

Reducing Cache Miss Rate Using Thread Oversubscription to Accelerate an MPI-OpenMP-Based 2-D Hopmoc Method

F. L. Cabral[1(✉)], C. Osthoff[1], and S. L. Gonzaga de Oliveira[2]

[1] Laboratório Nacional de Computação Científica (LNCC), CENAPAD, Petrópolis, Brazil
{fcabral,osthoff}@lncc.br

[2] Universidade Federal de São Paulo, São José dos Campos, SP, Brazil
sanderson.oliveira@unifesp.br

Abstract. This paper applies the MPI-OpenMP-based two-dimensional Hopmoc method using the explicit work-sharing technique with a recently proposed mechanism to reduce implicit barriers in OpenMP. Specifically, this paper applies the numerical algorithm to yield approximate solutions to the advection-diffusion equation. Additionally, this article splits the mesh used by the numerical method and distributes them to over-allocated threads. The mesh partitions became so small that the approach reduced the cache miss rate. Consequently, the strategy accelerated the numerical method in multicore systems. This paper then evaluates the results of implementing the strategy under different metrics. As a result, the set of techniques improved the performance of the parallel numerical method.

Keywords: Parallel computing · OpenMP · MPI · High performance computing · Thread oversubscription

1 Introduction

Researchers use partial differential equations to describe relevant problems in different areas. Examples include scientific and engineering applications in aerospace, petroleum, environmental, chemical and biomedical engineering, geoscience, and elsewhere. In more specific problems, practitioners use partial differential equations to describe flow problems inherent in geology, oceanography, and the propagation of pollutants in different media such as water and air. Specifically, we consider the advection-diffusion equation $u_t + vu_x = du_{xx}$, where v and d are positive constants of velocity and diffusion, respectively.

A recent publication [1] used the two-dimensional Hopmoc method to approximate the solution of advection-diffusion equations with dominant convection [2]. The Hopmoc method calculates the convection process for each time semi-step,

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
O. Gervasi et al. (Eds.): ICCSA 2022, LNCS 13375, pp. 337–353, 2022.
https://doi.org/10.1007/978-3-031-10522-7_24

eliminating the second term on the left side of the original equation, reducing the equation to the problem $u_t = du_{xx}$. Afterward, the method calculates the diffusion process by dividing the computational mesh into two subsets calculated alternately in each time semi-step, one with an explicit approach and the other with an implicit approach. Although it is an implicit method, this approach does not apply a linear equation systems solver.

Algorithms of this nature enable parallelism and other strategies. Furthermore, the two-dimensional Hopmoc method increases the application performance and low-cost approximate solutions using highly refined meshes [1]. These characteristics contribute to studying the results more accurately and closer to reality.

This paper studies the effect that thread oversubscription (i.e., a quantity of threads higher than the number of available threads) has on the performance of an MPI-OpenMP-based numerical method while also considering the influence of synchronization. Specifically, this paper analyzes the performance of a parallel resource thread oversubscription strategy applied in an MPI-OpenMP-based numerical algorithm in multicore computing environments. The study focuses on the potential benefits of oversubscription for reducing cache misses and increasing application speedup. Thus, the paper uses the 2-D Hopmoc method to approximate the solution to the advection-diffusion equation. Furthermore, the study provides two implementations of the approach for comparison. A naive version uses regular parallel work-sharing loops with barriers at the end of each semi-step. A second implementation, called MPI-OMP-EWS, uses fine-grain synchronization between consecutive regions, allowing a point-to-point synchronization between threads working on adjacent data.

This paper organizes the remainder of the text as follows. We present the related work in Sect. 2. Afterward, we describe the two-dimensional Hopmoc method implemented with OpenMP and MPI [3] in Sect. 3. In Sect. 4, we explain two variants of the Hopmoc algorithm. In Sect. 5, we describe how we performed the tests. In Sect. 6, we show the results. Finally, in Sect. 7, we address the conclusions.

2 Related Work

Practitioners have developed different techniques to reduce cache miss in applications executed in multicore systems (e.g., [4–6]). Cabral et al. [7] showed the performance of the Hopmoc method in a shared memory computing environment Intel® MIC Xeon® Phi using the OpenMP programming model. The publication compared a naive version of the Hopmoc algorithm with a strategy that used the explicit task chunk technique. The study concluded that this second alternative reduced overhead and spin times. Cabral et al. [7–11] employed a total variation diminishing scheme to increase the accuracy of the Hopmoc method.

Cabral et al. [9] implemented the Hopmoc method using the explicit work-sharing technique and more efficient OpenMP directives. The authors referred to this approach as the OMP-EWS strategy.

Cabral et al. [11] employed an explicit work-sharing approach alongside a specific synchronization mechanism. The strategies achieved reasonable speedups in both multi-core and manycore architectures.

A recent publication [1] proposed a mechanism to accelerate the hybrid MPI-OpenMP-based two-dimensional Hopmoc method in a multicore system. Furthermore, the study used a hybrid version of the two-dimensional Hopmoc algorithm based on MPI and OMP-EWS. The article referred to the strategy as MPI-OMP-EWS. The paper compared three two-dimensional numerical methods on different Intel® processors.

In general, publications avoid thread oversubscriptions because the technique introduces prohibitive overheads due to excessive thread preemptive scheduling, context switch costs, loss of locality, and contention on shared resources. On the other hand, previous studies have successfully explored the thread oversubscription technique [12–15].

