Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 739107, 20 pages
http://dx.doi.org/10.1155/2015/739107

Research Article

Performance of a Code Migration for the Simulation of Supersonic Ejector Flow to SMP, MIC, and GPU Using OpenMP, OpenMP+LEO, and OpenACC Directives

C. Couder-Castañeda,[1,2] H. Barrios-Piña,[3] I. Gitler,[1] and M. Arroyo[1]

[1] *Departamento de Matemáticas, Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional (ABACUS-CINVESTAV-IPN), P.O. Box 14-740, 07000 México, DF, Mexico*
[2] *Centro de Desarrollo Aeroespacial del Instituto Politécnico Nacional, Belisario Domínguez 22, 06010 México, DF, Mexico*
[3] *Tecnológico de Monterrey, Avenida General Ramón Corona 2514, 45201 Zapopan, JAL, Mexico*

Correspondence should be addressed to H. Barrios-Piña; hector.barrios@itesm.mx

Received 28 July 2014; Revised 9 April 2015; Accepted 4 June 2015

Academic Editor: Jan Weglarz

A serial source code for simulating a supersonic ejector flow is accelerated using parallelization based on OpenMP and OpenACC directives. The purpose is to reduce the development costs and to simplify the maintenance of the application due to the complexity of the FORTRAN source code. This research follows well-proven strategies in order to obtain the best performance in both OpenMP and OpenACC. OpenMP has become the programming standard for scientific multicore software and OpenACC is one true alternative for graphics accelerators without the need of programming low level kernels. The strategies using OpenMP are oriented towards reducing the creation of parallel regions, tasks creation to handle boundary conditions, and a nested control of the loop time for the programming in offload mode specifically for the Xeon Phi. In OpenACC, the strategy focuses on maintaining the data regions among the executions of the kernels. Experiments for performance and validation are conducted here on a 12-core Xeon CPU, Xeon Phi 5110p, and Tesla C2070, obtaining the best performance from the latter. The Tesla C2070 presented an acceleration factor of 9.86X, 1.6X, and 4.5X compared against the serial version on CPU, 12-core Xeon CPU, and Xeon Phi, respectively.

1. Introduction

Currently, the development of parallel applications in science is conducted for heterogeneous architectures which are a set of distinct processing units that share one memory system. Usually, these units are a set of processors with multiple cores or processing units such as graphics cards (GPUs) or coprocessors, like the Xeon Phi, also known as Many Integrated Core (MIC), designed to accelerate the computing time.

Due to the diversity of architectures, the type of parallel programming employed is heavily dependent on the type of hardware. For this reason, when developing a parallel application to be migrated for various platforms, the programmer must have three fundamental questions in mind: the performance regarding the programming effort, that is, the evaluation of the development cost in terms of the execution time reduction, the energy efficiency, and the ease of maintenance/modification of the source code.

For example, the parallel programming at low level on Graphics Processing Units (GPUs) (at the kernel level) can be complicated and requires much development time. This type of programming can lead to a lack of productivity and error prone, thereby usually not being acceptable for industrial projects where the development time is a critical decision factor. In order to mitigate the development effort, the next-generation compilers help to create low-level code through directives written in a higher level of abstraction. However, even when these compilers are currently more advanced and simplify the developer's work, they still require good coding design to achieve an efficient parallel algorithm. Benefits of the compiler can be evidenced by simplifying the tedious low-level coding work.

Within the methodology of a parallel application, the most difficult part begins even before the first line is coded. Having a successful parallel algorithm is essential to improve performance, thereby reducing processing time. If the design of the application is not significantly converted to parallel, the implementation on a GPU or MIC will not have much benefit. Other options and different possibilities for parallelization should be explored [1].

Currently, OpenMP could be considered as standard for scientific programming on symmetric multiprocessing systems (SMP), and even it can be used transparently in the Xeon Phi architecture [2]. OpenMP is sustained by a combination of function and compiler directives [3, 4]. OpenMP has proven to be a powerful tool for SMP due to several reasons: it is highly portable; it allows medium granularity; each thread can access the same shared memory; it has its own private memory; and it also has a greater level of abstraction than MPI models [5]. Specifically for Computational Fluid Dynamics (CFD), applications of OpenMP have proven their effectiveness better than MPI [6]. Meanwhile, OpenACC, with a very similar methodology than OpenMP, has recently appeared as a viable alternative for specialized low-level languages, such as CUDA C and OpenCL for fluid dynamics GPU based applications, with very good results [7, 8]. The granularity in OpenACC is considered fine for its origin linked to CUDA.

The application to be accelerated in the present work is a serial code for simulating an ejector supersonic flow, commonly used in oil and gas industry. The ejectors confine a fluid flow under controlled conditions and then discharge it to an intermediate pressure between high pressure fluid from the nozzle and low pressure fluid from the suction. The flow through an ejector is governed by the principle of momentum conservation. Ejectors are seen like a venturi that works on the transmission of energy caused by the impact of a fluid moving with a very high velocity against another slow-moving or steady flow. This impact generates a mixture of fluids moving at a moderately high velocity, finally obtaining a higher pressure than the slow-moving fluid.

Ejectors are commonly employed to extract gases from the reservoirs where vacuums are produced, appliances such as condensers, evaporators, vacuum-based distillation towers, and refrigeration systems, where the extracted gases are generally noncondensable, as air. Ejectors are also used for mixing flows in sulfitation processes of sugar mills. Figure 1 shows the components of a typical ejector.

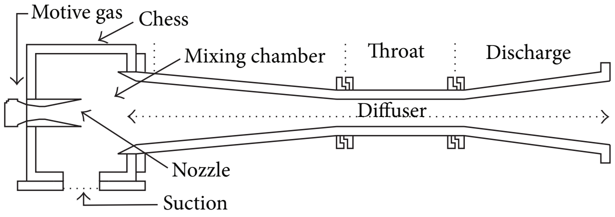


Figure 1: Different parts of a typical ejector.

Previous numerical simulations for the ejector diffuser using the present source code were conducted by Couder-Castañeda [9], where the geometry of the diffuser, the numerical scheme, and the initial and boundary conditions are described in detail. Couder-Castañeda parallelized the code using a message-passing methodology based on the JPVM; however, the code was tested only in four processors because the required number of messages is intensive. Thus it is not scalable. After the work of Couder-Castañeda, convolutional PML boundary conditions were improved in the code to absorb pressure waves at the outflow boundary [10]. The simulations have been costly in terms of computing time, resulting in undesirable time-delays to get results. For example, on an Intel Xeon 2.67 GHz, 3 seconds of real-time simulation consumes about 35 hours of serial computing time. For this reason, in this work other coding alternatives to reduce the computing time are sought, while maintaining the source code intact due to the complexity of the numerical scheme.

This paper is organized as follows. Section 2 focuses on the design of the parallel algorithm, where both OpenMP and OpenACC designs are described in detail. In Section 3, performance experiments of the designs are shown for different architectures, that is, OpenMP on conventional multicore CPU and Xeon Phi and OpenACC on GPU. Section 4 shows validation tests of the code porting for the case of the ejector flow, where the total energy and the Mach number are compared. The paper ends with the conclusions in Section 5.

2. Design of the Parallel Algorithm

Details of the numerical scheme used in the original source code can be consulted in [9]. As an explicit finite difference numerical method is used and the set of equations is solved by a predictor-corrector scheme, the code is highly parallelizable. However, it is necessary to be cautious about the parallel implementation in order to get a significant reduction of computing time. First, the number of loops to be parallelized must be analyzed. Figure 2 illustrates the flow diagram which corresponds to the predictor step and demonstrates the execution of the loops labelled as C_1, C_2, C_3, C_4, C_5, C_6, and C_7. On the other hand, Figure 3 shows the corrector step where the loops are C_8, C_9, C_{10}, C_{11}, C_{12}, C_{13}, C_{14}, C_{15}, and C_{16}.

In the predictor step, the loops are used as follows:

(i) One loop to calculate the predicted values of the flow primitives with backward finite differences (C_1).

(ii) Four light loops to apply boundary conditions (from C_2 to C_5).

(iii) One loop to calculate stress forces (C_6).

(iv) One loop to calculate the predicted flow variables (C_7).

In the corrector step, the loops are used as follows:

(i) One loop to calculate the new corrected values of the flow primitives with forward finite differences (C_8).

(ii) Four light loops to apply boundary conditions (from C_9 to C_{12}).

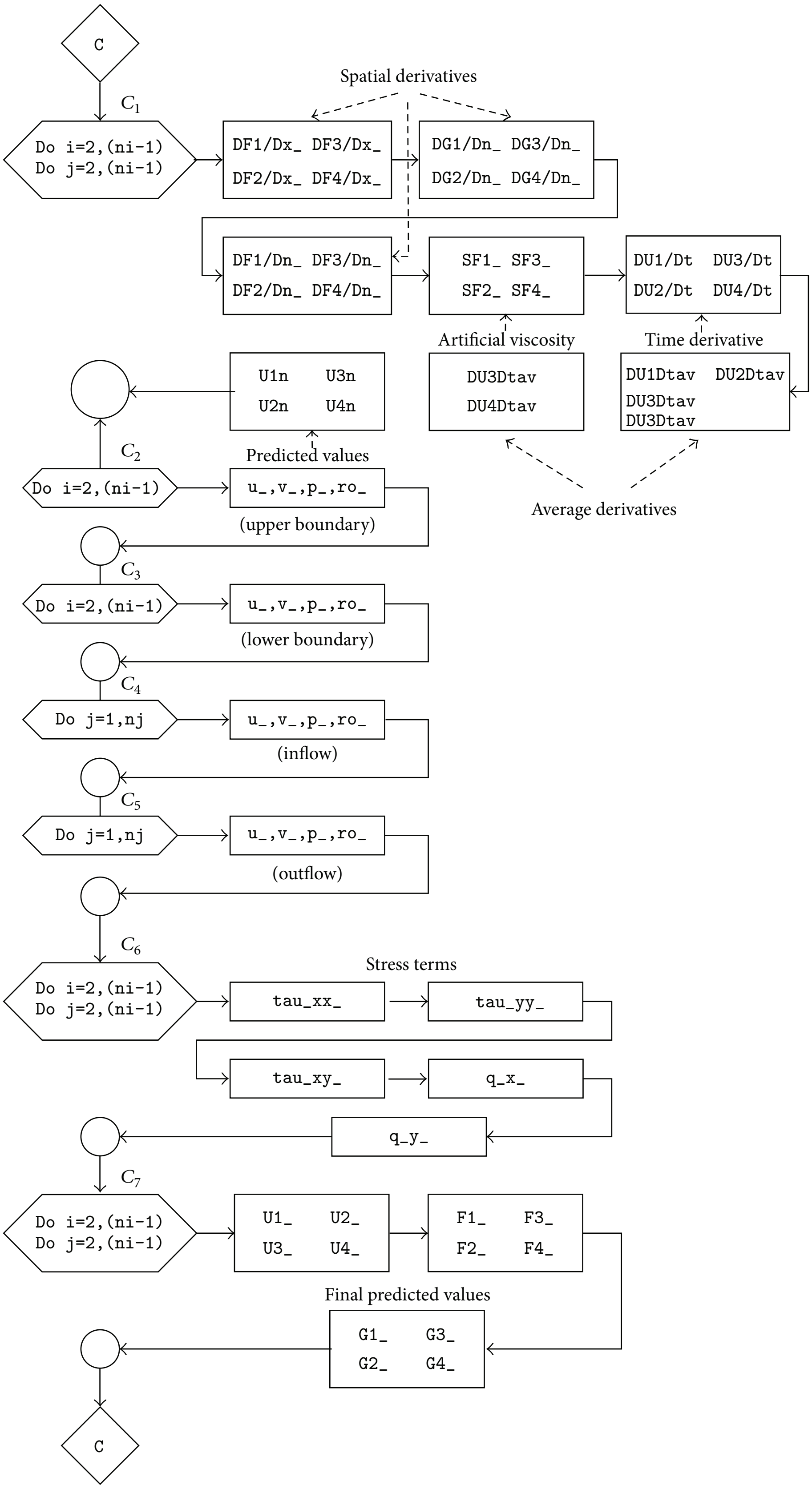


Figure 2: Flow diagram of the predictor step of the numerical algorithm.

