Received: 15 March 2019 | Revised: 26 September 2019 | Accepted: 17 November 2019
DOI: 10.1002/cpe.5642

WILEY

SPECIAL ISSUE PAPER

An evaluation of MPI and OpenMP paradigms in finite-difference explicit methods for PDEs on shared-memory multi- and manycore systems

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

[1]CENAPAD, LNCC, Petrópolis, RJ, Brazil
[2]DCC, UFLA, Lavras, MG, Brazil
[3]EIC, CEFET, Rio de Janeiro, RJ, Brazil
[4]IC, UFF, Niterói, RJ, Brazil

Correspondence
Carla Osthoff, Av. Getulio Vargas, 333 - Quitandinha, Petrópolis, RJ 25651-075, Brazil.
Email: osthoff@lncc.br

Present Address
Carla Osthoff, 333 Getulio Vargas Avenue, Quitandinha, 25651-075 Petrópolis-RJ, Brazil

Funding information
National Council for Scientific and Technological Development; Intel®

Summary

This paper focuses on parallel implementations of three two-dimensional explicit numerical methods on Intel® Xeon® Scalable Processor and the coprocessor Knights Landing. In this study, the performance of a hybrid parallel programming with message passing interface (MPI) and Open Multi-Processing (OpenMP) and a pure MPI implementation used with two thread binding policies is compared with an improved OpenMP-based implementation in three explicit finite-difference methods for solving partial differential equations on shared-memory multicore and manycore systems. Specifically, the improved OpenMP-based version is a strategy that synchronizes adjacent threads and eliminates the implicit barriers of a naïve OpenMP-based implementation. The experiments show that the most suitable approach depends on several characteristics related to the nonuniform memory access (NUMA) effect and load balancing, such as the size of the MPI domain and the number of synchronization points used in the parallel implementation. In algorithms that use four and five synchronization points, hybrid MPI/OpenMP approaches yielded better speedups than the other versions did in runs performed on both systems. The pure MPI-based strategy, however, achieved better results than the other proposed approaches did in the method that employs only one synchronization point.

KEYWORDS

high-performance computing, hybrid MPI/OpenMP programming, MPI, multicore architectures, parallelism, parallel processing

1 | INTRODUCTION

The current propensity for multicore architectures and the widespread use of manycore systems emphasize the demand for parallel solutions. Developers face a difficult decision regarding which programming model to use since researchers have proposed new languages, application program interfaces (APIs), and other alternatives for operating these architectures more efficiently. Open Multi-Processing (OpenMP) and message passing interface (MPI) are two dominant models commonly employed in parallel solutions. A developer should use OpenMP, MPI, or a hybrid version depending on the system configurations in context. The type of problem in context and its size also influence the performance of applications that employ these programming models. Practitioners have performed comparisons involving single shared memory systems, shared memory clusters, and distributed memory clusters. As the industry launches new architectures, the literature requires up-to-date performance evaluations of state-of-the-art parallel solutions for programming the new systems.

Presently, multicore and manycore clusters are the most favored alternative for the arrangement of high-performance computing (HPC) infrastructures, owing to the scalability, performance, and expense ratio of these architectures. Practitioners typically program these architectures

using MPI[1] on distributed memory, OpenMP on shared memory, and MPI with OpenMP on hybrid shared or distributed memory systems. Attaining a high parallel performance gain, however, is not simple in practice. Much of the success of the OpenMP model is because it has a simple interface. OpenMP requires less programming effort than MPI. Additionally, much of the synchronization and data sharing is transparent for the OpenMP user. OpenMP is also efficient for sharing data between threads because this standard can access shared memory directly without the need for communicating explicitly and copying data on the same node. However, publications report that achieving high efficiency with OpenMP when using a large number of cores and threads is not trivial.[2] Additionally, it is not clear if all applications and hardware platforms can effectively benefit from an MPI/OpenMP approach (see the work of Bassi et al[3] and references therein). Bassi et al[3] cited five reasons: (1) the code should be fully parallelized, including in the OpenMP context; (2) the use of OpenMP involves overhead due to the initialization of parallel and work-sharing regions; (3) explicit and implicit synchronizations of some of the work-sharing constructs also cause overhead; (4) the penalization caused by false sharing of cache lines among processors also generates a time delay; and (5) the API suffers from the reduction of the memory bandwidth of cache-coherent nonuniform memory access (NUMA) hardware used for the most recent and popular multicore processors. Bassi et al[3] also explained that some of these arguments emphasize the dominant influence of the hardware characteristics on parallel efficacy. The authors considered that developers should take into account the number of NUMA regions within a node. Additionally, the authors should consider the resulting latency when transferring data from the random access memory of one of these NUMA regions to the cache memory of the core associated with a different region. OpenMP offers no instruction for making the deployment of data in a specific NUMA region. Developers can either use a first contact strategy or employ some operating system call, such as numactl, which provides the ability to control NUMA scheduling and memory placement policies. Developers can partially control the affinity of data to a socket. Even if they can boost the OpenMP performance, the efficacy of these strategies may change with the platform hardware and the operating system. Additionally, the compiler influences the efficacy of OpenMP.[3]

Previous work has found that a naïve OpenMP-based implementation of an explicit method for solving partial differential equations (PDEs) is inefficient.[2] Thus, a previous publication[4] proposed an OpenMP-based one-dimensional (1D) Hopmoc method in tandem with total variation diminishing (TVD)[5] for numerical time integration of evolutionary differential equations. We evaluated the approach in executions performed on machines featuring Intel® Many Integrated Core and Xeon® Scalable Processor architectures. The approach achieved low speedups when a small number of time steps were set. Nevertheless, the parallel approach boosts data locality and, consequently, generates high cache hit rates. Additionally, we investigated 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 involving a small number of time steps. As a result, Cabral et al[2] proposed a scheme consisting of synchronizing only adjacent threads. The approach employs an explicit work-sharing (EWS) strategy to reduce the OpenMP scheduling time. The OpenMP-based 1D TVD-Hopmoc method partitions among threads an array that represents the computational mesh instead of allowing the OpenMP API to perform thread scheduling implicitly. Thus, the OMP-EWS scheme avoids barriers using an explicit synchronization mechanism. Specifically, a thread waits only for two adjacent threads in the OMP-EWS approach. Therefore, the approach diminishes the OpenMP spin time required in executions performed on shared-memory multicore and manycore architectures. Despite the satisfactory performance, the OMP-EWS approach employs a busy waiting loop for thread synchronization. On the other hand, MPI performs communication between processes automatically. This feature motivated us to compare the performance of the APIs in implementing explicit parallel methods.

Developers perform communication among processes explicitly with MPI. Moreover, a large set of functions in the MPI API permits high performance and adjusts, which is not available in OpenMP. Designing MPI-based implementations for distributed memory systems is a common practice. Nevertheless, because there is a limited amount of message exchange in the parallel implementation discussed in the present study, we investigated whether the MPI API can also perform well on shared-memory systems. Our motivation is that each process communicates only with its neighboring processes in the parallel implementations. For example, although a hybrid MPI/OpenMP implementation achieved better results than the pure MPI strategy did, Bassi et al[3] claimed that a discontinuous Galerkin solver is well suited for a pure MPI-based implementation because of how their parallel implementation partitions the domain. Thus, we investigated the selection of the best paradigm for each specific application running on multicore or manycore platforms. Specifically, this paper evaluates the use of pure MPI, the OMP-EWS approach,[2] and hybrid MPI/OMP-EWS approaches with scatter and compact thread binding policies aiming at improving the parallel efficiency of three explicit finite-difference (FD) methods for solving PDEs running on Intel® Many Integrated Core and Xeon® Scalable Processor shared-memory architectures. We apply the approaches to Laplace's, heat conduction, and advection-diffusion equations.

In executions performed on the machine featuring an Intel® Xeon® Scalable Processor shared-memory architecture, the hybrid MPI/OMP-EWS version of the three explicit FD methods employed two MPI processes (one process per socket), setting the number of threads from 1 to 48. In particular, we used the next neighbor halo exchange communication pattern.

In executions performed on the machine featuring an Intel® Many Integrated Core shared-memory architecture, we evaluated two hybrid MPI/OMP-EWS versions. First, each single physical CPU core had an MPI process with four threads. Thus, each process had dedicated level 1 cache memory, and the compact and scatter thread binding policies had distinct results. We varied the number of processes from 1 to 68 in the approach. We refer to this strategy as the MPI/OMP-EWS core approach. Thus, we executed this hybrid MPI/OMP-EWS version in conjunction with scatter and compact thread binding policies. Second, each tile was allocated to an MPI process. A tile contains two physical CPU cores that shares the same level 2 cache memory (containing 1 MB on the machine used in this study). In this case, we varied the number of processes from 1 to 34, with eight threads per process. We refer to this strategy as the MPI/OMP-EWS level 2 (L2) approach. This hybrid MPI/OMP-EWS

approach delivered the same results when applied in tandem with scatter and compact thread binding policies. Our goal was to verify which of these approaches was best suited to be employed with the three 2D explicit FD methods evaluated.

This paper is organized as follows. In Section 2, an overview of recent publications using HPC shared-memory resources is provided. In Section 3, the three methods used in the computational experiment are reported. In Section 4, the parallel approaches evaluated in this study are described. In Section 5, benchmarking results obtained from multicore and manycore systems are shown and analyzed. Finally, In Section 6, the main conclusions of this computational experiment are presented.

2 | RELATED WORK

The widespread use of multicore processors requires performance evaluations of algorithms based on pure MPI-based versions against OpenMP-based implementations on multicore architectures. Previous publications described that the best model depends on the context.

Mallón et al[6] presented a comparative performance evaluation of MPI and OpenMP on an HP Integrity Superdome system with 64 Montvale Itanium two dual-core processors (total 128 cores) at 1.6 GHz and using 1 TB of memory. The authors employed two applications to evaluate the performance and scalability of these parallel programming solutions: a matrix multiplication kernel and the Sobel edge detection kernel. When 128 cores were used, both platforms suffered from remote memory access and poor bidirectional traffic performance in the cell controller. OpenMP dominated MPI in three scenarios. The first example was an iterative solver that tested regular communications in sparse matrix-vector multiplications. The second scenario was a series of executions of 1D Fast Fourier transform procedure on a three-dimensional (3D) mesh that evaluated aggregated communication performance. The third example was a large integer sorting algorithm. The last application evaluated the integer computation performance and aggregated communication throughput. Speedups using OpenMP were usually higher than those of MPI due to its direct shared memory access, which avoids memory copies as in MPI. MPI yielded better results in two cases. The first case was a parallel code that assessed floating-point performance without meaningful communication, whereas the second case was a simplified multigrid kernel that performed short-distance and long-distance communication.

For a similar purpose, Chan and Yang[7] compared pure MPI with OpenMP in eight parallel benchmark problems taken from the NASA Advanced Supercomputing Division: a block tridiagonal solver, the conjugate gradient method, an embarrassingly parallel benchmark, a solver of a 3D PDE using the fast Fourier transform, the bucket sort for sorting integers, a simulated computational fluid dynamics (CFD) application that split the problem into block lower and upper triangular systems, a V-cycle multigrid method for solving a 3D scalar Poisson equation, and a CFD application involving a scalar pentadiagonal solver. The authors compared pure MPI and OpenMP. OpenMP outperformed MPI in most cases in execution time. OpenMP scalability was also better than MPI in executions performed with a large number of cores. The communication overhead in the MPI standard showed the main weakness of the programming model in these problems. Communication overhead is the fraction of time that an application spends communicating with the team of processes instead of performing productive work. The authors used eight different problems, and some affected communication more than others.

