An Evaluation of Direct and Indirect Memory Accesses in Fluid Flow Simulator

Stiw Harrison Herrera Taipe[1(✉)], Thiago Teixeira[1], Weber Ribeiro[1], Andre Carneiro[1], Marcio R. Borges[1], Carla Osthoff[1], F. L. Cabral[1], and S. L. Gonzaga de Oliveira[2]

[1] National Laboratory for Scientific Computing (LNCC), MCTIC, Av. Getulio Vargas, 333 - Quitandinha, Petrópolis, RJ 25651-076, Brazil
{stiw,tteixeira,webergdr,andrerc,mrborges,osthoff,fcabral}@lncc.br
[2] Universidade Federal de São Paulo (UNIFESP), Av. Cesare Monsueto Giulio Lattes, 1201 - Eugênio de Melo, São José dos Campos - SP, São José dos Campos, SP 12247-014, Brazil
sanderson.oliveira@unifesp.br
https://www.gov.br/lncc/pt-br, https://www.unifesp.br/

Abstract. In petroleum reservoir simulations, the level of detail incorporated into the geologic model typically exceeds the capabilities of traditional flow simulators. In this sense, such simulations demand new high-performance computing techniques to deal with a large amount of data allocation and the high computational cost of computing the behavior of the fluids in the porous media. This paper presents optimizations performed on a code that implements an explicit numerical scheme to provide an approximate solution to the governing differential equation for water saturation in a two-phase flow problem with heterogeneous permeability and porosity fields. The experiments were performed on the SDumont Supercomputer using 2nd Generation Intel®Xeon®Scalable Processors (formerly Cascade Lake architecture). The paper employs a direct memory data access scheme to reduce the execution times of the numerical method. The article analyzes the performance gain using direct memory access related to indirect access memory. The results show that the optimizations implemented in the numerical code remarkably reduce the execution time of the simulations.

Keywords: Transport in Porous Media · Biphasic Flow · Data Structure · Performance

1 Introduction

Numerical simulations of gas and oil reservoirs use mathematical models to describe the physical processes of fluid flows in a porous medium. Natural porous media, such as aquifers and reservoirs, present a high degree of heterogeneities

Supported by organization LNCC/SEPAD.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
O. Gervasi et al. (Eds.): ICCSA 2023, LNCS 13956, pp. 38–54, 2023.
https://doi.org/10.1007/978-3-031-36805-9_3

in their petrophysical properties (porosity, permeability, etc.). These heterogeneities extend from the pore scale to the field scale, and governing equations incorporate them for flow problems based on geostatistical models [2,6,9,10].

The variability and complex connectivity of the petrophysical properties affect the subsurface flow and transport behavior, and simulations of realistic problems must include these characteristics [11]. For simulations in this field of research, one needs to discretize huge domains in meshes that employ very refined polytopes. Consequently, one faces a very demanding computational problem that needs high-performance computing to be approximately solved.

The geometric representation of the reservoirs, which can present complex shapes, stimulates the adoption of irregular meshes in their discretization. On the other hand, despite less fidelity to geometric representation, applying regular grids to discretize the domains allows more simplified and faster calculations. Additionally, the resolution of the problem investigated in this study is highly scalable. Thus, it is possible to compensate for the loss of precision by increasing the number of polytopes in the mesh. The scalability of the code also allows us to refine the mesh until reaching reasonable accuracy of the solution, thus compensating for the considerable numerical diffusion present in first-order upwind schemes. Furthermore, one can incorporate a hyperbolic conservation nature law to design better approximate code solutions [26]. In future it will be very interesting to see the comparison them on another architectures.

Domain decomposition techniques divide a domain into subdomains. Thus, one can use Message Passing Interface (MPI) to calculate them in parallel [25]. The number of domain divisions depends on computational and communication costs. The higher the computational cost of the method, the greater the number of possible domain partitions. The higher the communication cost generated by the method stencil, the lower the number of subdomain partitions. In a previous paper [11], we presented a domain decomposition technique implementation based on block decomposition [18]. Specifically, we used the scheme to test different vertical and horizontal cross-sections to evaluate the best block decomposition given a certain number of processes. The scheme presented satisfactory performance.

Frequently, in numerical method codes for two-phase fluids flow in porous media, lines in the input file consist of the finite component and its neighbors [5]. Thus, when loading the file, the project needed another data structure to reference the location of the neighbors, resulting in an indirect access memory scheme [5]. The results from a previous publication [27] showed that, after the domain technique implementation, the computational environment performance bottleneck was, not surprisingly, the memory access time due to the indirect memory access in the data structure.

The present paper employs a data structure to provide direct memory access for the numerical fluid-flow method aiming to reduce memory access time and, consequently, improve code performance. We organized the manuscript as follows. In Sect. 2, we present related work. Afterward, in Sect. 3, we describe the mathematical model. In Sect. 3, the paper shows the approaches of the new data

structure strategy. In Sect. 4, we describe the tests performed. Finally, in Sect. 5, the conclusion and future work are presented.