Iancu et al. [12] evaluated the impact of task oversubscription on the performance of MPI, OpenMP, and UPC implementations of the NAS Parallel Benchmarks on UMA and NUMA multi-socket architectures. The authors evaluated explicit thread affinity management against the default Linux load balancing and considered sharing and partitioning system management strategies. The authors observed that sharing the available cores between applications provided better throughput than explicit partitioning. Modest levels of oversubscription improved system throughput by 27% and provided better performance isolation of applications. Their results indicated that the determining behavioral factor when applications share a system is the granularity of the synchronization tasks.

Navarro et al. [13] addressed the problem of the underutilization of the CPU core where a host thread operates. The authors evaluated a host thread blocking strategy combined with oversubscription that delegated to the operating system the duty of scheduling threads to available CPU cores to guarantee that all cores executed practical work. The authors observed that allowing oversubscription controlled by the operating system could be favorable under specified scenarios.

Jiang et al. [14] examined a shared memory multi-threaded approach to streamline tracing that targets data-intensive architectures. Their study pointed out that thread oversubscription of streamlined tasks is an effective technique for hiding I/O latency.

Huang et al. [15] conducted a comprehensive study of the overhead of thread oversubscription. The authors found that the direct cost of context switching does not cause a noticeable performance slowdown in most applications. Furthermore, the authors observed that oversubscription could be both constructive and destructive to the performance of CPU caches.

3 Hybrid Version of the Hopmoc Method Based on MPI and OpenMP

Hopmoc is a numerical strategy that provides approximate solutions to convection-dominated advection-diffusion equations. The method integrates

features from the Hopscotch algorithm [16] and the modified method of characteristics (MMOC) [17]. Thus, the approach allows the separation of variables through characteristic curves.

The Hopmoc algorithm divides the computational mesh into two distinct sets. Then, the algorithm explicitly updates the first set of unknowns in the first time semi-step. Subsequently, the method implicitly updates the second set of unknowns in the second time semi-step. The algorithm alternates the two groups of unknowns at each time step. Thus, the method does not apply a linear system. Each iteration of the main while loop represents a time step. This main loop executes a set of loops responsible for alternately performing explicit and implicit calculations of the two time semi-steps of each time step. Algorithm 1 depicts the pseudocode of the Hopmoc method.

```
1  begin
2      Each MPI process computes its partition mesh
3      Each OMP thread computes its local
4      mesh inside the partition mesh allocated to a process
5      while time < FinalTime do
6          ...
7          lock
8          for columns in the thread local mesh do
9              for all lines in the input mesh do
10                 MMOC or an explicit or implicit time semi-step
11         unlock
12         Each OMP thread waits for its neighbours
13         ...
```

Algorithm 1: Overview of the Hopmoc method.

The hybrid version of the Hopmoc method is a strategy that uses both OpenMP threads and MPI processes to split the workload and allow for efficient parallelization [1]. The implementation divides the input mesh into smaller partitions. The approach permanently assigns each of these partitions to a process. The strategy guarantees the first level of parallelism, using MPI processes exclusively. Next, each MPI process divides its workload among the available threads. Specifically, the approach parallelizes all inner for loops in the main outer while loop. This process guarantees the second level of parallelism, therefore, using threads exclusively.

4 Two Implementations of the Hopmoc Method

In this section, we present two implementations of the Hopmoc method. The two variations differ in the way thread parallelism is applied. Thus, we divided this section into two parts. The first, presented in Sect. 4.1, is intended to explain

the naive implementation of the Hopmoc method. In Sect. 4.2, we describe the hybrid version of the MPI-OpenMP-based two-dimensional Hopmoc algorithm.

4.1 Naive Implementation

The thread-level parallelism of this implementation uses standard OpenMP directives. Furthermore, inner the outer loop, each for loop immediately precedes an OpenMP directive **#pragma omp parallel for**. Algorithm 2 shows an overview of the parallelization used in the naive implementation.

```
1  begin
2      while time < FinalTime do
3          #pragma omp parallel for
4          for all columns in the input mesh do
5              for all lines in the input mesh do
6                  Modified method of characteristics
7          #pragma omp parallel for
8          for all columns in the input mesh do
9              for all lines in the input mesh do
10                 First explicit time semi-step
11         #pragma omp parallel for
12         for all columns in the input mesh do
13             for all lines in the input mesh do
14                 First implicit time semi-step
15         #pragma omp parallel for
16         for all columns in the input mesh do
17             for all lines in the input mesh do
18                 Second explicit time semi-step
19         #pragma omp parallel for
20         for all columns in the input mesh do
21             for all lines in the input mesh do
22                 Second implicit time semi-step
```

Algorithm 2: Overview of the parallel naive implementation.

4.2 MPI-OMP-EWS Strategy

The MPI-OMP-EWS-based two-dimensional Hopmoc method relies on a parallelization mechanism consisting of two main techniques: explicit work-sharing (EWS) and synchronization between adjacent threads (AdjSync) [1]. The EWS technique explicitly divides the mesh partitions allocated to the MPI process among the available threads. The approach assigns permanently to each thread

a mesh partition. Furthermore, the MPI-OMP-EWS strategy performs a 1-D partitioning of the mesh. For example, consider a mesh composed of $10^4 \times 10^4$ stencil points. In this example, each thread receives 10^4 mesh lines. Moreover, each thread holds 10^4 mesh lines divided by the number of threads set for the execution. For example, when the run uses 1,000 threads, each thread receives a partition composed of $10^4 \times 10$ stencil points.