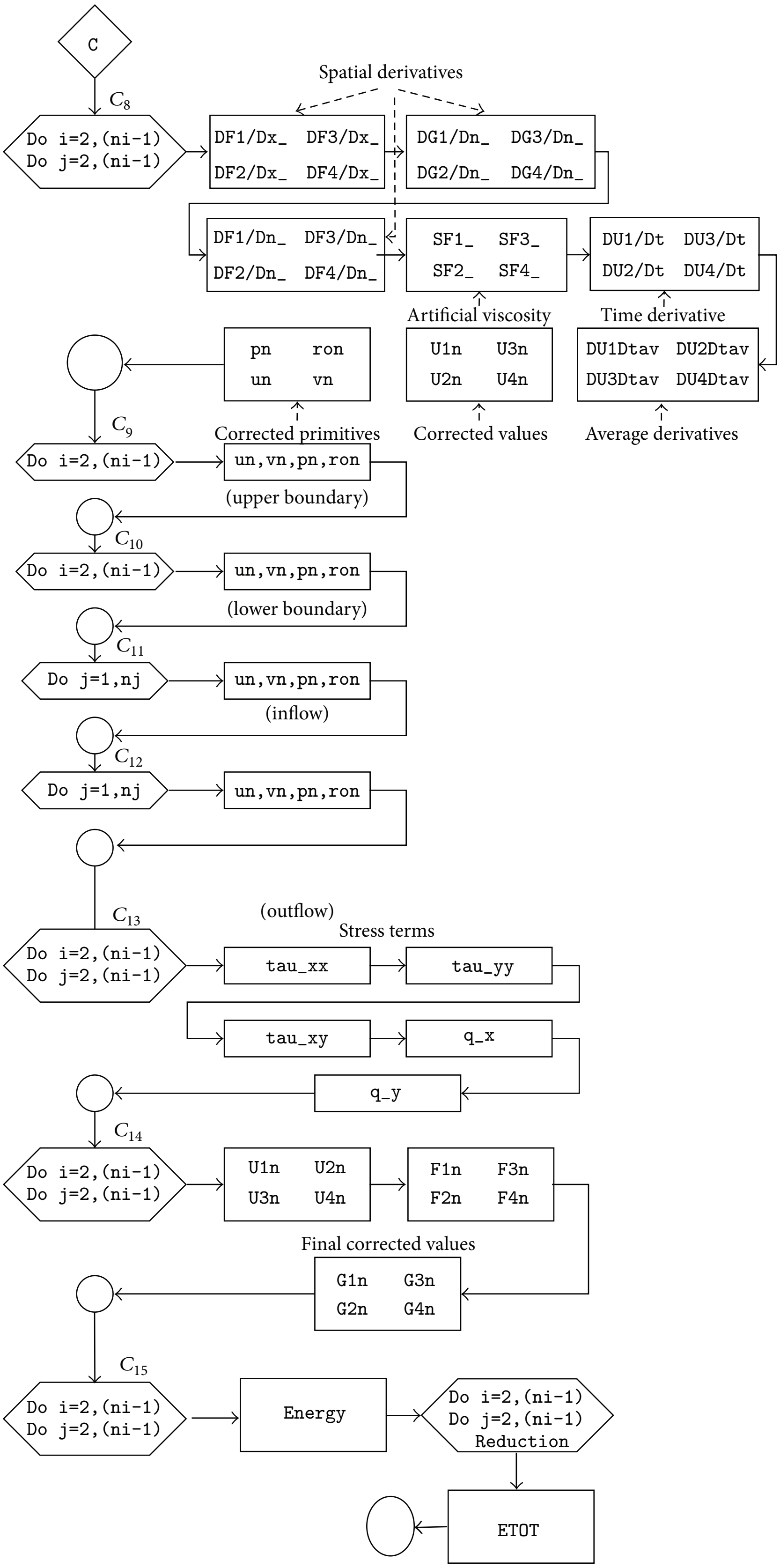


Figure 3: Flow diagram of the corrector step of the numerical algorithm.

Table 1: Intensity values of loops of the original source code. The underlined intensities correspond to loops candidates to be parallelized.

Loop	Intensity
C_1	$\underline{1.63}$
C_2	$\boxed{0.50}$
C_3	$\boxed{0.50}$
C_4	$\boxed{0.36}$
C_5	$\boxed{0.00}$
C_6	$\underline{2.69}$
C_7	$\underline{1.86}$
C_8	$\underline{1.86}$
C_9	$\boxed{0.50}$
C_{10}	$\boxed{0.50}$
C_{11}	$\boxed{0.36}$
C_{12}	$\boxed{0.00}$
C_{13}	$\underline{2.14}$
C_{14}	$\underline{2.69}$
C_{15}	$\underline{1.86}$
C_{16}	$\underline{1.40}$

(iii) One loop to calculate stress forces (C_{13}).

(iv) One loop to calculate the final flow variables (C_{14}).

Additionally, one more loop is considered to calculate the energy per cell (C_{15}), and another loop with a sum reduction serves to calculate the total energy (C_{16}). Thus, the code requires 16 loops to complete one time step. In order to get a significant reduction in computing time, the loops must be computationally intensive. Intensity is not the only metric, but for codes that use finite differences it is very useful. The intensity of a loop is the ratio of floating point operations to the memory accesses required by the loop.

In the PGI FORTRAN compiler, the parameter `-Minfo=intensity` can be employed to calculate intensity I. The intensities obtained for loops C_1 to C_{16} are shown in Table 1. It is necessary to clarify that the intensity I is defined as $I = f/m$, where f is the number of floating point operations and m is the number of data movements; for example, `A(:)=B(:)+C(:)` has an intensity of 0.333 (3 memory accesses, 1 operation) or `A(:)=C0+B(:)*(C1+B(:)*c2)` has an intensity of 2.0 (2 memory accesses, 4 operations); more examples could be seen in [11]. All the intensities in the loops were calculated manually following this definition to verify the results given by the PGI compiler. Additionally, the computing time required by loops was measured using the PGI profiling tool. We observe that intensity is directly proportional to computing time, which means that loops with greater intensity are loops that consume more computing time.

The intensities obtained are shown in Table 1. The implications of these results have to be managed carefully, because it is necessary to take into account the characteristics of the hardware. For this problem we consider that every loop with an intensity level $I \geq 1.0$ is a candidate to be parallelized, but it is also possible to parallelize them if they are part of a larger program. Loops with $I \leq 1.0$ generally are loops for which accelerating is not recommended but it depends on the platform properties. The loops of the present source code with intensities greater than 1.0 (values underlined in Table 1) are ready to be parallelized. The loops with intensities less than 1.0 are those where the boundary conditions are handled (values boxed in Table 1). In these cases, the loops are data dependent, which means that once one loop has finished, the next can start, with the exception of specific loops of boundary conditions that can be executed concurrently.

2.1. Design in OpenMP. The methodology begins with the OpenMP design, since it is more widely used and is compiled directly for multicore systems. Currently, OpenMP is considered the de facto standard to express parallelism in symmetric multiprocessing systems. According to Calvin et al. [2], OpenMP can be employed transparently in the Xeon Phi architecture. Furthermore, it is based on a combination of compilation directives and functions. OpenMP has proven to be a powerful tool for the development of scientific applications requiring parallelization, because (1) it is highly portable across multiple platforms, (2) it allows the development of applications with medium granularity, (3) each processing thread created with the same directive has its own private memory while also being able to access the shared memory, and (4) it is considered to have a higher level of abstraction than the message-passing model [3, 12, 13]. Two possible disadvantages that the use of OpenMP can has are that the application can be affected by problems of cache coherence, and uncontrolled and simultaneous access to the shared memory can lead to false sharing problems between execution threads.

The main characteristics of OpenMP are as follows:

(i) The OpenMP codes run on shared memory machines. It could be expected that they run also in GPUs in the 4.0 specification.

(ii) It has high portability.

(iii) It permits both medium grain and fine grain parallelism (vectorization level).

(iv) Each thread sees the same global memory but has its own private memory.

(v) It has implicit messaging (through shared variables).

Some disadvantages we can find are as follows:

(i) The placement policy of data can cause problems for not experts developers.

(ii) Overheads can become an issue when the size of the parallel loop is too small.

(iii) Threads are executed in a nondeterministic order.

(iv) Explicit synchronization is required.

In order to use OpenMP, the code should include the compilation directives to generate the parallel loops, thereby distributing the computation automatically [13]. The loops could be parallelized using Listing 1 or Listing 2. The computing is distributed implicitly; therefore the partitioning of

```
!$OMP PARALLEL DO SHARED(···) &
!$OMP & FIRSTPRIVATE(···)
   ...
!$OMP END PARALLEL
```

Listing 1: The parallel DO is implemented more efficiently than a general parallel region containing a loop.

```
!$OMP PARALLEL SHARED(···) &
!$OMP & FIRSTPRIVATE(···)
  !$OMP DO
    ...
  !$OMP END DO
!$OMP END PARALLEL
```

Listing 2: Parallel region containing a loop.

```
!$OMP PARALLEL DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
!$OMP END PARALLEL DO
```

Listing 3: Opening and closing of the parallel region.

```
!$OMP PARALLEL
  !$OMP DO
  !$OMP END DO

  !$OMP DO
  !$OMP END DO
!$OMP END PARALLEL
```

Listing 4: Parallel region persistent between loops.

the loop is effectuated automatically using a balancing algorithm. It can be specified by the developer but for our application the decision is left to the scheduler.

We have two options for parallelizing the loops that have to be considered; the first is the handling of loops inside a parallel region (Listing 3), and the second is to create one parallel DO for each loop (Listing 4). Both options are viable because even when a parallel region is closed the threads remain active and when a new parallel region is reopened the overhead is not significant.

On the other hand, the parallelization of the loops `DO/FOR` in the highest possible level can lead to a better performance, thereby implying the parallelization of the outer-most loop and the encompassing of multiple loops in the parallel region. In general, the creation of loops inside parallel regions reduces the overload for the parallelization by avoiding the creation of a parallel DO.

For example, the code showed in Listing 1 is more efficient than the code showed in Listing 2.

Nevertheless, the parallel constructor in OpenMP of the FORTRAN code shown in Listing 3 is less efficient than the code shown in Listing 4, because of the creation of the parallel DO; however the overhead creation is minimal, but, in finite difference codes for fluid flows, where millions of iterations are necessary over the time, a minimal overhead creation could not be negligible.

Implementations using Listing 3, Figure 4(a), can be seen in high-performance algorithms based on finite difference methods, as the SEISMIC_CPML algorithm [14, 15]. However, it misses the benefits shown in Listing 4, Figure 4(b), where a better performance can be obtained. In this case, the parallel region is persistently maintained throughout the time iteration. Additionally, we prefer to code by the way of Listing 4 because the readability of the code could be improved. This is due to the behavior of the variables that are only declared once at the beginning of the parallel region (`SHARED,PRIVATE`...). With the use of the scheme of Listing 4, the code contains 16 execution loops in one time iteration and only closes and reopens at the beginning of the time iteration. In this way, the opening and closing of 16 parallel regions on each iteration are avoided. It is important to clarify that the compiler cannot joint parallel regions automatically, because it could modify the logic of the flow work. If the parallel region is continuously opened and closed, it is obvious that the creation of a parallel region (spawning in multiple threads) will imply computational resources consumption, even if the threads are not destroyed. The time consumption to create a parallel region depends on compiler optimizations; thus this is a responsibility of the developer.