2 Contextualization and Related Work

One can classify memory access into three types: direct, indirect, and semi-direct. One uses indirect access in applications that access irregular meshes and present unsatisfactory performance in data access due to the high cache miss rate. Semi-direct access occurs when a part of the data is available directly in the array, whereas the application accesses another part through a function. This form can reduce the cache miss or help in finding the necessary data [15]. Herschlag et al. [12] compared these forms of data access. Clearly, the data structure and the mode to access data are critical for the performance of an application. Memory access can vary depending on the purpose of the application. Although direct access shows better results than the other accesses, semi-direct access can be more advantageous depending on the number of variables used in the calculations. The difference between these accesses is how the process accesses data. In direct access, the application accesses data directly in the array without using an index to inform the memory region where the data is. This form of access presents better performance when avoiding cache misses. However, it may not be feasible in irregular meshes, where it is not trivial to identify which data to use without employing a function to fetch data. In indirect access, the application uses a procedure to identify the location of the data (e.g., [8,17,21]. Although necessary in some applications, this type of access can generate a cache miss. Semi-direct access occurs when part of the data is available directly in the array, and the application needs to fetch only portion of the data through a function. This form of access can reduce the cache miss and help in the search for the data (e.g., [15]).

Domain decomposition techniques divide a domain into subdomains. Thus, they can be calculated in parallel using Message Passing Interface (MPI). The number of domain divisions depends on computational and communication costs. The higher the computational cost of the method, the greater the number of possible domain partitions.

Previous work [11] employed the domain decomposition method to split the 3-D reservoir onto n sub-domains that are solved using each Message Passing Interface (MPI) process. The paper analyzed the performance of the domain decomposition strategies and identified communication bottlenecks as the mesh size and the number of computational nodes increased. Results show that the optimizations implemented in the numerical code remarkably reduced the execution time of the simulations.

In the context of three-dimensional meshes, Shashkov and Steinberg [22] and Silvester and Ferrari [23] describe how to work with oil reservoir domains using regular meshes. In this scenario, the application access each control volume using only the indices (i, j, k) of a three-dimensional matrix (e.g., [4,28]). The idea is to solve the partial differential equation by discretizing the computational domain through a Cartesian mesh.

3 Numerical Fluid-Flow Simulator Data Structure

The fluid flow simulator used in this work consists of two primary modules [3]: the velocity transport modules. They are responsible for solving the system of nonlinear partial differential equations. Furthermore, the modules consider the pressure, Darcy velocity, and water saturation [26]. This paper adopted the IMPES methodology (implicit in pressure and explicit in saturation) to solve the decoupled systems of partial differential equations [1,3,26]. Briefly, the simulation uses two numerical methods. They interact with each other, as illustrated in Fig. 1. The importance of working with subsystems instead of operating with the complete system of partial differential equations is to make it possible to use suitable numerical methods for each one.

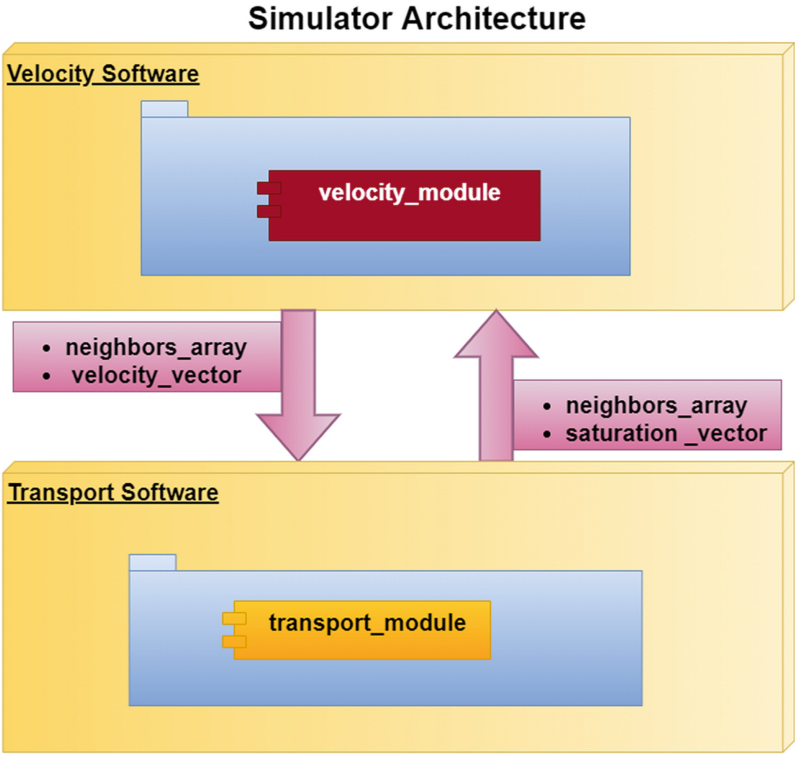


Fig. 1. Simulator architecture with velocity and transport modules

The numerical fluid flow simulator uses data from the computational grid with information on the parameters and domains of the reservoir. The velocity and transport modules perform the computations using the respective initial and boundary conditions [3]. Initially, the velocity module uses a finite element method to find the velocity field. Subsequently, the module sends the results

to the transport module through an array that contains information about the velocities and locations of neighboring volumes of each control volume. Then the transport module receives this information to find the water saturation. The transport module sends back the new results to the velocity module to start a new iteration. This paper evaluates the performance of the transport module to reduce the execution time for data collection.