The AdjSync mechanism is responsible for synchronizing the threads available in the process. With this approach, synchronization does not wait for all threads to reach the same synchronization point as performed in the naive implementation. Instead, the strategy only synchronizes adjacent threads. Furthermore, the approach only synchronizes each thread with its two adjacent threads, i.e., a previous thread and a succeeding thread.

Algorithm 3 shows an overview of the parallelization used in the implementation based on the MPI-OMP-EWS strategy. Before each inner for loop, the thread informs its neighbors of an execution block (see line 3 in Algorithm 3). Afterward, the thread removes this lock and waits until its neighbors perform the same operation (see line 7 in Algorithm 3).

```
1 begin
2     ...
3     lock
4     for columns in the partition mesh do
5         for all lines in the input mesh do
6             MMOC or an explicit or implicit time semi-step
7     unlock
8     ...
```

Algorithm 3: Pseudocode excerpt of parallel Hopmoc method using MPI-OMP-EWS.

The implementation allocates one MPI process to each machine's CPU in executions without thread oversubscription. Additionally, we executed the hybrid MPI-OMP-EWS strategy using from one thread to the maximum number of OpenMP threads per MPI process.

5 Description of the Tests

The first machine used in the tests featured a computer with two Intel® Xeon® processors E5-2698 v3, each composed of 32 physical cores, hyperthreading disabled, and total memory of 112 GB. The operating system was the CentOS Linux version 7.3.1611 (Core). We refer to this computer as the M1 machine. We extracted results from the Intel® VTune™ Amplifier performance profiler in executions performed on this machine.

The second machine used in the experiments a computer with four Intel® Xeon® Gold 6230R, each composed of 26 physical cores, with hyperthreading enabled (two threads per core) and total memory of 384 GB. We refer to this computer as the M2 machine. We used the mpiicpc (Intel® C++ compiler) with the optimization flags -O3 and -xHost to generate the executable codes in the two computers.

We conducted an exploratory investigation to determine the number of iterations. With more than 1,000 iterations, there was no change in speedup in the two versions of the two-dimensional Hopmoc method used in this study. With less than 1,000 iterations, the speedup was worse than with 1,000 iterations. This feature occurred because of a lack of workload. Thus, the simulations performed 1,000 iterations in each execution of the two-dimensional Hopmoc method.

At the beginning of an experiment, the experiment allocated an MPI process to each CPU. This allocation meant that the execution used two (four) MPI processes in the experiment performed on the M1 (M2) machine. Afterward, the test increased the number of threads from 1 to 32 (52) in the simulation conducted on the M1 (M2) computer system. When the run used 32 (52) threads per CPU, each physical core had one (two) thread(s) in the experiment performed on the M1 (M2) machine, recalling that we enabled the hyperthreading on the M2 high-performance unit. Subsequently, the test over-allocated threads, maintaining 32 (52) threads per process in the experiment performed on the M1 (M2) machine. Afterward, the simulation increased the number of MPI processes from 2 to 200 (74), totaling 3,200 (3744) threads in the experiment performed on the M1 (M2) machine. Above these quantities, the standard MPI aborted the execution, indicating a system resource limit. The test incremented the number of MPI processes by two (four) in the experiment executed on the M1 (M2) machine so that each CPU had the same number of threads.

The runs had exclusive access to the hardware, i.e., no other tasks were running simultaneously. We calculated the speedup as the ratio between serial version and parallel implementation times. The experiments ensured convergence as the 2-D Hopmoc method is convergent [18] and because the odd-even Hopscotch method shows unconditional numerical stability [19].

6 Results and Analysis

This section shows speedup metrics for two high-performance systems and L1, L2, and L3 cache miss for one of the systems. Additionally, we measured synchronization overhead concerning the overall execution and spin times regarding CPU time. Sections 6.1 and 6.2 show the experiment performed on the M1 and M2 machines, respectively.

6.1 Experiment Performed on the M1 Machine

We executed the MPI-OMP-EWS-based implementation with a mesh composed of $10^3 \times 10^3$ stencil points. In this test, the thread oversubscription technique

did not benefit the performance of the MPI-OMP-EWS-based implementation. Despite the small mesh partitions in each thread, the high overhead costs required to handle the high number of threads outweighed the cache miss rate reduction.

Additionally, we executed the MPI-OMP-EWS-based implementation using a mesh composed of $10^5 \times 10^5$ stencil points. Again in this test, the thread oversubscription technique did not benefit the performance of the MPI-OMP-EWS-based implementation. Thread oversubscription would require many threads to improve the MPI-OMP-EWS-based implementation executed with a large-scale mesh. With a high number of threads, the mesh partition of each thread would probably be small enough to reduce cache miss rates. However, the Intel® VTune™ Amplifier performance profiler indicated high cache miss rates even when the system reached the limits of MPI processes and OpenMP threads.