The scheme shown in Figure 4(b) uses only one parallel region. This scheme was selected to develop the code with the OpenMP directives. For the loops for boundary conditions, C_2 to C_5 and from C_9 to C_{12}, where intensities are $I < 1.0$, we have two options. The first is to perform parallelization by distributing the loop between available processing units (cores), as with the other loops with intensities greater than 1. This first alternative could not be convenient because a false sharing can occur; nevertheless as the loops are part of a large program, it seems to be a viable alternative. The second option, which is preferred in the present work, is to create one task for one loop (four loops to manage the boundary conditions). In our case, from one thread up to four can be assigned to perform the tasks. If the task is considered computationally nonintensive, one thread can perform all the tasks (all the loops), or if the task takes much time to be completed by one thread, other threads could be assigned to other tasks (see Figure 5).

Finally, it is possible to maintain the parallel region open, even during the time iteration. As shown in Figure 6(b), all the threads have control of the time iteration, thereby maintaining the parallel region open during the entire simulation time. It means that just one parallel region is created during all the execution. In this way, the possibility of overhead is reduced to a minimum. With this last design, only one

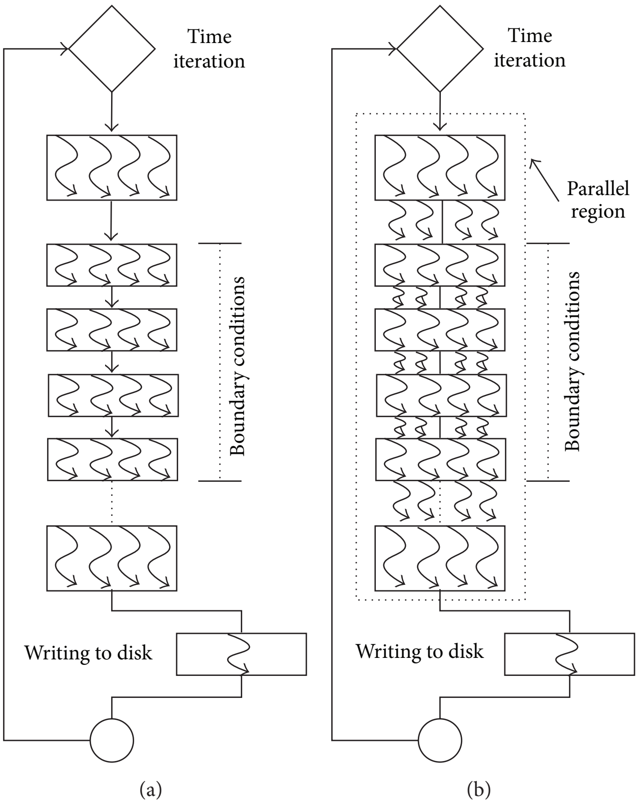


Figure 4: Design in OpenMP. (a) Opening and closing of parallel regions between loops, (b) persistent parallel regions between loops.

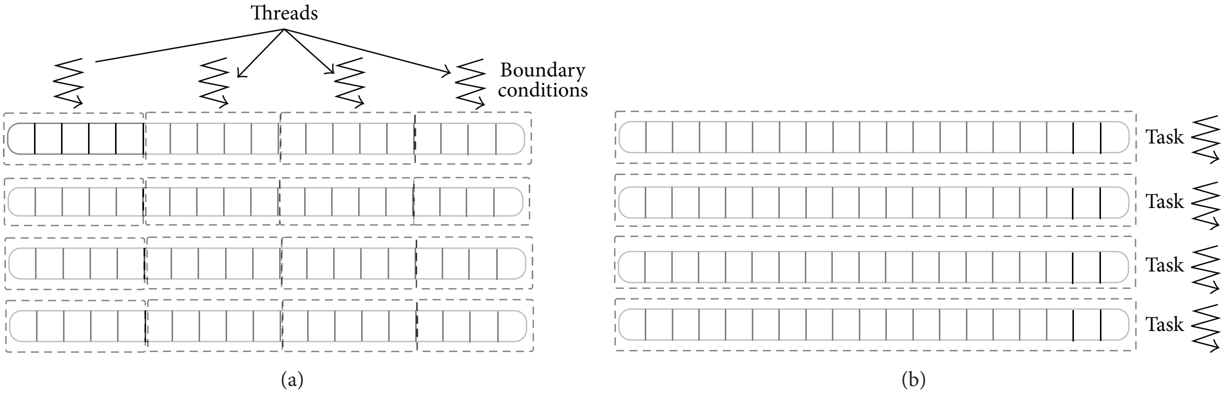


Figure 5: Implementation of the boundary conditions. (a) Distribution of the loops among the threads, (b) assignment of one loop to one task to avoid possible false sharing due to a computational intensity < 1.

parallel region is opened during the whole execution of the program and the loops with intensities below 1 are managed with tasks; however it is necessary to mention that there are more than one design possibilities.

Another important issue taken into account is the collapsing of loops. The collapse directive is used to increase the total number of iterations that will be partitioned across the available number of threads. By reducing the granularity, the number of parallel iterations to be done by each thread is therefore increased. If the amount of work to be done by each thread is not vectorizable (after collapsing is applied), the parallel scalability of the application may be improved. This technique was compared against vectorization of the inner loop.

When the loops are collapsed, the number of iterations distributed among the threads is $(nj-1)\times(ni-1)$ (see Listing 5). Without collapsing, the number of iterations distributed among the threads is $(nj - 1)$. In Listing 6 the loops are collapsed and the operations are vectorized; however, the number of floating point operations should be sufficient to

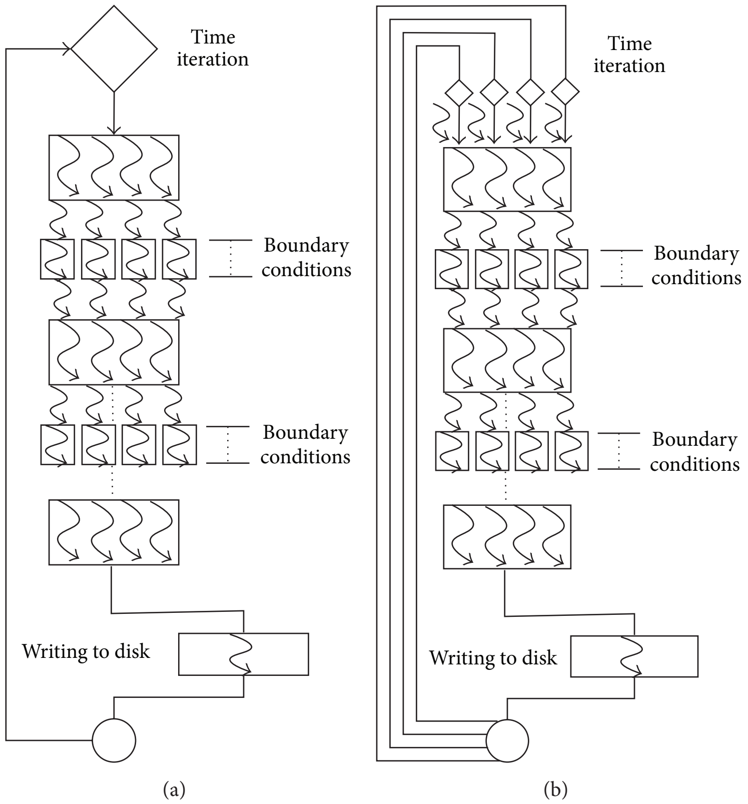


Figure 6: Design in OpenMP with boundary condition handling via tasks. (a) A boundary condition is handled by only one thread; (b) all threads handle the time iteration creating only one parallel region for the entire program execution.

```
!$OMP DO COLLAPSE(2)
DO j=2,(nj-1)
    DO i=2,(ni-1)
    ...
    END DO
END DO
```

Listing 5: Iterations of the loops are joined to be executed in parallel by the team of threads.

```
!$OMP DO SIMD COLLAPSE(2)
DO j=2,(nj-1)
    DO i=2,(ni-1)
    ...
    END DO
END DO
```

Listing 6: Iterations of the loops are joined to be executed in parallel and the internal operations are intended to be vectorized.

achieve the vectorization. If the operations are not sufficient, the compiler is not able to generate a vectorized code and a warning would be shown. In our code this is the case.

```
!$OMP DO SIMD
DO j=2,(nj-1)
    DO i=2,(ni-1)
    ...
    END DO
END DO
```

Listing 7: Iterations of the outer loop are distributed among the threads and the operations are intended to be vectorized; as the inner loop exists, it is intended to be vectorized.

For this reason the directive `OMP DO SIMD COLLAPSE(2)` cannot be used. The Intel FORTRAN Compiler version 15.0 produces the warning `#13379: loop was not vectorized with “simd”` and the PGI compiler produces the warning `Loop not vectorized: may not be beneficial`. In Listing 8, the directive `OMP DO SIMD COLLAPSE(2)` could be applied with better success because an additional inner loop is included.

To distribute the iterations of the outer loop among the threads and to vectorize the operations (in general the inner loop), the directive `OMP DO SIMD` (Listing 7) could be used. For some compilers, the clause `SIMD` is not necessary because an automatic vectorization is applied.

```
!$OMP DO SIMD COLLAPSE(2)
DO j=2,(nj-1)
    DO i=2,(ni-1)
        DO k=2,(nz-1)
        ...
        END DO
    END DO
END DO
```

Listing 8: Iterations of the outer loops are joined to be executed in parallel and the internal operations are intended to be vectorized. As the inner loop exists, it is intended to be vectorized.

The possibilities to parallelize the loops applicable to our present code are shown in Listings 5 and 7.

2.1.1. OpenMP on the Xeon Phi. One advantage of the parallel implementation in OpenMP is the possibility of migrating to a MIC type architecture practically without modification [4, 16]. The migration only requires the transfer of variables and data to the coprocessor and then the transfer of the execution of the parallel region. When the parallel regions are opened, it is important to avoid the transfer of data between CPU and coprocessor because this produces latency, which will significantly reduce the overall performance of the application. For this reason, the code should avoid any transfers during the execution of a time iteration.

Since the GPUs have been on the market during much time, more scientific applications have been migrated and tried on them than on the MICs. Specifically, the efficiency of the GPUs has been proven for finite difference-based algorithms [17–19]. MPI with CUDA C is used in [19]. There exist some similarities because the authors solve a PDE on time as we do, but their implementation is low level (kernels) for elastodynamic equations in a huge domain. Our application is for fluid flow equations in a relatively small domain; however, the flow chart shown is analogous. The GPUs do not offer an easy programming when they are used with low-level kernels, available on the Xeon Phi coprocessors, because the GPUs are not complete cores. In fact, the GPU cores can be seen as a group of small mathematical coprocessors that efficiently handle applications with very fine granularity. Similarly, we expect that applications which show a positive result using GPUs should benefit even more than the Xeon Phi architecture due to the presence of the vectorization concept, which is similar to CUDA architecture. The flexibility of the Xeon Phi allows the migration and provides support to a large number of applications requiring multiprocessing without the vectorization concept. Additionally, code generation for the GPU requires more refinement of the application to achieve the desired performance. This refinement means much development time, but, with the use of OpenACC, the job of coding low-level kernels is reduced.

One important aspect to be considered for understanding the expected performance of a Xeon Phi is the MT technology. With MT, each core can handle four threads for reducing the inherent latency to a micro-architecture multicore. This should not be confused with hyper threading (HT), defined as rapidly commutating between two threads that use the same core; HT can be deactivated via BIOS (for CPU). The Xeon Phi multithread technology cannot be deactivated and therefore we have to deal with it. Due to the inherent MT, available on the Xeon Phi coprocessor, the testing will consist of 1 up to 4 threads per core, the maximum that can be efficiently handled.