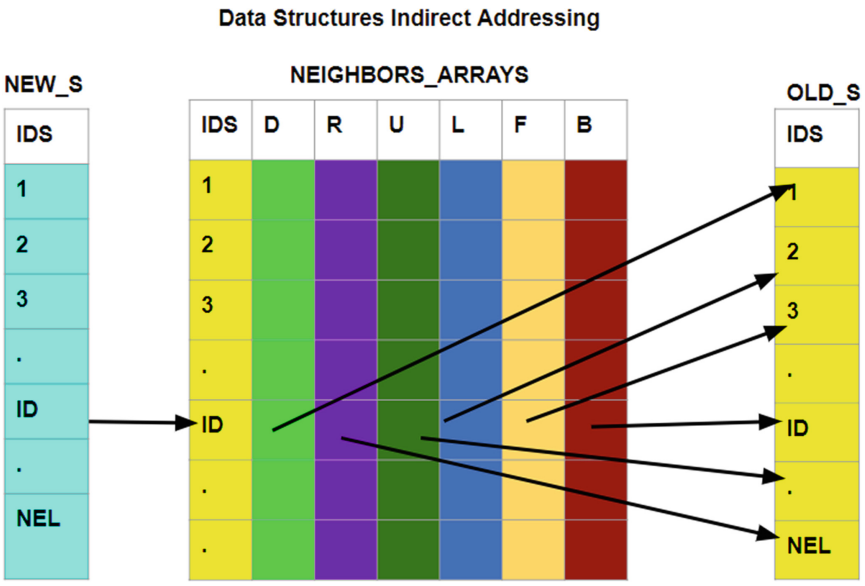


Fig. 2. Indirect access array.

Figure 2 illustrates the indirect access data structure used in the indirect access version of the numerical fluid-flow simulator. The array NEW_S contains the volumes' indices traversed by the numerical method. The variable IDS contains unique identifiers for each control volume, numbered from 1 to the total number of control volumes (NEL), and OLD_S is the array containing the data from the previous step. The numerical method developed by correa et al. [5] uses a 6-neighbor stencil to calculate the saturation of a control volume. Information about the location of neighbors is stored in the array $NEIGHBOR_ARRAY$, which is a two-dimensional array in which the first column contains the volume identifiers and the remaining columns D, R, U, L, F, and B are the abbreviations of down, right, up, left, front and back, respectively, and contain the data identifiers of the neighbors. At each iteration, the numerical method runs through the identifiers of each volume. Additionally, the numerical method uses the array $NEIGHBOR_ARRAY$ to locate the identifiers of the neighbors, which, in turn,

point to the data in the computer vector OLD_S, generating a high cache miss rate.

Algorithm 1. Numerical method with indirect access array.

input : $S_{old1} \ldots S_{oldN}$, **MVIZ**
output: $S_{new1} \ldots S_{newN}$ (saturation array)

Function `Numerical Method(`$S[\,]$`):`

for $EL \leftarrow 1$ *to* $NUMEL$ **do**

$NEW_S(EL) \leftarrow OLD_S(EL)$
$-(DT/DX) * V_{xp}(EL) * (OLD_S(EL) - OLD_S(\mathbf{MVIZ}(EL, 4))$
$-(DT/DX) * V_{xm}(EL) * (OLD_S(\mathbf{MVIZ}(EL, 2)) - OLD_S(EL)$
$-(DT/DY) * V_{yp}(EL) * (OLD_S(EL) - OLD_S(\mathbf{MVIZ}(EL, 5))$
$-(DT/DY) * V_{ym}(EL) * (OLD_S(\mathbf{MVIZ}(EL, 6)) - OLD_S(EL)$
$-(DT/DZ) * V_{zp}(EL) * (OLD_S(EL) - OLD_S(\mathbf{MVIZ}(EL, 1))$
$-(DT/DZ) * V_{zm}(EL) * (OLD_S(\mathbf{MVIZ}(EL, 3)) - OLD_S(EL)$

end
return NEW_S

Algorithm (1) presents only one loop varying the iterator from 1 to the total number of control volumes. The algorithm updates each polytope of the array NEW_S. Each control volume of the array NEW_S points to a location in the array **MVIZ**. In turn, the array **MVIZ** references the respective saturation array OLD_S of each neighbor volume of the control volume and $V_{xp}, V_{xm}, V_{yp}, V_{ym}, V_{zp}, V_{zm}$ are the velocity vectors of each control volumes.

Figure 2 illustrates the data access. When updating the saturation of a position of the array NEW_S, it is necessary to access the data of the saturation of two neighbor volumes using the array **MVIZ**. The matrix **MVIZ** contains locations of two neighbor control volumes that reference for positions of data in the array OLD_S, generating a high cache miss rate. It is possible to reduce this high cache miss rate using an adequate data structure, as used in this paper.

The direct access data structure used in the direct access version of the numerical method fluid-flow simulator uses a three-dimensional matrix that contains the NEW_S saturation data, and a three-dimensional matrix that contains the saturation data from the previous step OLD_S.

At each iteration, the numerical method developed in Correa and Borges work [5] uses a 6-neighbor stencil to calculate the saturation of a control volume. When searching for the saturation of a control volume of the matrix NEW_S, the method accesses the saturation data of two neighbor volumes through positions *(i-1,j,k)*,*(i+1,j,k)*,*(i,j-1,k)*,*(i,j+1,k)*,*(i,j,k-1)*,*(i,j,k+1)* of the matrix OLD_S, which are located contiguously in memory, reducing the cache miss rate.