Figure 1 shows the speedup of executions of the two implementations evaluated in this study. The simulation used a two-dimensional mesh composed of $10^4 \times 10^4$ points, employing 10^{-4} as a variation of the mesh spacing. We performed executions with up to 3,200 threads. However, the figure shows the results of runs with 1,100 threads in the plot. Specifically, the behavior of the results does not differ when the runs used more than 1,100 threads. The serial versions of the naive and MPI/OMP-EWS implementations took 1423 s and 1420 s, respectively. Up to approximately 500, the speedup of the two approaches was very close (≈6x) with slight superiority of the implementation with the MPI-OMP-EWS strategy. The speedup of the MPI-OMP-EWS approach increased to more than 25x when executed with more than 500 threads until approximately 1,000 threads. However, the speedup of the naive implementation remained at approximately 6x.

The naive version does not have the mesh partitioned as performed in the implementation using the MPI-OMP-EWS strategy. We implemented the former with the directive **#pragma omp parallel for** as reported in Sect. 4.1 and the latter as described in Sect. 4.2. Thus, the naive version distributes loop iterations among existing threads. Therefore, there is no data partitioning, and each thread does not know in advance its data partition, as performed in the implementation that uses the MPI-OMP-EWS strategy. Thus, the speedup of the naive version yielded a maximum of 6.5x with 32 threads.

Figures 2 and 3 show the L1 and L2 cache miss counts from executions of naive and MPI-OMP-EWS-based implementations of the two-dimensional Hopmoc method. This test varied the number of threads from 1 to 3,200. Figure 4 shows the results of the last level cache (LLC) miss rate for executions of the two implementations evaluated in this study. The results yielded by the two approaches in runs with up to 500 threads are very similar $\left(\approx 3 \cdot 10^9\right)$. With executions using more than 500 threads, the cache miss rate in LLC memory with the naive implementation increases. With runs with threads ranging from 500 to 700, the cache miss rate in LLC memory of the code based on the MPI-OMP-EWS strategy decreases significantly, reaching less than 10^9 in executions using more than 700 threads. With runs with the implementation based on the

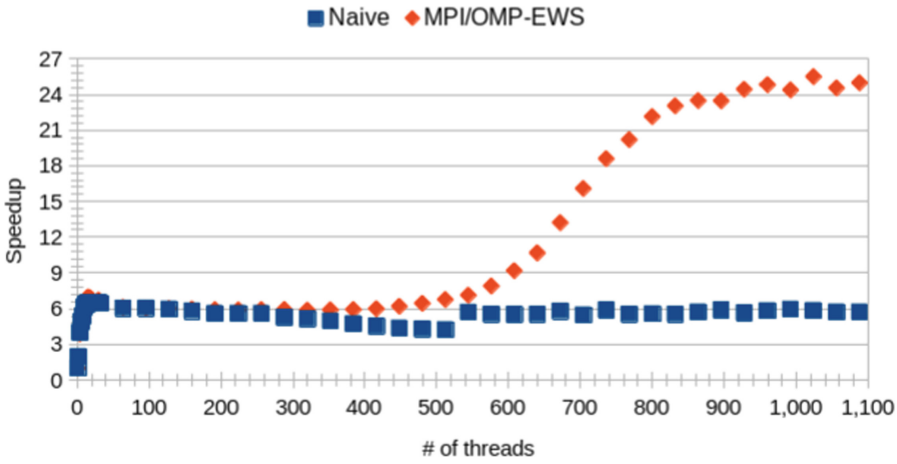

Fig. 1. Speedups from executions of naive and MPI-OMP-EWS implementations of the two-dimensional Hopmoc method using from 1 to 1,100 threads.

MPI-OMP-EWS approach using more than 500 threads, possibly, the mesh partitions are too small. Thus, we suppose the mesh partitions remain at the same cache memory level. We consider that the use of tread oversubscription creates mesh partitions very small. Consequently, we assume that the process efficiently allocates the small mesh partitions in the same cache memory level. This process possibly requires a short time to create and schedule these threads.

As previously mentioned, Fig. 4 shows that the cache miss rate in LLC memory becomes low when using more than 700 threads in the implementation based on the MPI-OMP-EWS strategy. Even with this number of threads, the implementation speedup was ≈16x (see Fig. 1). The speedup of the implementation was more than 25x when using more than 1,000 threads. Therefore, the cache hit rate in LLC memory does not explain this speedup gain. Thus, when the implementation employed 700 to 1,000 threads, the cache hit rate becomes higher in L2 cache-memory and subsequently in L1 cache memory (see Figs. 2 and 3).

Figure 5 shows the results of the ratio of Spin time to CPU time. Spin time is the time in synchronization barriers (i.e., the time the code waits for all threads to be synchronized and ready to work) in routines that process threads, such as the creation and scheduling. CPU time is the metric that represents the amount of time that a CPU uses for processing instructions of a process. Thus, CPU time is neither Spin time nor Overhead time, but the processing time in calculations and procedures specific to the method. Figure 5 shows that the ratio between Spin time and CPU time in the naive execution was always higher than in the MPI-OMP-EWS-based version.