As previously mentioned, the widespread use of multicore processors also demands performance evaluation of hybrid MPI and OpenMP implementations. Several publications have evaluated hybrid MPI and OpenMP implementations when applied to improve the parallel efficiency of applications running on clusters of multicore nodes. Drosinos and Koziris[8] compared the performance of three programming paradigms for the parallelization of nested loop algorithms on symmetric multiprocessor (SMP) clusters. The authors concluded that hybrid approaches could be more beneficial in some cases than the monolithic pure message-passing model. The reason is that hybrid approaches adequately utilize the resources of a hierarchical parallel platform, such as an SMP cluster.

Along the same line, Jones and Yau[9] proposed a hybrid MPI and OpenMP approach to improve the parallel efficiency of the ordered-subsets expectation-maximization algorithm for reconstructing 3D positron emission tomography images. The algorithm reduced the interprocessor data exchange time, which was the dominant limiting factor of parallel efficiency of the MPI model when a large number of processors were used for the shared memory and the single system image distributed shared memory architectures. The hybrid MPI and OpenMP approach achieved consistent improvement in terms of speedup on a large number of parallel processors compared to the pure MPI approach.

Many publications have also investigated hybrid MPI and OpenMP implementations on manycore architectures. Jeffers et al[10] evaluated parallel implementations of an explicit FD solution of the Poisson equation. Similar to our study, the authors compared a hybrid implementation (an MPI/OpenMP approach and an MPI-based implementation with threads) with a pure MPI-based implementation on a manycore architecture. The study employed a stencil mesh of size 3000×3000 using 1, 2, 4, and 8 nodes simultaneously, where each node was an Intel® Xeon® Phi Knights Landing accelerator. The authors concluded that the hybrid version yielded better results than the pure MPI-based implementation.

In the same vein, Cafaro et al[11] investigated the behavior of parallel shared-memory implementations of the space-saving algorithm concerning accuracy and performance on manycore and multicore processors, including an Intel® Phi 7120P accelerator. Specifically, the authors evaluated a hybrid MPI/OpenMP version (ie, MPI internode and OpenMP intranode) against a pure MPI-based version of their algorithm using up to 512 cores. The hybrid MPI/OpenMP implementation delivered considerable better performance than the pure MPI version of the algorithm. The results showed that the MPI/OpenMP version of the algorithm substantially improved the parallel speedup and efficiency due to the use of the

shared-memory approach. The authors concluded that the Intel® Phi accelerator was not useful for their algorithm because it exhibited a highly limited data locality, and the noncontiguous memory access restricts the exploitation of the cache hierarchy.

Similarly to the previous publications, Koleva-Efremova[12] compared the performance and scalability of the pure MPI-2 API with the hybrid MPI-2 and OpenMP implementation in executions performed on the supercomputer system Avitohol. The author used two Intel® benchmark tests: ping-pong operations and one-sided communication. The hybrid MPI/OpenMP implementation provided better memory consumption compared with the pure MPI-2 version.

The MPI shared memory model, supported by the Intel® MPI Library since version 5.0.2, permits changes to existing MPI codes aiming at accelerating communication between processes on shared-memory nodes. Brinskiy et al[13] evaluated the use of this model. The authors concluded that the model could benefit applications that use the next neighbor-halo exchange communication pattern.

Researchers have investigated the NUMA architecture in specific applications. Bassi et al[3] described a hybrid MPI/OpenMP implementation of a discontinuous Galerkin solver. The authors restricted the use of MPI calls when performing communication between nodes. The authors showed that the performance is heavily dependent on hardware platforms, as well as on computational details, such as either the polynomial order of space discretization or the number of computational elements. The authors' hybrid strategy generally was more efficient than the pure MPI-based version. The same authors concluded that the best performance could be obtained only with the proper choice of the number of MPI partition and OpenMP threads within a single node. Bassi et al[3] recommended considering the NUMA architectures to avoid the reduction of the memory bandwidth of CPUs. The authors also suggested that placing an MPI process in each NUMA region obtains optimal use of the hardware.

Publications in another group of related work evaluated the performance gains of hybrid MPI/OpenMP implementations that rely on load-balancing techniques. Similar to our implementation, these publications improved performance by decreasing memory synchronization overhead. Vu and Alaghband[14] proposed a hybrid MPI/OpenMP parallel method that enables frequent pattern mining on a cluster containing multiple multicore nodes. The authors employed a multistrategy load-balancing approach to balance the mining workload among all cores. The experimental results showed that the hybrid method improved performance related to the pure MPI approach.

Although a hybrid MPI/OpenMP implementation, in general, outperforms a pure MPI-based version, the best model depends on specific characteristics. Ferretti et al[15] reported that a pure OpenMP-based implementation of the cross-motif search algorithm is extremely inefficient, and cannot scale appropriately. Then, the authors described a hybrid shared-memory version of the algorithm using MPI. The same authors concentrated on the dependence of performance in the hybrid approach on the workload unbalance. A ''slave'' module was responsible for processing all protein pairs received from a ''master'' module. The communication between the master modules used shared-memory OpenMP directives, whereas the communication between master and slave modules employed MPI. Ferretti et al[15] evaluated their algorithm with executions performed on a custom cluster comprised of four servers interconnected with a gigabit ethernet backbone. The authors concluded that their hybrid implementation vastly outperformed the pure OpenMP implementation. In some cases, however, the pure MPI implementation performed better than the hybrid version of the algorithm.

Usually, parallel simulations show an irregular structure and dynamic load patterns. Adaptive message passing interface (AMPI) supports dynamic load balancing, processor virtualization, and fault tolerance for MPI applications. Diener et al[16] investigated how to improve the memory access locality of a hybrid MPI/OpenMP multiphysics simulation application in two different strategies: manually fixing locality (NUMA) issues and by using the AMPI runtime environment. Both strategies showed different trade-offs. AMPI does not require hybrid parallelization to obtain the resource utilization of MPI/OpenMP, but users are demanded to employ AMPI and the use of global variables. The custom C++ memory allocator provided the most portable solution but required significant implementation effort. In both solutions, scaling was much better than executing the pure MPI-based implementation. With this experiment, Diener et al[16] observed the influence of memory locality on load balance, A better locality of reference leads to lower stall time due to memory access, resulting in reasonable use of the resources. The same authors reported that AMPI leads to similar locality gains, achieving better performance and scalability than the simplistic hybrid MPI/OpenMP and pure MPI-based implementations.

Similar to the publications described in this section, the present study evaluated parallel models of methods when running on multicore and manycore architectures. Along the same lines of the investigations conducted in previous publications,[3,13,16] we implemented hybrid approaches to reduce the NUMA effect. For a similar purpose as other investigations,[3,8-13] this study analyzed hybrid configurations to evaluate the impact of message passing operations for a large number of processes. Similar to investigations conducted in previous publications,[14-16] this study evaluated the performance of a hybrid MPI/OpenMP approach concerning load-balancing and synchronization techniques.

3 | THREE 2D EXPLICIT FD METHODS

This section describes implementations of three 2D explicit FD methods for solving PDEs. Consider the bidimensional advection-diffusion equation in the following form:

$$u_t + v_1 u_x + v_2 u_y = d(u_{xx} + u_{yy}), \tag{1}$$

with the appropriate initial and boundary conditions, where u is the variable of interest (eg, temperature for heat transfer and species concentration for mass transfer), v_1 and v_2 are positive velocity constants, d is the diffusivity, and $0 \leq x, y \leq 1$. The Hopmoc method (see the work of Oliveira et al[17] and references therein) carries out steps of the odd-even Hopscotch method,[18] but the Hopmoc method performs its semisteps

along characteristic lines in a semi-Lagrangian scheme based on the modified method of characteristics (MMOC).[19] The Hopmoc method divides the set of stencil points into two subsets during the implementation of the integration step. Then, the method alternately performs two distinct (one explicit and one implicit) updates on each variable during the iterative process. Each update demands an integration semistep.

The 2D Hopmoc method employs the FD operator:

$$\mathcal{L}_h\left(u_{i,j}^n\right) = d\left[\frac{u_{i-1,j}^n - 2u_{i,j}^n + u_{i+1,j}^n}{(\Delta x)^2} + \frac{u_{i,j-1}^n - 2u_{i,j}^n + u_{i,j+1}^n}{(\Delta y)^2}\right]. \tag{2}$$

Thus, both consecutive time semisteps can be written as

$$\overline{u}_{i,j}^{n+\frac{1}{2}} = \overline{\overline{u}}_{i,j}^n + \delta t\left[\theta_{i,j}^n \cdot \mathcal{L}_h\left(\overline{\overline{u}}_{i,j}^n\right) + \theta_{i,j}^{n+1} \cdot \mathcal{L}_h\left(\overline{u}_{i,j}^{n+\frac{1}{2}}\right)\right] \tag{3}$$

and

$$u_{i,j}^{n+1} = \overline{u}_{i,j}^{n+\frac{1}{2}} + \delta t\left[\theta_{i,j}^{n+1} \cdot \mathcal{L}_h\left(\overline{u}_{i,j}^{n+\frac{1}{2}}\right) + \theta_{i,j}^n \cdot \mathcal{L}_h\left(u_{i,j}^{n+1}\right)\right], \tag{4}$$

where $\overline{\overline{u}}_{i,j}^n$ is the variable of interest u evaluated in the previous step, $\theta_{i,j}^n = 1$ (= 0) if $n+i+j$ is odd (even),[17] $(\overline{x}_i^{n+\frac{1}{2}}, \overline{y}_i^{n+\frac{1}{2}}) = (x_i - v_1\delta t, y_j - v_2\delta t)$, $(\overline{\overline{x}}_i, \overline{\overline{y}}_i) = (x_i - 2v_1\delta t, y_j - 2v_2\delta t)$, and $\Delta t = t_{n+1} - t_n$ ($\delta t = \frac{\Delta t}{2} = t_{n+\frac{1}{2}} - t_n$) represents a time (semi-) step when considering a typical FD discretization for Equation (1), for T time steps.

The Hopmoc method performs three main steps for every time step with size Δt.[17] At time step t_n, the method obtains $\overline{\overline{u}}$ for all stencil points $(\overline{\overline{x}}_i, \overline{\overline{y}}_j)$, for $i = 1, \ldots, N$ and $j = 1, \ldots, N$ using an interpolation method. At time semistep $t_{n+\frac{1}{2}}$, the method calculates $\overline{u}_{i,j}^{n+\frac{1}{2}}$ using the explicit (implicit) operator for stencil points for which $n+i+j$ is odd (even). At time semistep t_{n+1}, the method calculates $u_{i,j}^{n+1}$ using the implicit (explicit) operator for stencil points for which $n+i+j$ is odd (even). Because the method calculates neighbor stencil points at the previous time semistep, the implicit approach does not require a linear system to be solved in the same way as the odd-even Hopscotch method performs.

In the present simulations, each thread or process depends only on two neighboring threads, because we limited the velocity in the experiments (ie, $v = 1$). Thus, the method finds the foot of the characteristic line in a neighboring thread, or even in the same thread.

Algorithm 1 outlines the five double loops and corresponding synchronization points at each time step used in the implementation of the Hopmoc method. The loop in lines 4 through 28 describes a time step in the algorithm. The first double loop outlined in line 6 performs the MMOC step. The four double loops delineated in lines 9 through 22 carry out the four time semisteps of the method.

