures.

This OpenMP implementation defines an array that represents stencil points where each thread will operate. Thus, this implementation uses an explicit work-sharing strategy by previously defining this array with the objective of reducing the scheduling time. Using a lock array where each entry represents a thread, an approach that synchronizes adjacent threads replaces a synchronization time in barriers. These strategies permit the threads to attain a reasonable load balancing.

Our EWS-Sync TVD–Hopmoc method reached a speedup of approximately 150x (132x) when applied to a mesh composed of 10^5 stencil points in a simulation performed on an Intel® Xeon Phi™ Knights-Corner (Knights Landing) accelerator composed of 240 (272) threads. Moreover, this implementation attained a speedup of approximately 55x when applied to the same mesh in a simulation carried out on an Intel® Xeon® Scalable Processor.

We plan to provide further investigations with the objective of providing a better speedup in executions on this Intel® Xeon® Scalable Processor. Another 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 (or a matrix) to represent the stencil points so that the approach employed in the 1–D case of the TVD–Hopmoc method is still valid.

Acknowledgments

CNPq, CAPES, and 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. A. Holstad. The Koren upwind scheme for variable gridsize. Applied Numerical Mathematics, 37:459–487, 2001.
2. S.R.F. Oliveira, S.L. Gonzaga de Oliveira and M. Kischinhevsky. Convergence Analysis of the Hopmoc Method. Int. J. of Comput. Math., 86: 1375–1393, 2009.
3. F.L. Cabral, C. Osthoff, G.Costa, S.L. Gonzaga de Oliveira and D.N.Brandão, M. Kischinhevsky. Tuning up TVD HOPMOC method on Intel MIC Xeon Phi Architectures with Intel Parallel Studio Tools. Proceedings of the 8th Workshop on Applications for Multi-Core Architectures, 2017.
4. A. Harten. High resolution schemes for hyperbolic conservation laws. Journal of Computational Physics. (49): 357–393, 1983.
5. D.N. Brandão, S.L. Gonzaga de Oliveira, M. Kischinhevsky, C. Osthoff, F. Cabral (2018) A Total Variation Diminishing Hopmoc Scheme for Numerical Time Integration of Evolutionary Differential Equations. In: O. Gervasi et al. (eds) Computational Science and Its Applications – ICCSA 2018. ICCSA 2018, pp. 53–66. Lecture Notes in Computer Science, vol 10960. Springer, Cham.
6. F.L. Cabral, C. Osthoff, G.P. Costa, S.L. Gonzaga de Oliveira, D. Brandão, M. Kischinhevsky (2018) An OpenMP Implementation of the TVD–Hopmoc Method Based on a Synchronization Mechanism Using Locks Between Adjacent Threads on Xeon Phi(TM) Accelerators. In: Y. Shi et al. (eds) Computational Science – ICCS 2018. ICCS 2018, pp. 701–707. Lecture Notes in Computer Science, vol 10862. Springer, Cham.
7. F.W. Burton and M.R. Sleep. Executing functional programs on a virtual tree of processors. In Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (Portsmouth, N.H., Oct.). ACM, New York, N.Y., pp. 187–194, 1981.
8. R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), vol 46, Issue 5, Sept., pages 720–748, 1999.
9. P.H. Penna, M. Castro, P. Plentz, H.C. Freitas, F. Broquedis and J.F. Mehaut. BinLPT: A Novel Worload-Aware Loop Scheduler for Irregular Parallel Loops. Braziliam Simposium of High Perfomance Computing. 11: 527–536, 2017.
10. H. Ma, R. Zhao, X. Gao and Y. Zhang. Barrier Optimization for OpenMP Program. Proceedings of 10th ACIS Int. Conf. on Software Engineering, Artificial Intelligences, Networking, Parallel and Distributed Computing, 495–500, 2009.
11. D. Caballero, A. Duran and X. Martorell. An OpenMP Barrier Using SIMD Instructions for Intel Xeon Phi™ Coprocessor. OpenMP in the Era of Low Power Devices and Accelerators. IWOMP 2013. A.P. Rendell and B.M. Chapman M.S. Muller(Editors). Lecture Notes in Computer Science. 8122: 99–113, 2013.
12. F.L. Cabral, C. Osthoff, M. Kischinhevsky and D. Brandão. Hybrid MPI/OpenMP/OpenACC Implementations for the Solution of Convection Diffusion Equations with Hopmoc Method. Proceedings of 14th International Conference on Computational Science and Its Applications (ICCSA), 196–199, 2014.
13. Intel. Clockticks per Instructions Retired (CPI). Available at https://software.intel.com/en-us/vtune-amplifier-help-clockticks-per-instructions-retired-cpi. Visited in 2017-11-30.