Algorithm 2. Numerical method with direct access array.

input : $MSEW, MSOLD$
output: $MSNEW$ (saturation array)

Function `Numerical Method(`$S[\,]$`):`

```
for K ← 1 to NZ do
    for J ← 1 to NY do
        for I ← 1 to NX do
            MSNEW(I,J,K) ← MSOLD(I,J,K)
              −(DT/DX)∗Vxm(I,J,K)∗(MSOLD(I,J,K)−MSOLD(I−1,J,K))
              −(DT/DX)∗Vxp(I,J,K)∗(MSOLD(I+1,J,K)−MSOLD(I,J,K))
              −(DT/DY)∗Vym(I,J,K)∗(MSOLD(I,J,K)−MSOLD(I,J−1,K))
              −(DT/DY)∗Vyp(I,J,K)∗(MSOLD(I,J+1,K)−MSOLD(I,J,K))
              −(DT/DZ)∗Vzm(I,J,K)∗(MSOLD(I,J,K)−MSOLD(I,J,K−1))
              −(DT/DZ)∗Vzp(I,J,K)∗(MSOLD(I,J,K+1)−MSOLD(I,J,K))
        end
    end
end
return MSNEW
```

Algorithm (2) shows the direct access approach. Algorithm 1 traverses the three-dimensional matrix and accesses the volume's neighbors directly, which are located in positions *(i-1,j,k)*,*(i+1,j,k)*,*(i,j-1,k)*,*(i,j+1,k)*,*(i,j,k-1)*,*(i,j,k+1)*.

4 Results and Analysis

In this section, we evaluate the performance of the numerical fluid-flow simulator using the direct access data structure compared with the indirect access structure. The tests carried out in this study were conducted on the SDumont supercomputer, using two different architectures of nodes. The Cascade Lake architecture features two Intel® Xeon® Gold 6252 CPU (24 cores) processors running at 2.10 GHz and 384 GB of main memory. The Ivy Bridge architecture features two Intel® Xeon® E5-2695v2 Ivy Bridge (12c) processors running at 2.4 GHz and 64 GB of main memory. The nodes communicate between the processors through the INTEL® UPI channel, and we disabled the hyper-threading function.

As the present study evaluates the performance of MPI messages, we disabled the I/O operations and executed only the transport module. Thus, we disabled the velocity module and kept the input parameters constant. For the experiments presented in this study, we used a matrix containing $300 \times 300 \times 300$ control volumes using $NX = 300$, $NY = 300$, $NZ = 300$, and $NEL = 27,000,000$.

As shown in previous work [11], the present paper uses the domain decomposition technique called block decomposition [18]. Specifically, it used the scheme to test different vertical and horizontal cross-sections to evaluate the best block decomposition given a certain number of processes, which presents better performance.

This experiment used SDumont Supercomputer computational nodes for the Cascade Lake node architecture and the Ivy Bridge. The time measured for each test corresponds to an average of thirty runs.

Figure 3 presents the simulator execution time for 10, 24, 48, 96, and 150 MPI processes. The blue line represents computing load-balanced MPI processes domain decomposition for indirect access for Cascade Lake computer architecture node. The red line represents computing and communication load-balanced MPI processes domain decomposition for direct access Cascade Lake. The orange line represents computing load-balanced MPI processes domain decomposition for the indirect access Ivy Bridge computer architecture node. The green line represents computing and communication load-balanced MPI processes domain decomposition for direct access to Ivy Bridge.

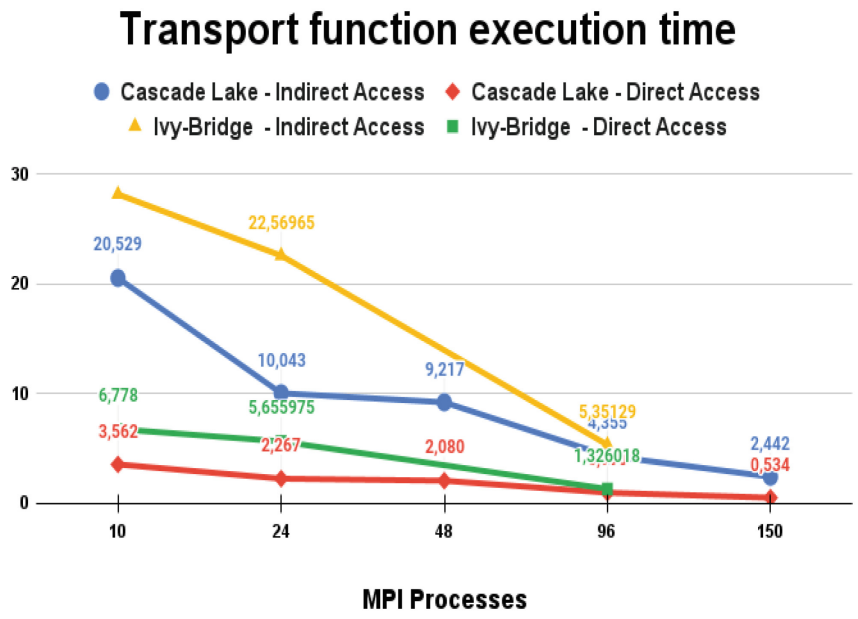


Fig. 3. Transport function execution time in the SDumont supercomputer.

As the number of MPI processes increases, the execution time gap between indirect and direct accesses decreases in both architectures. The main reason is that, for a fixed mesh size, as we increase the number of MPI processes, each MPI subdomain's volume decreases, reducing the number of cache misses on each core until a point there are no more cache misses at the last level cache. Still, the impact is more substantial on the Ivy Bridge architecture. The main reason is unsurprisingly the cache architecture.