Algorithm 1 2D Hopmoc method

```
1: [...]
2: k ← 0
3: t ← 0
4: while (t ≤ T) do
5:     [...]
6:     An outer and an inner loops that compute the MMOC step
7:     Synchronization point
8:     [...]
9:     An outer and an inner loops that respectively iterate i and j from 1 to N
10:        Compute the first explicit time semi-step if i + j + k is even
11:    Synchronization point
12:    [...]
13:    An outer and an inner loops that respectively iterate i and j from 1 to N
14:        Compute the first implicit time semi-step if i + j + k is odd
15:    Synchronization point
16:    [...]
17:    An outer and an inner loops that respectively iterate i and j from 1 to N
18:        Compute the second explicit time semi-step if i + j + k is even
19:    Synchronization point
20:    [...]
21:    An outer and an inner loops that respectively iterate i and j from 1 to N
22:        Compute the second implicit time semi-step if i + j + k is odd
23:    Synchronization point
24:    [...]
25:    k ← k + 1
26:    t ← k × Δt
27:    [...]
28: end while
```

The code of the five synchronization points outlined in Algorithm 1 depends on the approach used. There is no code for performing the synchronization point when a naïve OpenMP-based approach is used. The OpenMP API uses an implicit barrier synchronization mechanism in a naïve scheme. Furthermore, the OpenMP API uses implicit scheduling and synchronization mechanisms.

When using the OMP-EWS approach, a busy waiting loop performs the synchronization point. This synchronization mechanism is described in detail in Section 4.1. MPI calls carry out the synchronization point in the pure MPI-based approach. Details of this synchronization mechanism are described in Section 4.2. In the hybrid MPI/OMP-EWS version of this implementation, we employ the OMP-EWS approach, described in Section 4.1, within the MPI implementation, reported in Section 4.2. Details of the hybrid MPI/OMP-EWS approach are described in Section 4.3.

The odd-even Hopscotch method is now described. Consider the bidimensional heat conduction equation in the following form:

$$u_t = d(u_{xx} + u_{yy}), \tag{5}$$

with the appropriate initial and boundary conditions. The central FD scheme for Equation (5) with u_i^n as an approximation for $u(x_i, y_j, t_n)$ in the 2D odd-even Hopscotch method[18] is also provided by the FD operator in Equation (2). Thus, Equations (3) and (4) are the consecutive time semisteps of the method. Therefore, the 2D odd-even Hopscotch method is similar to Algorithm 1, except for the use of the MMOC method.

Consider the bidimensional Laplace equation in the form $d^2(u_{xx} + u_{yy}) = 0$ where the variable of interest u here is, for example, the voltage or the electric potential in a flat metal sheet. The Jacobi point iteration is a relaxation method because the approach replaces values at a stencil point by the average of the values around it. In particular, relaxation methods iterate toward convergence, employing a nearest-neighbor updating scheme. Our stencil for the Jacobi point iteration method consists of five points arranged in a cross.

Given an initial guess for the variable of interest (ie, $u_{i,j}^0$), and when approximating the second derivatives in both dimensions using a difference equation (and using the values obtained in the previous iteration u^k), one obtains

$$u_{i,j}^{k+1} = \left(u_{i+1,j}^k + u_{i,j+1}^k + u_{i-1,j}^k + u_{i,j-1}^k\right)/4. \tag{6}$$

The variable of interest at any stencil point at an iteration $k > 0$ is the average of the variables of interest at neighboring stencil points obtained in the previous iteration. The implementation updates $u_{i,j}^{k+1}$ only to interior points and leaves boundary values fixed.

Algorithm 2 outlines the double loop and the synchronization point used in the implementation of the Jacobi point iteration method. Lines 3 through 11 iterate a double inner loop (outlined in line 5) that performs the Jacobi point iteration in line 6. As previously mentioned, the code of the synchronization point depends on the approach used. In Sections 4.1, 4.2, and 4.3, the OMP-EWS, MPI-based, and hybrid approaches are described in detail, respectively.

Algorithm 2 Jacobi point iteration method for the 2D Laplace equation

```
1:  [...]
2:  k ← 0
3:  while (k ≤ MaxIterNr) do
4:      [...]
5:      An outer and an inner loops that respectively iterate i and j from 1 to N
6:          Compute Jacobi point iteration using equation 6
7:      Synchronization point
8:      [...]
9:      k ← k + 1
10:     [...]
11: end while
```

4 | OPENMP AND MPI APPROACHES FOR 2D EXPLICIT FD METHODS

In this section, the OpenMP and MPI approaches evaluated in this study are described. Specifically, several configurations of the implementations of three 2D explicit FD methods (Hopmoc, odd-even Hopscotch, and Jacobi point iteration methods) for solving PDEs in executions performed on two machines are outlined. The first machine featured an Intel® Xeon® Phi™ Knights Landing accelerator CPU 7250 at 1.4 GHz, comprised of 68 cores, with four threads per core, cluster mode SNC-4, and flat memory mode. We refer to this machine as the KNL machine. The second machine contained two sockets of an Intel® Xeon® Platinum 8160 CPU at 2.1 GHz where each socket contains 24 cores, with two threads per core. We refer to this machine as the SKL machine. In particular, we used the icpc (Intel® C++ compiler) and mpiicpc (MPI for C++) version 2018, with the optimization flag -O3. Additionally, in the Intel® Xeon® Scalable Processor, the -qopt-zmm-usage=high flag was set.

In the three versions of the implementation (OMP-EWS, pure MPI-based, and hybrid MPI/OpenMP approaches), the elapsed times are due to the number of double loops used in the three 2D explicit FD methods. The 2D Hopmoc and odd-even Hopscotch methods use five and four synchronization points, respectively, whereas the 2D Jacobi point iteration method uses only one synchronization point. The reason is that the

2D Hopmoc and odd-even Hopscotch methods employ five and four double loops in a time step, respectively, whereas the Jacobi point iteration method uses only one double loop. Consequently, the Hopmoc algorithm takes longer running times than the 2D odd-even Hopscotch method. In the same vein, the 2D odd-even Hopscotch method takes longer execution times than the Jacobi point iteration algorithm.

In Sections 4.1 and 4.2, the pure OMP-EWS and MPI-based approaches are described, respectively. In Section 4.3, the MPI/OMP-EWS approach is discussed.

4.1 | A 2D OpenMP-based EWS approach with an explicit synchronization strategy

This 2D OpenMP-based version of the three explicit FD methods for solving PDEs determines a static array of Booleans that designates the unknowns. Additionally, the implementation copes with load imbalance between threads by explicitly subdividing the array into the team of threads. Therefore, these approaches carry out thread scheduling only at the beginning of the execution. Thus, the explicit parallel methods for solving PDEs do not use the OpenMP *parallel for* directive because each thread already has its data. A thread sets (releases) its corresponding entry in this array to inform its two adjacent threads that the data cannot (can) be accessed. We refer to this implementation as the OMP-EWS approach.

Algorithm 3 A fragment of pseudocode that describes the explicit synchronization mechanism employed in the OMP-EWS approach

```
1: tid ← omp_get_thread_num()
2: nt ← omp_get_num_threads()
3: size ← (n − 2)/nt
4: remainder ← (n − 2)%nt
5: localStart ← tid · size + 1
6: localEnd ← localStart · size − 1
7: if (tid + 1 ≤ remainder) then
8:     localStart ← localStart + tid
9:     localEnd ← localEnd + tid + 1
10: else if
11:     thenlocalStart ← localStart + remainder
12:     localEnd ← localEnd + remainder
13: end if
14: nid ← tid + 1
15: boolean lock[nt+2]
16: lock[nid] ← false
17: #pragma omp master
18: {
19:   lock[0] ← false
20:   lock[nt+1] ← false
21: }
22: #pragma omp flush (lock) // update lock array
23: [...]
24: while (time ≤ FinalTime) do
25:     [...]
26:     // lock mechanism: inform the adjacent threads that this thread is performing a task in the shared memory
27:     lock[nid] ← true
28:     for (j ← 1; i ≤ n; i ← i + 1) do
29:         for (i ← localStart; i ≤ localEnd; i ← i + 1) do
30:             // do some work
31:         end for
32:     end for
33:     [...]
34:     // release the shared memory to adjacent threads
35:     lock[nid] ← false
36:     // verify if the shared memory is locked awaits until it is released
37:     while (lock[nid + 1] ∨ lock[nid − 1]) do
38:     end while
39:     [...]
40: end while
```

Algorithm 3 shows a fragment of pseudocode that outlines how we synchronize adjacent threads in this approach. Line 15 in the pseudocode describes how we define the array of locks. As the first (last) thread has no neighbor to the left (right) side, the first (last) entry of the lock array is unset. When the mesh size is not a power of the number of threads, we set the variables localStart and localEnd of all threads so that each thread receives only one more row of the matrix (see lines 7–13 in Algorithm 3). In particular, the implementation is a thread-safe code. Algorithm 3 improves our 1D version of the OMP-EWS approach.[2]

Algorithm 3 also describes the explicit synchronization mechanism employed in the 2D explicit FD methods evaluated in this study and how we synchronize adjacent threads. The pseudocode describes how we replace the OpenMP synchronization barrier mechanism defining a range from variables localStart to localEnd for each thread in the team. Lines 28 through 32 in Algorithm 3 represent the synchronization points described in Algorithms 1 and 2. The busy waiting loop in lines 37 and 38 in Algorithm 3 replaces the implicit barrier synchronization mechanism employed in the naïve OpenMP approach.

Some spin time may be desirable instead of increasing the thread context switches. Long spin time, however, can reduce productive work. The OpenMP barrier directive recognizes a synchronization point so that threads in a parallel code fragment will not run after the OpenMP synchronization barrier until all other threads in the team terminate their tasks. Thus, one can use the no-wait clause, and include an OpenMP barrier directive outside the loop. Even with these directives, however, all threads in the team synchronize at the same location.[2]

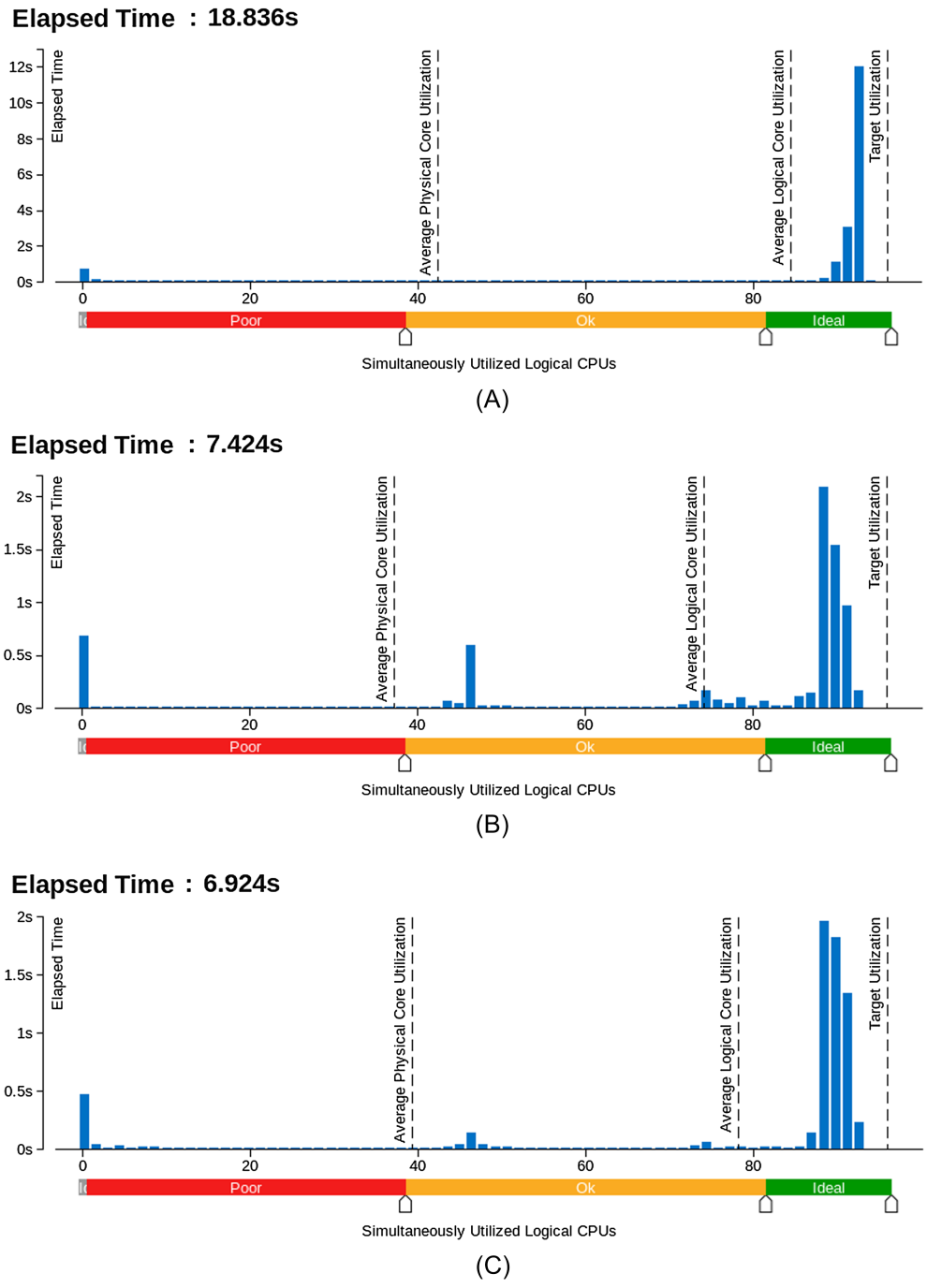

FIGURE 1 CPU usage histograms generated in executions of the OMP-EWS approach of three 2D explicit numerical methods. A, 2D Hopmoc method; B, 2D dd-even Hopscotch method; C, 2D Jacobi point iteration method