Although the Xeon Phi coprocessors are a complete computer by itself, they are able to execute programs natively independent of the CPU. The programming model recommended by Intel is the offload, where the CPU controls the Xeon Phi card, sending the tasks to be completed while the results are transferred back to the CPU. This model is employed since it is considered the most flexible, similar to the one used with the GPU. It is necessary to clarify that LEO extensions were preferred to offload the data, because although OpenMP 4.0 has capabilities of offloading, not all compilers implement it entirely. For example, OpenMP 4.0 will be apparently implemented completely in GCC 5.0 but in earlier versions it is partially implemented.

As happens in OpenMP for the CPU, what must be minimized is the number of parallel regions that are created in the loop of time, because unlike the CPU the threads created in the MIC for a parallel region are destroyed when the control returns to the CPU. However, the information must be registered on the CPU every certain number of iterations in order to write the simulation data to permanent storage. In this way, in a given instant, the parallel region must be abandoned in order to give back the control to the CPU. For this reason, it is not possible to use the model of Figure 6(b), where the parallel region is persistent throughout the whole program execution.

To minimize the number of parallel regions is important because much computing time is needed to create them. Schmidl et al. [20] showed that, to create a parallel region on an Intel Xeon Phi with 240 threads, 27.56 microseconds were needed. In the present code, we have 16 loops and, for example, a numerical simulation of 3 million of iterations implies 48 millions of `parallel for/DO`, which means 22 minutes of overload approximately (environ 2% of the overall time).

It is possible to use a design whereby the region is maintained open when disk writing is necessary. In Figure 7(a) we show a conventional design, where for each time iteration a parallel region is created on the MIC (this design will be denoted by D1 herein). In Figure 7(b) an improved design is shown, designed by reducing the number of created parallel regions. This is achieved by abandoning the execution of the MIC when disk writing is needed (this design will be denoted by D2 herein). In other words, control is given to the CPU only when transferring data from simulation to disk is effectuated. The corresponding implementation of Figure 7(b) can be seen in the Listing 9.

2.2. Design in OpenACC. Once the OpenMP model is completed and since this algorithm is a finite difference algorithm

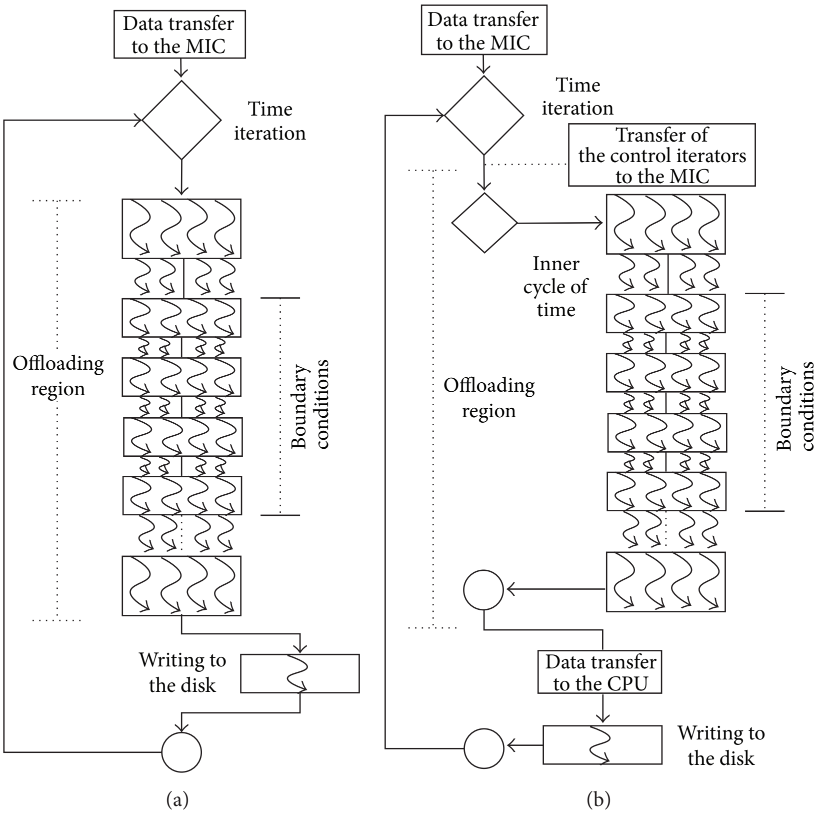


FIGURE 7: Design in OpenMP for the MIC architecture. (a) For each time iteration, one parallel region is opened and unloaded; (b) the parallel region is maintained until it is necessary to write data to the permanent memory; at that point control is returned to the CPU followed by a reentry to the MIC.

without complex reductions, it seems easy to migrate it to the OpenACC model. Both models have many similarities that can be consulted extensively in [21].

Researches have shown that, to obtain the best performance on the GPU, the transfers between CPU and GPU must be reduced at minimum. These transfers are transfers between kernels of the persistent variables. In OpenACC, the directive data specifies a region of the code where the data will be persistent on the GPU. Any region enclosed in the data region will use a persistent variable on the GPU, without the necessity of transferring the variables themselves to the compute region (kernel). The design used in this work is shown in Figure 8. This design is analogous to the one shown previously in Figure 7(a). However, the GPU has a limited residence time of the functions; that is, the card necessarily returns the control to the CPU after executing the kernel, in contrast to MIC.

The code structure which corresponds to the design of Figure 8 is shown in Listing 10.

The format of the loop (C_{16}) to calculate the total energy with a reduction is shown in Listing 11.

3. Performance Experiments

This section is focused on the experiments to evaluate both the performance of OpenMP on the conventional multicore CPU and Xeon Phi and the performance of OpenACC on the TESLA C2070 GPU. The time step Δt is 1×10^{-6} s and the code is executed to simulate 3.0 s of real time, requiring 3 millions of time iterations. The computational domain is composed of 1121×41 discrete points in x and y directions, respectively. Based on this mesh size, the computational domain can be considered as small, which has to be taken into account for the calculation of the performance.

This problem is considered as strong-scaling, because the computational domain is fixed and the number of processing elements is increased (threads mapped to cores); therefore the problem is governed by Amdahl's law. In strong-scaling, a program is considered to scale linearly if the speed-up is equal to the number of the processing elements used. In general, it is hard to achieve good strong-scaling with a big number of processes, since the communication overhead for most algorithms increases in proportion to the number of processes used. For the present problem the processes are threads.

3.1. Experiments on CPU. The configuration of the workstation is

(i) Dual Intel(R) Xeon(R) CPU X5650 @ 2.67 Ghz,

(ii) 12 real cores (one core can handle two threads when HT is enabled),

```
! dir$offload_transfer target(mic:0)
in(U1,U2,U3,U4: alloc_if(.true.)
free_if(.false.))
!dir$offload_transfer target(mic:0)
in(U1_,U2_,U3_,U4_: alloc_if(.true.)
free_if(.false.))
!dir$offload_transfer target(mic:0)
in(F1,F2,F3,F4: alloc_if(.true.)
free_if(.false.))

DO WHILE (.TRUE.)
...
!dir$ offload target(mic:0)
nocopy(DF1Dx,DF2Dx,DF3Dx,DF4Dx) &
& nocopy(F1,F2,F3,F4,U1,U2,U3,U4,
G1,G2,G3,G4) &
& nocopy(a_x,b_x,K_x,deltae) &
...
& in(k,Timetotal) out(k_shared) &
& out(Timetotal_shared,etot) &
& out(ro,u,v,P,T,M,ET)

!$OMP PARALLEL DEFAULT(NONE) &
!$OMP & SHARED(DF1Dx,DF2Dx,DF3Dx,DF4Dx) &
!$OMP & SHARED(F1,F2,F3,F4,
U1,U2,U3,U4,G1,G2,G3,G4) &
!$OMP & SHARED(a_x,b_x,K_x,deltae) &
...
DO WHILE (K <= nk)
!$OMP DO COLLAPSE(2)
DO j=2,(nj-1)
         DO i=2,(ni-1)
         ...
         END DO
END DO
...
!$OMP END PARALLEL
PRINT *, 'EXITING THE MIC TO WRITE DATA';
...
...
IF (k_shared >=nk) exit;
    k = k_shared+1;
    TimeTotal = TimeTotal_shared;
END DO;
```

Listing 9: Code fragments corresponding to Figure 7(b); first all the variables are transferred to the MIC.

(iii) 12 GB RAM,

(iv) CentOS 6.6 Operating System,

(v) Intel FORTRAN Compiler 15.0.0.

Before the performance experiments begin, the developer should keep in mind the effects of the HT and the vectorization on the performance [22]. Thus, the experiments are conducted with HT enabled and HT disabled. Since there are 12 physical cores, from 1 to 12 kernel threads are created to analyze the behavior of the performance when HT is disabled. On the other hand, when HT is enabled, from 1 to 24 kernel threads are created. It is not necessary to create more threads than cores (or logical cores) in the system; otherwise only overhead will be created.

For experiments with HT disabled, the strategy is to assign one thread to one core, and the compact scheduling affinity was used (see Figure 9). The compact strategy keeps all threads running on a single physical processor mapped one to one. Performance experiments for only 30,000 time iterations show that the scatter affinity is slower than the compact affinity by 30% to 10% (from 2 to 12 threads); therefore it is not included in the performance experiments. The compact affinity is desirable as long as all threads in the application repeatedly access different parts of a large array

```
!$acc data &
!$acc copyin(deltae,a_x,b_x,K_x) &
 !$acc local(DF1Dn,DF2Dn,DF3Dn,DF4Dn) &
...
!WHILE OF TIME
...
!$acc kernels
...
  !$acc loop independent
  DO j=2,(nj-1)
          !$acc loop independent
          DO i=2,(ni-1)
              ...
          END DO
  END DO
...
!$acc end kernels

IF (MOD(k,it_display) == 0) THEN
!$acc update host(U,V,ro,P,T,M,ET) async
CALL LAYER_WRITE(DBLE(k))
END IF

END DO ! end of while in time

!END OF WHILE

!$acc end data
```

Listing 10: Format of the OpenACC directives used to parallelize the loops.

```
DO i=2,(ni-1)
!$acc loop reduction(+:ETOT)
    DO j=2,(nj-1)
       ETOT = ETOT + (ET(i,j)*VOL(i,j));
    END DO
END DO
```

Listing 11: Format of the OpenACC directives used when a reduction is required. For the outer loop the directive acc loop has to be deleted.

(the case of the present application). When HT is enabled (see Figure 10), the scatter and the compact affinities give similar performance when the 24 kernel threads are created. However, the behavior of the performance of scatter affinity is similar to the performance with HT disabled until 12 threads, because the physical allocation is equal.

The performance experiments were carried out with vectorization of the inner loop and collapsing the loops. Sometimes the performance of nested loops can be improved by collapsing them, since this increases the total number of iterations that will be distributed over the available threads, thereby increasing the granularity. Nevertheless, for this problem the vectorization of the inner loop results in a better performance than collapsing by 5% to 7%. Vectorization is the process by which the implementation of an algorithm is converted from scalar to vectorial such that one single operation is executed over a group of contiguous values, all at the same time. In our particular case, the vectorization only applies to large floating point operations (inner loop) [23]. Thus, when the loops are collapsed, the granularity is reduced and the vectorization could not be applied; that is, by collapsing the loops, computing is insufficient for vectorization.