We used the Intel® VTune Profiler tool to analyze the hardware events available on Intel® processors by a unit called the Performance Monitoring Unit (PMU), which counts various events that occur during execution and makes them available for profiling tools [24]. The events used are related to cache hit and miss counts generated by instructions (Intel® Xeon® Cascade Lake Gold 6252) and microinstructions (Intel® Xeon® E5-2695v2 Ivy Bridge).

In Sect. 4.1, we present the analysis that shows the reduction of cache miss rates of the algorithm with direct access compared to the algorithm with indirect access. In the Sect. 4.2, we show the comparison of the behavior of the simulator between Intel® Xeon® Cascade Lake Gold 6252 and Intel® Xeon processors textsuperscript® E5-2695v2 Ivy Bridge.

4.1 VTUNE Analysis: Cache Miss Evaluation

In this section, we present a study of the cache miss of the simulator running on a computational machine of the Santos Dumont supercomputer and using the direct access structure concerning the indirect access structure. The computational system used in this study features two Intel® Xeon® Gold 6252 processors, each containing 24 cores. Each L1 and L2 cache is accessed by only one core [7]. The LLC (last-level cache), is shared by all cores. The initial test uses 10 MPI processes, and each process executes in a core. With this test, we can analyze the cache miss rate in the LLC.

Due to the non-inclusive nature of LLC, the absence of a cache line in LLC does not indicate that the line cannot be present in any of the cores' private caches. Thus, the architecture design uses a snoop filter to search the cache lines in the L2 or mid-level cache (MLC) of the cores when the LLC does not contain the required data. On CPUs from the generation before Intel® Xeon® Skylake family CPUs, the shared LLC itself solves this task. With this feature, which allows for a reduction in the size of the LLC and an increase in the size of the L2 memory cache, compared to architectures before Skylake-SP, there is lower latency in read/write operations involving the cache [7].

In architectures prior to the Skylake-SP architecture, all of the lines in the L2 cache for each core were also stored in the inclusive and shared LLC cache. In Skylake-SP and Cascade Lake, each physical core has a 1 MB private L2 cache and all other cores share a larger LLC cache that can reach 38.5 MB (typically 1.375 MB/core). All lines stored in the L2 cache for each core may not be present in the non-inclusive and shared LLC cache. A larger L2 cache increases the hit ratio from the L2 cache, which results in lower memory latency and reduced demand on the LLC cache [7]. Nodes with these architectures comprise the SDumont supercomputer [16]. We collected some hardware events associated with memory usage employing the Intel® VTune Profiler tool.

Table 1. Instructions for collecting events in the Cascade Lake architecture.

Collect Cache miss events in Cascade Lake architecture	
L1 MISS COUNT	*MEM_LOAD_RETIRED.L1_MISS_PS*
L2 MISS COUNT	*MEM_LOAD_RETIRED.L2_MISS_PS*
LLC MISS COUNT	*MEM_LOAD_RETIRED.LLC_MISS_PS*

Table 1 shows the hardware events available in the Cascade Lake processor microarchitecture, which have load instructions terminated with at least one micro-operation that has generated faults in L1, L2, and LLC. Table 2 shows the collected results.

Table 2. Vtune analysis on one node with 10 processes.

Description	indirect access	direct access	Indirect/Direct ratio
L1 Miss Count	5808374246	1028930867	5.645
L2 Miss Count	3015415943	341943556	8.818
LLC Miss Count	2864200480	272019040	10.529
Transport time (s)	20,529	3,562	5,764

Table 2 exhibits that the cache miss count of L1 (L2) for indirect access is 5.6 (8.8) times higher than direct access. The results of the experiments show that the algorithm with direct access is 5.7x times more efficient than the code with indirect access.

Table 3. Vtune analysis on one node with 24 processes.

Description	indirect access	direct access	Indirect/Direct ratio
L1 Miss Count	5672670175	1002030060	5.661
L2 Miss Count	3362811788	365603489	9.198
LLC Miss Count	3392237440	200014000	16.960
Transport	10.043	2.267	4.431

Table 3 shows cache miss count using 24 processes, i.e., the number of physical cores of a processor of a computational node. Concerning the values of the Table 2, which shows tests with 10 MPI processes, the proportion remains constant for the L1 cache, and there is an increase of approximately 5% for the L2 cache and an increase of approximately 60% for LLC. This impact does not occur in the reduction of the execution time.

Table 4. Vtune analysis on one node with 48 processes.

Description	indirect access	direct access	Indirect/Direct ratio
L1 Miss Count	6129683885	1002030060	6.117
L2 Miss Count	3644880207	365603489	9.969
LLC Miss Count	3296230720	72005040	45.778
Transport Time	9.217	2.080	4.432

Table 4 shows the cache miss count for 48 processes, i.e., the number of physical cores of two processors of the computational node. Concerning Table 3, with tests with 24 MPI processes, there was an increase of approximately 5% for the L1 cache. Additionally, the proportion remains constant for the L2 cache, and one observes a growth of around 300% for the LLC. However, a similar impact did not occur in the reduction of the execution time.

Table 5. Vtune analysis on two nodes with 96 processes.