Figure 1 illustrates histograms of effective CPU utilization for the OMP-EWS approach of the three 2D explicit FD methods. We extracted histograms from the Intel® VTune™ Amplifier performance profiler in executions performed on the SKL machine. The histograms display percentages of the wall time required when a specific number of logical and physical cores were running simultaneously. The performance profiler adds spin and overhead times to the idle CPU utilization value. The figure shows that the three OMP-EWS approaches of the 2D explicit FD methods used an ideal number of threads simultaneously when performed on the SKL machine. In the experiments, we set $N = 10^4$ and $T = 100\,000$.

Figure 1 shows that the OMP-EWS approach of the three explicit FD methods effectively uses from 37 to 42 (out of 48) physical cores, on average (from 78% to 88%). The figure also shows that the OMP-EWS approach of the three explicit FD methods effectively uses from 74 to 85 (out of 96, which is the target utilization) logical cores, on average (from 77% to 85%). Thus, the OMP-EWS version of the three explicit FD methods shows effective physical and logical core utilization. In particular, a naïve OpenMP-based implementation of the three explicit FD methods uses approximately ten physical and 20 logical cores (21%). Additionally, the performance profiler showed that the OMP-EWS version of the three explicit algorithms took from 90% to 96% (from 4% to 10%) of the required time in parallel (serial) regions.

4.2 | Pure MPI-based explicit methods for solving PDEs

A parallel algorithm should balance data distribution. A parallel algorithm should allocate, as much as possible, the same number of entries to each process. Additionally, a parallel algorithm should minimize communication. Our parallel algorithm considers the data in a 1D block partition. Thus, each process communicates with two neighbors.

MPI provides a selection of various communication modes that permit the developer to manage the choice of the communication protocol.[1] We employed nonblocking communication in this study using the MPI_Isend() and MPI_Irecv() routines. The routines do not block the communication and return immediately even if the communication has not terminated yet. The instructions avoid deadlocks and do not rely on buffering. In contrast, this is not the case with the standard MPI communication mode, which, to circumvent a deadlock, depends on whether the operating system has buffers. In the standard MPI communication mode, MPI decides whether outgoing messages will be buffered.[1] Thus, the same code can work on a machine, but a deadlock can occur on other machines.[1] In particular, this implementation is iteration safe because it does not need to verify whether the process has an odd or even rank.

Algorithm 4 shows a fragment of pseudocode that describes how we programmed this MPI-based version of the 2D explicit FD methods for solving PDEs. The algorithm uniformly balances the load distribution between processes. The algorithm distributes the load at the beginning of the code.

Algorithm 4 sends line 0 to the northern neighbor and line $n + 1$ to the southern neighbor. Similarly, the algorithm receives in line 1 the data from the northern neighbor and in line n the data from the southern neighbor. Then, the algorithm waits for these four operations before performing the next step of the numeric method. For example, after executing the MMOC routine, the 2D Hopmoc algorithm executes the code referring to the MPI routines and then calculates the explicit operator of the first semistep of the Hopmoc method. Then, the algorithm runs the MPI code, computes the implicit operator of the first semistep, and so on. We placed the MPI code in the same locations of the busy waiting loop of the OMP-EWS approach.

Figure 2 illustrates the times computed in MPI routines concerning the execution of three 2D explicit numerical methods. We extracted the plots from the Intel® Trace Analyser and Collector (ITAC) performance profiler in executions performed on the SKL machine. The figure represents ratios of all MPI calls concerning the rest of the codes in the applications. The same figure lists the most active MPI functions from all MPI calls in the applications. In the experiments, we set $N = 10^4$ and $T = 10\,000$.

The parallelism in this context is that the execution uses more than one MPI process. The pure MPI approach creates an MPI process on each logical core of the machine. Thus, the pure MPI approach employs 96 (272) MPI processes on the SKL (KNL) machine. There is no time spent in OpenMP function calls. From an MPI process, a code is serial when it is not within OpenMP threads. Thus, Figures 2A and B show that the 2D Hopmoc and Hopscotch methods used approximately 86% in serial codes because the approaches employ five and four double loops (and the same number of synchronization points) in a time step, respectively. In contrast, Figure 2C shows that the 2D Jacobi point iteration method required approximately 97% of the time in a serial code. The reason is that the code employs only one double loop and one synchronization point. For the same reason, Figures 2A and 2B (2C) show that the 2D Hopmoc and odd-even Hopscotch (Jacobi point iteration) methods used approximately 14% (3%) of the running time required in MPI calls. Thus, the 2D Hopmoc and odd-even Hopscotch methods compute longer times in the MPI_Isend, MPI_Irecv, and MPI_Wait calls than the Jacobi point iteration method does. The reason is that the two former methods employ a larger number of synchronization points than the latter does. This feature is due to how we implemented the three explicit FD methods, that is, these implementations call MPI routines between double loops. Thus, the implementation with more loops and more synchronization points (the Hopmoc method) takes shorter times in serial regions and longer times with MPI calls than the other methods. On average, among the three explicit FD methods, each synchronization point required appropriately 3.8% of the execution time in MPI calls.

The implementation computed most of the time required by MPI calls in the MPI_Wait routine. The reason is that we employed nonblocking communication using MPI calls. In particular, the MPI_Wait routine ensures that the execution continues only after the MPI_recv routine has completed its task.

Algorithm 4 A fragment of pseudocode that describes an MPI-based approach of an explicit method for solving PDEs

```
1: pid ← MPI_Comm_Rank()
2: np ← MPI_Comm_Size()
3: size ← (n − 2)/np
4: remainder ← (n − 2)%np
5: localStart ← pid · size + 1
6: localEnd ← localStart · size − 1
7: if (pid + 1 ≤ resto) then
8:     localStart ← localStart + pid
9:     localEnd ← localEnd + pid + 1
10: else if
11:     thenlocalStart ← localStart + remainder
12:     localEnd ← localEnd + remainder
13: end if
14: nN ← finalLocal − inicioLocal + 1
15: [...]
16: while (time ≤ FinalTime) do
17:     [...]
18:     for (j ← 1; i ≤ n; i ← i + 1) do
19:         for (i ← localStart; i ≤ localEnd; i ← i + 1) do
20:             // do some work
21:         end for
22:     end for
23:     MPI_Isend(&U[line 1], n-2, MPI_DOUBLE, north, 1, MPI_COMM_WORLD, &request[0])
24:     MPI_Isend(&U[line nN], n-2, MPI_DOUBLE, south, 1, MPI_COMM_WORLD, &request[1])
25:     MPI_Irecv (&U[line 0], n-2, MPI_DOUBLE, north, 1, MPI_COMM_WORLD, &request[2])
26:     MPI_Irecv (&U[line nN + 1], n-2, MPI_DOUBLE, south, 1, MPI_COMM_WORLD, &request[3])
27:     MPI_Wait(&request[2] , &status)
28:     MPI_Wait(&request[3] , &status)
29:     [...]
30: end while
```

The MPI_Comm_rank routine determines the rank of the calling process in the communicator. The MPI_Finalize routine terminates an MPI execution and appears only once at the end of the code. These routines spend a negligible amount of time because the application calls them only once.

4.3 | Parallel 2D explicit FD methods for solving PDEs based on an MPI/OMP-EWS approach

The MPI and OpenMP approaches rely on a thread or process to wait for its two neighbors to reach the same point in the code at the end of a loop. In the OMP-EWS approach, the while loop performs the task using a busy-waiting technique. In contrast, the MPI approach carries out the task using message exchange. The hybrid version uses both schemes.

The hybrid version of the three explicit FD methods employs the OMP-EWS version described in Section 4.1 as intrasocket parallelism and the MPI-based implementation described in Section 4.2 as intersocket parallelism in executions performed on the SKL machine. In executions performed on the KNL machine, the hybrid version of the three explicit FD methods employs the MPI/OMP-EWS core (L2) approach as intercore and intracore (intertile and intratile) parallelism. The threads carry out all work in these implementations, that is, OpenMP calls create threads, distribute workload (ie, work-sharing), etc. Thus, most of the time required in the execution is performed by OpenMP functions. Thus, the percentage of time required in MPI calls is smaller in the hybrid version (see Figure 3) than in the pure MPI-based approach (see Figure 2).

We extracted images from the ITAC performance profiler in executions of the hybrid MPI/OpenMP approach performed on the SKL machine. Figure 3 illustrates the time required in MPI and OpenMP calls regarding the execution of three numerical methods. In this context, ITAC considers the time required in the MPI calls of the two MPI processes of this experiment. As each MPI process contains 48 threads (resulting in 96 threads computing OpenMP functions), the performance profiler also considers the time required in the OpenMP function calls. Additionally, ITAC considers the rest of the code as serial. In the experiments, we set $N = 10^4$ and $T = 200\,000$.