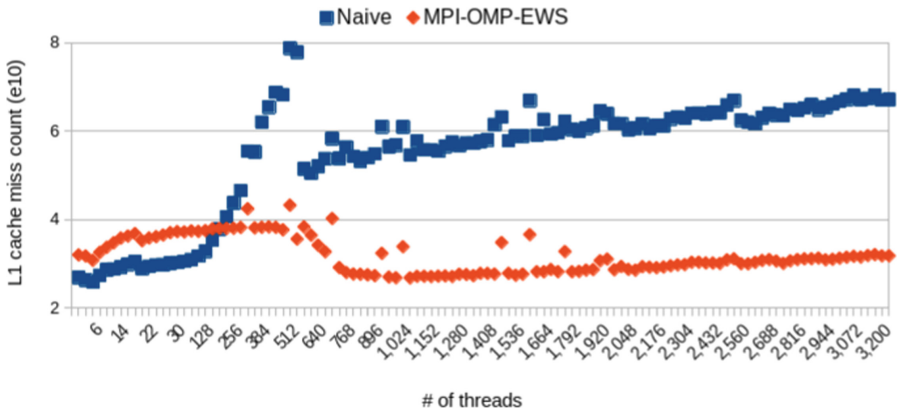

Fig. 2. L1 cache miss count from executions of naive and MPI-OMP-EWS implementations of the two-dimensional Hopmoc method using from 1 to 3,200 threads.

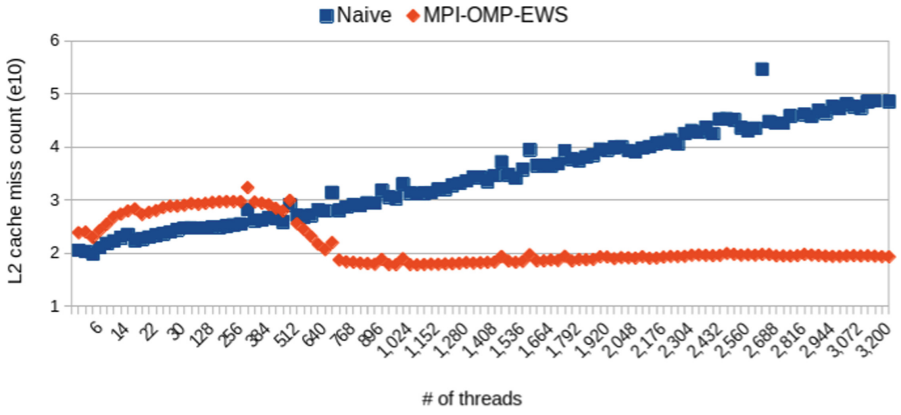

Fig. 3. L2 cache miss count from executions of naive and MPI-OMP-EWS implementations of the two-dimensional Hopmoc method using from 1 to 3,200 threads.

Figure 6 exhibits the results of the ratio of overhead time to CPU time. Overhead time is the time in the routines of the operating system. The figure also shows that the performance profiles change in executions using more than 500 threads. The proportion between overhead time and CPU time always grew. This proportion increased significantly when the runs used more than 500.

Figure 6 shows that the computational cost of creating, allocating, and scheduling several threads did not significantly increase the total running time. Specifically, for approximately 500 to 2,000 threads, the overhead time

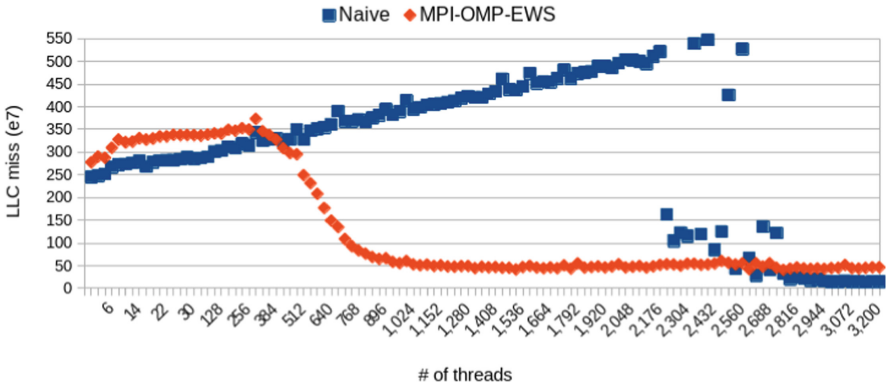

Fig. 4. LLC miss count from executions of naive and MPI-OMP-EWS implementations of the two-dimensional Hopmoc method using from 1 to 3,200 threads.

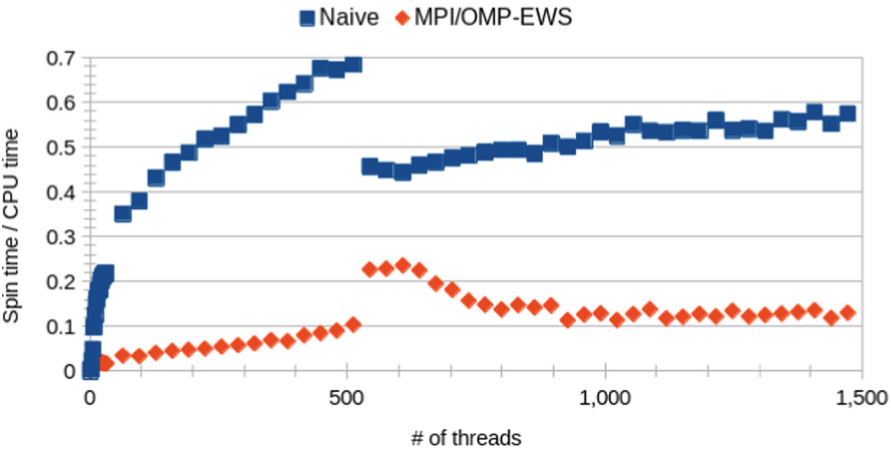

Fig. 5. Spin time by CPU time ratio of executions of the naive and MPI-OMP-EWS-based implementations of the two-dimensional Hopmoc method with up to 1,500 threads.

represented less than 2% of the productive processing time of the method. This result points out the effectiveness of the thread oversubscription technique in the MPI-OMP-EWS strategy: reducing mesh segments to their adequacy to the cache memory size compensated for the time to over-allocate threads (preemptive scheduling and context switching between threads).