Description	indirect access	direct access	Indirect/Direct ratio
L1 Miss Count	7789333673	2180665418	3.572
L2 Miss Count	3813300914	591448304	6.447
LLC Miss Count	2880201600	0	–
Transport Time	4.355	0.990	4.397

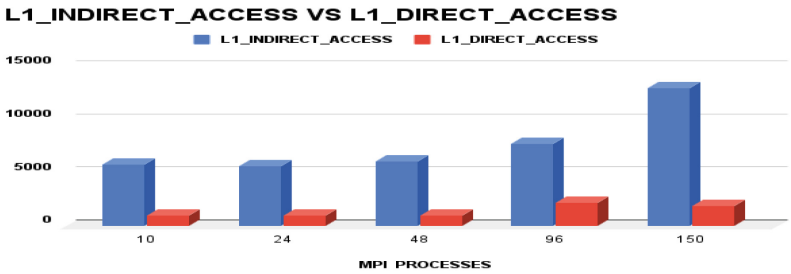
Table 5 shows a cache miss count for 96 processes, i.e., the physical cores of two processors of two computing nodes. Concerning Table 5, which shows the results with 48 MPI processes, there is a decrease of approximately 40% for the L1 cache, a reduction of approximately 30% for the L2 cache, and miss cache does not happen in LLC. However, a similar impact does not occur in proportion to the decrease in execution time.

Table 6. Vtune analysis on four nodes with 150 processes.

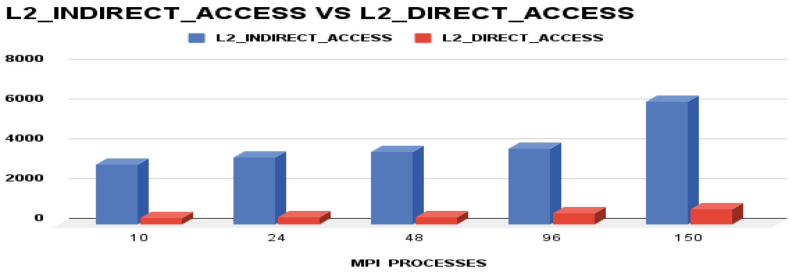
Description	indirect access	direct access	Indirect/Direct ratio
L1 Miss Count	13018290537	1870256106	4.107
L2 Miss Count	6153633441	765821510	3.670
LLC Miss Count	0	0	–
Transport Time	2.373	0.534	4.442

Table 6 shows a cache miss count for 150 processes related to the execution in four computer systems, with 38 processes in two nodes and 27 processes in the other two nodes. This division was necessary so that it was possible to divide the mesh into 150 processes. Concerning Table 4, which shows tests with the execution of 96 MPI processes, there is an increase of approximately 20% for

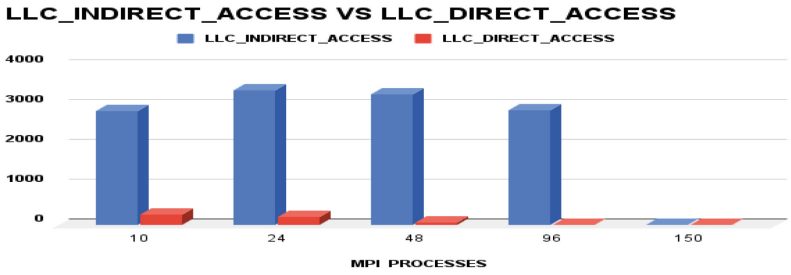
the L1 cache, a decrease of about 45% for the L2 cache, and cache miss did not happen in LLC. However, a similar impact did not occur in proportion to the decrease in execution time.



(a) Cache miss in L1



(b) Cache miss in L2



(c) Cache miss in LLC

Fig. 4. Cache miss evaluation.

Figure 4 shows plots that relate the cache miss rate as a function of the number of processes, which varies from 10 to 150, for the simulator executed with indirect access relative to direct access. Figure (a), (b), and (c) represents the cache miss rates for the L1, L2, and LLC memory caches. In the plots, the values are much higher for the simulator with indirect access.

4.2 Cache Miss and Cache Hit Evaluations on Intel® Xeon® E5-2695v2 Ivy Bridge and Intel® Xeon® Cascade Lake Gold 6252

In this section, we present numerical experiments to verify that the performance gains generated by direct access concerning indirect access are still more accentuated in a pre-Skylake generation architecture. The investigation evaluates the difference between the execution time for direct and indirect accesses for the Intel® Xeon® E5-2695v2 Ivy Bridge architecture concerning the Intel® Xeon® Cascade Lake Gold 6252 architecture. We analyzed the code with the Intel® Vtune [24] tool to validate the simulator on the two architectures.

Table 7. Instructions for collecting events (cache miss) in the Ivy-Bridge and Cascade Lake architecture.

Hardware events in Ivy-Bridge architecture	
L1 MISS COUNT	*MEM_LOAD_UOPS_RETIRED.L1_MISS_PS*
L2 MISS COUNT	*MEM_LOAD_UOPS_RETIRED.L2_MISS_PS*
LLC MISS COUNT	*MEM_LOAD_UOPS_RETIRED.LLC_MISS_PS*
Hardware events in Cascade Lake architecture	
L1 MISS COUNT	*MEM_LOAD_RETIRED.L1_MISS_PS*
L2 MISS COUNT	*MEM_LOAD_RETIRED.L2_MISS_PS*
LLC MISS COUNT	*MEM_LOAD_RETIRED.LLC_MISS_PS*

Table 7 shows the hardware events that report how many microinstructions of load generated cache miss in L1, L2, and LLC.

Table 8. Instructions for collecting events (cache hit) in the Ivy-Bridge and Cascade Lake architecture.