Figures 3A and 3B (3C) show that the 2D Hopmoc and odd-even Hopscotch (Jacobi point iteration) methods use approximately 4% (1%) of the computational time in MPI calls. The hybrid MPI/OpenMP approach uses two MPI processes to perform intersocket parallelism in executions performed on the SKL machine. Similar to the pure MPI-based versions, the hybrid approaches computed most of the time required by MPI

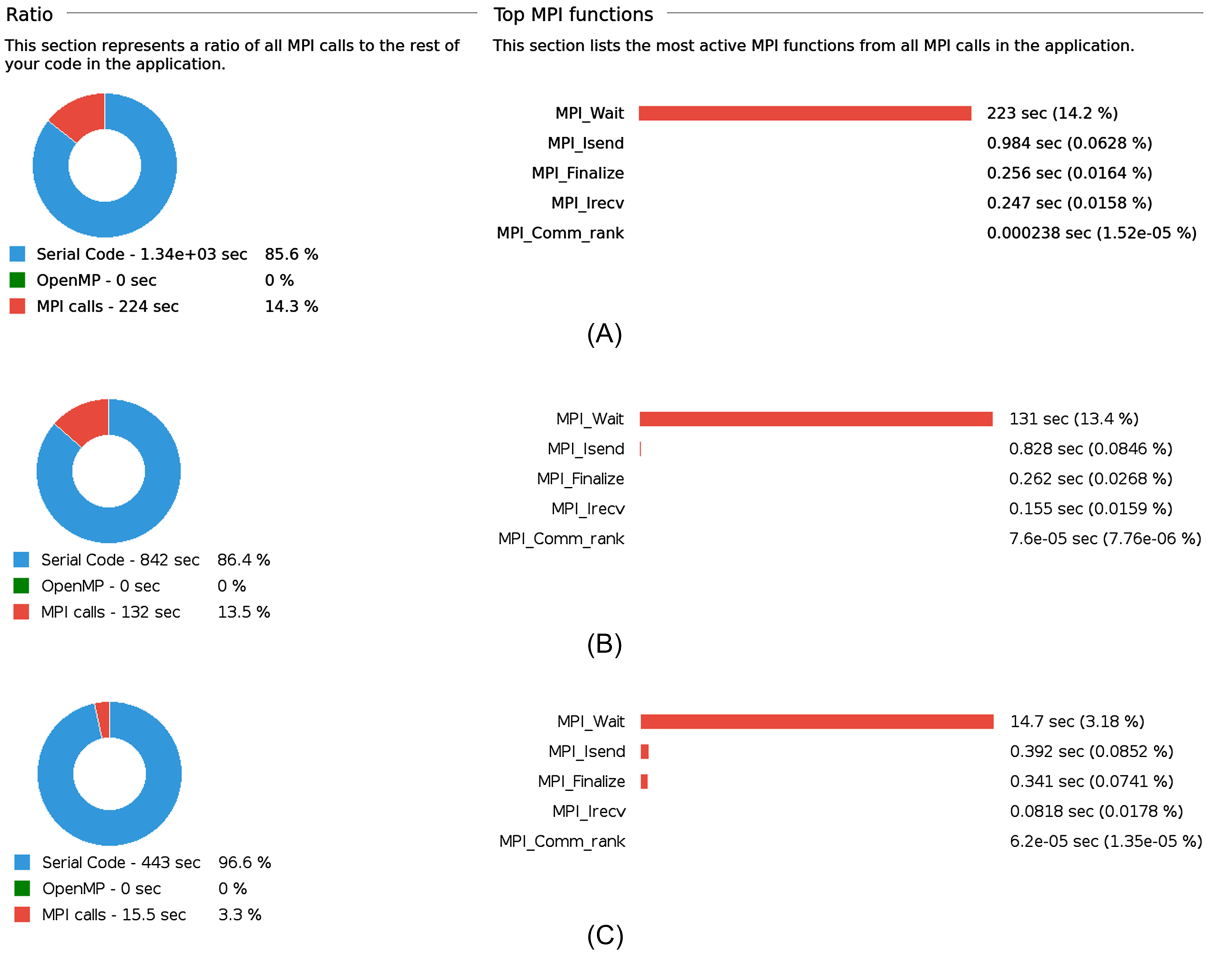

FIGURE 2 CPU usage plots generated in executions of the pure MPI-based implementation of three 2D explicit numerical methods. A, 2D Hopmoc method; B, 2D odd-even Hopscotch method; C, 2D Jacobi point iteration method

calls in the wait routine. The reason is that we employed nonblocking communication using MPI routines. In particular, the pure MPI-based approaches compute a longer time in MPI calls (see Figures 2A, 2B, and 2C) than the MPI/OMP-EWS approaches do, because the latter uses only intersocket MPI communication. Consequently, the hybrid MPI/OpenMP implementation takes most of the required time in OpenMP functions that perform intrasocket parallelism (among 48 threads on each socket).

5 | RESULTS AND ANALYSIS

In this section, we compare the performance of the pure MPI implementation and the improved OpenMP-based implementation with hybrid MPI/OpenMP approaches in three 2D explicit FD methods for solving PDEs on shared-memory multicore and manycore systems. Specifically, in this section, we discuss the results of several configurations of the implementations of the 2D Hopmoc, odd-even Hopscotch, and Jacobi point iteration methods running on the KNL and SKL machines. We evaluated each configuration with compact and scatter thread binding policies.

- Cabral et al[2] proposed a 1D scheme consisting of synchronizing only adjacent threads. We evaluate the 2D version of the approach. The improved OpenMP approach employs an EWS strategy to reduce the OpenMP scheduling time. The OpenMP-based 2D Hopmoc method partitions among threads an array that represents the computational mesh instead of allowing the OpenMP API to perform thread scheduling implicitly. The OMP-EWS scheme avoids barriers using an explicit synchronization mechanism. Thus, a thread waits only for two adjacent threads in the OMP-EWS approach. The approach reduces the OpenMP spin time required in runs performed on shared-memory multicore and manycore architectures.
- A pure MPI-based approach.
- In executions performed on the SKL machine, the hybrid MPI/OMP-EWS version of the three explicit FD methods employs two MPI processes (one process per socket), setting the number of threads from 1 to 48 for each MPI process. We used the next-neighbor halo exchange communication pattern.
- In executions performed on the KNL machine, we evaluated two hybrid MPI/OMP-EWS versions:

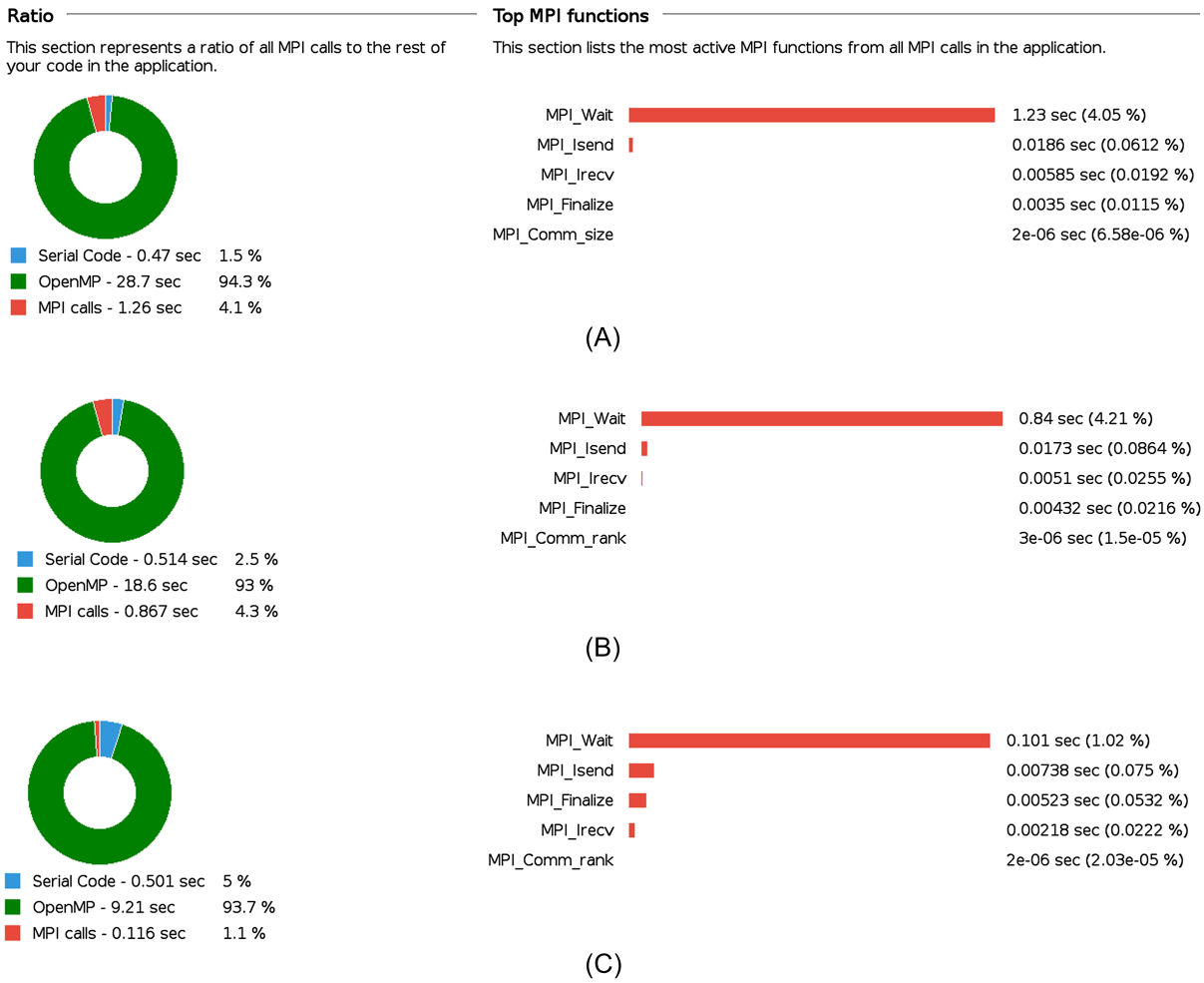

FIGURE 3 CPU usage plots generated in executions of the MPI/OMP-EWS approach of three 2D explicit numerical methods. A, 2D Hopmoc method; B, 2D odd-even Hopscotch method; C, 2D Jacobi point iteration method

1. Each single physical CPU core has an MPI process with four threads. Thus, each process (and its four threads) has dedicated level 1 cache memory. Therefore, the compact and scatter thread binding policies have distinct results. We varied the number of processes from 1 to 68 in the approach. We refer to this version as the MPI/OMP-EWS core approach. Thus, we executed this hybrid MPI/OMP-EWS version in conjunction with scatter and compact thread binding policies.
2. Each tile was allocated to an MPI process. A tile comprises two physical CPU cores (thus, a process and its eight threads) that share the same level 2 cache memory (containing 1 MB on the machine used in this study). We varied the number of processes from 1 to 34, with eight threads per process. We refer to this version as the MPI/OMP-EWS L2 approach. This hybrid MPI/OMP-EWS approach presented the same results when applied with scatter and compact thread binding policies.

In this section, we show the times computed for the MPI routines, and the time required in OpenMP calls concerning the execution of the implementations of the numerical methods evaluated in this study. Specifically, in this section, we show the results of eight configurations of the implementation of the 2D Hopmoc method applied to the advection-diffusion Equation (1) for a Gaussian pulse with amplitude 1.0, the odd-even Hopscotch method applied to the heat conduction Equation (5), and the Jacobi point iteration method applied to the 2D Laplace equation.

Concerning the Hopmoc and odd-even Hopscotch methods, the runs used 10^6 stencil points and $T = 10^5$. The experiments ensured convergence as the 2D Hopmoc method is convergent,[20] and because the odd-even Hopscotch method shows unconditional numerical stability.[21]

We did not execute the Jacobi point iteration method until a steady-state phase was reached because we defined a fixed number of iterations (100 000) to analyze the behavior of the parallel approaches. DuChateau and Zachmann[22] provef the convergence of the method (see chapter 11 of their work). In particular, the matrix representation of the algorithm must have a spectral radius smaller than 1. The Jacobi point iteration method always converges when applied to the Laplace equation. Thus, the convergence was guaranteed because we used a uniform mesh. Other methods converge faster than the Jacobi point iteration method does. Nevertheless, the Jacobi point iteration method always converges, and one can easily implement a parallel version of the algorithm. In Sections 5.1 and 5.2, we report the results of several implementations of three numerical methods in experiments performed on the KNL and SKL machines, respectively.