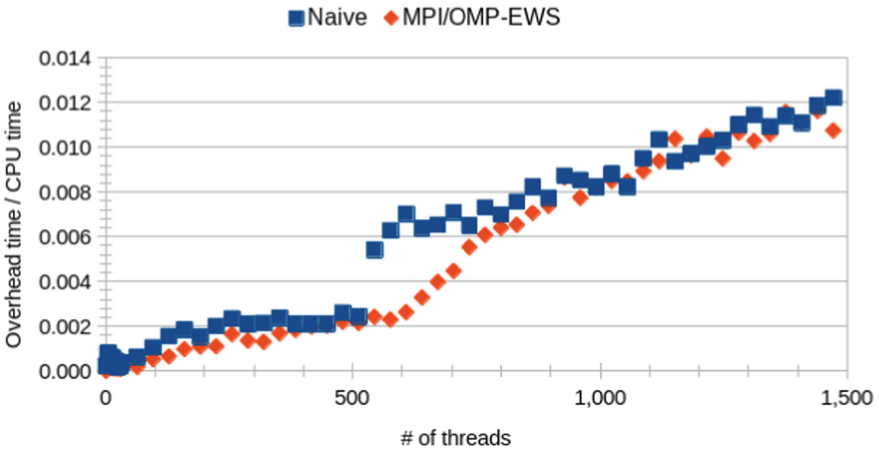

Fig. 6. Overhead time by CPU time ratio of executions of the naive and MPI-OMP-EWS implementations of the two-dimensional Hopmoc method with up to 1,500 threads.

The naive implementation does not divide the mesh again within each process. The reason is that the directive **#pragma omp parallel for** is responsible for the parallelization. Thus, one cannot realize the effect of the thread oversubscription technique. On the other hand, this is remarkable in the tests within the implementation based on the MPI-OMP-EWS strategy. In addition to explicitly dividing the mesh and distributing them to processes, this implementation also explicitly splits the mesh and distributes them to threads.

Figure 7 shows the CPU time of the implementations. The figure shows another result that points out the effectiveness of the strategy. The figure shows two distinct CPU time behaviors for the two versions of the Hopmoc method evaluated in this study. With runs using less than 500 threads, the two curves have a very similar profile. With runs using more than 500 threads, the two curves differ. There is a drop with discontinuity, and then the curve rises again with less slope in the curve representing the CPU time of the naive implementation. The CPU time is reduced continuously in the variant with the MPI-OMP-EWS strategy. Then, the results stabilize with executions with more than 700 threads.

These results reaffirm and agree with those presented in the plots that show the speedup, L1, L2, and last level cache rate miss (see Figs. 1, 2, 3, and 4). While there is a reduction in the cache miss in LLC, there is a significant decrease in CPU time for the Hopmoc method based on the MPI-OMP-EWS strategy, which directly leads to a higher speedup (see Fig. 1).

We previously mentioned that the naive implementation does not have an explicit second division inside each process as is performed in the MPI-OMP-EWS

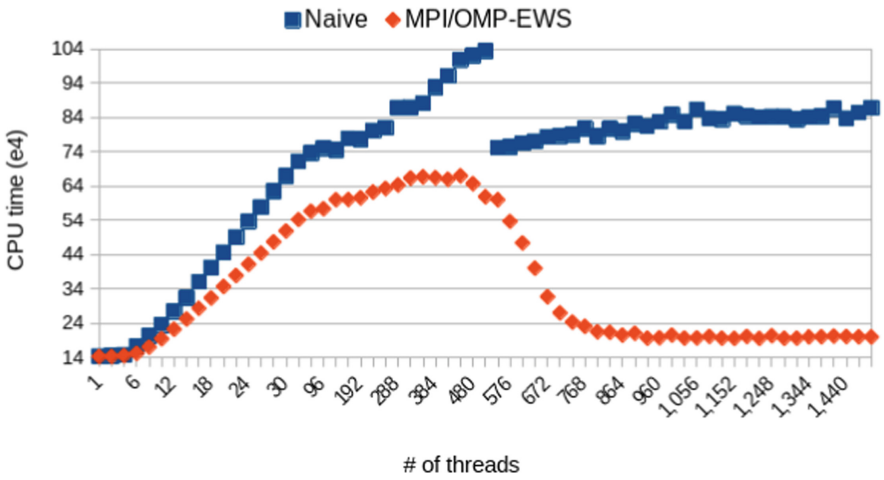

Fig. 7. CPU time of naive and MPI-OMP-EWS-based implementations of the two-dimensional Hopmoc method with up to 1,500 threads.

strategy. Thus, the naive implementation is less sensitive to cache miss reduction and, consequently, to its benefits.

Figure 8 shows the running times of other simulations performed with the two implementations evaluated in this study. The simulation used a two-dimensional mesh composed of $2 \times 10^4 \times 2 \times 10^4$ points, employing 2×10^{-4} as a variation of the mesh spacing. The serial versions of the naive and MPI/OMP-EWS implementations took 5654 s and 5848 s, respectively. The two versions took similar running times when using up to 600 threads. When using more than 600 threads, the MPI-OMP-EWS-based implementation was remarkably faster than the naive implementation.