Hardware events in Ivy-Bridge architecture	
L1 HIT COUNT	*MEM_LOAD_UOPS_RETIRED.L1_HIT_PS*
L2 HIT COUNT	*MEM_LOAD_UOPS_RETIRED.L2_HIT_PS*
LLC HIT COUNT	*MEM_LOAD_UOPS_RETIRED.LLC_HIT_PS*
Hardware events in Cascade Lake architecture	
L1 HIT COUNT	*MEM_LOAD_RETIRED.L1_HIT_PS*
L2 HIT COUNT	*MEM_LOAD_RETIRED.L2_HIT_PS*
LLC HIT COUNT	*MEM_LOAD_RETIRED.LLC_HIT_PS*

Table 8 informs the hardware events used to collect the number of load microinstructions that will not generate any miss cache in L1, L2, and LLC.

The cache miss counts and the cache hit counts for L1, L2, and LLC in the Ivy-Bridge architecture was collected so that it was possible to calculate the

miss/hit ratio. Table 9 shows the ratio for the two versions of the method, i.e., using direct and indirect accesses. The last line shows that the version with direct access is approximately four times faster than the version with indirect access. The indirect/direct ratio of the miss/hit ratio is always greater than one for the three cache levels, evidencing that the version with direct access presents less cache error concerning the hits when compared to the implementation with indirect access.

Table 9. Vtune analysis on one node with 10 processes using Ivy-Bridge architecture.

Vtune analysis on intel® Ivy-Bridge architecture			
Description	indirect access	direct access	Indirect/Direct ratio
L1 Miss Count/L1 Hit Count	0.018	0.017	**1.069**
L2 Miss Count/L2 Hit Count	2.835	1.243	**2.280**
LLC Miss Count/LLC Hit Count	1.557	0.953	**1.634**
Transport time(seg)	28.151	6.778	**4.152**

The cache miss count and the cache hit count for the L1 and L2 memory caches of the Ivy-Bridge architecture were collected Table in 9, so that it would be possible to calculate the miss/hit ratio, which gives us the cache miss rate concerning the hit rate.

Table 10. Vtune analysis on one node with 10 processes using Intel® Xeon® Cascade Lake Gold 6252.

Vtune analysis on intel® Xeon® Cascade Lake Gold 6252 architecture			
Description	indirect access	direct access	Indirect/Direct ratio
L1 Miss Count/ L1 Hit Count	0.008	0.000	**13.320**
L2 Miss Count/L2 Hit Count	1.099	0.492	**2.232**
Transport time(seg)	20.529	3.562	**5.764**

Table 10 shows the ratio for the two implementations of the simulator, i.e., using direct and indirect memory-access. The last line shows that the version with direct access is approximately 5.8 times faster than the version with indirect access. The indirect/direct ratio of the miss/hit ratio is always greater than 1 for the two cache levels, evidencing that the version with direct access presents less cache error concerning the hits compared with the implementation employing indirect access.

Table 11 shows a ratio for Ivy Bridge/Cascade Lake of the percentages of the cache miss concerning the cache hits for levels L1 and L2. All the numbers are higher than 1 for both direct and indirect accesses versions. The Table emphasizes the value for the L1 level of the implementation with direct access (26.091), suggesting that this version takes more advantage of the cache hierarchy than the version with indirect access in the Cascade Lake processor architecture, which

Table 11. Ratio of Ivy Bridge/Cascade Lake of the cache miss rate concerning cache hit rate for L1 and L2 memory caches.

Description	indirect access	direct access
L1	2.094	**26.091**
L2	2.579	2.525

reflects in the execution times. The direct access version is approximately 5.8 times faster on Cascade Lake (see Table 10). Furthermore, it is around four times faster on Ivy Bridge (Table 9), confirming what we mentioned in Sect. 4.1.

5 Conclusions and Future Works

Adopting the direct access data structure strategy improved the performance and scalability of the numerical fluid-flow simulator concerning using an indirect access data structure for distributed memory architectures. We presented a Vtune cache miss analysis. The analysis showed that the main reason for the results was not surprisingly the low cache miss rate when using the direct access data structure. We also showed that the impact of decreasing the number of cache misses is even more substantial for previous Intel® Xeon® Gold 6252 processors architectures that present inclusive cache LLC.

The experiments presented in this paper shows that by implementing direct access data, we can further improve the execution time of the numerical fluid-flow simulator that uses the optimized domain decomposition technique presented in previous work [11].

We intend to improve the communication between MPI processes using advanced techniques, such as the adaptive MPI, based on a previous investigation [13] to provide further scalability to the present study. Moreover, the present study is part of a research effort to develop a computational environment for academic studies to provide scalability to the new numerical methods developed in the oil and gas field. We are developing a scalable code written in the Fortran language to approach the solution of the advection problem using the well-known first-order finite volume method upwind [14]. We also plan to implement numerical methods of higher order in the code, as recommended in previous publications [3,5].

As can be seen in the Algorithm 1, from the direct access data structure, the calculation of each control volume of the three-dimensional matrix is carried out through the path of the indices (i, j, k) of the saturation matrix. In future work, we intend to evaluate the strategies developed by Respondek [19] to increase the locality of access to data in the saturation matrix. We also want to develop research to evaluate the performance of the code for vectorization instructions such as Intel(AVX) and develop studies for other processor architectures such as the ARM architecture [20].

Authorship Statement. The authors confirm that they are the sole liable persons responsible for the authorship of this work, and that all material that

has been herein included as part of the present paper is either the property (and authorship) of the authors or has the permission of the owners to be included here.