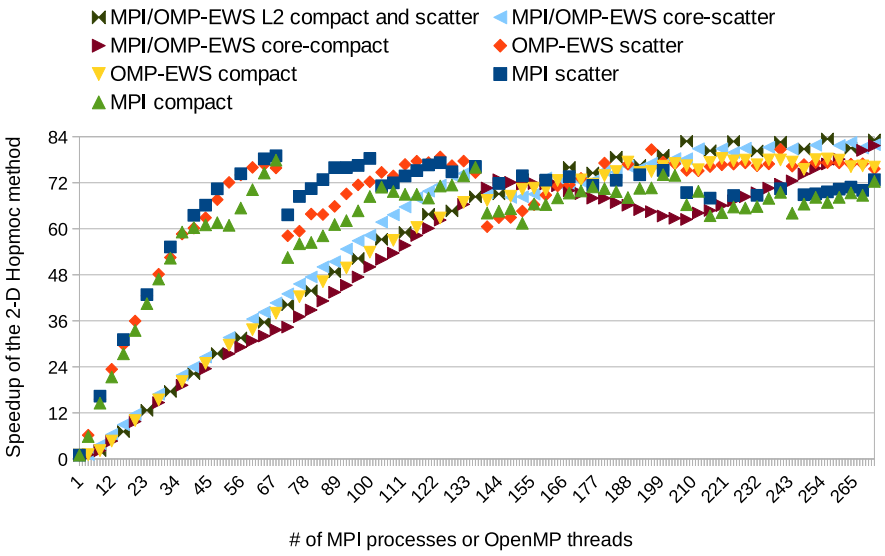

FIGURE 4 Speedups of the OMP-EWS, pure MPI-based, and four MPI/OMP-EWS approaches of the 2D Hopmoc method with scatter and compact thread binding policies in runs performed on a machine featuring an Intel® Xeon® Phi™ Knights Landing accelerator

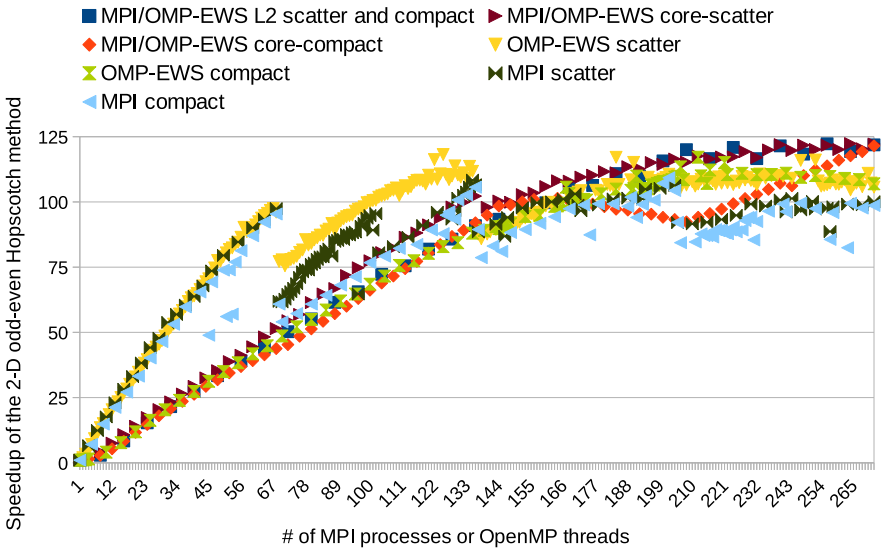

FIGURE 5 Speedups of the OMP-EWS, pure MPI-based, and four MPI/OMP-EWS approaches of the 2D odd-even Hopscotch method with scatter and compact thread binding policies in runs performed on a machine featuring an Intel® Xeon® Phi™ Knights Landing accelerator

5.1 | Experiments on the KNL machine

Figures 4 and 5 illustrate the results of eight configurations of the implementation of the 2D Hopmoc and odd-even Hopscotch methods in runs performed on the KNL machine, respectively. The figures show that the OMP-EWS approach delivered better speedups than the pure MPI approach did. In general, the former yielded shorter execution times to synchronize OpenMP threads than the latter did to synchronize MPI processes.

Figures 4 and 5 show that the OMP-EWS approach used in conjunction with the scatter thread binding policy delivered slightly better speedups than the same approach in tandem with the compact thread binding policy did. The reason is that the former achieved better load balancing than the latter. Nevertheless, the OMP-EWS approach with the compact thread binding policy yielded more regular results than the use of the scatter thread binding policy. The reason is that the former employed hyperthreading even when only a few threads were used, whereas the latter employed hyperthreading after 68 threads were used in the machine. In particular, the scatter thread binding policy employed (two threads per) physical cores when the application used up to 68 (136) threads in the machine. Thus, when up to two threads were used per core (ie, up to 136 threads), the version with scatter thread binding policy also yielded better results than the use of the compact thread binding policy.

Figures 4 and 5 also show that the hybrid approaches achieved better speedups than the pure OMP-EWS approach did. The hybrid approaches yielded better cache locality due to having an MPI process allocated to a structure that shared the same level of cache memory. In particular, the hybrid intertile approach had eight threads in the same level 2 cache memory, whereas the hybrid intercore approach had four threads in the same level 1 cache memory in runs performed on the KNL machine.

Figures 4 and 5 also show that the hybrid approaches evaluated in this study yielded better results than the pure MPI-based version did when used to implement parallel versions of the 2D Hopmoc and odd-even Hopscotch methods. The hybrid approaches used fewer processes than the pure MPI-based version did. Furthermore, the hybrid versions employed only MPI processes for interdomain communication. Thus, the hybrid

approaches delivered less process communication than the pure MPI-based version did, recalling that the implementations of the 2D Hopmoc and odd-even Hopscotch methods employed five and four synchronization points at each time step, respectively.

Figures 4 and 5 show that, in general, the MPI/OMP-EWS core-scatter approach returned better speedups than the MPI/OMP-EWS core-compact version did. In this case, the compact thread binding policy yielded more of a NUMA effect than the scatter thread binding policy did because the latter had better load balancing than the former. The MPI/OMP-EWS core approach with the compact thread binding policy yielded different results from the other versions evaluated when from 136 to 204 threads were used because of the loading imbalance caused by how the operating system distributed MPI processes in adjacent cores (ie, cores that shared a large number of resources).

Figures 4 and 5 also show that the MPI/OMP-EWS L2 approach used in conjunction with the scatter or compact thread binding policy yielded slightly better speedups than the MPI/OMP core-scatter approach did. A large domain, such as used in the MPI/OMP-EWS L2 approach, significantly attenuated the NUMA effect when the implementation employed several synchronization points. The reason is that a tile comprises two physical CPU cores that share the same level 2 cache memory. Therefore, the MPI/OMP-EWS L2 approach yielded a high cache hit rate. Specifically, the figures show that the hybrid approaches evaluated yielded similar results when 272 threads were used because the strategies occupied all the cores (and their threads) in this case. The MPI/OMP-EWS L2 and MPI/OMP core-scatter approaches provided regular results because the operating system allocated the MPI processes in domains where threads share a level of cache memory. Four (eight) OpenMP threads of a MPI process shared a level 1 (2) cache memory in the MPI/OMP-EWS core (L2) approach.

The results yielded by the MPI/OMP-EWS L2 approach were similar to the results obtained by the MPI/OMP-EWS core version with the compact thread binding policy when up to 136 threads were used. In contrast, the results yielded by the MPI/OMP-EWS L2 approach were similar to the results obtained by the MPI/OMP-EWS core version with the scatter thread binding policy when more than 136 threads were used. The reasons for this behavior were the load imbalance and the NUMA effect. Furthermore, these results were related to the way binding policies distribute the MPI processes. The NUMA effect occurs depending on how the application distributes the MPI processes. Because the KNL machine features multichannel dynamic random-access memory, the chance of ocurring the NUMA effect increases.

The previous results apply when the code employs four (odd-even Hopscotch method) or five (Hopmoc method) double loops and synchronization points in a time step. The results differ when the code uses only one double loop and synchronization point in an iteration.

Figure 6 illustrates the results of eight configurations of the implementation of the 2D Jacobi point iteration method in runs performed on the KNL machine. Although the OMP-EWS approach of the 2D Hopmoc and odd-even Hopscotch methods yielded better speedups than the pure MPI-based approach did in experiments performed on the KNL machine (see Figures 4 and 5), the pure MPI-based approach of the Jacobi point iteration method achieved better results than the OMP-EWS strategy did in executions carried out on the same machine. Furthermore, Figure 6 shows that the OMP-EWS approach does not compare favorably with the other versions when the implementations of the method employ only one synchronization point. The busy waiting synchronization used in the OMP-EWS approach reads an array of locks. When reading an entry (of the array) in a thread lying in a different domain, the synchronization scheme employed in the OMP-EWS approach yielded a considerable NUMA effect. Furthermore, the small executable file generated by the implementation of the Jacobi point iteration method benefited from the use of the MPI model.

Figure 6 also shows that the hybrid approaches achieved better results than the OMP-EWS version did. The hybrid approaches avoided the NUMA effect. This result occured because the hybrid approaches used MPI for interdomain communication. Thus, a thread rarely read an entry (in the array of locks) that lay in a different thread. Nevertheless, the pure MPI-based approach yielded less of a NUMA effect than the hybrid approaches did. Consequently, Figure 6 shows that the pure MPI-based approach delivered better results than the other versions did. Specifically, the pure MPI-based approach with the scatter process binding policy yielded, in general, better speedups (with a speedup of 104x)

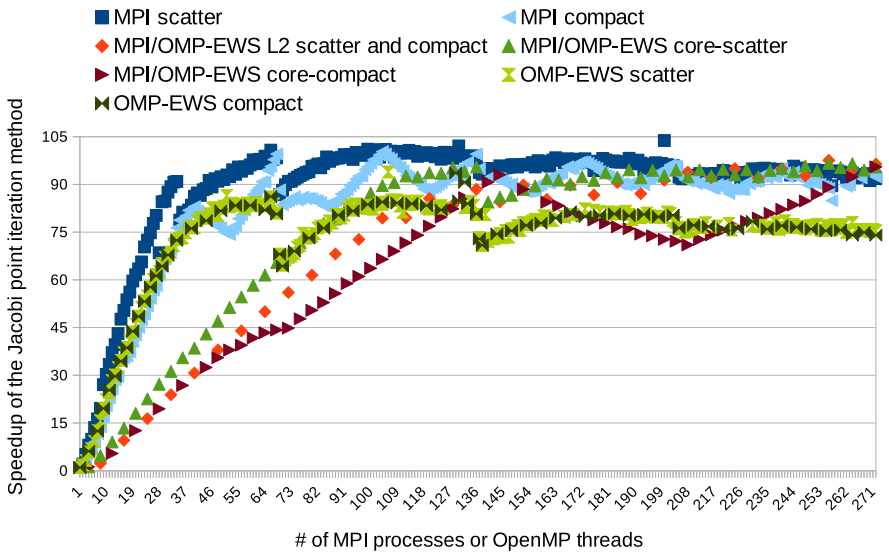

FIGURE 6 Speedups of the OMP-EWS, pure MPI-based, and four MPI/OMP-EWS approaches of the 2D Jacobi point iteration method with scatter and compact thread binding policies in runs performed on a machine featuring an Intel® Xeon® Phi™ Knights Landing accelerator

than the hybrid versions did when considering the runs used up to 204 threads. The pure MPI-based approach outperformed the other strategies because the Jacobi point iteration method used only one double loop in an iteration (whereas the 2D Hopmoc (odd-even Hopscotch) method employed five (four) double loops in a time step). Consequently, the Jacobi point iteration method presented a smaller executable file as the 2D Hopmoc and odd-even Hopscotch methods did. Therefore, the Jacobi point iteration method required a smaller amount of level 1 instruction cache memory than the two other algorithms did. This feature favored the use of the MPI standard. The reason is that researchers designed the model for these cases.

Figure 6 shows that the pure MPI-based implementation used in conjunction with the scatter process binding policy yielded, in general, better speedups than the same approach in tandem with the compact process bind policy did when up to 204 processes were used. The scatter process binding policy employed up to three processes per physical core when the application used up to 204 processes in the machine. Furthermore, both process binding policies achieved similar results when more than 204 threads were used. In this case, as the approaches occupied all the cores (and their threads), differences in load balancing were reduced.