6.2 Experiment Performed on the M2 Machine

Figure 9 shows the speedup for the two implementations evaluated in this study. The simulations used a two-dimensional mesh composed of $10^4 \times 10^4$ points, employing 10^{-4} as a variation of the mesh spacing. The serial versions of the naive and MPI/OMP-EWS implementations took 1382 s and 1174 s, respectively.

The MPI-OMP-EWS-based implementation increased the speedup from 1 to 68 threads. We recall that 104 physical cores and 208 threads comprise the machine. Up to 416 threads, the speedup was approximately 33x. Then, the speedup increased to 165x with 3,744 threads. The serial version of the MPI-OMP-EWS-based version probably had a high cache miss rate. We used the

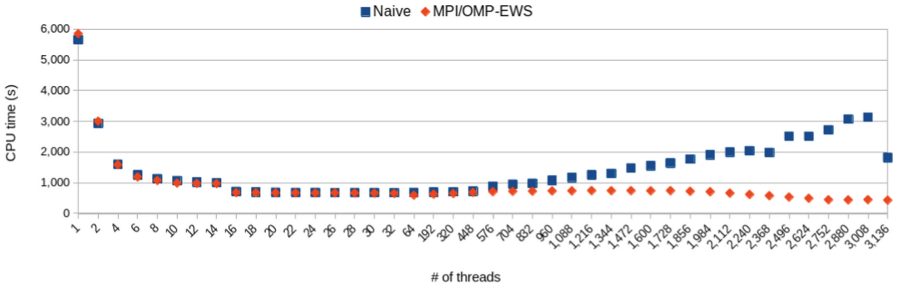

Fig. 8. Running times from executions of naive and MPI-OMP-EWS implementations of the two-dimensional Hopmoc method using from 1 to 3,136 threads.

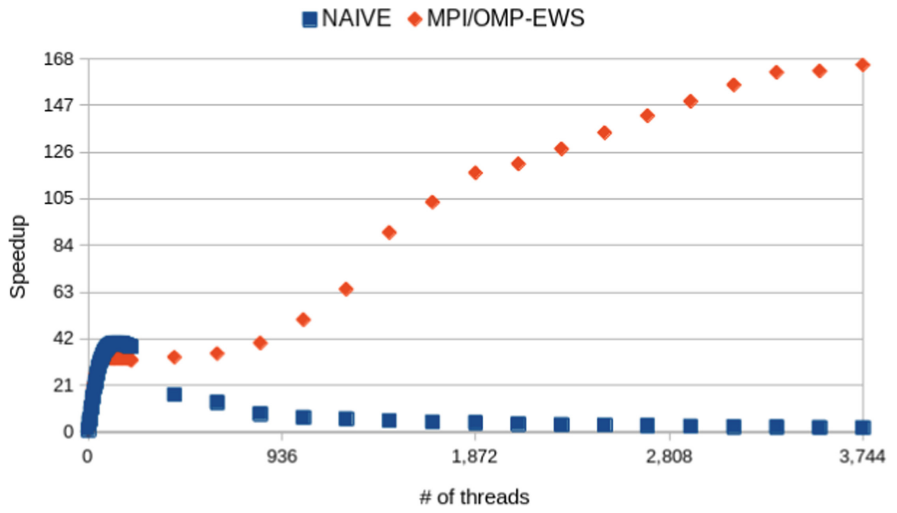

Fig. 9. Speedups from executions of naive and MPI-OMP-EWS-based implementations of the two-dimensional Hopmoc method using from 1 to 3,744 threads.

thread oversubscription technique to reduce this problem. Consequently, the use of the approach improved the performance of the implementation.

The naive implementation yielded a speedup of 40x using 106 to 248 threads. Then, the speedup of the implementation decreased, returning a speedup of 2x with 3,744 threads.

7 Conclusions

We performed an experiment using the MPI-OMP-EWS strategy in the two-dimensional Hopmoc algorithm in executions performed on a computer with two Intel® Xeon® processors E5-2698 v3, composed of 32 physical cores. The MPI-OMP-EWS-based implementation yielded a speedup of 7x running with 32 threads. When using the thread oversubscription technique, the speedup of the implementation increased to 25x, almost reaching a speedup equal to the number of physical cores of the machine.

We also experimented with the MPI-OMP-EWS-based implementation of the two-dimensional Hopmoc algorithm executions on a computer with four Intel® Xeon® Gold 6230R, composed of 104 physical cores and 208 threads. The MPI-OMP-EWS-based implementation yielded a speedup of 35x running with 208 threads. When using the thread oversubscription technique, the speedup of the implementation increased to 165x.

The focus of this study was to reduce the sizes of the numerical method mesh partitions so that they can fit in the same level of cache memory. As a result, communication between the segments would be less costly. The implementation achieved this objective using the thread oversubscription technique in the experiments.

Although the overhead time of allocation and scheduling (preemptive scheduling and context switching between threads) of threads, the experiment achieved a satisfactory result with the MPI-OMP-EWS strategy applied to implement a parallel version of the two-dimensional Hopmoc method in tandem with the thread oversubscription technique. Thus, this paper showed that the MPI-OMP-EWS process enables an efficient performance of the 2-D Hopmoc method using thread oversubscription. The approach allowed it by eliminating the implicit barriers of the OpenMP standard and reducing the spin time. The conclusion is that for fine-grain synchronization, oversubscription of threads through MPI is beneficial.