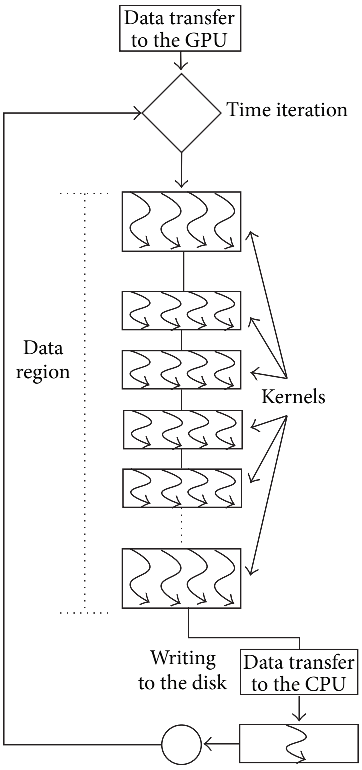


Figure 8: Design of the OpenACC for the GPU architecture.

For the handle of the boundary conditions, an improvement by 1% and 1.5% was obtained using tasks, compared with loops parallelized conventionally; thus the gain is low. About the use of a persistent parallel region the gain is by 0.5% and 1.0%. Thus, there is not a noticeable overhead when many parallel regions are used, and therefore one or many parallel regions could be used.

The best performance is gathered when the compact affinity, vectorization of the inner loop, tasks for the boundary conditions, and HT disabled were considered. The corresponding computing times with their respective speed-up factors are shown in Table 2 for 1 to 12 threads.

The behavior of the computing time with HT enabled and disabled is shown in Figures 11(a) and 11(b), respectively. We denote that there is a difference in computing time when the HT is enabled and disabled. When the HT is enabled the serial execution is slower by 20%. This result is in

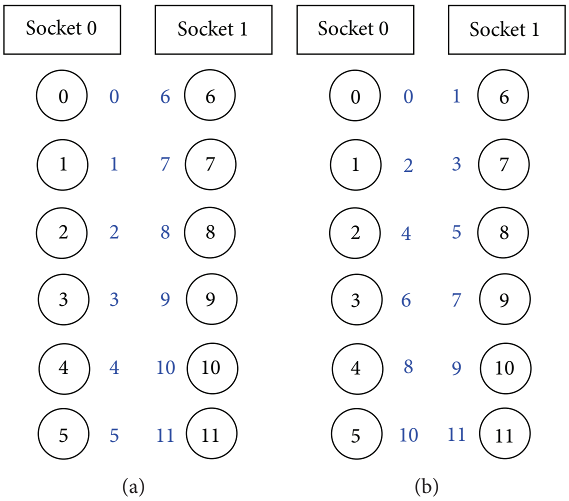


Figure 9: Affinities behavior on a SMP with 2 sockets and 6 cores by socket, with the HT disabled; (a) compact affinity, (b) scatter affinity.

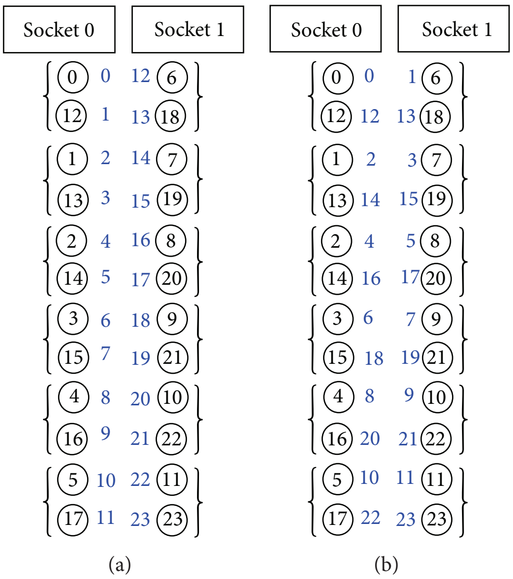
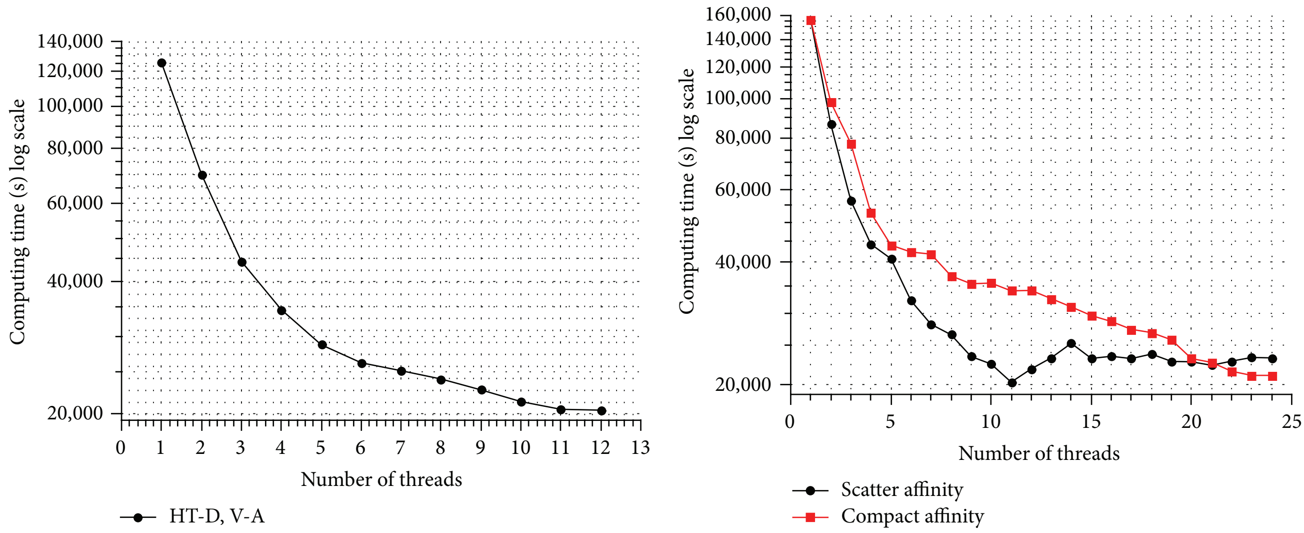


Figure 10: Affinities behavior on a SMP with 2 sockets and 6 cores by socket, with the HT enabled; (a) compact affinity, (b) scatter affinity. Curly brackets mean that the virtual cores are handled by the same physical core.

agreement with Zhang et al. [13], who found that sometimes it is better to execute an OpenMP application using only a single thread per physical processor. Leng et al. [24] also confirmed that the HT can lead to performance degradation for some HPC applications. Empirical studies showed that when HT is used for intensive floating point calculation codes, which need to share variables among the threads, an overhead can be introduced. To achieve similar performance with only one thread without the HT, it is necessary to create two threads and assign them to the same core. Nevertheless, for applications where the threads do not share the computational domain a gain in performance was observed by Couder-Castañeda et al. [25].

The graph of Figure 12(a) depicts a behavior governed by Amdahl's law with a maximum speed-up factor of 6.14X. The speed-up factors obtained let us to approximate the serial fraction of our code between 0.05 and 0.10. This also was estimated by measuring the computing time of the different parts of the code.

With HT enabled each core can handle 2 threads and reports 24 logic cores; therefore execution threads from 1 to 24 are created to analyze the performance behavior. In this way, we tried to determine if a better performance can be achieved with HT [26–29]. When HT is enabled, the best speed-up factor obtained is 6.72 for the scatter affinity and 7.40 for the compact affinity, considering as reference the computing time of the optimized serial version executed with the HT enabled. In this case, the use of HT does not increase the performance; thus the HT does not really provide a benefit for this case. This behavior seems to be normal



(a) Computing time obtained with HT disabled (12 threads)

(b) Computing time obtained with HT enabled (24 threads)

Figure 11: Computing times obtained on the Dual Xeon CPU.

Table 2: Computing times and speed-up factors obtained using OpenMP on the Xeon CPU with HT disabled. The speed-up factors are calculated with best optimized serial version as reference. With HT: hyper threading, V: vectorization, D: deactivated, and A: activated.

Threads	HT-D V-A	Speed-up
1	34 h 48 m 32 s	(1.00X)
2	19 h 26 m 58 s	(1.79X)
3	12 h 20 m 02 s	(2.82X)
4	09 h 35 m 11 s	(3.63X)
5	07 h 58 m 53 s	(4.36X)
6	07 h 15 m 54 s	(4.79X)
7	06 h 58 m 14 s	(4.99X)
8	06 h 40 m 30 s	(5.21X)
9	06 h 18 m 21 s	(5.52X)
10	05 h 55 m 56 s	(5.87X)
11	05 h 41 m 52 s	(6.11X)
12	05 h 40 m 23 s	(6.14X)

because the application is floating point intensive and two threads are sharing the same core FPU (floating point unit).

These results showed in Table 3 emphasize the effects of the vectorization which works correctly when the number of floating point operations by core is sufficient and can be vectorized.

3.2. Experiments on the Xeon Phi. The code with OpenMP directives was also tested for performance on the Xeon Phi (model 5110P) with the following general characteristics:

(i) 60 active cores (each core can handle 4 threads),

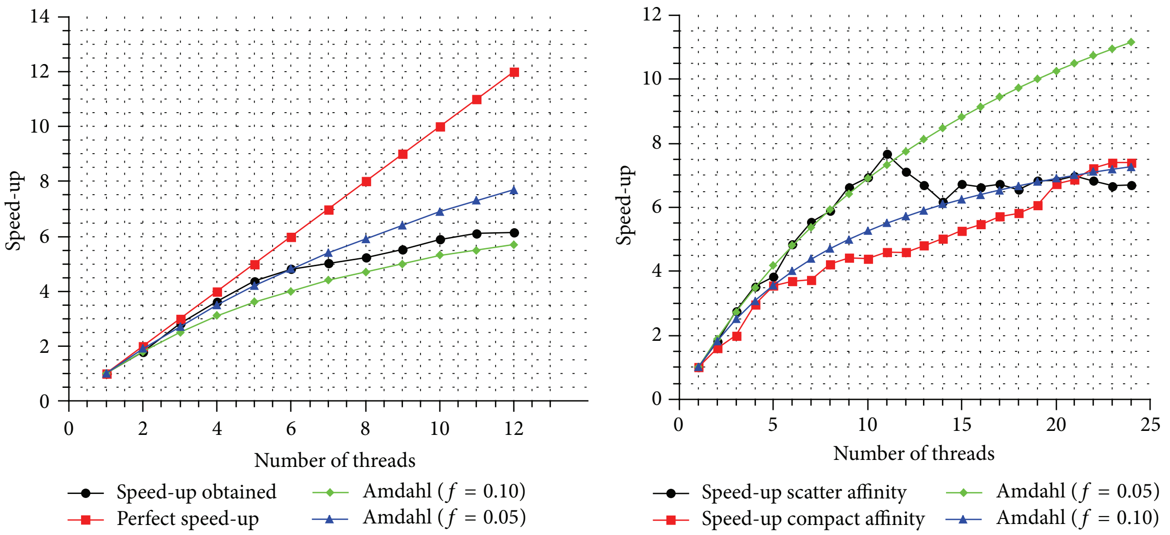
(ii) frequency 1 GHz,

Table 3: Computing times and their corresponding speed-up factors obtained with HT enabled for the compact and scatter affinities. HT: hyper threading, V: vectorization, and A: activated.