Acknowledgement. The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources for the SDumont supercomputer. The use of HPC resources contributed significantly to the research results reported in this paper. URL: http://sdumont.lncc.br. This study was financed in part by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* - Brasil (CAPES) Finance Code 001.

References

1. Abreu, E.C.d., et al.: Modelagem e simulação computacional de escoamentos trifásicos em reservatórios de petróleo heterogêneos. Ph.D. thesis, Universidade do Estado do Rio de Janeiro (2007)
2. Borges, M., Furtado, F., Pereira, F., Souto, H.A.: Scaling analysis for the tracer flow problem in self-similar permeability fields. Multiscale Model. Simul. **7**(3), 1130–1147 (2009)
3. Carneiro, I.B., Borges, M.R., Malta, S.M.C.: Aplicação de métodos de alta ordem na resolução de problemas bifásicos. Proc. Ser. Braz. Soc. Comput. Appl. Math. **6**(2) (2018)
4. Causon, D., Mingham, C.: Introductory Finite Difference Methods for PDEs. Bookboon (2010)
5. Correa, M., Borges, M.: A semi-discrete central scheme for scalar hyperbolic conservation laws with heterogeneous storage coefficient and its application to porous media flow. Int. J. Numer. Meth. Fluids **73**(3), 205–224 (2013)
6. Dagan, G.: Flow and Transport in Porous Formations. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-75015-1
7. Mulnix, D.L.: Intel®Xeon®processor scalable family technical overview (2017). https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html
8. Dupuis, A., Chopard, B.: Lattice gas: an efficient and reusable parallel library based on a graph partitioning technique. In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B. (eds.) HPCN-Europe 1999. LNCS, vol. 1593, pp. 319–328. Springer, Heidelberg (1999). https://doi.org/10.1007/BFb0100593
9. Durlofsky, L.J.: Numerical calculation of equivalent grid block permeability tensors for heterogeneous porous media. Water Resour. Res. **27**(5), 699–708 (1991)
10. Gelhar, L.W., Axness, C.L.: Three-dimensional stochastic analysis of macrodispersion in aquifers. Water Resour. Res. **19**(1), 161–180 (1983)
11. Herrera, S., et al.: Optimizations in an numerical method code for two-phase fluids flow in porous media using the sDumont supercomputer. In: CILAMCE 2021-PANACM 2021 Proceedings of the XLII Ibero-Latin-American Congress on Computational Methods in Engineering and III Pan-American Congress on Computational Mechanics, ABMEC-IACM Rio de Janeiro, Brazil, 9–12 November 2021. CILAMCE (2021)
12. Herschlag, G., Lee, S., Vetter, J.S., Randles, A.: GPU data access on complex geometries for D3Q19 lattice Boltzmann method. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 825–834. IEEE (2018)

13. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24644-2_20
14. LeVeque, R.J.: Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems. SIAM (2007)
15. Martys, N.S., Hagedorn, J.G.: Multiscale modeling of fluid transport in heterogeneous materials using discrete Boltzmann methods. Mater. Struct. **35**, 650–658 (2002)
16. Osthoff, C., et al.: A arquitetura do supercomputador sDumont e os desafios da pesquisa brasileira na área de computação de alto desempenho. In: Anais da XI Escola Regional de Alto Desempenho de São Paulo, pp. 1–5. SBC (2020)
17. Pan, C., Prins, J.F., Miller, C.T.: A high-performance lattice Boltzmann implementation to model flow in porous media. Comput. Phys. Commun. **158**(2), 89–105 (2004)
18. Parashar, M., Yotov, I.: An environment for parallel multi-block, multi-resolution reservoir simulations. In: Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems (PDCS 98), Chicago, IL, International Society for Computers and their Applications (ISCA), pp. 230–235 (1998)
19. Respondek, J.: Matrix black box algorithms - a survey. Bull. Pol. Acad. Sci. Tech. Sci. **70**(2), e140535 (2022)
20. Ryzhyk, L.: The Arm architecture. Chicago University, Illinois, EUA (2006)
21. Schulz, M., Krafczyk, M., Tölke, J., Rank, E.: Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high-performance computers. In: High Performance Scientific and Engineering Computing: proceedings of the 3rd International FORTWIHR Conference on HPSEC, Erlangen, 12–14 March 2001, vol. 21, pp. 115–122. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-55919-8_13
22. Shashkov, M.: Conservative Finite-Difference Methods on General Grids, vol. 6. CRC Press (1995)
23. Silvester, P.P., Ferrari, R.L.: Finite Elements for Electrical Engineers. Cambridge University Press (1996)
24. Intel Corporation Site: performance monitoring events supported by Intel performance monitoring units (PMUs). https://perfmon-events.intel.com/
25. Message Passing Interface Forum: MPI: a message passing interface. In: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, pp. 878–883 (1993)
26. Tuane, V.L.: Simulação Numérica tridimensional de escoamento em reservátorios de petróleo Heterogêneos. Master's thesis, LNCC/MCT, Petrópolis, RJ, Brasil (2012)
27. Tuszyński, J., Löhner, R.: Parallelizing the construction of indirect access arrays for shared-memory machines. Commun. Numer. Meth. Eng. **14**(8), 773–781 (1998)
28. Vázquez-Cendón, M.E.: Solving Hyperbolic Equations with Finite Volume Methods. U, vol. 90. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14784-0