We intend to continue this study to confirm the possibility of extracting a high performance in parallelism and efficiently using the physical resources of a multicore system. We plan to conduct new experiments with the parallel techniques used in this study. We also plan to continue this study by employing the strategy in other numerical methods executed in different microarchitectures, such as the third-generation Intel® Xeon® Scalable processors.

Acknowledgments. CAPES - Coordenação de Aperfeioamento de Pessoal de Nvel Superior (Coordination for Enhancement of Higher Education Personnel, Brazil) supported this study.

References

1. Cabral, F.L., Gonzaga de Oliveira, S.L., Osthoff, C., Costa, G.P., Brandão, D.N., Kischinhevsky, M.: An evaluation of MPI and OpenMP paradigms in finite-difference explicit methods for PDEs on shared-memory multi- and manycore systems. Concurrency Comput. Pract. Exp. **32**(20), e5642 (2020). e5642 cpe.5642
2. Oliveira, S., Kischinhevsky, M., Gonzaga de Oliveira, S.L.: Convergence analysis of the Hopmoc method. Int. J. Comput. Math. **86**, 1375–1393 (2009)
3. MPI Forum. MPI forum (2022). https://www.mpi-forum.org/. Accessed 20 Mar 2022
4. Prisaganec, M., Mitrevski, P.J.: Reducing competitive cache misses in modern processor architectures. Int. J. Comput. Sci. Inf. Technol. (IJCSIT) **8**(6), 49–57 (2016)
5. Cassales, G., Gomes, H., Bifet, A., Pfahringer, B., Senger, H.: Improving parallel performance of ensemble learners for streaming data through data locality with mini-batching. In: IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Yanuca Island, Cuvu, Fiji. IEEE (2020)
6. Alperen, A., Afibuzzaman, M., Rabbi, F., Ozkaya, M.Y., Catalyurek, U., Aktulga, H.M.: An evaluation of task-parallel frameworks for sparse solvers on multicore and manycore CPU architectures. In: ICPP 2021: 50th International Conference on Parallel Processing, pp. 1–11 (2021)
7. Cabral, F.L., Osthoff, C., Costa, G.P., Brandão, D., Kischinhevsky, M., Gonzaga de Oliveira, S.L.: Tuning up the TVD-HOPMOC method on Intel MIC Xeon Phi architectures with Intel Parallel Studio tools. In: 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), pp. 19–24. IEEE (2017)
8. 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
9. Cabral, F.L., Osthoff, C., Costa, G.P., Gonzaga de Oliveira, S.L., Brandão, D., Kischinhevsky, M.: An OpenMP implementation of the TVD–Hopmoc method based on a synchronization mechanism using locks between adjacent threads on Xeon Phi (TM) accelerators. In: Shi, Y., et al. (eds.) ICCS 2018. LNCS, vol. 10862, pp. 701–707. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93713-7_67
10. Cabral, F., et al.: An improved OpenMP implementation of the TVD–Hopmoc method based on a cluster of points. In: Senger, H., et al. (eds.) VECPAR 2018. LNCS, vol. 11333, pp. 132–145. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-15996-2_10
11. Cabral, F.L., et al.: Fine-tuning an OpenMP-based TVD–Hopmoc method using Intel® parallel studio XE tools on Intel® Xeon® architectures. In: Meneses, E., Castro, H., Barrios Hernández, C.J., Ramos-Pollan, R. (eds.) CARLA 2018. CCIS, vol. 979, pp. 194–209. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16205-4_15
12. Iancu, C., Hofmeyr, S., Blagojevic, F., Zheng, Y.: Oversubscription on multicore processors. In: IEEE International Symposium on Parallel Distributed Processing (IPDPS), Atlanta, GA, pp. 1–11. IEEE (2010)

13. Navarro, A., Vilches, A., Corbera, F., Asenjo, R.: Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. J. Supercomput. **70**, 756–771 (2014)
14. Jiang, M., Essen, B.V., Harrison, C., Gokhale, M.B.: Multi-threaded streamline tracing for data-intensive architectures. In: Childs, H., Pajarola, R., Vishwanath, V. (eds.) 4th IEEE Symposium on Large Data Analysis and Visualization, LDAV, Paris, France, pp. 11–18. IEEE Computer Society (2014)
15. Huang, H., et al.: Towards exploiting CPU elasticity via efficient thread oversubscription. In: HPDC 2021: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, pp. 215–226 (2021)
16. Gordon, P.: Nonsymmetric difference equations. SIAM J. Appl. Math. **13**, 667–673 (1965)
17. Douglas, J., Jr., Russell, T.F.: Numerical methods for convection-dominated diffusion problems based on combining the method of characteristics with finite element method or finite difference procedures. SIAM J. Numer. Anal. **19**, 871–885 (1982)
18. Robaina, D.: BDF-Hopmoc: an implicit multi-step method for the solution of partial differential equations based on alternating spatial updates along the characteristic lines. Ph.D. thesis, Instituto de Computação, Universidade Federal Fluminense, July 2018. (in Portuguese)
19. Gourlay, A.R.: Hopscotch: a fast second-order partial differential equation solver. IMA J. Appl. Math. **6**(4), 375–390 (1970)