Threads	HT-A, V-A, compact	HT-A V-A, scatter
1	43 h 22 m 22 s (1.00X)	43 h 19 m 58 s (1.00X)
2	27 h 17 m 46 s (1.59X)	24 h 05 m 16 s (1.80X)
3	21 h 38 m 07 (2.00X)	15 h 41 m 05 (2.76X)
4	14 h 37 m 51 s (2.96X)	12 h 15 m 17 s (3.54X)
5	12 h 12 m 27 s (3.55X)	11 h 18 m 27 s (3.83X)
6	11 h 44 m 25 s (3.69X)	08 h 55 m 26 s (4.86X)
7	11 h 36 m 13 s (3.74X)	07 h 46 m 44 s (5.57X)
8	10 h 15 m 49 s (4.23X)	07 h 21 m 59 s (5.88X)
9	09 h 49 m 17 s (4.42X)	06 h 31 m 26 s (6.64X)
10	9 h 51 m 28 s (4.40X)	06 h 16 m 28 s (6.91X)
11	09 h 26 m 09 s (4.60X)	05 h 33 m 41 s (7.79X)
12	09 h 27 m 33 s (4.59X)	06 h 03 m 58 s (7.14X)
13	09 h 01 m 54 s (4.8X)	06 h 29 m 24 s (6.68X)
14	08 h 37 m 13 s (5.03X)	07 h 01 m 00 s (6.18X)
15	08 h 12 m 43 s (5.28X)	06 h 27 m 47 s (6.70X)
16	07 h 56 m 28 s (5.46X)	06 h 31 m 54 s (6.63X)
17	07 h 35 m 52 s (5.71X)	06 h 26 m 55 s (6.72X)
18	07 h 27 m 38 s (5.81X)	06 h 35 m 04 s (6.58X)
19	07 h 09 m 47 s (6.06X)	06 h 20 m 00 s (6.84X)
20	06 h 26 m 38 s (6.73X)	06 h 19 m 51 s (6.84X)
21	06 h 18 m 32 s (6.87X)	06 h 12 m 58 s (6.97X)
22	06 h 00 m 01 s (7.23X)	06 h 20 m 40 s (6.83X)
23	05 h 51 m 48 s (7.40X)	06 h 24 m 12 s (6.77X)
24	05 h 51 m 49 s (7.40X)	06 h 27 m 04 s (6.72X)

(iii) 8 GB of DDR5 memory,

(iv) up to one theoretical Teraflop in double floating point precision.



(a) Speed-up factor estimated using the computing times showed in Figure 11(a)

(b) Speed-up factor estimated using the computing times showed in Figure 11(b)

FIGURE 12: Comparison of the obtained speed-up factors versus Amdahl's law with a serial fraction of $f = 0.05$ and $f = 0.10$.

The first consideration regarding this implementation is that the Xeon Phi coprocessor supports 4 threads per processing core; therefore the optimum number of execution threads per core is necessary to be determined. Generally, this optimal number of threads greatly depends on the algorithm and the memory management within the application [30, 31]. Moreover, on the Phi architecture, more than one thread per core helps to hide the inherent latencies in the applications. For example, while one thread awaits its memory resource assignment, other threads could be scheduled on the core. On the conventional Xeon CPU architecture, various developers have found that in general HPC applications do not benefit from the HT technology [32]. Nevertheless, this is not applicable to the Phi technology where the multithreading (MT) can not be disabled as the HT can be disabled via BIOS.

The Xeon Phi could be used in native or `offload` mode. In native mode the Phi could be seen as stand-alone computer and the applications can run directly on the coprocessor. In the `offload` mode the Phi operates as a slave computer and the information and control are transferred and handled by the hosts. In the present case, the Phi is used in the same manner as a conventional GPU.

When the Xeon Phi coprocessor is used in `offload` mode, the method to observe the performance begins with the creation of different numbers of threads from $(n - 1)$ up to $4 \times (n - 1)$, where n is the number of physical cores in the Phi. Then, as recommended by Intel, four experiments should be executed creating $(n - 1)$, $2 \times (n - 1)$, $3 \times (n - 1)$, and $4 \times (n - 1)$ threads, respectively. This serves to determine if the increasing number of threads per core improves the performance. Multiples of $(n - 1)$ are employed instead of multiples of n because one core is left available for the operating system services. Therefore, we conducted the experiments using 59, 118, 177, and 236 execution threads on the Xeon Phi with a balanced affinity, since each core can handle up to 4 threads. Regarding the vectorization, the `$SIMD` directive was required to vectorize the internal loop. Table 4 reports the results obtained in execution time of the different configurations.

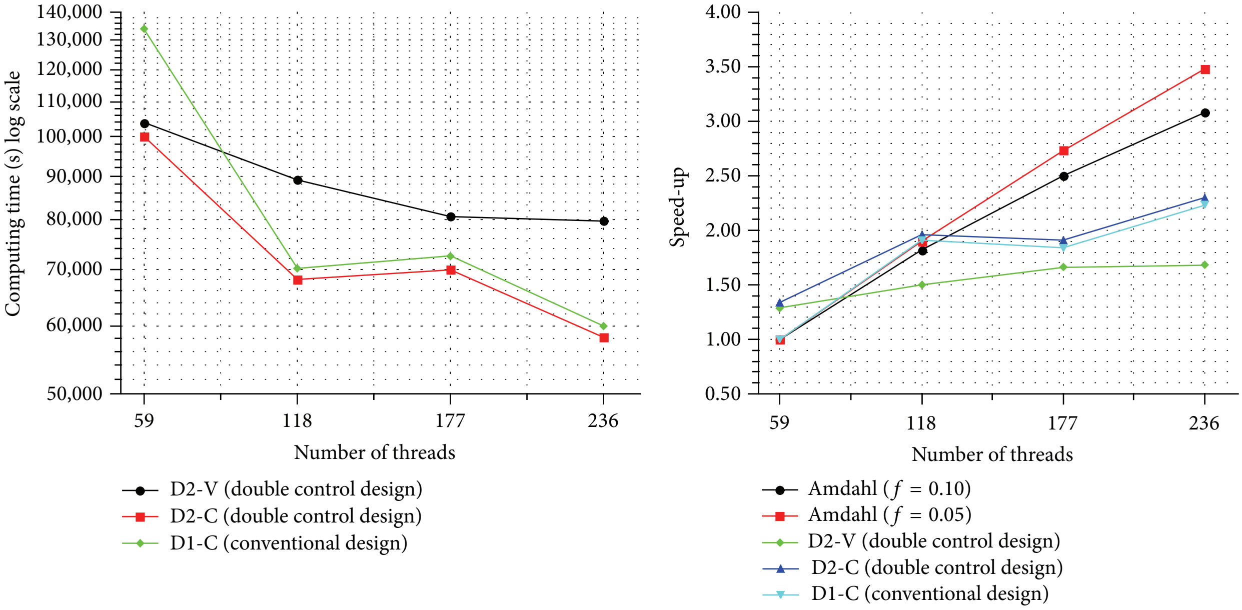
Figure 13 shows the graphs which correspond to Table 4. In this case, the condition of 59 execution threads was considered as reference to calculate the speed-up factors. If the conventional design D1 is used, the number of threads is increased from 59 up to 236 threads, since 4 threads per core is the optimal number handled by the Phi cores. The experiments were conducted in a similar way for the collapsed design and the vectorized design D2.

From Figure 13, the best speed-up factor is reached by the collapsed design D2, as expected, since the times the parallel region has created were reduced. With this, the improvement in performance is between 0.05X and 0.34X. This result suggests that the overload for creating one parallel region on the coprocessor can be considered lightweight, but when several parallel regions are created (millions) this can cause overhead for this case by 2%. If this design is implemented in algorithms where the number of iterations is much higher [33], the benefit would be even more evident, for example, wave-propagation models.

The maximum speed-up factor obtained when using four threads per core is 2.30X, in reference with the one-thread-per-core design on the Phi and 4.30X in reference with the serial version on the CPU. This is a result of the low number of data. Effects of vectorization are not denoted in the performance, because the data handled by the internal loops are not too many. Moreover, the number of data is not sufficient to continue scaling. For this reason, in contrast with the experiments carried out on the conventional CPU, the best performance is shown by collapsing the loops without vectorization. Table 5 compares the speed-up factors for the MIC

Table 4: Computing times and the corresponding speed-up factors obtained by using OpenMP mounted on the Xeon Phi coprocessor. D1: design 1, D2: design 2, C: collapsed, and V: vectorized.

Threads	D1-C	D2-C	D2-V
59	37 h 07 m 47 s (1.00X)	27 h 47 m 51 s (1.34X)	28 h 52 m 20 s (1.29X)
118	18 h 54 m 38 s (1.91X)	18 h 54 m 38 s (1.96X)	24 h 43 m 53 s (1.50X)
177	20 h 09 m 13 s (1.84X)	19 h 24 m 13 s (1.91X)	22 h 24 m 06 s (1.66X)
236	16 h 40 m 19 s (2.23X)	16 h 10 m 10 s (2.30X)	22 h 07 m 53 s (1.68X)



(a) Computing time

(b) Speed-up

Figure 13: Computing times and the corresponding speed-up factors compared to Amdahl's law with $f = 0.05$ and $f = 0.10$. As can be seen, the Xeon Phi stops scaling in two threads per core for all the proposed designs (for this domain size). Unlike the conventional CPU the Phi does not show improvements with the vectorization, due to the decrease of vectorization effects with many cores; that is, there are no sufficient floating point operations per core.

against the other architectures in their best version regarding the performance.

3.3. Experiments on the GPU. The GPU experiments were carried out using a Tesla C2070 card with the following characteristics:

(i) 14 multiprocessors (SM),

(ii) 32 cores per multiprocesador (448 GPU cores total),

(iii) 6 GB of DDR5 global memory,

(iv) up to 515 theoretical Gflops in double floating point precision,

(v) up to one theoretical Teraflop in single floating point precision,

(vi) 1.15 GHz frequency of the GPU cores.

The experiment used the PGI version 15.3 64-bit compiler with the compilation flags: `-Minline -Minfo=acc, accel, intensity -acc -fast`. The execution time obtained was `3 h 31 m 47 s`, which is the best execution time reached for all architectures. Table 6 compares the speed-up factors for the C2070 against the other architectures in their best version regarding the performance.

4. Validation of the Code Porting

The level of decomposition (granularity) is highly influenced by the type of architecture employed. This work employed three different architectures and two programming paradigms to obtain the maximum performance from each platform. Moreover, the validation of the code is a work that must be addressed as well, since inherent errors can occur during program development.

The supersonic ejector flow is a very complex problem due to the physics of the flow and the geometry of the ejector. A transformation of grid coordinates has to be used to characterize the expansion section of the diffuser. Thus, an appropriate transformation to generate a boundary-fitted coordinate system was implemented and validated [9]. On the other hand, as the flow is supersonic, pressure waves are present on the flow and have to be absorbed at the open

Table 5: Comparison of the speed-up factors taking as reference the MIC.

MIC	Computing time	Versus CPU serial vectorized without HT	Versus CPU (HT) OpenMP vectorized	GPU C2070
5110P	16 h 10 m 10 s (1.0X)	2.15X	−3.5X	−4.5X

Table 6: Comparison of the speed-up factors taking as reference the GPU.

GPU	Computing time	Versus CPU serial vectorized without HT	Versus CPU (HT) OpenMP vectorized	Versus MIC
C2070	03 h 31 m 47 s (1.0X)	9.86X	1.6X	4.5X

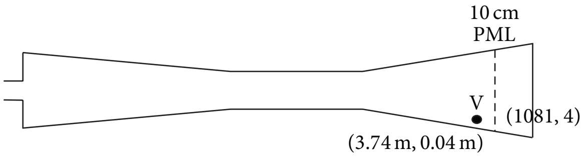


Figure 14: Location of the numerical visor near the outflow region. The thickness of the CPML absorbing zone is 10 cm (10 × Δx).

boundaries, which is not trivial. For this reason, an unsplit convolutional PML boundary condition was developed to absorb pressure waves [10]. Moreover, the numerical scheme considers Direct Numerical Simulation (DNS), which implies high computational cost even at low Reynolds numbers. This is one of the reasons that an algorithm with a high level of optimization must be used for this problem.