The MPI/OMP-EWS core approach allocated one core per process, whereas the MPI/OMP-EWS L2 version allocated one tile per process (ie, the version allocated one process every two cores) (in particular, a process occupied more space than threads). Thus, the MPI/OMP-EWS core approach required more space than the MPI/OMP-EWS L2 version because the latter involved a larger domain than the former did.

In executions using from 205 to 272 processes, the pure MPI-based, MPI/OMP-EWS L2 and core-scatter approaches yielded similar results. Additionally, the pure MPI-based and hybrid versions achieved similar speedups when the maximum number of threads of the machine was used. The reason is that the strategies involved all the cores (and their threads) in the experiment. These cases attenuated differences in the NUMA effect and load balancing among the various versions evaluated in this study.

The pure MPI-based and hybrid approaches with the compact process binding policy did not present regular results because of the NUMA effect (see Figures 4, 5, and 6). These approaches return great differences in speedups when executed with the number of threads or processes set as multiples of the cores contained in the machine and the next number of threads or processes. The reason is that this KNL machine used sub-NUMA cluster SNC-4 mode of cache operation.

5.2 | Experiments on the SKL machine

Figures 7, 8, and 9 show the results of seven implementations of the 2D Hopmoc, odd-even Hopscotch, and Jacobi point iteration methods in runs performed on the SKL machine, respectively. The figures illustrate the results of the pure MPI-based and OMP-EWS versions, naïve OpenMP, and the MPI/OMP-EWS approach of the three explicit FD methods with scatter and compact thread binding policies. The figures show that the naïve OpenMP-based implementations of the three numerical methods did not provide competitive results when compared with the results yielded by the other versions evaluated in this study.

Figures 7, 8, and 9 reveal that the pure MPI-based approach was more efficient than the OMP-EWS approach in runs performed on the SKL machine when considering the three explicit FD methods evaluated in this study. The pure MPI approach was more efficient in synchronizing the processes than the OMP-EWS version in synchronizing threads between the two sockets contained in this machine. The figures also reveal that the fewer the number of synchronization points in the code, the more the pure MPI approach yielded better results than the OMP-EWS approach. In addition to the number of synchronization points employed in the code, the code size influenced the runtime. A small executable file generated high instruction cache hit rates in the level 1 cache in the memory hierarchy, benefiting the MPI version.

Figure 7 shows that the MPI scatter approach of the 2D Hopmoc method yielded remarkably better results than the other versions of the method did in executions that used only one socket. An application that employs the scatter thread binding policy uses physical cores before

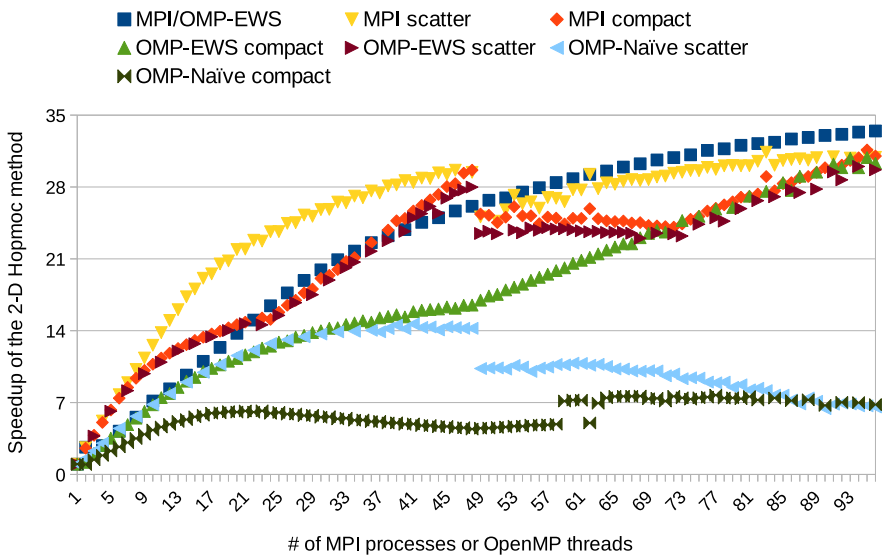

FIGURE 7 Speedups of the OMP-EWS approach, pure MPI based, and naïve versions of the 2D Hopmoc method with scatter and compact thread binding policies in runs performed on a machine featuring an Intel® Xeon® Scalable Processor

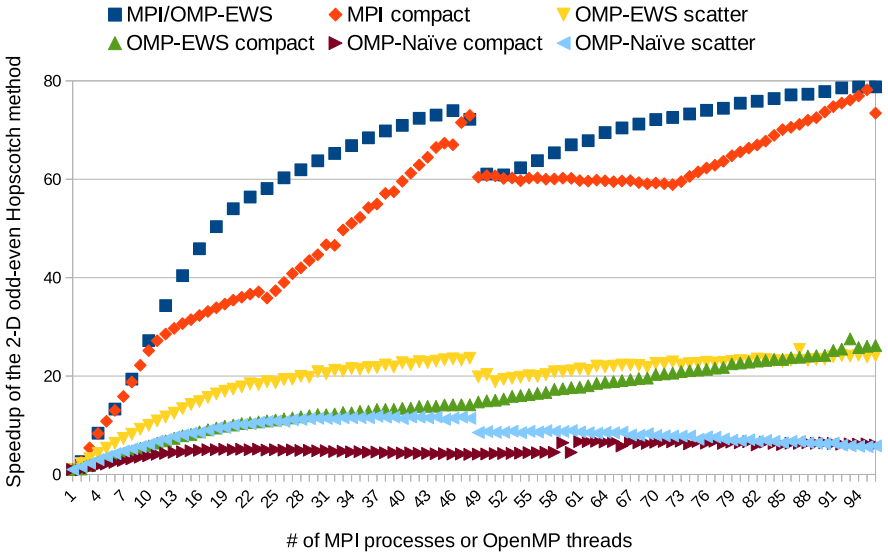

FIGURE 8 Speedups of the pure MPI-based implementation, the OMP-EWS, MPI/OMP-EWS, and naïve approaches of the odd-even Hopscotch method with scatter and compact thread binding policies in runs performed on a machine featuring an Intel® Xeon® Scalable Processor

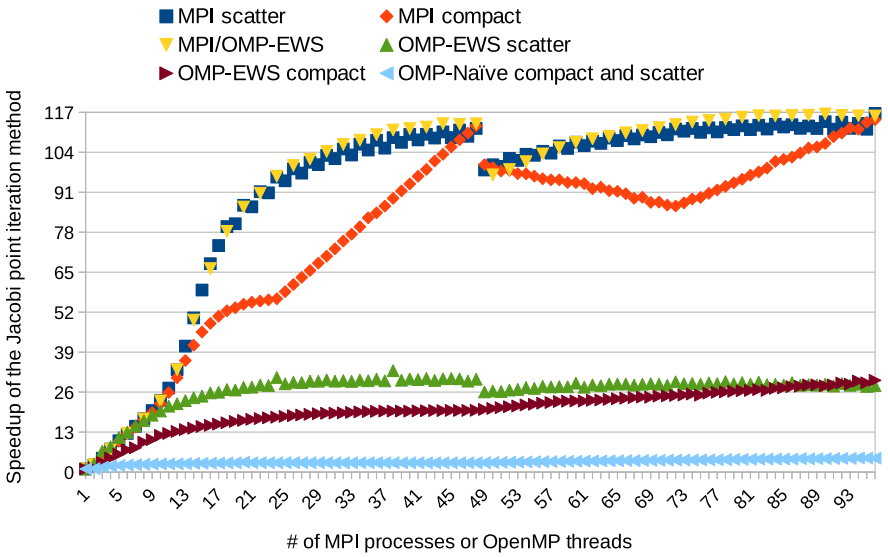

FIGURE 9 Speedups of the pure MPI-based implementation, OMP-EWS, MPI/OMP-EWS, and naive approaches of the 2D Jacobi point iteration method with scatter and compact thread binding policies in runs performed on a machine featuring an Intel® Xeon® Scalable Processor

using hyperthreading. Figure 9 shows that a code that employed few synchronization points and generated a small executable file diminished this effect regarding the results yielded by the hybrid MPI/OMP-EWS approach. Furthermore, Figures 7 and 9 show that the MPI scatter approach achieved, in general, better results than the MPI compact approach did when the approaches did not use the maximum number of threads in the socket (in the machine). As previously mentioned, an application that applies the scatter process binding policy uses physical cores before using hyperthreading. An application that uses the compact process binding policy employs hyperthreading with more than one process. Thus, the former yields a better load balancing than the latter in such cases.

Figures 7 and 8 show that the OMP-EWS approach with the compact thread binding policy yielded similar results compared to the naïve OpenMP-based approach in conjunction with the scatter thread binding policy did in executions using only one socket. The reason is that the former employed hyperthreading even when only a few threads were used, whereas the latter employed hyperthreading after 48 threads were used on the SKL machine. The OMP-EWS approach with the compact thread binding policy yielded notable better speedups than the naïve OpenMP-based approach in runs carried out using both sockets. Furthermore, Figure 9 shows that the former yielded better results than the latter did when employed in a method that used only one synchronization point and presented a small code.

Concerning the 2D Hopmoc (see Figure 7) and odd-even Hopscotch (see Figure 8) methods, the MPI/OMP-EWS approach achieved better speedups than the other versions did in executions performed on the two sockets contained in the machine. In these cases, the MPI/OMP-EWS version yielded more efficient synchronization than the other approaches did because the former performed MPI inter-socket and OpenMP intra-node communications. Additionally, the hybrid MPI/OMS-EWS approach shared the same level 3 cache memory in runs carried out on the SKL machine. Thus, the hybrid version yielded a better cache locality than the other approaches did in executions carried out on the SKL machine.

Regarding the Jacobi point iteration method (see Figure 9), although the MPI/OMP-EWS approach yielded slightly better results than the pure MPI scatter version in executions with several different numbers of threads, the latter achieved a better speedup than the former did when the maximum number of threads was used in runs performed on the SKL machine. The reason for the results is the load imbalance and the NUMA

TABLE 1 Parallel approaches listed in order of the best results in the runs performed on two machines

Machine	Hopmoc and odd-even Hopscotch methods	Jacobi point iteration method
Intel® Xeon® Phi™ Knights Landing Processor 7250	MPI/OMP-EWS L2 scatter and compact	MPI scatter
Intel® Xeon® Platinum 8160 Processor	MPI/OMP-EWS	

effect. The OMP-EWS approach required more time in the busy waiting synchronization than the message exchange operations performed by the MPI API. The same level of cache memory contained MPI processes. Additionally, in general, MPI processes only performed send and receive calls between sockets. The Jacobi point iteration method used only one double loop in an iteration, resulting in a small executable file. Thus, its code presented high instruction cache hit rates in the level 1 cache memory. This characteristic favors the MPI model. In particular, the lower the number of synchronization points used in the code, the higher the speedup obtained in the experiments on the SKL machine (33× within the Hopmoc method (see Figure 7), 79× within the Hopscotch method (see Figure 8), and 117× within the Jacobi point iteration method (see Figure 9)), recalling that the SKL machine comprises 48 cores and 96 threads. A superlinear speedup can happen for several reasons, but in the case of the Jacobi point iteration method running on the SKL machine, it seems that the serial execution took longer than it should because of the misuse of the level-1 data cache. Additionally, the pure MPI-based and hybrid MPI/OMP-EWS approaches achieved high cache hit rates in executions with a large number of processes and threads, because each process or thread was responsible for little data. The Jacobi point iteration method generated a small code by employing only a double loop. Consequently, the method achieved high instruction cache hit rates. The same result did not occur on the KNL machine because the size and cache architecture are different from the SKL machine. Table 1 shows the best parallel approaches used in the implementations of three explicit FD methods in the executions performed on two machines.