Time series signals of the primitive flow variables were stored to analyze the quality of the calculations. For this reason, a numerical visor was located near one of the sloped walls, where the diffuser section is expanded (see Figure 14). The visor was located at the near flow region walls, a critical region where distortions of the flow structure could appear due to the presence of spurious fluxes coming from the open boundary. The simulations were carried out over 3 millions of iterations and the data were stored each 30,000 iterations, thereby generating 100 data points per primitive flow variable. The 10 cm zone at the end of the ejector is highlighted, where a Convolutional Perfectly Matched Layer (CPML) of the open boundary is employed to absorb pressure waves.

A numerical simulation using a serial processing of one core provided the reference solution to be compared with the parallel solution of the same core. To compare the serial and the parallel solutions, the computed total energy of the system (E_t) was analyzed, since this physical quantity integrates all the primitive variables as follows:

$$E_t = \frac{1}{2}\left(u^2 + v^2\right) + \frac{1}{\gamma - 1}\frac{p}{\rho}, \qquad (1)$$

where u and v are the vector velocity components in the x and y directions, respectively, p is the pressure, ρ is the density of the fluid, and γ is the adiabatic dilation coefficient. The first term of the right hand of (1) represents the kinetic energy per unit of mass and the second term represents the internal energy per unit of mass.

In addition, the computed Mach number (M) from both solutions was also compared, because it characterizes the flow under consideration.

Table 7: Total energy and Mach number obtained in the visor V for all three architectures at the end of the 3 s of simulation period.

	E_t ($\mathrm{m}^2/\mathrm{s}^2$)	M
Reference	3366417302.615026	1.644826
GPU	3366759849.759678	1.644971
MIC	3367625209.964143	1.643896
CPU multicore	3366417302.615055	1.644826

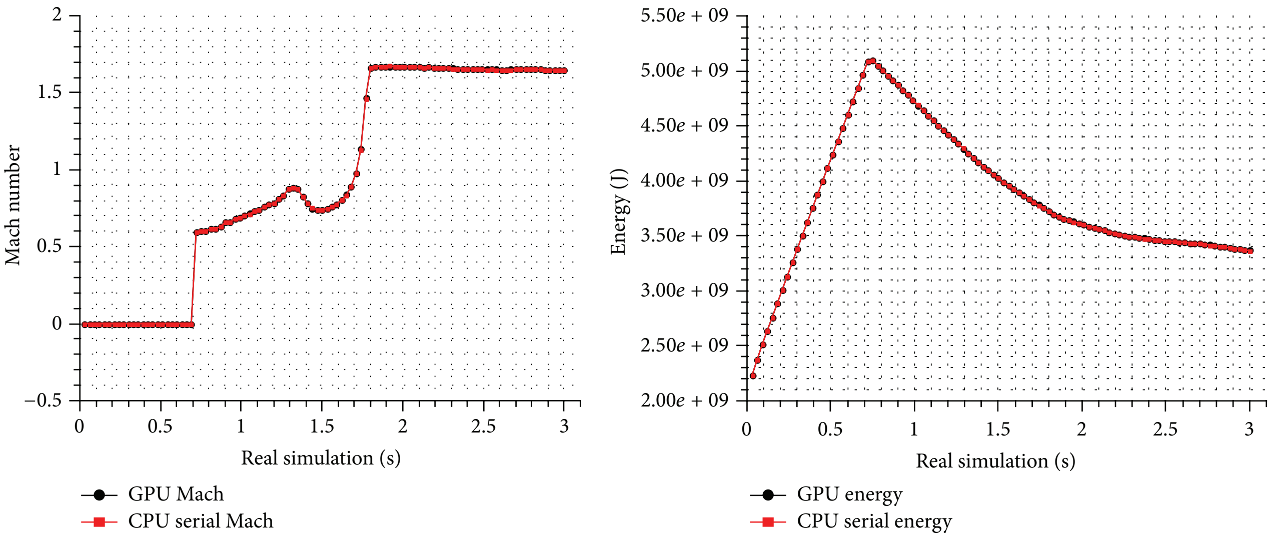
Table 8: Relative percentage error for total energy and for Mach number obtained in the visor V for all three architectures at the end of the 3 s simulation period.

	Error for E_t (%)	Error for M (%)
GPU	1.01754×10^{-4}	8.81552×10^{-5}
MIC	3.58811×10^{-4}	5.65409×10^{-4}
CPU	8.92365×10^{-15}	0.00000

For the sake of clarity, only the results of the experiments which reached the best performance from the GPU and MIC platforms are discussed (see Figures 15 and 16). The curves of Figures 15 and 16 together with the calculated correlation coefficients show good agreement between the porting parallel solutions and the serial reference solution. However, we observe small differences among the results of the platforms. Even within the same platforms where we introduced processing threads, variations are observed.

The numerical algorithm must be robust and stable enough during long periods of time in order to demonstrate numerical stability. Table 7 shows the differences between the magnitudes of total energy and Mach number computed in the visor V at the end of the 3 s of simulation, for all three architectures. The corresponding relative percentage errors are also shown in Table 8 regarding the reference serial solution. It can be observed that the CPU multicore architecture provides the most accurate solution regarding the reference serial solution, where the error is negligible for both the total energy and the Mach number. For the GPU and MIC architectures, the MIC has the greatest error for both the total energy (3.58811×10^{-4}) and the Mach number (5.65409×10^{-4}).

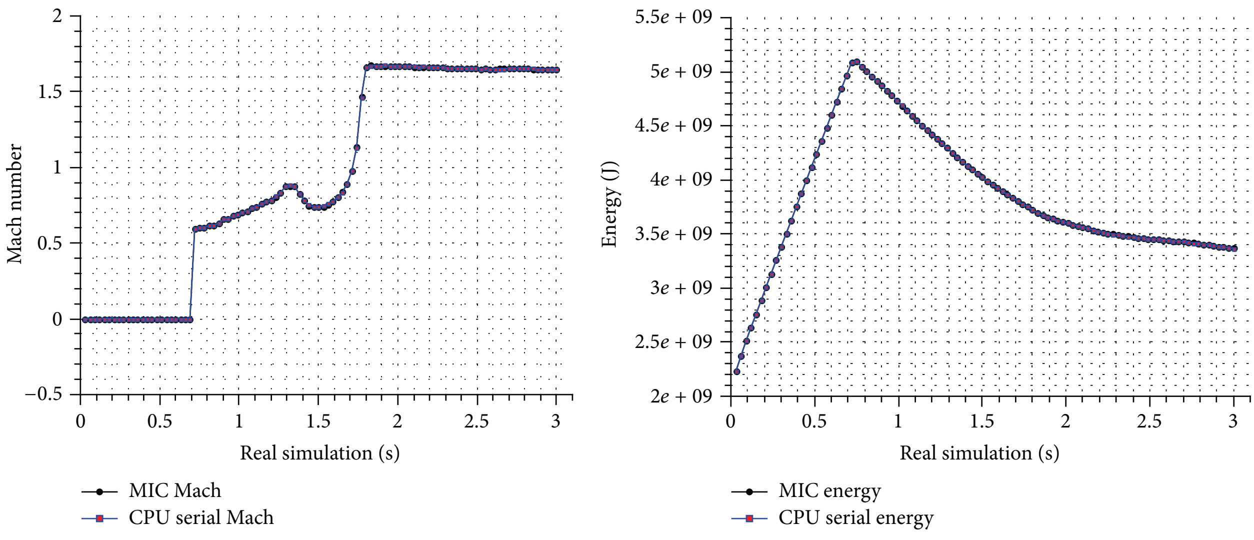
Figure 17 shows different snapshots of the Mach number in the diffuser of the ejector at different times. Patterns were obtained on the GPU C2070. Snapshots of the Mach number in the ejector diffuser are reproduced at 0.75 s, 1.5 s, 2.25 s, and 3 s of real time. The contour levels show the instants when the compression, transfer, and expansion occur in the diffuser.



(a) Mach number, correlation coefficient = 0.999999483128566

(b) Total energy, correlation coefficient = 0.999998594006821

Figure 15: Comparison of the total energy and the Mach number of the visor V. Simulations on GPU compared against the serial reference solution.



(a) Mach number, correlation coefficient = 0.999998575641890

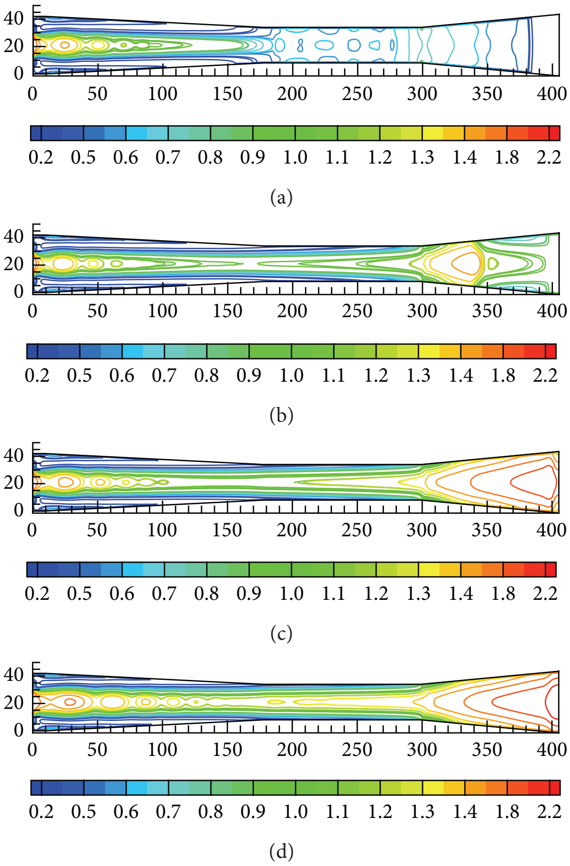
(b) Total energy, correlation coefficient = 0.999999702775720

Figure 16: Comparison of the total energy and the Mach number of the visor V. Simulations on MIC running with the design D2 at 236 threads compared against the serial reference solution.

5. Conclusions

The need of reducing computing time of scientific applications led to the development of accelerators and coprocessors to meet this demand. There are now tools based on directives as OpenACC and OpenMP that reduce the arduous task of low-level programming. However, in spite of the fact that a methodology based on directives reduces the programming work, it is necessary to explore different design strategies to take advantage of the maximum potential of the available architectures. The application ported in this work was a serial source code for a supersonic ejector flow. This work showed the effectiveness of OpenMP, by reducing the computing time in very good agreement with Amdahl's law. Regarding OpenACC, the well-known strategy that could be applied is to maintain the data regions active among calls to the kernels, thereby avoiding the data transfer between CPU and GPU. Similarly, the persistent variables are maintained when working on the MIC in offload mode. The overload, caused by adding directives in OpenMP for the parallel programming on CPU and on the MIC, is relatively low and does not limit scalability.

In terms of programmability and productivity, we show that OpenMP and OpenACC are viable alternatives for



(a)

(b)

(c)

(d)

Figure 17: Snapshots of the Mach number obtained at different times: (a) 0.75 s, (b) 1.5 s, (c) 2.25 s, and (d) 3.0 s.

the development of parallel codes for scientific applications. The expected performance is satisfactory, considering that OpenMP and OpenACC are paradigms that express parallelism and the parallel code is generated by the compiler, which finally implies less development time with an acceptable performance.

6. Future Work

As future work, OpenMP and OpenACC can be mixed in order to port the application to a multi-GPU or multicoprocessor architecture integrated in the same node.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work has been supported in part by the project ABACUS-CONACyT under Grant no. EDOMEX-2011-C01-165873. The authors would like to thank the CNS (National Supercomputing Center, http://www.cns-ipicyt.mx/) for the facilities given for using the Xeon Phi coprocessor.

References

[1] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Longman, Boston, Mass, USA, 1995.