6 | CONCLUSIONS

This paper evaluated several MPI and OpenMP approaches of three 2D explicit FD methods for solving PDEs in executions performed on two machines. We analyzed a naïve OpenMP approach, the OMP-EWS approach, and a pure MPI-based approach. Additionally, we evaluated hybrid MPI/OpenMP approaches intending to reduce the NUMA effect. Specifically, we appraised hybrid approaches with MPI processes in specific domains.

The results revealed that a naïve OpenMP-based implementation of three 2D explicit FD methods for solving PDEs was inefficient. The implicit OpenMP barrier synchronization mechanism requires significant spin and overhead times, limiting the performance of the application. The reason is that the OpenMP standard employs a barrier synchronization mechanism that waits for all threads to reach the same point at the code. The OpenMP API provides neither semaphores nor a barrier synchronization mechanism that allows the developer to choose specific threads to wait for their task to be accomplished. In particular, a lock operation in OpenMP can be set and unset only by the same thread that originated the lock. Send and receive operations between communicators in MPI perform automatic synchronization only between the processes involved in the dependency.

The results showed that, in general, the OMP-EWS approach did not compare favorably with the other versions appraised. In general, the MPI/OMP-EWS and pure MPI-based approaches yielded better speedups than the pure OMP-EWS approach of the 2D explicit FD methods for solving PDEs evaluated in this study. The OMP-EWS approach of the three explicit FD methods for solving PDEs evaluated suffered from high overhead when synchronizing threads.

The hybrid approaches evaluated reduced the NUMA effect. A large domain, such as the schemes used in the hybrid approaches evaluated in this study, significantly attenuated the NUMA effect when the 2D explicit FD method employed several synchronization points at each time step. The reason is that the hybrid approaches applied MPI processes for interdomain communication and OpenMP threads for intradomain communication. For example, the MPI/OMP-EWS L2 version used MPI processes for intertile communication and OpenMP threads for intratile communication. In this strategy, two physical CPU cores shared the same level 2 cache memory. Therefore, the MPI/OMP-EWS approach yielded a high cache hit rate. Furthermore, in general, the use of the scatter binding policy yielded better speedups than the compact binding policy did. The former delivered better load balancing for employing hyperthreading after occupying all physical cores in the machine, whereas the latter uses hyperthreading even when using few threads. Furthermore, the hybrid MPI/OpenMP approaches yielded better speedups than the pure OpenMP- and MPI-based solutions of 2D explicit FD methods that employed four (Hopmoc method) and five (odd-even Hopscotch method) synchronization points at each time step. In contrast to the common practice of using OpenMP-based solutions on shared-memory machines, the pure MPI-based approach achieved, in general, better results than the pure OpenMP-based and hybrid approaches when the 2D explicit FD method employed only one synchronization point at each iteration in executions performed on the SKL and KNL machines (see Table 1). A small executable file generated by the Jacobi point iteration method also favored the use of the MPI model. Thus, the approach attenuated the NUMA effect and produced high cache hit rates.

The results in this study for the SKL and KNL machines indicated that the NUMA effect depended on the size of the MPI domain. In methods that employed several synchronization points, as in the case of the 2D Hopmoc and odd-even Hopscotch methods, the MPI/OMP-EWS L2 approach delivered, in general, better results than the MPI/OMP-EWS core version in executions performed on the KNL machine. As another example, the hybrid MPI/OMP-EWS approach yielded better results than the other approaches, recalling that the MPI/OMP-EWS approach

employed MPI process for intersocket communication and OpenMP threads for intrasocket communication in executions performed on the SKL machine. The use of a large domain favored the MPI model, reducing the NUMA effect.

The results in this study for the SKL and KNL machines indicated that the NUMA effect also depended on the executable size. The pure MPI-based approach of the Jacobi point iteration method, in general, superseded the other approaches in runs carried out on the KNL and SKL machines. The small size of the executable file favored reducing the NUMA effect in the pure MPI-based approach. Furthermore, the use of only one synchronization point at each iteration in the Jacobi point iteration method benefited from the use of the pure MPI-based approach.

The results presented in this study raise several questions. In a future investigation, we will provide explorations of the reasons the pure OMP-EWS approach of the 2D Hopmoc and odd-even Hopscotch method, in executions carried out on the KNL machine, yielded better results than the pure MPI-based approach (see Figures 4 and 5). In this case, we intend to study whether the cache miss rate for MPI is higher than for OpenMP, as indicated by Brinskiy et al.[13] Furthermore, we plan to confirm whether the largest difference in time between the two approaches is because of communication overhead. Specifically, we want to confirm whether MPI calls result in a higher execution cost than the time required in the busy waiting loop used in the OMP-EWS approach of these two methods. We want to verify whether this effect occurs because of the use of several synchronization points in these methods.

An application that employs the scatter thread binding policy uses physical cores before using hyperthreading, whereas an application that employs the compact binding policy already uses hyperthreading with more than one thread. This behavior explains why the OMP-EWS approach with the scatter thread binding policy yielded, in general, better results than the compact thread binding policy when using up to half of the threads available on the two machines in this computational experiment. Nevertheless, concerning the Jacobi point iteration method, the approach achieved similar results in tandem with both thread binding policies in runs performed on the KNL machine. We intend to investigate this behavior in a future study.

In future studies, we intend to verify whether a code that occupies small instruction cache space yields superlinear speedup. We also plan to examine whether the results presented in this study occur in other Intel® Xeon® Scalable Processors and Knights Landing accelerators. We also need to check whether the same results happen when a different compiler is used. Additionally, we plan to appraise loop tiling techniques, which are significant compiler optimizations.

Along the same lines, we intend to implement the hybrid approaches combining with MPI and threads. We also want to evaluate the MPI shared memory model in the implementations of the 2D explicit FD methods assessed.

We also plan to execute the three 2D explicit FD methods evaluated in this study with large-scale meshes in executions performed in a supercomputer environment, such as the Santos Dumont supercomputer, using a large number of nodes and cores. In this context, we intend to investigate the use of the adaptive MPI implementation, which was designed to execute the MPI efficiently with hundreds of thousands of cores.

ACKNOWLEDGMENTS

The authors would like to thank the Núcleo de Computação Científica at Universidade Estadual Paulista (NCC/UNESP) for letting them execute the simulations on its heterogeneous multicore cluster. This work was supported by Intel® through the projects entitled Intel Parallel Computing Center, Modern Code Partner, and Intel/Unesp Center of Excellence in Machine Learning. The National Council for Scientific and Technological Development supported a scientific initiation scholarship for the fourth author. We executed our simulations in a machine located in the Núcleo de Computação Científica at Universidade Estadual Paulista (NCC/UNESP).The resources of the NCC/UNESP were partially funded by Intel(R).

ORCID

Frederico L. Cabral https://orcid.org/0000-0003-0467-3424
Sanderson L. Gonzaga de Oliveira https://orcid.org/0000-0003-4863-542X

REFERENCES

1. MPI Forum. MPI forum. https://www.mpi-forum.org/. Accessed 2019.
2. Cabral F, Osthoff C, Souto R, et al. Fine-tuning an OpenMP-based TVD-Hopmoc method using Intel® Parallel Studio XE Tools on Intel® Xeon® Architectures. Paper presented at: 5th Latin American Conference on High Performance Computing (CARLA); 2018; Bucaramanga, Colombia.
3. Bassi F, Colombo A, Crivellini A, Franciolini M. Hybrid OpenMP/MPI parallelization of a high-order discontinuous Galerkin CFD/CAA solver. In: Papadrakakis M, Papadopoulos V, Stefanou G, Plevris V, eds. *VII European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS Congress*. Crete Island, Greece: National Technical University of Athens; 2016:7992-8012.
4. Cabral F, Osthoff C, Souto RP, et al. An improved OpenMP implementation of the TVD-Hopmoc method based on a cluster of points. Paper presented at: 13th International Conference on High Performance Computing for Computational Science (VECPAR); 2018; São Pedro, Brazil.
5. Brandão DN, Gonzaga de Oliveira SL, Kischinhevsky M, Osthoff C, Cabral FL. A total variation diminishing hopmoc scheme for numerical time integration of evolutionary differential equations. Paper presented at: 18th International Conference on Computational Science and Its Applications; 2018; Melbourne, Australia.
6. Mallón DA, Taboada GL, Teijeiro C, et al. Performance evaluation of MPI, UPC and OpenMP on multicore architectures. In: Ropo M, Dongarra J, Westerholm J, eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface. European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*. Berlin, Heidelberg: Springer; 2009:174-184. Lecture Notes in Computer Science; vol. 5759.

7. Chang MK, Yang L. Comparative analysis of OpenMP and MPI on multi-core architecture. Paper presented at: 44th Annual Simulation Symposium; 2011; Boston, MA.
8. Drosinos N, Koziris N. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. Paper presented at: 18th International Parallel and Distributed Processing Symposium; 2004; Santa Fe, NM.
9. Jones MD, Yao R. Parallel programming for OSEM reconstruction with MPI, OpenMP, and hybrid MPI-OpenMP. Paper presented at: 2004 IEEE Symposium Conference Record Nuclear Science; 2004; Rome, Italy.
10. Jeffers J, Reinders J, Sodani A. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. 2nd ed. Burlington, MA: Morgan Kaufmann; 2016.
11. Cafaro M, Pulimeno M, Epicoco I, Aloisio G. Parallel space saving on multi- and many-core processors. *Concurrency Computat Pract Exper*. 2018;30(7):e4160.
12. Koleva-Efremova V. Testing performance and scalability of the pure MPI model versus hybrid MPI-2/OpenMP model on the heterogeneous supercomputer Avitohol. In: Georgiev K, Todorov M, Georgiev I, eds. *Advanced Computing in Industrial Mathematics. BGSIAM 2017*. Sofia, Bulgaria: Springer; 2019:93-105. Studies in Computational Intelligence, vol. 793.
13. Brinskiy M, Lubin M, Dinan J. MPI-3 shared memory programming introduction. In: Jeffers J, Reinders J, eds. *High Performance Parallelism Pearls volume two: Multicore and Many-Core Programming Approaches*. Burlington, MA: Morgan Kaufmann; 2015:305-319.
14. Vu L, Alaghband G. A load balancing parallel method for frequent pattern mining on multi-core cluster. In: Proceedings of the Symposium on High Performance Computing (HPC). Alexandria, VA: Society for Computer Simulation International; 2015.
15. Ferretti M, Musci M, Santangelo L. MPI-CMS: a hybrid parallel approach to geometrical motif search in proteins. *Concurrency Computat Pract Exper*. 2015;27:5500-5516.
16. Diener M, White S, Kale LV, Campbell M, Bodony DJ, Freund JB. Improving the memory access locality of hybrid MPI applications. In: Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI). Chicago, IL:ACM; 2017:11.
17. Oliveira SRF, Gonzaga de Oliveira SL, Kischinhevsky M. Convergence analysis of the Hopmoc method. *Int J Comput Math*. 2009;86:1375-1393.
18. Gordon P. Nonsymmetric difference equations. *J Soc Ind Appl Math*. 1965;13:667-673.
19. Jr Douglas J, Russell TF. 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*. 1982;19:871-885.
20. 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* [in Portuguese]. Niterói, Brazil: Universidade Federal Fluminense; 2018.
21. Gourlay AR. Hopscotch: a fast second-order partial differential equation solver. *IMA J Appl Math*. 1970;6(4):375-390.
22. DuChateau P, Zachmann DW. *Partial Differential Equations*. 3rd ed. New York, NY: McGraw Hill; 2011.

How to cite this article: Cabral FL, Gonzaga de Oliveira SL, Osthoff C, Costa GP, Brandão DN, Kischinhevsky M. An evaluation of MPI and OpenMP paradigms in finite-difference explicit methods for PDEs on shared-memory multi- and manycore systems. *Concurrency Computat Pract Exper*. 2019;e5642. https://doi.org/10.1002/cpe.5642