[2] C. Calvin, F. Ye, and S. Petiton, "The exploration of pervasive and fine-grained parallel model applied on Intel Xeon Phi coprocessor," in *Proceedings of the 8th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC '13)*, pp. 166–173, IEEE, Compiègne, France, October 2013.

[3] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[4] M. Curtis-Maury, X. Ding, C. D. Antonopoulos, and D. S. Nikolopoulos, "An evaluation of openmp on current and emerging multithreaded/multicore processors," in *OpenMP Shared Memory Parallel Programming: Proceedings of the International Workshops, IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1–4, 2005, Reims, France, June 12–15, 2006*, vol. 4315 of *Lecture Notes in Computer Science*, pp. 133–144, Springer, Berlin, Germany, 2008.

[5] H. Brunst and B. Mohr, "Performance analysis of large-scale openmp and hybrid mpi/openmp applications with vampir ng," in *OpenMP Shared Memory Parallel Programming*, M. Mueller, B. Chapman, B. Supinski, A. Malony, and M. Voss, Eds., vol. 4315 of *Lecture Notes in Computer Science*, pp. 5–14, Springer, Berlin, Germany, 2008.

[6] A. Amritkar, S. Deb, and D. Tafti, "Efficient parallel CFD-DEM simulations using OpenMP," *Journal of Computational Physics*, vol. 256, pp. 501–519, 2014.

[7] B. P. Pickering, C. W. Jackson, T. R. Scogland, W.-C. Feng, and C. J. Roy, "Directive-based GPU programming for computational fluid dynamics," *Computers & Fluids*, vol. 114, pp. 242–253, 2015.

[8] A. Kucher and G. Haase, "Many-core sustainability by pragma directives," in *Large-Scale Scientific Computing*, vol. 8353 of *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pp. 448–456, Springer, 2014.

[9] C. Couder-Castañeda, "Simulation of supersonic flow in an ejector diffuser using the jpvm," *Journal of Applied Mathematics*, vol. 2009, Article ID 497013, 21 pages, 2009.

[10] R. Martin and C. Couder-Castaneda, "An improved unsplit and convolutional perfectly matched layer absorbing technique for the navier-stokes equations using cut-off frequency shift," *Computer Modeling in Engineering and Sciences*, vol. 63, no. 1, pp. 47–77, 2010.

[11] J. Levesque and G. Wagenbreth, *High Performance Computing: Programming and Applications*, CRC Press, 2010.

[12] Z. Krpic, G. Martinovic, and I. Crnkovic, "Green HPC: MPI vs. OpenMP on a shared memory system," in *Proceedings of the 35th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO '12)*, pp. 246–250, 2012.

[13] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss, "An adaptive OpenMP loop scheduler for hyperthreaded SMPs," in *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS '04)*, 2004.

[14] R. Martin, D. Komatitsch, and A. Ezziani, "An unsplit convolutional perfectly matched layer improved at grazing incidence for seismic wave propagation in poroelastic media," *Geophysics*, vol. 73, no. 4, pp. T51–T61, 2008.

[15] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *Journal of Computational Physics*, vol. 229, no. 20, pp. 7692–7714, 2010.

[16] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "Openmp programming on intel r xeon phi tm coprocessors: An early performance comparison," 2012.

[17] J. Zhou, D. Unat, D. J. Choi, C. C. Guest, and Y. Cui, "Hands-on performance tuning of 3D finite difference earthquake simulation on GPU fermi chipset," in *Proceedings of the 12th International Conference on Computational Science (ICCS '12)*, pp. 976–985, June 2012.

[18] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2 '09)*, pp. 79–84, ACM, New York, NY, USA, March 2009.

[19] D. Michéa and D. Komatitsch, "Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards," *Geophysical Journal International*, vol. 182, no. 1, pp. 389–402, 2010.

[20] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. Muller, "Assessing the performance of OpenMP programs on the intel xeon phi," in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds., vol. 8097 of *Lecture Notes in Computer Science*, pp. 547–558, Springer, Berlin, Germany, 2013.

[21] S. Wienke, C. Terboven, J. C. Beyer, and M. S. Müller, "A pattern-based comparison of OpenACC and OpenMP for accelerator computing," in *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds., vol. 8632 of *Lecture Notes in Computer Science*, pp. 812–823, Springer International Publishing, Cham, Switzerland, 2014.

[22] D. K. Ojha and G. Sikka, "A study on vectorization methods for multicore SIMD architecture provided by compilers," in *ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of Computer Society of India- Vol I*, vol. 248 of *Advances in Intelligent Systems and Computing*, pp. 723–728, Springer, 2014.

[23] J. Francés, S. Bleda, A. Márquez et al., "Performance analysis of SSE and AVX instructions in multi-core CPUs and GPU computing on FDTD scheme for solid and fluid vibration problems," *The Journal of Supercomputing*, vol. 70, no. 2, pp. 1–13, 2013.

[24] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," in *Proceedings of the Linux HPC Revolution Conference*, 2002.

[25] C. Couder-Castañeda, J. C. Ortiz-Alemán, M. G. Orozco-del-Castillo, and M. Nava-Flores, "Forward modeling of gravitational fields on hybrid multi-threaded cluster," *Geofísica Internacional*, vol. 54, no. 1, pp. 31–48, 2015.

[26] U. Ranok, S. Kittitornkun, and S. Tongsima, "A multithreading methodology with OpenMP on multi-core CPUs: SNPHAP case study," in *Proceedings of the 8th Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI '11)*, pp. 459–463, IEEE, Khon Kaen, Thailand, May 2011.

[27] J. H. Abdel-Qader and R. S. Walker, "Performance evaluation of OpenMP benchmarks on Intel's quad core processors," in *Proceedings of the 14th WSEAS International Conference on Computers*, vol. 1, pp. 348–355, July 2010.

[28] W. Zhong, G. Altun, X. Tian, R. Harrison, P. C. Tai, and Y. Pan, "Parallel protein secondary structure prediction schemes using Pthread and OpenMP over hyper-threading technology," *Journal of Supercomputing*, vol. 41, no. 1, pp. 1–16, 2007.

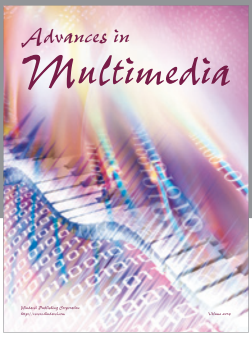
[29] G. Bernabé, R. Fernández, J. M. García, M. E. Acacio, and J. González, "An efficient implementation of a 3D wavelet transform based encoder on hyper-threading technology," *Parallel Computing*, vol. 33, no. 1, pp. 54–72, 2007.

[30] A. E. Eichenberger, C. Terboven, M. Wong, and D. an Mey, "The design of openmp thread affinity," in *OpenMP in a Heterogeneous World*, vol. 7312 of *Lecture Notes in Computer Science*, pp. 15–28, Springer, Berlin, Germany, 2012.

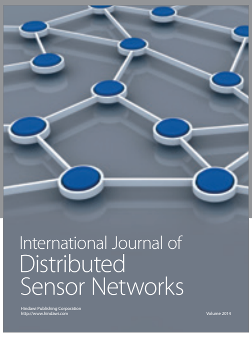
[31] C. Allande, J. Jorba, A. Sikora, and E. César, "A performance model for openMP memory bound applications in multisocket systems," *Procedia Computer Science*, vol. 29, pp. 2208–2218, 2014.

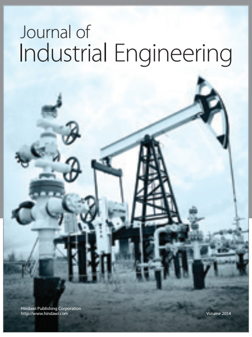
[32] P. Gepner, M. F. Kowalik, D. L. Fraser, and K. Waćkowski, "Early performance evaluation of new six-core intel xeon 5600 family processors for HPC," in *Proceedings of the 9th International Symposium on Parallel and Distributed Computing (ISPDC '10)*, pp. 117–124, July 2010.

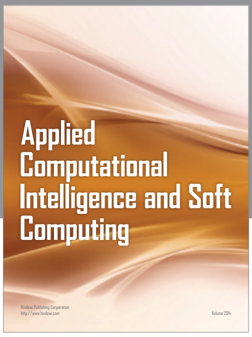
[33] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J. M. Cela, and D. P. Scarpazza, "3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors," *Scientific Programming*, vol. 17, no. 1-2, pp. 185–198, 2009.

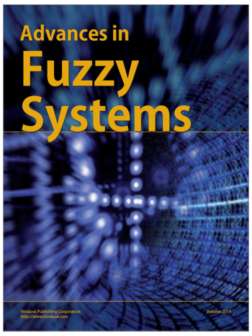

Advances in
Multimedia

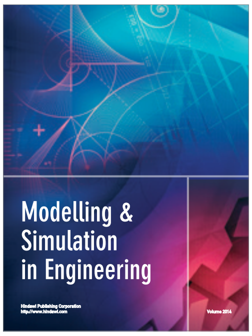
The Scientific
World Journal

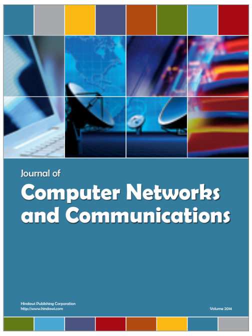

International Journal of
Distributed
Sensor Networks

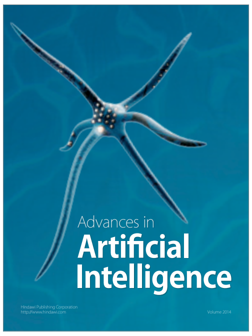

Journal of
Industrial Engineering

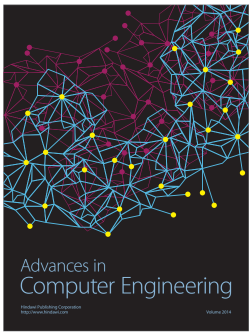

Applied
Computational
Intelligence and Soft
Computing


Advances in
Fuzzy
Systems

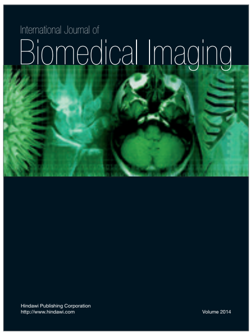

Modelling &
Simulation
in Engineering

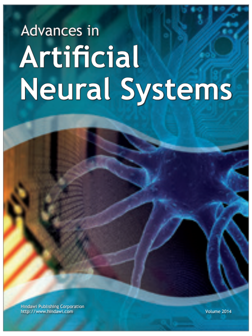

Journal of
Computer Networks
and Communications

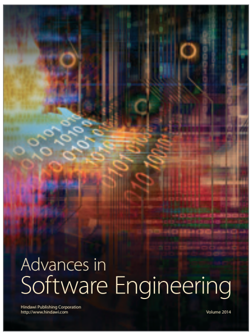

Advances in
Artificial
Intelligence


Advances in
Computer Engineering

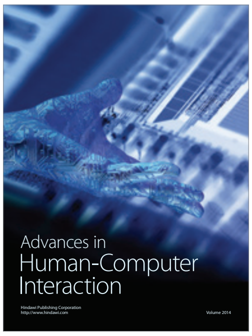
International Journal of
Computer Games
Technology

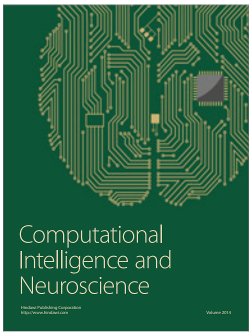

International Journal of
Biomedical Imaging


Advances in
Artificial
Neural Systems


Advances in
Software Engineering

Journal of
Robotics


Advances in
Human-Computer
Interaction


Computational
Intelligence and
Neuroscience

International Journal of
Reconfigurable
Computing

Journal of
Electrical and Computer
Engineering